Multiparty Computation over $\mathbb{Z}/2^k\mathbb{Z}$

Ph.D. Thesis

Daniel Escudero

Abstract

The main topic of study in this thesis is *Secure Multiparty Computation*, or MPC for short. This is a set of techniques that enable a set of mutually-distrustful parties to securely compute a given function of their choice, without leaking any information about the inputs provided to the computation beyond what is leaked from the output itself. These tools are extremely useful in many applications where privacy of data is required, but different operations on this data must still be performed.

The field of MPC has a rich and fruitful research history. Starting with the seminal work of Yao in 1982, a large body of works has taken care of expanding the knowledge barrier in MPC in many directions, including the development of new techniques, discovering new inherent limitations, improving the efficiency of existing techniques, among others.

In spite of such a long and successful series of studies, most MPC techniques share in common a simple yet restrictive limitation. Mathematics play a central role in the development of MPC protocols, and in particular, most MPC techniques require the desired computation to be performed over a "nice enough" algebraic structure. This is typically represented by means of *finite fields*, which include the natural case of arithmetic modulo 2 over $\{0, 1\}$, and more generally include integer arithmetic modulo a prime number p. Unfortunately, this type of arithmetic is not very natural for many applications, and moreover, it is not "directly compatible" with modern computer architectures where native operations are typically arithmetic modulo 2^{32} or 2^{64}.

In this thesis we explore the task of designing MPC protocols when the computation domain is a ring of the form $\mathbb{Z}/2^k\mathbb{Z}$, that is, integers modulo 2^k. This algebraic structure is not a field, and in fact, it has a lot of undesirable properties that complicate the task of protocol design in this setting. On the positive side, this ring is more directly compatible with native datatypes in modern computer architectures such as `int32` or `int64`, which can naturally lead to further improvements in the efficiency of different MPC protocols. Additionally, arithmetic modulo a power of two is more "compatible" with binary arithmetic, which is a central building block in many applications.

The results of this thesis include a series of MPC protocols in a wide variety of security scenarios for computation over $\mathbb{Z}/2^k\mathbb{Z}$. These settings include passive and active corruption for t corruptions where $t < n/3$, $t < n/2$ and $t < n$. Special cases where the number of parties is small are also considered, and specialized protocols for different subcomputations that appear thoroughly in many applications are presented, taking complete advantage of the fact that the computation domain is $\mathbb{Z}/2^k\mathbb{Z}$, instead of a finite field.

As a bonus, and to compare our novel techniques with previous works over finite fields, several existing techniques over these domains that can be considered essential in the area are presented.

Abstrakt

Hovedemnet for denne afhandling er Sikker Flerparts Beregning, eller MPC fra engelsk "Secure Multiparty Computation". MPC omfatter en række teknikker, der tillader en gruppe indbyrdes-mistroisk deltagerer, at beregne en vilkårlig funktion, uden at lække noget information funktionens input, ud over hvad der naturligt lækkes via funktionens output. Disse teknikker er yderst brugbare i mange applikationer hvor data privathed er et krav, men hvor beregninger på disse data er nødvendige.

MPC som forskningsfelt har en rig historie. Med en begyndelse i det skelsættende værk af Yao fra 1982, har en stor samling forskning udvidet forståelsen for MPC i mange forskellige retninger med udvikling af nye teknikker, opdagelse af fundamentale begrænsninger, effektivisering af eksisterende teknikker, o.s.v.

På trods af denne lange og succesfulde række af forskningsværker, deler mange af de teknikker der bruges i MPC stadig en fundamental begrænsning. Matematik spiller en central rolle i udviklingen af MPC protokoller, og specielt kræver de fleste MPC protokoller og teknikker en algebraisk struktur der er "pæn nok". Dette betyder oftest, at endelige legemer, såsom modulo 2 over 0, 1 og mere generelt modulo et primtal p, anvendes. Denne type aritmetik er dog ikke naturlig i mange sammenhæng, og hvad mere, er ikke "direkte kompatibel" med moderne computer arkitekturer, hvor aritmetik foretages modulo 2^{32} eller 2^{64}.

I denne afhandling undersøges det, at udvikle MPC protokoller hvor domænet for beregninger er en ring af formen $\mathbb{Z}/2^k\mathbb{Z}$, altså, heltal modulo 2^k. Denne algebraiske struktur er ikke et legeme og indeholder en masse negative egenskaber der vanskeliggør protokol design. På den mere positive side, så er denne ring direkte kompatibel med de datatyper der er indbygget i moderne computere, såsom int32 og in64, hvilket gør beregninger mere effektive. Hvad mere, aritmetik modulo 2^k er mere kompatibel med binær aritmetik (aritmetik modulo 2), som er en naturlig byggeblok i mange applikationer.

Resultaterne der præsenteres i denne afhandling indebærer en række af MPC protokoller for beregning modulo 2^k for mange forskellige sikkerhedsmodeller. Disse sikkerhedsmodeller omfatter både "passive" og "aktiv" sikkerhed, og med korrumperings tærskler på både $t < n/3$, $t < n/2$ og $t < n$. Derudover undersøgers der også scenario hvor antallet af beregnings parter lille, og under-protokoller der anvendes i mange forskellige typer beregninger bliver præsenteret. I disse udnyttes der fuldt ud, at beregningsdomænet er $\mathbb{Z}/2^k\mathbb{Z}$ i stedet for et endeligt legeme. De nye teknikker i denne afhandling bliver sammenlignet med tilsvarende teknikker for endelige legemer.

Som en bonus, så gives der også en omfattende præsentation af eksisterende teknikker for sikker beregninger modulo 2^k.

Preface

I cannot believe I have finally managed to assemble together a +200 pages thesis. This has been quite a fun journey, full of countless ups and downs, and it just feels nice to see this stage coming to an end, leaving room for what will come next.

I grew up in a small city in a not-so-small country called Colombia. Somehow, I have managed to finish a Ph.D. under the supervision of one of the most renowned cryptographers around the globe, Ivan Damgård. This has been a long story of sadness, frustration, happiness, and many other emotions that tend to contradict each other. I felt the need of writing down my experience through this journey, my emotions, crucial events and central people that helped me through this path. This preface is intended to give an overview of this adventure. Lots of things can change in the course of 4.5 years, and I am very happy for being able to write this foreword from the position in which I am right now: writing a dissertation. Additionally, there are several actors who played a fundamental role in the development of this story, and this preface also serves as a place to honor them.

I apologize with the reader in advance for the rather informal style in which the preface below is written. It felt more organic, more natural, and more affine to how my thoughts were lying in my head. I also apologize for the density of the text, which feels more like a diary than a formal Ph.D. thesis preface.

Undergrad and Early Master Years

I studied Mathematics at university, and somehow managed to get involved with the exciting field of Cryptography thanks to my professor, bachelor/master thesis advisor, and friend, Daniel Cabarcas. His guidance and support were crucial as I was starting to learn concepts outside textbooks and exercises. Back then, learning cryptography was only getting more and more exciting: as a soon-to-be mathematicians, the field contained many of the great areas I enjoyed studying and learning about like number theory, discrete mathematics, algebraic geometry, graph theory, and many others, but on top of this it also allowed me to get started with an area that a part of me had been desiring for a while, which is computer science.

Back at the time, no university in my country offered a proper computer science degree (something that has changed in recent years thanks to many forces getting together,

among them, crucially, that of Daniel Carbarcas), and the closest thing was Systems Engineering. After two failed years of attempting to become a chemical engineer (before turning into mathematics), I had decided I did not want to become any kind of engineer. However, I still liked computers, and I enjoyed all basic programming courses I had a the math department, although I wanted to get myself much more involved with the topic. Given this, cryptography, for me, was also a way to learn about more applied subjects, while still being able to use and learn advanced mathematical tools and constructions.

Daniel Cabarcas advised me well in the field of Multivariate Public Key Cryptography, enabling me to travel abroad for the first time for a conference and publish my first paper [12] when I was still an undergrad. I am extremely grateful to Daniel Cabarcas, not only for all the help and guidance he provided during my early academic years, but for all the lessons he taught me about other non-cryptography-related topics such as academia in Colombia, settling in the US, and many, many others.

In the same note, although they are not cryptographers, I would like to thank Professors Carlos Velez and Fernando Morales, who taught me so much through their courses and our personal interactions. I'm grateful for the wonderful friendship we have maintained even to these days.

Looking for a PhD

I managed to learn a lot about multivariate public key cryptography and eventually I felt very confident in that particular area. However, all my focus was put entirely on this specific field, which limited quite a bit the options I had for studying abroad, which was one of my oldest dreams at the time. It was not until I met Aisling Connolly and Pooya Farshim, both great professional cryptographers and amazing people, that I started to learn about all the other exciting ideas and opportunities present in other parts of cryptography. I met Aisling and Pooya at CryptoCO 2016, a cryptography school that took place in Bogotá, Colombia. Walking around the city, we had the chance to talk about future plans, goals and aspirations, and very quickly I got a whole world full of options outside multivariate public key cryptography, but still, inside cryptography itself. We sat at a cafe in La Candelaria, and Pooya wrote for me in a piece of paper (which I still have with me) different places to apply to, book recommendations, websites, resources, etc.

The support I obtained from Daniel Cabarcas, and then by Aisling and Pooya, were crucial for my next step. Several universities were listed in the note that Pooya gave me, and I applied to many of them. One of them, however, was Aarhus University, were I ended up studying my PhD. I wrote Claudio Orlandi with a motivation letter plus a recommendation letter from Pooya, and I was invited to visit the university in November 2016. I was excited, but unfortunately, the visit turned out to be some kind of a disaster! (or at least, as usual, that is how it felt to me). First, the presentation talk I gave was rushed and I did not give a good impression of myself. What was worst is that, even though I felt quite confident in the subfield I had been working on, it was during my visit that I noticed that my background in cryptography in general was very low. I did not even know what MPC was, or that it even existed. I also did not know about Shamir secret sharing, zero knowledge proofs, or many other "essential" concepts in cryptography. Besides, I was

very disconnected from practice: I did not even know that SHA was a family of hash functions, and the acronym AES did not ring any kind of a bell.

The lack of background prevented me from having any meaningful conversations with members of the group, when I visited. I talked with Ivan for only like 5 minutes, before he left for a trip. I only met once with Claudio, my host, and the conversation was not very fruitful. Other members of the group were also friendly but, again, conversations would be rather superficial. My week visiting Aarhus very quickly became simply me getting to know the university and the city, rather than having any meaningful academic engagements.

In spite of this, however, two months after my visit I received a notice that informed me that I had been accepted as a Ph.D. student under the supervision of Ivan Damgård and co-supervision of Jesper Nielsen. This came as a surprise, an extremely pleasant one, so I started planning for my next steps.

PhD: Early Years

Right after joining Aarhus University, my background was still a bit too weak, so my first full year was basically devoted to me building an appropriate profile for studying cryptography at the level in which my current position required. This was a painful and frustrating experience: I had the title of a Ph.D. student, but all other Ph.D. students around me were simply doing so well! We had weekly meetings were we shared to each other what we were all working on, and, while my reports were typically of the kind "just reading papers/books", many other members would talk about submissions to great conferences, progress on exciting projects, and many other cool things. Sometimes I would feel so frustrated after this meeting that my whole day would be essentially lost in terms of productivity.

I met very little with my advisor, which made me reach some depressing conclusions like that he was disappointed of his new student. Today I understand that all this was *imposter's syndrome*, and that most of us go through it in one way or another. However, knowing this does not necessarily make it less painful (and, in fact, it does not prevent me from feeling it as of today, from time to time).

The task of getting an adequate level of prior knowledge in the area of secure multiparty computation, which was the field I wanted to work on, quickly became a daunting task. Most of the resources were papers, which were either too old or too complex if the only thing an uninitiated novice like me wanted was to get a general idea of the field and some of its constructions and results. As a statement of how hard I tried to get this knowledge, I must confess I read several "essential resources" in full detail, like the book [34], the original UC paper by Canetti [24], the simulation based tutorial by Yehuda Lindell [67], and many more. I still preserve some of the printed papers and books with a bunch of notes and highlighted passages, many of which, I remember, meant absolutely nothing back at the time. Furthermore, many of which, even today, I do not fully understand.

It is motivated by this difficulty I went through that I decided to add to my thesis a general

and rather intuitive introduction to (secret-sharing-based) secure multiparty computation, which can be found in Part I. I like disseminating knowledge, and I already started sharing knowledge to a broader audience through a set of blog posts, for example [49]. I find this thesis to be a chance to get a more formal write-up on the topic, which may serve as a starting point to a more solid stand-alone resource in the future. This should prove useful for future students, or in general, people in the desire of learning MPC, with a similar background to the one I had when I started in the field.

PhD: Middle Years

It was not until Ivan came to me with the seeds to build the SPDZ2k protocol [32], that I got engaged in serious research activities. At that point, Peter Scholl, now professor at Aarhus University, joined the group as a postdoc, and he happened to be an expert in the particular MPC setting that SPDZ2k was aiming to. His guidance proved crucial in the development of my research skills in the area of MPC, and it is thanks to him that I slowly acquired some level of self-confidence so that the painful experience that my Ph.D. had been so far finally became something I could enjoy.

After my first project, several other works started to appear. I began to establish new research collaborations, think about new ideas and learn about new topics. These "golden years" gave me the hope and confidence I was so fiercely looking for, and allowed me to picture myself ahead, plan for visits, internship, and other research activities for the nearby future. Multiple wonderful things happened in my personal life during this time also. For example, I got married, and became the uncle of two wonderful little girls.

PhD: Final Years

With the appearance of COVID in 2019, several of my plans started to fail. For example, I was planning to visit Abhi Shelat at Northeastern University for around six months in mid 2020, and due to the pandemic this had to be canceled. Many other projects had to be cut off, but the hardest hit came when we were sent to work from home. Being away from my family, and living in a small studio in Aarhus at the moment, I decided to travel back home on April 2020 before doing so became impossible. This was a good move: I was living with my wife, we had a good working environment and we managed to keep the productivity. As time passed, however, the effects of the isolation became more evident. The time difference with Europe prevented me from joining regularly our group meetings, and several joint projects started to get delayed due to the difficulties of collaborating via a video call. Conferences were not happening in person anymore, so establishing new research connections was also, for me at least, an impossible task.

For June 2020 I was already in the need of returning to Denmark (since in-office working was being slowly redeployed), but traveling restrictions made it impossible for me to do so before September 2020. After returning to Denmark, I had a few weeks of great research interactions with different members of my group, but this did not last much,

since in November 2020 we were, once again, sent to work from home. I only remember Christmas being a very dark time. After some cold months of isolation and not a lot of productiveness, we decided to travel back to Colombia in February 2021, where I stayed until the end of my Ph.D. studies. Being at home, naturally, brought many positive aspects, but it only kept feeding the feeling of being isolated from my group and from the research community in general.

Due to these factors, I do not feel as confident as I used to do about one year ago. I sincerely look at the future, and hope things really improve, for the sake of me and all the people whose confidence and motivation took a considerable hit due to the different consequences of the pandemic.

Acknowledgments

A large percentage of the individuals I have engaged with in a meaningful conversation during the course of my Ph.D. have contributed greatly in one way or another. I would like to thank all the members of the Cryptography and Security group at Aarhus University, who showed me that creating such a fraternity where we could all be friends, share ideas with each other, have lunch together, party, play games, and so much more, was possible within a research group. This would not be possible without the amazing leader and founder of the group, my advisor Ivan Damgård, to whom I am not only grateful for his amazing job as an advisor and for serving as an "on-demand living crypto library", but also because of his patience and confidence in me. It happened more than once that I reached his office (or sent a long email) with the topic "am I doing OK?", "are you happy with me as your student?", "I am doing things slower than expected, right?", and every single time he would have a positive and concise answer that would get me back on track. We also had several music jam sessions together, which will last in my memory forever, with the dream of them happening again in the future.

Peter Scholl played a crucial role in my development as a PhD. He was, besides Ivan, the one who made me feel that it was OK to make "stupid" questions, to simply not know things. I would knock on Peter's office more than once per day, sometimes asking simple questions, some other times just looking to chat, and he would always be there smiling and open for a conversation. I had the pleasure to see him grow as a researcher, to see him become a professor, an event for which I was extremely proud, not only for him, but also for Aarhus University and my research group. We held several meetings (even during the pandemic) where I would talk about my insecurities, future plans, current concerns, and many other topics, and he would always excel at his job of helping me regain confidence. I thank Peter for all this and much more, and wish him all success in the prolific career he is building. He is a strong researcher, but most importantly, a great advisor and an amazing friend (I only complain that he declined so many times my invitations to play music).

I would like to also thank Anders Dalskov, who obtained his Ph.D. from my group on late 2020. Although I had a good relation in general with all the members of the group, it was with Anders with whom I would build the strongest friendship. We worked on many projects together, we complemented each other very well in terms of background

and expertise, and we really liked collaborating. Sometimes we would share a coffee at the office that would last hours, simply talking about work, cryptography, projects, life, science, politics, or any other topic that came to our minds. We shared movies, walks, and great moments, and I am just extremely happy I got the chance to establish such a strong bond with a nice person such as Anders. He definitely made my time in Denmark, a country so much different than where I was born and lived during all my pre-Ph.D. life, a wonderful journey.

There are so many important mentions, and unfortunately it is hard to remember them all. Claudio Orlandi was always there to guide me a provide a helping hand. Diego Aranha engaged in several conversations regarding academia in Latin America, and helped me reach conclusions and make decisions about my own future. Ronald Cramer allowed me to visit his group at CWI, Amsterdam more than once, and helped me a lot to grow as a researcher. Mark Abspoel, from the same group, also proved to be an excellent colleague, and we shared a lot even outside academia. Dorthe, our first secretary, was extremely helpful when I visited the group and when I joined, and Malene, our current group manager, continues being amazingly helpful and kind, for which I thank these two wonderful persons.

I thank the team involved in my summer internship in Bar Ilan University, on 2018. This includes Yehuda Lindell, who organized the event, but also Assi Barak, who would be extremely helpful and kind, and would even invite Anders and me for future visits. The other interns in this program were also extremely nice and I am happy I have managed to keep in touch with them. Special mention to Sameer Wagh, who is very much loved by my wife and I due to his amazing charisma and helpful attitude, and Eysa Lee, who has shared with me so many conversations about so many different topics that I find it hard to keep track of them today. I am also generally greateful to all the different coauthors I have engaged with, and to all the people I have held research and non-work related discussions during the course of my studies.

I would like to thank my mother Nibia and my siblings Angie, Suly and Juan, since they never doubted me and they supported me through every single step. They have radically different backgrounds than me, and there was always a technical barrier that disallowed me from getting too close to them in terms of what I was doing, conferences I was attending, projects I was working on, etc. However, their unconditional support was always present and it is something I always appreciated. To them, I have always been the top mathematician in the world, and it is through their eyes that more than once I have managed to recover myself from a momentary loss of self-confidence.

Now, I decided to save the most special mention for the end. I would like to thank my wife, Cristina Ochoa, for the invaluable support, not only through the PhD, but through every single step that I have taken since we met each other in 2012. Back then I was struggling with many aspects of my life, like economical, career-wise but also emotional and related to family. Cristina Ochoa appeared in a critical moment, where these and several other things were starting to take a noticeable toll. Ever since then, she has supported me with every single step in my life, no matter how big or how small.

Cristina Ochoa helped me with the idea of switching to Mathematics, and kept me motivated throughout the process. We have always shared everything to each other, and

she has been crucial in helping me reach different decisions and conclusions in many contexts. Cristina Ochoa was aware of the university courses I was taking and the topics I was interested in. She would provide feedback on all these different issues, and in spite of these being far from her field of interest and expertise, which are Social Sciences, she would make an effort in understanding as much as possible the different activities and projects I was being engaged in, and all the different concerns and struggles I was facing.

Cristina Ochoa supported the idea of me traveling abroad for studying a PhD, in spite of us spending so much time being physically together, until then. We continued sharing most of our time, however, but now through virtual means. It was thanks to her that I could manage the hard hit of moving countries, changing cultures, and starting a new Ph.D. program. She was aware of every single struggle I had, she provided great feedback and advice on these matters, and shared the pain and frustration with me when it was necessary. She celebrated times of success, condemned times of failures and uncertainty, and would be the crucial interlocutor that would simply listen to and comment on every single thought I had the need of sharing.

I do not think I could have gone through a Ph.D. program in a foreign country without counting on someone like Cristina Ochoa who could share the experience with me. This Ph.D. is as much hers as it is mine. Thank you[TAI].

Contents

Introduction

An enormous amount of natural questions surrounding information and communication have arisen throughout the history of humanity, questions that have played a pivotal role in bringing us to where we are today, boosting many areas and branches of arts and science. For example, the development of language as we know it and use it seamlessly today, the transmission of information and knowledge from generation to generation, and even the ability to "talk to ourselves", have enabled the construction of societal groups that boosted mutual development, allowing us to store and refine skills and expertise through education and communication.

After different groups started spreading, forming communities across noticeably far locations, the need of developing methods to transmit messages over long geographical distances became obvious. Several approaches have been considered throughout the history of humanity to approach this task. For example, with the construction of important roads and highways, like the Royal Road, built by Darius the Great of 550 BC, communication over long distances was truly boosted thanks to strategically placed posting stations that would relay information to each other. Other forms of long distance communication included visual cues, like fires or heliographs (like a mirror, used to reflect sunlight), and these have been used to warn attacks, ask for help, or send several other messages through some sort of binary code.

Thanks to several advances in topics like the ones described above, coupled with other major developments in fields like electricity and radio waves, we can enjoy today of countless amenities that were completely unthinkable to previous generations. It suffices to consider, for example, the deployment of radio-based communication, the widespread use of personal computing devices, or the astonishing global network called internet, together with all the applications that have been built on top, like electronic email, instant messaging, payment and banking services, and many, many more.

From our discussion above we see that information and communication have played a central role in an incredibly large portion of humanities' most relevant advances across a wide range of disciplines. Without the ability to communicate information between us, to store it, manipulate it, and interpret it, it is very hard to imagine any possible advance in other areas of knowledge, and in fact, it is even harder to define the concept of knowledge itself. However, because of this very same reason, information also becomes a source of great power, and due to the competitive nature of human beings, it becomes necessary to somehow "secure" it. This led to the development of *cryptography*.

There are multiple settings in which the need to secure information and communication becomes evident, with one of the most prominent examples throughout history being *war*. Conflicts constitute scenarios where a lot of distributed parties have to share knowledge with each other in order to reach their main goal, victory. Strategies, commands,

military intelligence, and many other forms of information have to be constantly stored, manipulated and communicated, and to maintain an advantage over the enemy it is typically the case that this data cannot be leaked beyond the internal parties authorized to access it. There are simply too many examples across the history of humanity that illustrate the use of cryptography to protect information in war-related contexts, and in fact, this has been, historically speaking, one of the most relevant use cases of these techniques. For a thorough exposition of many of such examples we refer the reader to Simon Singh's great resource, The Code Book [76].

In general, cryptography in the context of war emerges mostly as a method to hide information and communication from unwanted prying eyes. This is captured by the notion of *confidentiality*, and it is necessary since other entities could cause some *harm* should they have access to certain given knowledge. However, protecting information is not only relevant in scenarios where some damage can be caused if not done otherwise. Indeed, this notion has also been of major importance in less "destructive" settings. In some scenarios, even if no harm between humans was ever done and no concerns about bad use of information existed, ensuring confidentiality turns out to be important for certain system or social construct to exist. A good example of this is *economics*, whose development also contributed greatly to all of humanity advancements. Having the ability to trade goods and services, establish currencies, and founding the notion of a person's "worth", is one of the most important pillars of our development as a society. However, for such an infrastructure to exist, it is typically necessary to ensure the confidentiality of several types of information. For example, it is important to have competing companies as this ensures a good quality of service, but to achieve this it is necessary that different companies cannot spy on each other, as this allows them to face the challenges of the market and gain an advantage. A good analogy is a poker game, where, if all participants were to know each other's cards, the point of the game would be completely void, and in general this extends to any game or interaction that requires a secret to play.

Now, the concept of "securing" information is not necessarily restricted to hiding data, which is what the examples above are concerned with, but it can also deal with preventing information from being modified, or tampered with. It is natural to imagine why this is important in war or economic contexts, considering for example an instruction from a general that should not be modified, a record of a transaction that should not be transformed, or a dollar bill that should not be easy to tamper with. Without integrity of this data, orders could not be easily carried out through the chain of command, and several constructions of the economic infrastructure, like fiat currency, would lack any sense if they could easily be forged or tampered with.

Finally, the notion of securing information is also closely tied to the idea of ensuring it is available for use. In many settings, similar of even worse harm can be done by erasing information than by accessing it in an unauthorized manner. In our two main settings, war and economics, we see that intelligence gathered about an enemy is worthless if it cannot be accessed, and similarly, our modern economic infrastructure could not be supported if the records of each individual could be deleted at a given point. Availability may not pop up as the first idea when considering the task of securing information, but it is as important as the concepts of confidentiality and integrity discussed above.

Information and Communication in our Modern World

With the surge of the electronic and digital era, the concept of securing information and communication takes a radical jump. For centuries, many scenarios have appeared that have required humanity to address information security in one way or another, but all of these were "mechanical", or "physical". For example, an important message could in principle be hidden by placing it in a highly secure box, or the integrity of a document could be ensured (as we still do today) by asking for a signature. For more sensitive information, methods to "scramble" the data so that only the intended receiver could read it were designed, and these are accompanied by a long history of fractures and proposals.

In recent decades, however, the concept of securing information has engaged in profound changes. First, with the advent of electricity, information started taking completely new forms, and started being represented in a digital fashion. As a result, traditional, more physical techniques such as the use of safes, envelopes, locks, ink signatures, and so on, stopped being applicable in many scenarios, such as in the setting of sensitive communication over radio, or, even more importantly, the internet.

Fortunately, some techniques like "scrambling" information so that only an intended received could interpret and make sense of the result could still be applicable in these new settings. Mechanical and electronic devices were designed precisely to handle these processes efficiently, and although the art of designing these methods was always followed by some clever and unexpected way of completely undermining the security these provided, these techniques resulted effective in many use cases. However, the advent of modest computational power rendered some of the traditional techniques unusable, given that these methods to protect information, although hard to break by a human with pen and paper, fell within reach for modern computers. Naturally, people adapted to these challenges, designing new and more complex methods that could withstand, and even use in favor, the new wide range of available computational resources, and the propose-break game continued for decades.

We have seen then that new technologies and developments in different areas have had major impacts in how we conceive and interact with information and communication. However, one of the biggest advances that caused one of the most notable shocks was the invention of the *internet*, together with the adoption of *computers*, and more specifically *personal computers*. The idea of humans communicating with each other over previously unimaginable geographical distances, and the concept of individuals being able to perform complicated computations, and store, manipulate, share and interact with insurmountable amounts of information in personal devices, led to incredible advances in many areas and disciplines. Information could be more easily accessed, distributed and manipulated than ever, which could only open doors to new applications and developments.

Unfortunately, with the appearance of this new digital era of almost-instant communication, seemingly infinite amount of storage and computability resources, and limitless availability of information, many problems in terms of security started to become relevant, and these had a completely different shape when compared to analogous chal-

lenges from the past, which made them harder to approach. There are countless examples, but to mention a few, consider for instance the classical task of securing communication between two parties who are geographically separated. Traditionally, these two parties would set up a "scrambling" mechanism together with a way of ensuring integrity, and this would be done perhaps by meeting in person once or somehow sharing an initial secret that would enable confidential communications later on. Going through this process in modern times is not that feasible anymore, given that the pair of parties communicating could be something as ubiquitous as a client discussing sensitive information with his or her bank, or an individual retrieving medical records from his or her hospital. It is sadly not feasible for each of these institutions to perform a complicated setup with each individual user. This is made even more complex when factoring in identities and authentication, since it is much harder in internet scenarios, where all information is digital, to ensure that the party at the other end of the line is who they claim to be.

Another example that illustrates the difficulties of securing information in the modern digital era pertains the ridiculously large amount of *personal* information existing online. From the recent growth of social media to the extensive load of data gathered by organizations and institutions, information about individuals is frequently in transit over the internet and stored in different locations, many of these commonly available online. The concept of *privacy* becomes today more relevant than ever, and people are becoming more wary regarding the information present about them in external locations, which has many uses for third parties ranging from seemingly-harmless advertising, to discredit campaigns and censorship.

Cryptography

As we have illustrated above, the task of securing information and communication is as old as these two concepts themselves, and humanity has cleverly found different ways to deal with these problems, which was crucial for the fruitful development of a broad collection of areas and disciplines. These methods can be considered of a more artistic and heuristic nature rather than scientific, given that they typically consisted of different proposals that "seemed" to work well in particular contexts, but shortly after were proven to be insecure, not scalable, not applicable in other settings, or they simply become obsolete. This, however, seemed to match the pace in which knowledge, discoveries and inventions were being developed. Nonetheless, today's times are radically different. The list of challenges that emerge in regards to securing information that did not exist, or were even unthinkable, before the modern digital age, is essentially endless. As a result, modern solutions that could match the already established ways of science were in high demand.

Cryptography, as a science, is born with the intention of addressing the difficulties of securing information and communication in our modern context. As we have already mentioned, traditional techniques that today we regard as cryptography consisted mostly of ad-hoc methods to "scramble" information, which in modern terms can be associated with the concept of *encryption*. The task of hiding data is even more relevant today, as we have highlighted already with the examples if clients requesting medical records,

or users discussing sensitive information with financial institutions. In contemporary times, the requirements of confidentiality, integrity and availability are not restricted to governments and military, and instead, it is fair to say that every single individual making use of our current digital and distributed infrastructure is in need of these services.

Modern cryptography aims to provide users with these guarantees. Getting together ideas from a wide range of disciplines like mathematics, computer science, electrical engineering, communication science, and even physics, researchers in cryptography have managed to develop the necessary tools and foundations to support the notion of securing information in today's standards, which has enabled many of the applications we all-over enjoy nowadays.

One of the most relevant of such developments lies in the invention of *public key cryptography*. As we have mentioned, multiple techniques to protect data have been proposed since ancient times. Historically, many of these proposals have been proven to be insecure by means of different types of attacks, which led to new constructions aiming to improve over their predecessors. However, in the case that a system was considered secure at a given time, it was typically the case that a complicated setup or "ceremony" needed to be conducted to arrange what would be the *secret key*, which would enable further secure communication. In our modern digital era, constructions that are recognized as secure still need some sort of setup, which is complicated to carry out when there is a need for secure information and communication from a large collection of individuals, institutions, companies, organizations, etc. Public key cryptography [46, 48] was invented precisely as a way of securing information and communication without the need of dedicating a setup key for every single pair of interacting parties. This breakthrough has enabled the current online infrastructure we are accustomed to, permitting online payments, distributed accounts, banking, communication, and much more.

Additional to this, cryptography has also contributed in many other great ways to help us reach many of the achievements we enjoy today. Banks, governments, institutions and organizations are all able to operate at a large scale thanks to existing methods to protect the large amount of potentially sensitive information they work with. Authenticating online towards a website or service is only possible thanks to the amount of security measures put in place to prevent a different individual from doing so without authorization. All of these advances and developments are achieved via a rigorous, open and collaborative scientific process involving experts from all around the world. Unlike previous approaches to securing information and communication, where proposals were carried out by selective groups and kept in high secret with the belief that their disclosure would harm security, in modern times the parties researching and adopting these technologies tend to be open about their methods and are conscious of the value of doing so: more minds constructing, analyzing, deploying these techniques means better quality overall and more transparency in case of flaws, problems, or other complications. This collaborative nature, together with the rigor of the area, is in essence what makes of cryptography a scientific field.

Advanced Topics in Cryptography

A big part of the theory and practice of cryptography is devoted to the study of different technologies deployed in our world today. Standard topics include the task of encryption, which relates to hiding information, but it is also common to consider digital signatures and message authentication codes to ensure integrity, enable authentication and authorization, and many other subjects relevant for today's infrastructure. For centuries, these were essentially the main tasks associated with the idea of securing information and communication, and this continues being the case today—of course, with the added complications of digital and worldwide-distributed technologies. Research into correct implementations and deployments of these tools, possible attacks, improvements, enhancements in user experience, adaptation to modern more technologies and scenarios, and other relevant questions, is of high importance.

However, a big portion of the field efforts is devoted not to current systems or natural questions like simply hiding information or ensuring integrity. In recent decades, researchers in cryptography have been pushing the barriers of science by considering new problems and challenges that were not even imagined in previous times. A good example is the idea of *zero knowledge proofs*, which provides us with the ability of proving properties of certain given data, without revealing the data itself. For instance, an individual could be able to prove to a third party that the balance in his or her bank account is above a certain threshold without revealing the exact amount, or it could be able to prove knowledge of the answer to a given question or challenge, without announcing the answer itself. This may sound counterintuitive and perhaps even impossible, but such techniques have been under research for decades already, and it is fair to say that today they are crossing the boundary towards the real world due to their wide range of applications, most notably within the blockchain and cryptocurrency domains (which, on their own, are interesting applications of cryptographic techniques that are gaining popularity quite rapidly). This is because, in simple terms, zero knowledge proofs have the potential of enabling users to prove that certain transaction was made, without revealing the specifics of the transaction itself, among several other use cases. For more information on this, see for example [58].

Another interesting concept that at first glance seems impossible to instantiate is the idea of *computing on private data*. As we have previously commented on, it is natural to expect certain type of information to be kept hidden, such as monetary transactions, balances, highly-sensitive information, login credentials, medical records, private conversations, and many, many others. However, it is commonly the case that, although this data has to be kept private from unauthorized agents, it is ultimately revealed in one way or another to be processed by a legitimate party, and in general, the situation in which data has to be kept hidden perpetually is rare. In many cases, however, it is not the full hidden data that is needed to be retrieved, but rather, a derived property from this data. As an example, consider a set of encrypted medical records from a hospital. In order to compute the proportion of users having certain medical condition, the hospital (or the party executing this analysis) would need to decrypt all records and then perform the computation, potentially revealing much more information than simply the intended proportion.

In today's world, more and more information is collected every single day. With more devices recording data every single minute, and more information about individuals being stored as they interact with new services and technologies, information has never been so plentiful. However, not all of it is readily available due to security or privacy concerns, as illustrated by the different examples above. Researchers in cryptography have studied the task of performing computation over hidden or sensitive data in order to remove this obstacle, and enable advances in multiple disciplines without violating the notion of securing information. To achieve this, multiple ideas applicable to different settings have been proposed. For example, *functional encryption* enables users to encrypt their data, and derive special "keys" that can be used to decrypt not the entire information, but derived data like, for example, only a small piece of the full information or operations carried out on these.

Homomorphic encryption on the other hand promises its users to be able to encrypt, or completely hide data, while still being able to perform certain computations on these. The result of this process leads to an encryption of the result of the computation, which can then be decrypted. This way, throughout the whole procedure the data is always hidden, and the only value that is revealed is the final output. Naturally, this is a very powerful tool with countless applications, and it has received enormous attention from the cryptographic community. Unfortunately, in spite of inspiring recent breakthroughs like the first actual construction of such a scheme [55], and in spite of multiple works proposing new constructions and improving over previous ones, the efficiency of these techniques is still too low for a wide range of relevant applications. However, many use cases are already within reach, and the field evolves in a very rapid manner, so a more widespread adoption of these tools can be around the corner.

Another tool that aims at enabling data analysis and aggregation on private data, that is gaining a lot of popularity in recent years due to its simplicity, is *differential privacy* [47]. The main idea behind this technique lies in adding small "noise" to the data in such a way that information about individuals cannot be discerned from the published records, but overall, certain aggregate data can still be computed. The result will be only approximate, as it contains some errors derived from the noise added to individual records, and there is a trade-off between the level of privacy provided and the precision of the result.

Many other advanced techniques in cryptography with several relevant applications are of interest to researchers, and some of them are slowly making their way into our real world. Of particular interest to us is *secure multiparty computation*, which is proposed as an alternative tool instantiate the dreamful task of performing computations on hidden data. We discuss this set of techniques below.

Secure Multiparty Computation

Consider the following scenario. There is an individual, *Alejandra*, applying for a loan at a *Bank*, and for doing so she needs to present a lot of information about her, like her income, savings, investments and possessions. Alejandra is a very important businesswoman, and she does not want all of this information to be held by the bank in the case she gets rejected, so she asks the bank to publish the algorithm they use to determine

whether she applies for the desired credit so that she can check her situation first. On the other hand, the bank developed one of the most advanced credit analysis algorithms, and it is not willing to do this as that might imply they lose an important asset.

Imagine also this setting. A gender-equality campaign promoted by an NGO in certain city aims at determining whether men and women are being paid equally, in average, across several companies in the area. For doing so, the NGO would like these companies to send the information regarding salaries and benefits so that they can compute the desired statistics. However, although the NGO is only interested in learning averages across all companies, and although these might agree with the idea of calculating such figures, they do not feel comfortable sending all this information to a third party such as the NGO, which could harm their operations.

Both of the examples above constitute cases of computation on data that is intended to be kept private, and these can be brought together under the following description. There are n parties, $P_1, \dots, P_n$, each P_i having an input x_i that they wish to maintain secret. Furthermore, there is a function $f(\mathtt{X}_1, \dots, \mathtt{X}_n)$, and the parties want to learn the result $z = f(x_1, \dots, x_n)$. In the first example, $n = 2$, x_1 is the set of records from Alejandra, x_2 is the bank's algorithm and $f(\mathtt{X}_1, \mathtt{X}_2)$ is the function that applies algorithm $\mathtt{X}_2$ to $\mathtt{X}_1$, and in the second case n is the number of companies, each x_i is the information regarding employee salaries of the i-th company, and f is the function that determines average income based on gender.

As in general with the problem of computing on hidden data, the problems above seem hard to solve without the involved parties being willing to share their data. However, as we have mentioned already, researchers in cryptography have been working on different technologies to enable this type of computations. The parties in the scenarios above could in principle resort, for example, to homomorphic encryption techniques, encrypting their inputs and computing on the corresponding ciphertexts, but for reasonably meaningful functions $f(\cdot)$ this could be simply too inefficient. Alternatively, it is possible that differential privacy techniques can help, specially in the second scenario where the NGO wishes to compute different statistics on data coming from different parties, but this could potentially affect the precision of the study.

A different approach, called *secure multiparty computation*, or MPC for short, aims at developing much more efficient solutions to the problem above that do not undermine precision or correctness. Observe for example the following in the Alejandra vs bank scenario. Alejandra does not trust the bank to have her information completely, so she might send the bank a "hidden" version of her data. If homomorphic encryption is used, then the bank would be able to apply its analysis algorithm internally, but as we have already mentioned this could place an insurmountable computational barrier. Alternatively, we may notice that Alejandra, also interested in the output of the computation, can lend a hand to *interactively* compute the result together with the bank. A similar observation holds in the second scenario: although all the different companies could simply encrypt their records and send these to the NGO for computation,[1] we may notice that if the companies are willing to collaborate to jointly compute the function, savings in efficiency could be achieved

[1] It is worth noticing that for the situation in which the computation is simple additions, which is potentially the case in this example, homomorphic encryption is actually quite efficient.

In contrast to other approaches like homomorphic encryption, which work by hiding data completely from anyone not holding a special key and allow anyone to compute on the hidden data, MPC leverages the fact that there are multiple parties in the computation, and these entities can together compute the desired function. This is achieved via an interactive protocol executed by the parties that guarantees privacy of the data throughout the whole computation. We will discuss in Chapter 1 the details of what these guarantees exactly mean, but for now, it suffices to say that if a function is computed using an MPC protocol, then all of its inputs remain private and only the result of the computation is revealed. Unlike the case of homomorphic encryption, the computation is carried out by the parties holding the inputs themselves and it cannot be delegated to any other entity holding "encryptions".[2]

Secure multiparty computation was introduced in 1982 [80] when Yao presented the concept of *Garbled circuits*, which is a particular way of securely evaluating a function using MPC. Since then, many different approaches have been proposed by cryptography researchers in the literature, leading to a fruitful line of exploration that has produced many interesting theoretical and practical results. Today, we have a solid understanding in regards to what type of protocols with what form of security can exist, and many of the state-of-the-art techniques can be used already for a wide range of applications and use cases. For example, the gender-equality study described above actually happened in Boston [66].

Several other applications of MPC have reached the realm of our real world, such as the Danish sugar beet AUCTION in 2008 [21], the tax fraud detection process in Estonia in 2015 [20], and many more. Furthermore, many other relevant use cases are considered regularly by researchers, and several prototypes are already under development. These applications include, for example, custody of cryptographic material, training/evaluating ML models.[3], securing databases, secure statistics, e-voting, and many more. Finally, what is also interesting is that these technologies are getting attention from institutions, organizations, and companies beyond academia, with some notable examples being Google, VISA, Facebook, IBM, Intel and Microsoft. Moreover, many start-ups and well-established companies are aiming at developing products based on secure multiparty computation, such as Sharemind, Galois, Cape Privacy, Unbound tech, Partisia and Inpher.

Arithmetic Circuits

There are quite a few general approaches for designing secure computation protocols. For example, since the introduction of the idea by Yao [80], Garbled circuits have been improved in several directions and today they are much more efficient and promising than their previous counterparts. These techniques are useful when the parties are geographically separated and have a high latency connection between them given that it requires a small amount of communication rounds, although it typically demands high bandwidth. Another method consists of using homomorphic encryption techniques, which is again

[2] As we will see in Section 5.1, this is not a real limitation in MPC since secure computation can still be outsourced, albeit with different security guarantees as in homomorphic encryption.

[3] See for example the blog post *Privacy-Preserving Training/Inference of Neural Networks, Part 2*. https://bit.ly/3eRKlgM

particularly well-suited for highly distributed computations, but as we have discussed already tend to impose a computational burden. Finally, a very popular approach, which is the main technique for secure multiparty computation we will focus on in this thesis, consists of making use of *linear secret-sharing schemes*, which enable the parties to hold distributed versions of the intermediate results of the computation, maintain this invariant throughout the evaluation process, until the result is reached. Details on this idea are presented in subsequent sections.

Importantly, a common and central idea across all of the techniques mentioned above consists of first representing the desired computation as an *arithmetic circuit*, and then designing a method to process such circuit securely. An arithmetic circuit is a representation of a function that consists of wires and gates. The input wires are fed with the inputs to the computation and then processed through gates to obtain the values for the internal wires, which account for the intermediate results of the computation, until the output wires are reached, where the result of the computation is obtained. More details are discussed in Chapter 1, but for now, it suffices to know that the gates represent operations over a ring, an algebraic structure admitting an addition and a multiplication operation. These constitute the allowed intermediary processes that can be applied to the data, which is itself represented as elements over this ring.

A typical choice of ring is the set of integers modulo a prime p, which constitutes what is called a *field*, since every non-zero element admits a multiplicative inverse. For example, the set $\{0, 1\}$ with the operations `AND` and `XOR` constitutes precisely a field (integers modulo 2), and arithmetic circuits defined over this structure, also known as binary circuits, are highly important in many use cases. On the other hand, for applications involving integer arithmetic, it is common to consider integers modulo a large prime p given that, if the integers used in the computation can be guaranteed to be smaller than p, then reduction modulo p does not play any effect and the resulting arithmetic becomes in essence simple integer arithmetic, which is useful in numerous scenarios.

Most of the literature in the area of secure multiparty computation has focused on arithmetic circuits over fields exclusively. The reason for this is two-fold. First, as described above these structures already enable a good set of applications with immense relevance in practice. Second, fields, having the very important property that every non-zero element is invertible, are much more amenable to work with and permit the use of several techniques and constructions that would not be possible if these properties did not hold. Notwithstanding the above, there are other algebraic structures that could prove useful in specific scenarios, or perhaps they can provide certain efficiency improvements, so it is worth considering secure multiparty computation protocols over these.

Secure Computation over $\mathbb{Z}/2^k\mathbb{Z}$

A useful algebraic structure that permeates all the results in this thesis is the set of integers modulo a power of two 2^k, denoted by $\mathbb{Z}/2^k\mathbb{Z}$. This structure is *not* a field given that not all elements admit a multiplicative inverse: for example, there is no number such that, when multiplied by 2, we get the unity 1 modulo 2^k. As a result, the vast

majority of the existing techniques in the literature do not apply in this case. However, there are several reasons why one would like to consider this structure. For example:

Compatibility with existing computer architectures. Although computation modulo a large prime p can be used for applications involving integer arithmetic, it is not directly compatible with existing computer architectures, and the reduction modulo p operation has to be implemented in software. For certain choices of p (e.g. Mersenne primes), it is possible to design very lightweight reduction programs, but, still, it would be more desirable to use some type of arithmetic that typical hardware architecture use, for example, types like `int32` or `int64`; this corresponds precisely to arithmetic over $\mathbb{Z}/2^k\mathbb{Z}$ with $k = 32$ and $k = 64$, respectively.

Synergy with binary computation. It is common that, even if an application mostly involves integer arithmetic, binary computation is still needed in different parts of the process. Arithmetic modulo 2^k has certain "synergy" with arithmetic modulo 2^ℓ for any $\ell \leq k$, in particular for $\ell = 1$ which is the binary case, given that if two integers are congruent modulo 2^k then they are congruent modulo 2^ℓ too. This turns out to lead to several benefits when converting from the $\mathbb{Z}/2^k\mathbb{Z}$ domain to $\{0, 1\}$ and back, among several other gains.

Knowledge barrier in secure MPC. Another important motivation to study secure multiparty computation over $\mathbb{Z}/2^k\mathbb{Z}$ is to determine whether there are certain inherent limitations of this type of rings that prevent them from having natural secure MPC protocols over them.

As we have mentioned already, most of the existing constructions in the literature, specially these tolerating adversarial deviation from the protocol specification, operate over finite fields, with no clear way to extend or adapt them to the ring $\mathbb{Z}/2^k\mathbb{Z}$. For instance, existing protocols will typically rely on invertibility of non-zero elements for security arguments, or require the non-existence of zero divisors (that is, elements that can be multiplied with a non-zero element so that the result is zero). To cite some concrete examples, protocols for dishonest majority like [19, 41, 43, 62, 63] frequently make use of *one-time message authentication codes*, which for security require equations of the type $\mathtt{x} \cdot \delta + \gamma = 0$ to have only one solution for $\mathtt{x}$ if $\delta \neq 0$. This is not the case if the equation is modulo 2^k, since for example, for $\delta = 2^{k-1}$ and $\gamma = 0$, all even numbers are possible solutions. Another illustration of this is protocols based on Shamir secret-sharing like [16, 17, 42], which require polynomial interpolation theorems that do not exactly hold in other non-field rings such as $\mathbb{Z}/2^k\mathbb{Z}$.

Main Results of this Thesis

From what we have discussed above, the general problem of how to design natural secure multiparty computation protocols over $\mathbb{Z}/2^k\mathbb{Z}$, study their limitations and explore potential applications where these techniques can be beneficial, can be considered to be an open problem. In this thesis, specifically in Part II, we present a series of techniques and protocols that operate over the ring $\mathbb{Z}/2^k\mathbb{Z}$, ranging over a wide variety of MPC settings like dishonest majority, honest majority, two-thirds honest majority, and

also small number of parties. We also discuss advanced primitives for secure computation that illustrate the benefits of working over $\mathbb{Z}/2^k\mathbb{Z}$ in several applications. We list these constructions in a more detailed manner when we discuss, in page 14, the organization of this thesis. It is also important to mention that these results are based on research papers written and published by different teams of researchers, all including the author of this thesis. These are described in more detail in page 16.

About this Thesis

Before we dive into the main contents of this thesis, which start in Part I below, we discuss some general aspects of it such as broad preliminaries, the organization of the document (which in particular lists in more detail the contributions of this thesis) and the works this thesis is based on.

General Preliminaries and Notation

Most of the notation used in this paper will be introduced "on the fly", that is, it will be presented in each relevant section where it is used for the first time. However, some general notation that we can introduce from now consists of the following:

- The set $\mathbb{Z}/M\mathbb{Z}$ denotes the ring of integers modulo M, whose representatives are taken over the set $\{0, \dots, M-1\}$.
- All vectors, denoted by bold lowercase letters like $\mathbf{x}$ and $\mathbf{y}$, are column vectors by default. This is particularly relevant when dealing with multiplications with matrices.
- $x \in_R A$ means that x is sampled uniformly at random from the set A.
- For a positive integer ℓ, $[\ell]$ denotes the set $\{1, \dots, \ell\}$.
- For a k-bit integer x, we denote by $(x[k-1], \dots, x[0])$ its bit decomposition, that is, $x[i] \in \{0,1\}$ for $i \in \{0, \dots, k-1\}$ and $x = \sum_{i=0}^{k-1} x[i] \cdot 2^i$.

Finally, whenever some research work involving the author of this thesis is cited, it will be referred to as *original works*.

Organization of the Thesis

This thesis is divided into two main parts: Part I, which lays down the foundations of MPC necessary for our main contributions and also presents some general existing techniques for MPC over *finite fields*, and Part II, where the actual contributions of this thesis lie and includes the description of a series of MPC protocols over the ring $\mathbb{Z}/2^k\mathbb{Z}$ in a wide variety of security settings. Each of these parts is described more thoroughly below.

Part I: MPC Fundamentals. This first part contains some basic concepts and construction from *existing* secure multiparty computation literature. The purpose of this part is two-fold. First, and this is particularly the case in Chapter 1, it introduces the background tools necessary for the development of our contributions in Part II of this work. This includes crucial concepts such as the definition of secure multiparty computation itself, simulation-based security, types of security guarantees, different corruption scenarios, etc. Furthermore, the field-based protocols presented in Chapter 2 serve as a starting point for many of the $\mathbb{Z}/2^k\mathbb{Z}$-based protocols from Part II, so it is worth keeping them at hand for establishing differences and key ideas. Secondly, this part aims to serve as a tool for quick reference when looking for already "standard" concepts and techniques in the field of secure multiparty computation, and more specifically, linear-secret-sharing-based MPC.

Chapter 1: The Theory of Multiparty Computation. This chapter presents some basic concepts in MPC that are necessary for the development of the main results in this thesis. This includes the description of the simulation-based security model considered in MPC, some essential impossibility results, and some general approaches to the design of MPC protocols. The contents of this chapter are not tied to computation over fields.

Chapter 2: Some Essential MPC Constructions. Finally, this chapter presents a wide range of essential MPC protocols over fields for the cases of two-thirds honest majority ($t < n/3$), honest majority ($t < n/2$) and dishonest majority ($t < n$), including passive and active security.

Part II: MPC Techniques over $\mathbb{Z}/2^k\mathbb{Z}$. This second part includes the main contributions of this thesis, and it focuses solely on computation over the ring $\mathbb{Z}/2^k\mathbb{Z}$. All protocols considered in this part satisfy security against an active adversary.

Chapter 3: Two-Thirds Honest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$. This chapter presents a perfectly secure protocol with abort for computation over $\mathbb{Z}/2^k\mathbb{Z}$ that is secure against an adversary corrupting t parties, where $t < n/3$. Extensions to guaranteed output delivery are discussed.

Chapter 4: Honest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$. This chapter presents a statistically secure protocol with abort for computation over $\mathbb{Z}/2^k\mathbb{Z}$ that is secure against an adversary corrupting t parties, where $t < n/2$.

Chapter 5: MPC over $\mathbb{Z}/2^k\mathbb{Z}$ for a Small Number of Parties. In this chapter two protocols for computation over $\mathbb{Z}/2^k\mathbb{Z}$ with a small number of parties are considered. The first one supports four parties and one corruption, while the second one is designed for three parties and also one corruption. Both protocols are computationally secure, although they only make use of mild security assumptions like the existence of PRFs and cryptographic hash functions.

Chapter 6: SPDZ2k: Dishonest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$. A protocol in the dishonest majority setting ($t < n$) in the preprocessing model is discussed in this chapter.

This means that, for the description of the protocol, we assume different preprocessed data like multiplication triples plus some initialization correlation for the authentication of shared values.

Chapter 7: Primitives for Secure Computation using edaBits. Finally, with the aim of enabling practical use of the techniques above, we present in this ending chapter a series of tools to securely evaluate common primitives present in a wide range of applications, that are suitable for use with any of the protocols presented above.

Original Works this Thesis is Based on

Throughout this work, whenever we call a reference an "original work", we mean the author of this thesis is listed as an author in the cited publication. The main contributions of this thesis are based on original works published by the author, together with different colleagues, at several cryptography and security conferences. These are listed below in chronological order.

Year 2018

SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority [32]

Authors: Ronald Cramer, Ivan Damgård, *Daniel Escudero*, Peter Scholl and Chaoping Xing
Published: CRYPTO'18

Abstract: Most multi-party computation protocols allow secure computation of arithmetic circuits over a finite field, such as the integers modulo a prime. In the more natural setting of integer computations modulo 2^k, which are useful for simplifying implementations and applications, no solutions with active security are known unless the majority of the participants are honest.
We present a new scheme for information-theoretic MACs that are homomorphic modulo 2^k, and are as efficient as the well-known standard solutions that are homomorphic over fields. We apply this to construct an MPC protocol for dishonest majority in the preprocessing model that has efficiency comparable to the well-known SPDZ protocol (Damgård et al., CRYPTO 2012), with operations modulo 2^k instead of over a field. We also construct a matching preprocessing protocol based on oblivious transfer, which is in the style of the MASCOT protocol (Keller et al., CCS 2016) and almost as efficient.

In this thesis: This work presents an actively secure protocol for computation over $\mathbb{Z}/2^k\mathbb{Z}$ in the dishonest majority setting. In Section 6 we present the *online phase* of this protocol. The offline phase of the SPDZ$\mathbb{Z}_{2^k}$ protocol is left out of this thesis. This is instantiated in [32] making use of Oblivious Transfer, following the template of the MASCOT protocol [62].

Year 2019

Efficient Information-Theoretic Secure Multiparty Computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois Rings [2]

Authors: Mark Abspoel, Ronald Cramer, Ivan Damgård, *Daniel Escudero* and Chen Yuan
Published: TCC'19

Abstract: At CRYPTO 2018, Cramer et al. introduced a secret-sharing based protocol called SPDZ2k that allows for secure multiparty computation (MPC) in the dishonest majority setting over the ring of integers modulo 2^k, thus solving a long-standing open question in MPC about secure computation over rings in this setting. In this paper we study this problem in the information-theoretic scenario. More specifically, we ask the following question: Can we obtain information-theoretic MPC protocols that work over rings with comparable efficiency to corresponding protocols over fields? We answer this question in the affirmative by presenting an efficient protocol for robust Secure Multiparty Computation over $\mathbb{Z}/p^k\mathbb{Z}$ (for *any* prime p and positive integer k) that is perfectly secure against active adversaries corrupting a fraction of at most $1/3$ players, and a robust protocol that is statistically secure against an active adversary corrupting a fraction of at most $1/2$ players.

In this thesis: This work presents a generalization of Shamir secret-sharing to the so-called Galois rings, and it makes use of this tool to design MPC protocols in two settings: (1) perfect security against an active adversary corrupting $t < n/3$ parties, and (2) statistical security against an active adversary corrupting $t < n/2$ parties. Both settings are considered with guaranteed output delivery.
In this thesis, this work appears in Chapters 3 and 4, where we present MPC protocols over $\mathbb{Z}/2^k\mathbb{Z}$ with information-theoretic security. The results presented here are a restricted version of the ones in [2]. For example, we do not include guaranteed output delivery (beyond a short discussion in Section 3.4), we focus on the case $p = 2$, and several conceptual simplifications are made.

Year 2020

Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits [50]

Authors: *Daniel Escudero*, Satrajit Ghosh, Marcel Keller, Rahul Rachuri and Peter Scholl
Published: CRYPTO'20

Abstract: This work introduces novel techniques to improve the translation between arithmetic and binary data types in secure multi-party computation. We introduce a new approach to performing these conversions using what we call extended doubly-authenticated bits (edaBits), which correspond to shared integers in the arithmetic domain whose bit decomposition is shared in the binary domain. These can be used to considerably increase the efficiency of non-linear operations such as truncation, secure comparison and bit-decomposition.
Our edaBits are similar to the daBits technique introduced by Rotaru et al. (Indocrypt 2019). However, we show that edaBits can be directly produced much more efficiently than daBits, with active security, while enabling the same benefits in higher-level applications. Our method for generating edaBits involves a novel cut-and-choose technique that may

be of independent interest, and improves efficiency by exploiting natural, tamper-resilient properties of binary circuits that occur in our construction. We also show how edaBits can be applied to efficiently implement various non-linear protocols of interest, and we thoroughly analyze their correctness for both signed and unsigned integers.
The results of this work can be applied to any corruption threshold, although they seem best suited to dishonest majority protocols such as SPDZ. We implement and benchmark our constructions, and experimentally verify that our technique yield a substantial increase in efficiency. EdaBits save in communication by a factor that lies between 2 and 60 for secure comparisons with respect to a purely arithmetic approach, and between 2 and 25 with respect to using daBits. Improvements in throughput per second are slightly lower but still as high as a factor of 47. We also apply our novel machinery to the tasks of biometric matching and convolutional neural networks, obtaining a noticeable improvement as well.

In this thesis: We present the applications of edaBits in Chapter 7, including bit decomposition, truncation and integer comparison. We do not include in this thesis the experimental results from [50], nor the method to generate edaBits based on a sophisticated cut-and-choose-based analysis.

Year 2021

An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings [4]

Authors: Mark Abspoel, Anders P. K. Dalskov, *Daniel Escudero* and Ariel Nof
Published: ACNS'21

Abstract: Multiparty computation (MPC) over rings such as $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ has received a great deal of attention recently due to its ease of implementation and attractive performance. Several actively secure protocols over these rings have been implemented, for both the dishonest majority setting and the setting of three parties with one corruption. However, in the honest majority setting, no *concretely* efficient protocol for arithmetic computation over rings has yet been proposed that allows for an *arbitrary* number of parties.
We present a novel compiler for MPC over the ring $\mathbb{Z}_{2^k}$ in the honest majority setting that turns a semi-honest protocol into an actively secure protocol with very little overhead. The communication cost per multiplication is only twice that of the semi-honest protocol, making the resultant actively secure protocol almost as fast.
To demonstrate the efficiency of our compiler, we implement both an optimized 3-party variant (based on replicated secret-sharing), as well as a protocol for n parties (based on a recent protocol from TCC 2019). For the 3-party variant, we obtain a protocol which outperforms the previous state of the art that we can experimentally compare against. Our n-party variant is the first implementation for this particular setting, and we show that it performs comparably to the current state of the art over fields.

In this thesis: We present the three-party instantiation of this protocol in Section 5.4. We do not present the generic compiler, nor the instantiation using Shamir secret-sharing for any number of parties. We also do not include the experimental results.

Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security [36]

Authors: Anders P. K. Dalskov, *Daniel Escudero* and Marcel Keller
Published: USENIX'21

Abstract: In this work we introduce a novel four-party honest-majority MPC protocol with active security that achieves comparable efficiency to equivalent protocols in the same setting, while having a much simpler design and not relying on function-dependent preprocessing. Our initial protocol satisfies security with abort, but we present some extensions to achieve guaranteed output delivery. Unlike previous works, we do not achieve this by delegating the computation to one single party that is identified to be honest, which is likely to hinder the adoption of these technologies as it centralizes sensitive data. Instead, our novel approach guarantees termination of the protocol while ensuring that no single party (honest or corrupt) learns anything beyond the output.
We implement our four-party protocol with abort in the MP-SPDZ framework for multiparty computation and benchmark multiple applications like MNIST classification training and ImageNet inference. Our results show that our four-party protocol performs similarly to an efficient honest-majority three-party protocol that only provides semi-honest/passive security, which suggest that adding a fourth party can be an effective method to achieve active security without harming performance.

In this thesis: The four-party protocol from [36] is presented in Section 5.3. The only contribution of that work omitted from this thesis is the experimental results and the applications to privacy-preserving machine learning.

Efficient Information-Theoretic Multi-Party Computation over Non-Commutative Rings [51]

Authors: *Daniel Escudero* and Eduardo Soria-Vazquez
Published: CRYPTO'21

Abstract: We construct the first efficient, unconditionally secure MPC protocol that only requires black-box access to a non-commutative ring R. Previous results in the same setting were efficient only either for a constant number of corruptions or when computing branching programs and formulas. Our techniques are based on a generalization of Shamir's secret sharing to non-commutative rings, which we derive from the work on Reed Solomon codes by Quintin, Barbier and Chabot (*IEEE Transactions on Information Theory, 2013*). When the center of the ring contains a set $A = \{\alpha_0, \ldots, \alpha_n\}$ such that $\forall i \neq j, \alpha_i - \alpha_j \in R^*$, the resulting secret sharing scheme is strongly multiplicative and we can generalize existing constructions over finite fields without much trouble.
Most of our work is devoted to the case where the elements of A do not commute with all of R, but they just commute with each other. For such rings, the secret sharing scheme cannot be linear "on both sides" and furthermore it is not multiplicative. Nevertheless, we are still able to build MPC protocols with a concretely efficient online phase and black-box access to R. As an example we consider the ring $\mathcal{M}_{m\times m}(\mathbb{Z}/2^k\mathbb{Z})$, for which when $m > \log(n+1)$, we obtain protocols that require around $\lceil\log(n+1)\rceil/2$ less communication and $2\lceil\log(n+1)\rceil$ less computation than the state of the art protocol based on Circuit Amortization Friendly Encodings (Dalskov, Lee and Soria-Vazquez, *ASIACRYPT 2020*).
In this setting with a "less commutative" A, our black-box preprocessing phase has a less practical complexity of poly(n). We fix this by additionally providing specialized, concretely efficient preprocessing protocols for $\mathcal{M}_{m\times m}(\mathbb{Z}/2^k\mathbb{Z})$ that exploit the structure of the matrix ring.

In this thesis: From [51] we only use in Chapters 3 and 4 the definitions and results regarding Shamir-secret sharing over arbitrary rings. The protocols from these chapters are specialized to the case of computation over $\mathbb{Z}/2^k\mathbb{Z}$ and as such, they do not make use of the results from [51], which are about MPC over arbitrary (possibly non-commutative) finite rings.

Other Original Works not Included in this Thesis

Now we present other works by the author that are not present in this thesis.

Works Related to MPC over $\mathbb{Z}/2^k\mathbb{Z}$. First we describe other works related to the main topic of this thesis, that is, secure computation over $\mathbb{Z}/2^k\mathbb{Z}$. These are not included in this work, but still serve as relevant references in the topic.

New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning [44]

Authors: Ivan Damgård, *Daniel Escudero*, Tore Frederiksen, Marcel Keller, Peter Scholl and Nikolaj Volgushev
Published: S&P'19

Abstract: At CRYPTO 2018 Cramer et al. presented SPDZ2k, a new secret-sharing based protocol for actively secure multi-party computation against a dishonest majority, that works over rings instead of fields. Their protocol uses slightly more communication than competitive schemes working over fields. However, their approach allows for arithmetic to be carried out using native 32 or 64-bit CPU operations rather than modulo a large prime. The authors thus conjectured that the increased communication would be more than made up for by the increased efficiency of implementations.
In this work we answer their conjecture in the affirmative. We do so by implementing their scheme, and designing and implementing new efficient protocols for equality test, comparison, and truncation over rings. We further show that these operations find application in the machine learning domain, and indeed significantly outperform their field-based competitors. In particular, we implement and benchmark oblivious algorithms for decision tree and support vector machine (SVM) evaluation.

Asymptotically Good Multiplicative LSSS over Galois Rings and Applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$ [1]

Authors: Mark Abspoel, Ronald Cramer, Ivan Damgård, *Daniel Escudero*, Matthieu Rambaud, Chaoping Xing and Chen Yuan
Published: ASIACRYPT'19

Abstract: We study information-theoretic multiparty computation (MPC) protocols over rings $\mathbb{Z}/p^k\mathbb{Z}$ that have good asymptotic communication complexity for a large number of players. An important ingredient for such protocols is arithmetic secret sharing, i.e., linear secret-sharing schemes with multiplicative properties. The standard way to obtain these over fields is with a family of linear codes C, such that C, $C^\perp$ and C^2

are asymptotically good (strongly multiplicative). For our purposes here it suffices if the square code C^2 is not the whole space, i.e., has codimension at least 1 (multiplicative). Our approach is to lift such a family of codes defined over a finite field $\mathbb{F}$ to a Galois ring, which is a local ring that has $\mathbb{F}$ as its residue field and that contains $\mathbb{Z}/p^k\mathbb{Z}$ as a subring, and thus enables arithmetic that is compatible with both structures. Although arbitrary lifts preserve the distance and dual distance of a code, as we demonstrate with a counterexample, the multiplicative property is not preserved. We work around this issue by showing a dedicated lift that preserves *self-orthogonality* (as well as distance and dual distance), for $p \geq 3$. Self-orthogonal codes are multiplicative, therefore we can use existing results of asymptotically good self-dual codes over fields to obtain arithmetic secret sharing over Galois rings. For $p = 2$ we obtain multiplicativity by using existing techniques of secret-sharing using both C and $C^\perp$, incurring a constant overhead. As a result, we obtain asymptotically good arithmetic secret-sharing schemes over Galois rings. With these schemes in hand, we extend existing field-based MPC protocols to obtain MPC over $\mathbb{Z}/p^k\mathbb{Z}$, in the setting of a submaximal adversary corrupting less than a fraction $1/2-\varepsilon$ of the players, where $\varepsilon > 0$ is arbitrarily small. We consider 3 different corruption models. For passive and active security with abort, our protocols communicate $O(n)$ bits per multiplication. For full security with guaranteed output delivery we use a preprocessing model and get $O(n)$ bits per multiplication in the online phase and $O(n \log n)$ bits per multiplication in the offline phase. Thus, we obtain true linear bit complexities, without the common assumption that the ring size depends on the number of players.

Secure Evaluation of Quantized Neural Networks [37]

Authors: Anders P. K. Dalskov, *Daniel Escudero* and Marcel Keller
Published: PoPETs'20

Abstract: We investigate two questions in this paper: First, we ask to what extent "MPC friendly" models are already supported by major Machine Learning frameworks such as TensorFlow or PyTorch. Prior works provide protocols that only work on fixed-point integers and specialized activation functions, two aspects that are not supported by popular Machine Learning frameworks, and the need for these specialized model representations means that it is hard, and often impossible, to use e.g., TensorFlow to design, train and test models that later have to be evaluated securely. Second, we ask to what extent the functionality for evaluating Neural Networks already exists in general-purpose MPC frameworks. These frameworks have received more scrutiny, are better documented and supported on more platforms. Furthermore, they are typically flexible in terms of the threat model they support. In contrast, most secure evaluation protocols in the literature are targeted to a specific threat model and their implementations are only a "proof-of-concept", making it very hard for their adoption in practice. We answer both of the above questions in a positive way: We observe that the quantization techniques supported by both TensorFlow, PyTorch and MXNet can provide models in a representation that can be evaluated securely; and moreover, that this evaluation can be performed by a general purpose MPC framework. We perform extensive benchmarks to understand the exact trade-offs between different corruption models, network sizes and efficiency. These experiments provide an interesting insight into cost between active and passive security, as well as honest and dishonest majority. Our work shows then that the separating line between existing ML frameworks and existing MPC protocols may be narrower than implicitly suggested by previous works.

Secure training of decision trees with continuous attributes [5]

Authors: Mark Abspoel, *Daniel Escudero* and Nikolaj Volgushev
Published: PoPETs'21

Abstract: We apply multiparty computation (MPC) techniques to show, given a database that is secret-shared among multiple mutually distrustful parties, how the parties may obliviously construct a decision tree based on the secret data. We consider data with continuous attributes (i.e., coming from a large domain), and develop a secure version of a learning algorithm similar to the C4.5 or CART algorithms. Previous MPC-based work only focused on decision tree learning with discrete attributes (De Hoogh et al. 2014).
Our starting point is to apply an existing generic MPC protocol to a standard decision tree learning algorithm, which we then optimize in several ways. We exploit the fact that even if we allow the data to have continuous values, which a priori might require fixed or floating point representations, the output of the tree learning algorithm only depends on the relative ordering of the data. By obliviously sorting the data we reduce the number of comparisons needed per node to $O(N \log^2 N)$ from the naive $O(N^2)$, where N is the number of training records in the dataset, thus making the algorithm feasible for larger datasets. This does however introduce a problem when duplicate values occur in the dataset, but we manage to overcome this problem with a relatively cheap subprotocol. We show a procedure to convert a sorting network into a permutation network of smaller complexity, resulting in a round complexity of $O(\log N)$ per layer in the tree.
We implement our algorithm in the MP-SPDZ framework and benchmark our implementation for both passive and active three-party computation using arithmetic modulo 2^{64}. We apply our implementation to a large scale medical dataset of $\approx 290\,000$ rows using random forests, and thus demonstrate practical feasibility of using MPC for privacy-preserving machine learning based on decision trees for large datasets.

Other works. Finally, during the course of his Ph.D. studies, the author published other research articles that are not directly relevant for the main topic addressed in this thesis, namely secure multiparty computation over $\mathbb{Z}/2^k\mathbb{Z}$. These are listed below.

Rank Analysis of Cubic Multivariate Cryptosystems [11]

Authors: John B. Baena, Daniel Cabarcas, *Daniel Escudero*, Karan Khathuria and Javier A. Verbel.
Published: PQCRYPTO'18

Abstract: In this work we analyze the security of cubic cryptographic constructions with respect to rank weakness. We detail how to extend the big field idea from quadratic to cubic, and show that the same rank defect occurs. We extend the min-rank problem and propose an algorithm to solve it in this setting. We show that for fixed small rank, the complexity is even lower than for the quadratic case. However, the rank of a cubic polynomial in n variables can be larger than n, and in this case the algorithm is very inefficient. We show that the rank of the differential is not necessarily smaller, rendering this line of attack useless if the rank is large enough. Similarly, the algebraic attack is exponential in the rank, thus useless for high rank.

Efficient Protocols for Oblivious Linear Function Evaluation from Ring-LWE [14]

Authors: Carsten Baum, *Daniel Escudero*, Alberto Pedrouzo-Ulloa, Peter Scholl and Juan Ramón Troncoso-Pastoriza.
Published: SCN'20

Abstract: An oblivious linear function evaluation protocol, or OLE, is a two-party protocol for the function $f(x) = ax + b$, where a sender inputs the field elements a, b, and a receiver inputs x and learns $f(x)$. OLE can be used to build secret-shared multiplication, and is an essential component of many secure computation applications including general-purpose multi-party computation, private set intersection and more.
In this work, we present several efficient OLE protocols from the ring learning with errors (RLWE) assumption. Technically, we build two new passively secure protocols, which build upon recent advances in homomorphic secret sharing from (R)LWE (Boyle et al., Eurocrypt 2019), with optimizations tailored to the setting of OLE. We upgrade these to active security using efficient amortized zero-knowledge techniques for lattice relations (Baum et al., Crypto 2018), and design new variants of zero-knowledge arguments that are necessary for some of our constructions.
Our protocols offer several advantages over existing constructions. Firstly, they have the lowest communication complexity amongst previous, practical protocols from RLWE and other assumptions; secondly, they are conceptually very simple, and have just one round of interaction for the case of OLE where b is randomly chosen. We demonstrate this with an implementation of one of our passively secure protocols, which can perform more than 1 million OLEs per second over the ring $\mathbb{Z}_m$, for a 120-bit modulus m, on standard hardware.

Improved Threshold Signatures, Proactive Secret Sharing, and Input Certification from LSS Isomorphisms [9]

Authors: Diego F. Aranha, Anders Dalskov, *Daniel Escudero* and Claudio Orlandi.
Published: LATINCRYPT'21

Abstract: In this paper we present a series of applications stemming from a formal treatment of linear secret-sharing isomorphisms, which are linear transformations between different secret-sharing schemes defined over vector spaces over a field $\mathbb{F}$ and allow for efficient multiparty conversion from one secret-sharing scheme to the other. This concept generalizes the folklore idea that moving from a secret-sharing scheme over $\mathbb{F}_p$ to a secret sharing "in the exponent" can be done non-interactively by multiplying the share unto a generator of e.g., an elliptic curve group. We generalize this idea and show that it can also be used to compute arbitrary bilinear maps and in particular pairings over elliptic curves.
We include the following practical applications originating from our framework: First we show how to securely realize the Pointcheval-Sanders signature scheme (CT-RSA 2016) in MPC. Second we present a construction for dynamic proactive secret-sharing which outperforms the current state of the art from CCS 2019. Third we present a construction for MPC input certification using digital signatures that we show experimentally to outperform the previous best solution in this area.

Honest Majority MPC with Abort with Minimal Online Communication [35]

Authors: Anders Dalskov and *Daniel Escudero*.
Published: LATINCRYPT'21

Abstract: In this work we focus on improving the communication complexity of the

online phase of honest majority MPC protocols. To this end, we present a general and simple method to compile arbitrary secret-sharing-based passively secure protocols defined over an arbitrary ring that are secure up to additive attacks in a malicious setting, to actively secure protocols with abort. The resulting protocol has a total communication complexity in the online phase of $1.5(n-1)$ shares, which amounts to 1.5 shares per party asymptotically. An important aspect of our techniques is that they can be seen as generalization of ideas that have been used in other works in a rather *ad-hoc* manner for different secret-sharing protocols. Thus, our work serves as a way of unifying key ideas in recent honest majority protocols, to understand better the core techniques and similarities among these works. Furthermore, for $n = 3$, when instantiated with replicated secret-sharing-based protocols (Araki et al. CCS 2016), the communication complexity in the online phase amounts to only 1 ring element per party, matching the communication complexity of the BLAZE protocol (Patra & Suresh, NDSS 2020), while having a much simpler design.

Information-Theoretically Secure MPC against Mixed Dynamic Adversaries [39]

Authors: Ivan Damgård, *Daniel Escudero* and Divya Ravi.
Published: TCC'21

Abstract: In this work we consider information-theoretically secure MPC against an *mixed* adversary who can corrupt t_p parties passively, t_a parties actively, and can make t_f parties fail-stop. With perfect security, it is known that every function can be computed securely if and only if $3t_a + 2t_p + t_f < n$, and for statistical security the bound is $2t_a + 2t_p + t_f < n$.
These results say that for each given set of parameters (t_a, t_p, t_f) respecting the inequality, there exists a protocol secure against this particular choice of corruption thresholds. In this work we consider a *dynamic* adversary. Here, the goal is a *single* protocol that is secure, no matter which set of corruption thresholds (t_a, t_p, t_f) from a certain class is chosen by the adversary. A dynamic adversary can choose a corruption strategy after seeing the protocol and so is much stronger than a standard adversary.
Dynamically secure protocols have been considered before for computational security. Also the information theoretic case has been studied, but only considering non-threshold general adversaries, leading to inefficient protocols.
We consider threshold dynamic adversaries and information theoretic security. For statistical security we show that efficient dynamic secure function evaluation (SFE) is possible if and only if $2t_a+2t_p+t_f < n$, but any dynamically secure protocol must use $\Omega(n)$ rounds, even if only fairness is required. Further, general reactive MPC is possible if we assume in addition that $2t_a + 2t_f \leq n$, but fair reactive MPC only requires $2t_a + 2t_p + t_f < n$.
For perfect security we show that both dynamic SFE and verifiable secret sharing (VSS) are impossible if we only assume $3t_a + 2t_p + t_f < n$ and remain impossible even if we also assume $t_f = 0$. On the other hand, perfect dynamic SFE with guaranteed output delivery (G.O.D.) is possible when either $t_p = 0$ or $t_a = 0$ i.e. if instead we assume $3t_a + t_f < n$ or $2t_p + t_f < n$. Further, perfect dynamic VSS with G.O.D. is possible under the additional conditions $3t_a+3/2t_f \leq n$ or $2t_p+2t_f \leq n$. These conditions are also sufficient for dynamic perfect reactive MPC.

Improved single-round secure multiplication using regenerating codes [3]

Authors: Mark Abspoel, Ronald Cramer, *Daniel Escudero*, Ivan Damgård and Chaoping Xing.
Published: ASIACRYPT'21

Abstract: In 2016, Guruswami and Wootters showed Shamir's secret-sharing scheme defined over an extension field has a regenerating property. Namely, we can compress each share to an element of the base field by applying a linear form, such that the secret is determined by a linear combination of the compressed shares. Immediately it seemed like an application to improve the complexity of unconditionally secure multiparty computation must be imminent; however, thus far, no result has been published.
We present the first application of regenerating codes to MPC, and show that its utility lies in reducing the number of rounds. Concretely, we present a protocol that obliviously evaluates a depth-d arithmetic circuit in $d + O(1)$ rounds, in the amortized setting of parallel evaluations, with $o(n^2)$ ring elements communicated per multiplication. Our protocol is secure against the maximal adversary corrupting $t < n/2$ parties. All existing approaches in this setting have complexity $\Omega(n^2)$.
Moreover, we extend some of the theory on regenerating codes to Galois rings. It was already known that the repair property of MDS codes over fields can be fully characterized in terms of its dual code. We show this characterization extends to linear codes over Galois rings, and use it to show the result of Guruswami and Wootters also holds true for Shamir's scheme over Galois rings.

Part I

MPC Fundamentals

Chapter 1

The Theory of Multiparty Computation

The goal of this initial chapter is to introduce the reader to some of the most relevant existing theoretical concepts in the area of secure multiparty computation. First, in Section 1.1, we present a general and high level introduction to some of the most important concepts and ideas in MPC, like the notion of an adversary, and different settings and goals that are typically considered in MPC. Then, in Section 1.2, we present the idea of simulation-based security, which is the formal mathematical machinery necessary to properly define the concepts discussed in Section 1.1. This tool allows us to approach the task of securely computing a given function from a mathematical point of view, enabling us to obtain precise and explicit security results. Finally, we discuss in Section 1.3 some of the most relevant results in the theory of secure multiparty computation, which are related to the types of security notions that can be achieved in the three main distinctive settings: two-thirds honest majority, honest majority, and dishonest majority.

1.1 A General Introduction to MPC

In secure multiparty computation we consider a setting where n parties $P_1, \dots, P_n$, each P_i having an input x_i, want to securely compute a given function $f(\mathtt{X}_1, \dots, \mathtt{X}_n)$, in such a way that only the value $z = f(x_1, \dots, x_n)$ is learned, and nothing else about $x_1, \dots, x_n$ is revealed. This is intended to be achieved by means of an *MPC protocol*, which is a set of rules that the parties execute, involving some local computation plus some exchange of messages. These rules depend on the inputs of each party, and they are typically randomized, meaning that they usually depend on some random bits sampled by each party as well.

Ideally, nothing should be learned about the inputs $x_1, \dots, x_n$, except perhaps from what is leaked about the output z.[1] Towards formalizing this notion, it is useful to think of an *ideal world* in which there is a *trusted third party* who receives the inputs from the parties, computes the result z, and sends this to all the participants, promising to perform the correct computation and not to leak absolutely anything else besides z. The goal of a secure multiparty computation protocol is to instantiate this type of scenario without

[1] Notice that it might be the case that the output z reveals a lot of information about some of the inputs, which is obvious for instance if the function f is defined as $\mathtt{X}_1 = f(\mathtt{X}_1, \dots, \mathtt{X}_n)$. This is acceptable in the context of MPC given that the only goal is to protect the inputs $x_1, \dots, x_n$, *except possibly* for what is leaked by the output z itself.

the presence of a trusted third party, only involving communication among the parties holding the inputs themselves. In other words, a protocol must match the behavior of the ideal world in what we call the *real world*. As we will see in Section 1.2, this ideal/real world paradigm is not only useful to understand at a high level what the guarantees of a secure multiparty computation protocol should be, but it also serves as the core idea behind a proper mathematical formalization, which enables the use of rigorous definitions and theorems.

1.1.1 Adversaries and their Power

In a wide range of notions and constructions across all of cryptography, it is very common to formalize the idea of something being "secure" by considering the idea of an *adversary*, which is an entity who tries to break whatever property we are trying to protect, and should not be able to succeed with reasonable probability. This adversary is simply an algorithm, a mathematical object that can be formally defined. For example, in the case of encryption it is common to define security (at least, one particular notion among several other variants) by requiring that no (typically efficient) adversary can win at a "game" that is supposed to represent a real-world scenario where an attacker gets to interact with an already-deployed encryption scheme. In this game, the adversary gets to choose two different messages, and it gets an encryption with an unknown random key of one of these two plaintexts, chosen at random. It is the adversary's goal to determine which of the two messages was encrypted. If no adversary, which in essence means no algorithm, is able to significantly win at this game, then, intuitively, it must be the case that the encryption scheme is good at its job of hiding data, since encryptions of different messages look indistinguishable.

Many other notions in cryptography, like the security of digital signatures or key exchange mechanisms, are formalized via adversaries attacking the system, and secure multiparty computation is naturally no exception. Consider an execution of a secure multiparty computation protocol where n parties $P_1, \ldots, P_n$ engage in a set of communication and computation rules, exchange messages, and return a result at the end of this interaction. Who should be the adversary in such scenario? As the name implies, an adversary is a "rival" whose aim is to break a given security property we are trying to maintain, in this case, the fact that the inputs of the parties remain private, except from the output z. For example, we can consider one of the parties in the execution of the protocol to be the adversary, and what we could require is that this party, after the interaction with the other participants, this "adversarial party" does not learn anything about the other parties' inputs, besides what is leaked by the output of the computation.

Let us assume that a given secure computation protocol satisfies the notion that no party, regarded as the adversary, can learn anything about the inputs from the other parties, as considered above (assume for now that we can appropriately define the idea of "not learning more than what is leaked by the output", which is achieved via simulation-based security as considered in Section 1.2). A natural question is, what would such notion reflect in practice? In principle, it is very powerful: if any of the parties behaves as an adversary, trying to learn anything from the other parties' inputs (besides what is leaked by the output), this party will fail at doing so. However, it fails to capture a very

important "attack" that could easily happen in a practical scenario, and it has to do with the possibility of a *collusion attack*. Imagine that, among the n parties participating in the protocol, there are two which, for different reasons, might benefit from learning the inputs from the remaining parties. These two parties may decide to trust each other and *collude*, that is, work together, perhaps by sharing out-of-band messages to each other, if this somehow helps them in their task of learning information they are not allowed to gather. Alternatively, if by joining the internal data of different parties it is possible to break privacy, then an external attacker that breaks into several of the participating machines can learn private information.

In light of the scenario described above, which could easily appear in practice, it becomes important to somehow incorporate into our adversarial definition the idea of parties colluding, working together, sharing information to each other, in order to break the privacy of the remaining parties. One could in principle achieve this by considering different adversaries that somehow communicate to each other, but it turns out to be a much simpler way if, instead of following this approach, we consider a *single* adversary, as before, that, instead of simply playing the role of an individual party, it completely controls a given set of parties. This attacker plays a similar role as the "hacker" described in our previous example, and it also serves to model the case in which two or more parties collude voluntarily, since the strategy they follow in their collusion process can be modeled as an algorithm, which can be ultimately regarded an adversary on its own. Finally, notice that this notion strictly generalizes the one we considered initially above, where a protocol was secure if no single party acting as an adversary could violate privacy. This case corresponds to the scenario when an adversary corrupts a set containing only one single party.

With this idea of what the adversary role is in the execution of a secure multiparty computation protocol, we proceed to describe and categorize many of the possible different variants that such adversary can present. Before we do this, however, we introduce some notation that will be used throughout this work. Suppose that the n parties participating in a given protocol execution is $P_1, \dots, P_n$. The set of indexes in $[n]$ corresponding to corrupted parties is denoted by $\mathcal{C}$, while the set of indexes corresponding to the remaining parties, which are also called *honest parties*, is denoted by $\mathcal{H} = [n] \setminus \mathcal{C}$.

1.1.1.1 Possible Corrupted Sets

In our first naive notion of security a protocol was considered secure if *any* adversary, corrupting *any* single one of the parties $P_1, \dots, P_n$ participating in the protocol, cannot learn anything about the remaining parties' inputs. Observe that in this case the protocol should remain secure no matter what party is corrupted. If, for example, we only require security against adversaries corrupting one of the parties among $P_2, \dots, P_n$ (so P_1 is never corrupted), a simple and trivial protocol would consist of the parties sending their inputs to P_1, who computes the function and returns the result. This satisfies the security notion since we only require that the adversary does not learn any extra information about the honest parties' inputs, which is trivially guaranteed since P_1 is always honest. We see then that, depending on which parties are allowed to be corrupted by the adversary, different protocols might exist.

As we have mentioned in previous paragraphs, in our more general context the adversary is not restricted to corrupting one single party but rather it can corrupt a set of parties potentially having size greater than one. However, in order to define security, we require the protocol to protect the privacy of the honest parties' inputs in the event in which the adversary corrupts a given set of parties. A crucial question that appears under this consideration is then the following: which sets of parties can the adversary corrupt? In the single-corruption case from above these possible sets are $\{P_1\}, \dots, \{P_n\}$, and as before, if the collection of possible sets the adversary can corrupt is too simple (e.g. all the possible sets miss one specific party, which means that this party is always honest), the task of designing secure multiparty computation protocols may become trivial.

In general, the collection of possible sets the adversary can corrupt is a security property of a given protocol. Such collection, which is simply a set of subsets of $[n]$, is called an *adversarial structure*, and different secure multiparty computation protocols are designed with the goal of withstanding corruptions from different adversarial structures. Below we consider some relevant adversarial structures. Before we do this, however, we note that, if a set B is part of a given adversarial structure, then it makes sense to include any subset $A \subseteq B$ into the structure as well, given that a protocol cannot be secure against corruptions in A if it is insecure by the adversary corrupting a smaller subset. Given this, we define adversarial structures as antimonotone collections of subsets of $[n]$, meaning that if B is in the collection, every set $A \subseteq B$ has to be part of it too.

$\mathcal{Q}2$ and $\mathcal{Q}3$ Adversarial Structures. The ability to consider general adversarial structures is very useful in scenarios in which there is a lot of asymmetry among the "importance" of the different parties. For example, consider a setting with three parties P_1, P_2, P_3, and suppose for simplicity in the argument that they have no reasons or motivations to voluntarily collude. However, there is still the concern that an attacker breaks into some of these machines. Suppose now that P_1 is very well protected, but P_2 and P_3 have a weaker safeguards. In such a setting we might consider a protocol that withstands the adversarial structure $\{\{P_1\}, \{P_2\}, \{P_3\}, \{P_2, P_3\}\}$. This way, if the adversary wants to break the system then it has to corrupt a set of parties outside this structure, so either P_1 *and* one of P_2 or P_3. Since $\{P_2, P_3\}$ is part of the adversary structure, even if the attacker breaks into the two weaker machines P_2 and P_3, it cannot still breach privacy.

There is a long and important line of study into how to design secure multiparty computation protocols for general adversarial structures, starting with the work of [33]. However, among all possible adversarial structures, there are two types that are particularly important. We will make more explicit the relevance of these two particular structures later in Section 1.3.

$\mathcal{Q}2$ structures. An adversarial structure is $\mathcal{Q}2$ if, for every A_1 and A_2 in the structure, $A_1 \cup A_2 \neq \{P_1, \dots, P_n\}$.

$\mathcal{Q}3$ structures. An adversarial structure is $\mathcal{Q}3$ if, for every A_1, A_2 and A_3 in the structure, $A_1 \cup A_2 \cup A_3 \neq \{P_1, \dots, P_n\}$.

Threshold Adversarial Structures. In settings in which there is more "symmetry" among the parties, and there are no obvious reasons to put more weight into how easy it is for one party to get broken into, or into how likely it is that a given party colludes, with respect to another party, a natural adversarial structure is the *threshold structure*. This measures the adversary's capabilities by how many parties it can corrupt simultaneously, without making any distinction among the parties whatsoever. More concretely, a threshold adversary structure is parameterized by a value $0 < t < n$, and it consists of all the subsets of $[n]$ of size at most t. Intuitively, a protocol that is secure against such structure guarantees privacy of the honest parties' inputs as long as the adversary does not corrupt more than t parties.[2]

Below we discuss three types of threshold adversarial structures, depending on their threshold value t. The main importance of these distinctions will be made clear in Section 1.3, when we discuss several fundamental results in the feasibility of secure multiparty computation protocols depending on each of these cases.

Honest majority, $t < n/2$. In this case the adversary is assumed to corrupt at most $t < n/2$ parties, so, no matter what set of corrupted parties is chosen, the set of honest parties constitute a majority. It is easy to verify that the resulting adversarial structure is $Q2$.

Two-thirds honest majority, $t < n/3$. Now the adversary is assumed to corrupt even less parties, at most $t < n/3$. Here the set of honest parties is always at least two-thirds the total number of parties. It is easy to verify that the resulting adversarial structure is $Q3$.

Dishonest majority, $t < n$. This is the scenario where the no special bound on t holds, so t can take the largest possible value, $t = n-1$. In this case the adversary can corrupt any set containing all but one party, and a protocol secure in this setting would still guarantee privacy to this remaining party. This is the strongest possible setting: intuitively, each party knows individually that their inputs are secure, even if all the other parties collude. This is not the case with any of the previous scenarios (and in general, if $t < n - 2$), since in these cases the adversary can break the privacy of an individual party's input by corrupting a set with at least $t + 1$ parties.

We remark that, throughout this thesis, our only focus lies in threshold adversarial structures.

Remark 1.1 (Maximal vs non-maximal adversaries)**.** *Intuitively, when designing secure multiparty computation protocols in any of the threshold settings listed above, it is better to consider the maximum possible value of t that respects the given bound. For example, in the honest majority setting where $t < n/2$, the best is to choose t as the largest integer that respects this bound, i.e. $t = \lceil \frac{n}{2} - 1 \rceil$, since in this case the resulting protocol tolerates the largest number of corruptions while still falling within the honest majority setting.*[3]

[2] Notice that t lies between 1 and $n - 1$. If $t = 0$ then there are no corruptions and the task of secure multiparty computation becomes trivial. Also, if $t = n$, then all parties are corrupted so there are no honest parties' inputs to protect the privacy of.

[3] It is important to mention that having a gap between t and $n/2$ (or $n/3$) is sometimes useful as it allows

Motivated by the above, it is quite common, and in fact, we do so in several opportunities in this thesis, to assume for simplicity that the adversary corrupts exactly t parties, and that t is as large as possible while respecting the bound under consideration. At an intuitive level, this should be without loss of generalization given that, if a protocol is secure against an adversary that corrupts exactly, say, $\lceil \frac{n}{2} - 1 \rceil$ parties, then the protocol should remain secure even if the adversary corrupts less, since this means that now the adversary is less powerful.

Unfortunately, although such reasoning makes a lot of sense, the mathematical model under which the concept of MPC is formalized, which is discussed in Section 1.2, does not satisfied this property. More precisely, there are protocols that are secure against $\lceil \frac{n}{2} - 1 \rceil$ corruptions, but an adversary corrupting less than this amount can somehow break the protocol. This counter-intuitive nuisance is hardly an issue in practice, but it is important to be aware of it. We revisit this issue when we assume a maximal adversary in this thesis.

1.1.1.2 Type of Corruption

Our current corruption model is intended to represent a set of parties colluding, exchanging messages out-of-band, sharing their internal state, and possibly coordinated by an attacker, which is in fact how the proper adversarial model is formalized. Recall that a secure multiparty computation protocol is in essence a set of computation and communication rules that the parties have to follow in order to securely compute a given function. So far, although the adversary is able to see all the internal state of the corrupt parties, including messages received and sent by these, we have implicitly assumed that the corrupt parties follow the rules specified by the protocol faithfully. This corresponds to the notion of a *passive* of *semi-honest* adversary, and it is intended to reflect a setting in which all of the parties are assumed to follow the protocol instructions, but even if an attacker is able to access the internal information from a given subset of the parties (within certain adversarial structure), the protocol should guarantee privacy of the inputs from the remaining parties.

Unfortunately, it is in principle not possible for the parties to somehow verify that the other participants are following the protocol specification faithfully. This is because, ultimately, all the different parties see from other participants are messages which, although depend on their private inputs, are supposed to reveal nothing about these values from the security definition itself. From this, if an adversary can gain additional information by modifying the *behavior* of the corrupt parties during the execution of the protocol, perhaps in a way in which such misconduct goes undetected towards the honest parties, a need to protect secure multiparty computation protocols against such actions appears.

An adversary with the more advanced and realistic capabilities described above is referred to as an *active* or *malicious* adversary, and protocols that are secure against such type of adversarial behavior are the most ideal to deploy in much of the potential use cases for secure multiparty computation, given that it prevents corrupt parties from causing any harm during the execution of the protocol, even if they internally deviate from

the use of*packed secret sharing*, a technique to improve efficiency of MPC protocols in these scenarios [40, 52].

the specified rules, which in principle they would be able to do without being detected. However, although these protocols are stronger than their passively secure counterparts, they are naturally much harder to construct, plus they tend to add certain overhead in terms of performance.

So, to summarize:

Passive/semi-honest corruption. An adversary is said to be passive if the behavior of the corrupt parties during the protocol execution is exactly as specified by the protocol description. The adversary sees the internal state of the corrupt parties, including in/out messages, input and internal random coins, but it cannot alter their behavior.

Active/malicious corruption. An adversary is said to be active if it controls the corrupt parties completely, including possibly modifying their actions during the execution of the protocol.

In this work we will consider both passive and active adversaries. The description of existing MPC constructions (over fields) in Chapter 2 considers both cases, while the protocols (over $\mathbb{Z}/2^k\mathbb{Z}$) presented in Part II of this thesis are all set in the scenario in which the adversary is active.

1.1.2 Privacy Guarantees

Recapping what we have seen so far, our intuitive security definition for secure multiparty computation protocols requires that an adversary, corrupting (either passively or actively) a set from an adversarial structure, which is simply a collection of possible corruption sets, learns nothing from the honest parties' inputs, which, as will be made more precise in Section 1.2, is formalized by requiring that the protocol execution somehow looks "close" to an ideal world in which a trusted third party is used. In this section we explore in a bit more detail what the concept of these two executions being "close" means. The description here is rather verbal and intuitive, and it is only made more precise in Section 1.2 where we properly define the idea of simulation-based security.

Perfect security. In this case the real and ideal executions follow the *exact same* distribution, so, from the point of view of the adversary, nothing is learned about the honest parties' inputs from the protocol execution. This is regardless of the computational resources available to the adversary.

Statistical security. This is a slightly weaker notion, and it is also called unconditional security. In this case the real and ideal worlds have *statistically close* distributions, meaning that the distributions from the real and ideal worlds may not be the same, but are *very* close. More precisely, by controlling certain parameter of the given construction, it is possible to shorten the gap between these two distributions by any desired amount. To illustrate what this type of security entails, it is useful to think, as an example, of an MPC construction that achieves the following: the real and ideal worlds follow the exact same distribution, except that there is a very

small chance (regardless of its computational powers) in which the adversary can completely break privacy of the honest parties' inputs in the real world. In this case, security would be statistical.

Computational security. The two security notions above are very powerful, but as we will see in Section 1.3, they are not always possible to achieve. As a result, and motivated by practice, it is useful to restrict security to only *efficient adversaries*, that is, adversaries that make use of a bounded amount of resources, which are formally modeled as algorithms running in polynomial time (in terms of a security parameter). In a computationally secure protocol, the distributions from the real and ideal worlds are indistinguishable, but only as long as the adversary is not infinitely powerful, which is enough for practical use. As an example, consider an MPC protocol in which some party has to send an encryption of its input using a secret key. Although this might be hard to break for an adversary not knowing the secret key, an attacker with infinite resources can brute-force the ciphertext to recover sensitive information.

The first two notions, perfect and statistical security, are commonly referred to as *information-theoretic security*. It is typically the case that protocols satisfying these notions of security tend to be more efficient than computationally secure ones, given that they are usually simpler and do not rely on certain parameters that aim at making a given computational problem hard. However, as we have mentioned, information-theoretic security is not always possible to achieve.

1.1.3 Output Guarantees

Recall that, to define security of secure multiparty computation protocols, we have resorted to comparing the real world, where the execution of the given protocol takes place, to an ideal world where the desired function is computed by a trusted third party, who receives the inputs from the parties and promises to reveal only the output. Defined in this way, the parties have the guarantee in the real world that their inputs are as protected in the actual interaction as in the ideal world, where only the output is revealed. However, another subtle property of the trusted third party is that, as described above, it *always* returns the correct output of the computation to all of the parties. Unfortunately, as we will see in Section 1.3, this notion, which is called *guaranteed output delivery* in the literature, is not always possible to achieve in the real world. In some settings, for example, the best that can be achieved is that if the parties obtain a result, it is guaranteed to be the correct one, but it could be the case that no party obtains any result at all.

From the above, it becomes necessary to relax the requirements in the ideal world regarding the output of the computation. To this end, we present the following three notions related to this. We remark that, in any of the three concepts below, whenever the parties obtain an output it is guaranteed to be the *correct* one.

Guaranteed output delivery. As described above, in this case all the parties are guaranteed to obtain the output, regardless of the actions that the corrupt parties perform.

Fairness. In this case it could happen that the adversary causes the honest parties to not obtain the output, which is referred to as causing the parties to *abort*. However, if this happens, then the corrupt parties (and hence the adversary) are guaranteed to also *not* learn the output.

Security with abort. In this scenario it can be the case that the adversary denies the honest parties from learning the output, while the corrupt parties may still learn the result. Furthermore, we identify two possible variants: in *security with unanimous abort* all the honest parties are guaranteed to either receive the output or abort altogether, and in *security with selective abort*, which is even weaker, the adversary can choose which honest parties receive output and which honest parties abort.

If the adversary is passive, the corrupt parties will behave exactly as if they were honest parties, following the protocol specification, so in this case it always holds that the protocol terminates with the correct output, which in particular means that, trivially, guaranteed output delivery is obtained.

Additionally, we note that, although the most desired property is guaranteed output delivery, which ensures availability of the output under all possible attacks, the notion of fairness is already very useful in practice, as the adversary does not learn or gain anything from stopping the (honest) parties from learning the output. For example, if the computation under consideration is distributed voting, a protocol that simply satisfies security with abort may allow the adversary to first learn the outcome of the voting, and decide to deny the honest parties from learning this if desired. A protocol with fairness, however, may simply disrupt the computation (which is of course a problem of a different nature on its own), but the adversary cannot decide to cause an abort depending on the output of the computation.

1.1.4 Different Functions to be Computed

Another important consideration when designing secure multiparty computation protocols is what type of computations they are intended to operate with. In this section we provide a general discussion on the topic, differentiating between several important types of computations.

1.1.4.1 Public-Output vs Private-Output.

So far, our description of the function to be computed has been $f(\mathtt{x}_1, \ldots, \mathtt{x}_n)$, with z being the output of applying this function to the inputs of the parties, $x_1, \ldots, x_n$, which is the value that all parties learn at the end of the execution of a given secure multiparty computation protocol. This scenario is referred to as the *public-output* setting, since there is only one result, that all parties learn equally. Alternatively, we may consider the case where each party receives a different result. In this case, we regard the function as producing n different outputs $(z_1, \ldots, z_n)$, where each party P_i is intended to learn (only) z_i. Clearly, this scenario, called the *private-output* setting, is a generalization of

the public-output case (by taking $z_i = z$ for $i \in [n]$), but it is fortunately not much harder to achieve. The reason for this is that, given an MPC protocol for public-output functions, each party can simply provide as an additional input a secret-key only known to this participant, and the function to be computed can be modified so that all the outputs are returned to all parties, except that output z_i is "encrypted" under the key provided by party P_i. This way, only this party can recover its corresponding output.

In this thesis we sometimes consider the public-output case (specially when computing arithmetic circuits), and in some other occasions we consider the private-output setting (when in the context of the arithmetic black box model). See Section 1.2.6.2 for details.

1.1.4.2 Reactive vs Non-Reactive Functionalities

A *reactive* functionality is one that enables the parties to learn "partial results" of the computation, and continue the process in a way that perhaps depend on these intermediate results, plus possibly new inputs. A good example of this type of computations is given by, for example, a commitment scheme, which enables a party to commit to a given value without revealing its contents, to do so at a later stage without the ability of announcing a different value to the one committed earlier. On the other hand, in a *non-reactive* functionality the parties simply provide their inputs at the beginning of the protocol, and obtain the result at the end of the execution. There are many relevant applications that can be phrased as non-reactive functionalities, like data processing tasks, distributed voting and auctions, and many others.

Clearly, reactive secure multiparty computation is a more general setting than the non-reactive case. However, it is generally the case that one can obtain a secure multiparty computation for reactive functionalities from a protocol that only supports non-reactive ones by making use of a technique called *verifiable secret-sharing*. In short, this technique enables the parties to obtain a "shared state" of each checkpoint in a reactive computation using the non-reactive protocol, which can be reconstructed to obtain partial outputs. Although there are certain scenarios in which non-reactive computation is possible while reactive computation is not, it is generally the case that the two notions are back-to-back, so the difference between the two is typically irrelevant for the discussion of different secure multiparty computation protocols.

1.1.4.3 General vs Special-Purpose MPC

Another relevant distinction for secure multiparty computation protocols lies in whether they are designed to support *any* arbitrary function, or if they are tailored to specific functions. The former family is typically referred to as *general-purpose* MPC protocols, while the latter, being more targeted to particular computations, is called *special-purpose* MPC.

It is fair to say that most of the proposed secure multiparty computation protocols in the literature correspond to general-purpose constructions. However, at first glance, this

may sound as an extremely difficult task: how can a single protocol support *any* arbitrary computation? The catch is in the way that computations are represented, which involves the concept of *arithmetic circuits*, discussed below.

Arithmetic Circuits and General-Purpose MPC. As we have mentioned above, researchers in the field of secure multiparty computation have focused mostly in designing general-purpose MPC protocols, which are intended to securely compute *any* functionality that the parties wish to compute. This is possible thanks to an abstraction known as *arithmetic circuits*, which concisely captures any possible function in terms of rather simple operations over certain algebraic structure.

Consider a finite field $\mathbb{F}$, and let $f : \mathbb{F}^n \mapsto \mathbb{F}$ be any function defined over this field. A well known fact in field theory is that *every* such function can be written as an arithmetic circuits, which in formal terms is simply a directed acyclic graph with labeled nodes. Denoting by (i, o) a node that has fan-in i and fan-in o, every node is either of the type $(0, 1)$, $(2, 1)$, or $(1, 0)$. $(0, 1)$ nodes are called the *input gates*, and they model the inputs to the computation. $(2, 1)$ nodes are call the *operation gates*, and they represent field operations. A field has two main operations, addition $(+)$ and multiplication $(\cdot)$, and this is reflected in the fact that there are two types of gates *addition* and *multiplication gates*, each corresponding to a different operation. Finally, $(1, 0)$ nodes are the *output gates*.

Edges are also called *wires*, and in an actual execution of the function f, each wire has associated a value to it corresponding to an intermediate result of the computation. Wires outgoing from an input gate are associated to the actual value provided as input, corresponding to the given input gate. Wires leaving an operation gate are matched with the values corresponding to the result of applying the corresponding operation (addition or multiplication) to the (associated values to the) incoming wires to the gate at hand. Finally, the wire that goes into the output gate is precisely the result of the computation.

With this tool at hand, designing a general-purpose secure multiparty computation protocol reduces to constructing a protocol for securely computing arithmetic circuits exclusively, which is what most of the research in the field of MPC is concerned with. Furthermore, as we will see in Section 2.1, among all different techniques to design secure multiparty computation protocols, there is a promising general template known as secret-sharing-based MPC that, in a nutshell, works by letting the parties have a "hidden" representation of the inputs to the computation, together with some methods to obtain a hidden version of the output of each operation gate, assuming the inputs are already hidden. Eventually, the output is reached in hidden form, point in which the parties can "reveal" it so that the result of the computation is learned. From this template, the task of secure multiparty computation reduces to designing a method to (1) keep "hidden" versions of different values, (2) obtain a hidden version of the result of an addition or a multiplication, assuming the inputs are already hidden, and (3) reveal a hidden value. This, at a first glance, seems more feasible than the daunting task of securely computing *any* conceivable function.

General-purpose MPC protocols are particularly useful in theory as they show what type of computations are possible, which, accompanied with impossibility results, provide us

with a complete landscape of the sorts of computations we can hope for. However, their reachability in practice can be, in principle, more questioned. For instance, recall that these protocols work by first representing the desired computation as an arithmetic circuit over a finite field, which is simply a combination of additions and multiplications over this structure, but it is not at all clear what type of *practical* applications lend themselves to be *efficiently* expressed as an arithmetic circuit. As an starting example, applications that involve real-valued arithmetic, such as these in the domain of machine learning, for illustration, are more naturally written in terms of operations over the real numbers, including possibly additional processes that go beyond basic additions and multiplications (e.g. taking square roots, applying sine or cosine, or even non-mathematical operations such as flipping the bits of the given value in the bit-representation).

In spite of the above, many general-purpose MPC protocols are not a mere theoretical tool, as they tend to be the basis, or the starting point, of a wide range of more specialized protocols. This is achieved by adding certain subprotocols for specific operations that appear repeatedly across multiple applications, such as the case of real-valued arithmetic illustrated above. We will get the chance to discuss this in more detail in Chapter 7. Additionally, the existence of highly-efficient general-purpose MPC protocols has allowed the creation of several *MPC frameworks* that enable a set of parties interested in securely computing a given function to achieve this task with little-to-none knowledge of secure multiparty computation. This is achieved by enabling computation over arbitrary computer programs that, in essence, resemble an arithmetic circuit with several available sub-operations added on top. Popular frameworks of this kind include MP-SPDZ [61], SCALE-MAMBA [7], EzPC [28], among others, and it is fair to say that these implementations play a pivotal role into taking secure multiparty computation techniques from theory to practice.

Finally, we remark that the contents of this thesis are solely concerned with general-purpose secure multiparty computation. Only in Chapter 7 we discuss some MPC construction for certain specific circuits, but this is because they appear in a wide range of applications and serve as a building block, rather than them being a particular application on their own.

Special-Purpose MPC. A special-purpose secure multiparty computation protocol exploits particular properties of the given function in order to optimize the construction to the case at hand. This has major relevance in practice, but quite surprisingly, it also plays an important role in the theory of MPC.

First, in terms of practice, the benefits of considering special-purpose protocols are generally obvious: by exploiting the structure of the function to be computed it is typically the case that multiple savings in efficiency can be achieved with respect to the use of a more generic protocol for arbitrary computation. We remark, however, that many special-purpose MPC deployments have as a starting point more generic techniques to compute basic additions and multiplications, which come from the general-purpose MPC domain.

On the other hand, special-purpose secure multiparty computation constructions also have a tremendous impact in theory, by considering certain concrete functionalities. In

general, the theory of MPC, as far as feasibility or impossibility results is concerned, is not interested in secure multiparty computation protocols for very specific tasks such as image analysis or distributed voting. This is because, at the end of the day, one of the major results in the field of MPC is that secure multiparty computation of *any* function is generally possible, so designing special-purpose for these practically-oriented tasks is not that relevant in this direction.

Instead of considering special-purpose MPC protocols as a way of improving efficiency of more generic constructions, the main contribution of these type of protocols in terms of theory lies in simplifying the construction of other, possibly more general-purpose, protocols. More precisely, different useful functions that turn out to be crucial for the development of other protocols are identified, and as a result, advances and developments in regards to MPC constructions for these primitives, specifically, lead to general improvements across all other constructions that make use of these concrete functionalities. A good example of a particular function that is not only particularly useful on its own, but also serves as a major building block in many other secure multiparty computation protocols, is Oblivious Transfer. In short, this is a two-party functionality that receives two inputs, (m_0, m_1) and a bit b from two parties P_1 and P_2 respectively, and returns m_b to P_2, effectively allowing this party to learn only the chosen value m_b among m_0 and m_1, while P_1 does not learn which value was chosen. For a more detailed definition on this primitive, constructions and applications, we refer the reader to e.g. [64].

1.1.5 Efficiency Metrics

Finally, we discuss some of the efficiency metrics that we are typically interested in when designing secure multiparty computation protocols.

Computation complexity. A first measure is the amount of computation that each party has to carry out locally. Fully homomorphic encryption techniques are typically very costly in terms of computation, although they tend to have minimal overhead in terms of communication.

Communication complexity. Since MPC is a distributed application, it is important to measure how many bits need to be transmitted overall by all the parties during a protocol execution. Protocols based on garbled circuits tend to have a large communication complexity, although they round count is generally small.

Round count. Finally, orthogonal to communication complexity, which is affected by bandwidth resources, is the concept of round count, which is more relevant in terms of the latency between the parties. A communication round consists of one execution of the parties sending one message to each other. If a protocol involves many rounds, and there are parties having high-latency links between them, then the overall efficiency of the given MPC protocol might be poor. Secret-sharing-based MPC protocols, although they tend to be very reasonable in terms of communication complexity, suffer from a round count that is proportional to the number of layers present in the arithmetic circuit at hand.

1.2 Simulation-Based Security

One of the major achievements of modern cryptography lies in properly formalizing several constructions like encryption or digital signatures, so that rigorous mathematical reasoning could be applied on these. This way, concrete guarantees and relationship across different notions could be achieved, as exemplified by the idea of "provable security".

Secure multiparty computation appeared in the late 80's as an interesting task to study, having many potential applications and involving very interesting techniques. However, it took almost a decade until more formal approaches to secure multiparty computation appeared, which enabled a more rigorous treatment of the area. The "mathematical framework" under which the task of secure multiparty computation can be phrased is highly non-trivial, and is undoubtedly considered a contribution on its own.

1.2.1 High-Level Idea

There exist several different frameworks for formalizing secure multiparty computation protocols, like the stand-alone model [23], the UC framework [24] and the SUC framework [25], among others. However, although there are minor differences from one model to the other, what is common across all these approaches is that security is defined via *simulation*, an idea that is already common in the area of zero-knowledge proofs, for example, and serves the purpose of properly defining the notion of "not learning anything beyond $\mathtt{x}$". To provide a high level idea of what this technique is about, recall from Section 1.1 that the idea of a given secure multiparty computation protocol being secure is related to ensuring the adversary, who corrupts a subset of the parties, does not learn anything about the inputs from the honest parties, except perhaps from what is leaked by the output of the computation itself.

Formalizing the idea of an adversary not learning some data is not new in cryptography, as it has appears already, for example, in constructions such as encryption schemes, which are formalized using *game-based security*.[4] The main challenge in the MPC setting, however, is that the adversary does learn something about the data that is intended to be hidden, namely, the output of the computation. Furthermore, another major complication lies in the fact that secure multiparty computation is a distributed application, involving communication among the parties according to some specified pattern. The adversary gets to see all the messages exchanged with the corrupt parties during the protocol execution, and when the adversary is active, it even gets the power to modify the behavior of the corrupt parties. These complications put a barrier in the use of simpler game-based definitions widely used across cryptography.

The key idea to tackle the complication above, as we already hinted at in Section 1.1, is to consider an *ideal world* that captures the desired properties of the interaction, and

[4] In a nutshell, game-based security considers a scenario in which the adversary interacts with the system trying to distinguish certain data that is intended to be hidden, and security is formalized by requiring that no adversary can win at this "game" with high probability.

somehow requiring that the real world, where the actual protocol execution takes place, is indistinguishable from the ideal world. Typically, the ideal world consists of the parties sending their inputs to a third trusted party who computes the function and returns the result, and only the result, to the parties. Now, to claim that the two worlds are indistinguishable, a natural approach is to claim that the adversary cannot distinguish the real from the ideal world. However, this approached is doomed as these two worlds are trivially distinguishable from the point of view of the adversary: in the real world there are messages going to and from all the parties, there are several rounds, and there is no trusted parties, while in the ideal world there is a third trusted party that simply receives inputs and sends output. These two patterns look entirely different, so the adversary can clearly distinguish between the two.

This is where the idea of simulation kicks in. In the real world, the adversary, corrupting a subset of the parties, interacts with the honest parties and learns a result at the end of the execution. In the ideal world, the adversary will not directly interact with the trusted party. Instead, while the honest parties do interact with the third trusted party, the adversary interacts with some "virtual" honest parties that, unlike the actual honest parties, do not have access to the inputs intended to be kept hidden towards the adversary. These virtual honest parties are coordinated by a *simulator*, who also controls the corrupt parties in the ideal world, sending input and receiving output from the third trusted party. This way, the simulator effectively serves as an interface that enables the adversary to interact with the third trusted party in the ideal world, while having an interaction that equals the one from the real world.

To prove that a given secure multiparty computation protocol is secure, it is then necessary to define a simulator that acts as the interface sketched above, in such a way that the real world, where the adversary interacts with the actual honest parties holding the real inputs, is indistinguishable from the ideal world, where the adversary interacts with the virtual honest parties controlled by the simulator. Notice that the only power the simulator counts on in order to "fool" the adversary is the access to the third trusted party, which receives inputs and reveals solely the output. As a result, intuitively, this means that the adversary's experience in the real world can alternatively be "recreated" by having access only to the third trusted party, which, from a philosophical standpoint, instantiates the core idea of the adversary *only learning the output of the computation*, after the interaction with the honest parties in the real world.

With the intuitive approach outlined above, we now proceed to provide slightly more formal details on how simulation-based security works. We remark that, in this thesis, we focus only on the UC framework, leaving other simulation-based security notions such as stand-alone security aside. Furthermore, the description here is not intended to be fully self-contained, and the approach to the UC framework used in this section is taken from [34]. For a more complete treatment of the UC framework we refer the reader to this reference.

1.2.2 Interactive Agents

We begin by considering all the different entities involved in the formalization of a secure multiparty computation protocol and their security. Our starting point is the concept of an *interactive agent*, which, at an intuitive level, is a computational device that receives and sends messages, holds internal state and carries out computations. For example, the parties in a secure multiparty computation protocol are interactive agents, but in the framework under which these ideas are formalized several other interactive agents appear. Interactive agents can be formalized by the means of interactive Turing machines, which are simply traditional Turing machines (or algorithms) that, additionally to carrying out computations, can send and receive data to and from certain *communication ports*, which can be thought of as computer buses or channels.

1.2.2.1 Relevant Interactive Agents in the UC Framework

Now we describe the different interactive agents that appear in the UC framework. As we have mentioned already, the first natural interactive agents are the parties, which are the actual devices carrying out the computation, but several other interactive agents such as the simulator or the "trusted third party" appear. We discuss these below.

Parties. The parties, which we denote by $P_1, \ldots, P_n$, constitute the first natural example of interactive agents. Each party P_i, having certain input to the computation, proceeds according to the instructions of the protocol, performing local computation and sending/receiving data to/from the other parties, as required. At the end of this execution, each party P_i obtains the result of the computation.

Functionalities (in the real world). A functionality is simply an interactive agent that connects to the parties. It receives messages from them, performs local computation, maintains internal state, and sends messages back to the parties.

Although there is only one "type" of functionalities, these are used in two different contexts, with the first being in the real world, where the actual execution of the protocol takes place. In an execution of a secure multiparty computation protocol, the parties may be able to use certain "external" resources that may aid them during the computation. As an example, the parties may count on a third trusted party that, although it may not compute the whole desired function for the parties, may provide certain help like distributing some secret keys, or sending certain certificates. This can be formalized as a functionality that the parties talk to during the execution of the protocol at hand. Furthermore, what is crucial is that, as we will see in Section 1.2.5, if later on another protocol is developed that imitates the behavior of this functionality, then the parties can use this construction as a subroutine and the overall construction remains secure, without the need of a third trusted party to assumed to provide the secret-key or certificate service from the example above.

Furthermore, so far we have been implicitly assuming that the parties communicate among each other by means of special ports set between every pair of parties. However, now that we have introduced the concept of a functionality, it is convenient to consider communication instead as a functionality which receives messages from and sends messages to the parties. For example, a simple peer-to-peer network may be modeled as a functionality that acts as follows: party P_i sends a message of the type "send message m to party P_j", and the functionality sends to P_j the message "party P_i sends the message m".[5] Although this approach may seem as an unnecessary complication, it actually plays the important role of enabling flexibility in the way the parties talk to each other.[6] For example, in this thesis we consider as a basis a functionality that, in addition to modeling point-to-point encrypted and authenticated channels, supports an additional broadcast channel. More details are provided in Section 1.2.6.1.

Functionalities (in the ideal world). The second setting in which functionalities are used is in the ideal world. Recall that, in that world, there is a trusted third party that receives the inputs from the different parties and returns the result of the computation. This is precisely a functionality, which, as defined above, is an interactive agent that receives inputs from the parties, performs certain internal computation and sends a result to the parties.

In the most simple case, a functionality receives the inputs to a given function, evaluates this function internally, and returns the result to the parties. However, the concept of a functionality allows us to model much more complex interactions. For example, in Section 1.1.4.2 we discussed non-reactive computation, which enables the parties to obtain partial results and continue the computation afterwards. This can be captured by a functionality that receives inputs from the parties, stores some internal state, sends partial results, and continues in this fashion as indicated by the parties. We will discuss in Section 1.2.6 some basic functionalities we will use throughout this thesis.

Adversary. The adversary, denoted by $\mathcal{A}$, is modeled as another interactive agent, and it has ports communicating it to the each of the corrupt parties. If the corruption is passive, these ports are used to inform the adversary about the internal state of the corrupt parties, including the messages they have received. On the other hand, if the corruption is active, these ports are used to "fully control" the corrupt parties.

Environment. This entity plays a crucial role within the notion of simulation-based security. Intuitively, the environment is in charge of *distinguishing* the real world from the ideal world. We have mentioned in Section 1.2.1 that it is the adversary who cannot distinguish between the real and ideal worlds, but this is unfortunately insufficient. The main reason for this is that, if we simply require that the adversary cannot distinguish between

[5] Functionalities of this type are referred to as *communication resources* in [34], but we avoid this terminology in order to make it more clear that these functionalities are no different than the ones considered in the ideal world.

[6] Furthermore, an important low level detail of functionalities is that they leak certain information to the environment, which can be used to model the fact that, in practice, the adversary might be able to see certain metadata such as when an honest party sent a message to another honest party, its size, etc.

the two worlds, then, even though this would imply that the inputs from the honest parties are protected, it might be the case that the honest parties do not receive the correct output of the computation, which is also an important concern. This could occur since, to "fool" the adversary, the simulator only needs to create a similarly-looking interaction towards the adversary, but it could be that the honest parties in the real world end up computing a completely different result than in the real world, while in the ideal world they obtain the correct result. Since the adversary does not see these outputs as they belong to honest parties only, the two worlds would still be indistinguishable.

In order to address this issue, indistinguishability is defined in such a way that the inputs and the outputs of the computation are also taken into account. This is formalized by considering another agent, the *environment*, typically denoted by $\mathcal{Z}$, that indicates the parties which inputs to use, and receives from each party the result they obtained in the world under consideration (in the ideal world this corresponds to the correct result of the computation, while in the real world this is the result the party computes in the execution of the given protocol). Under this new consideration, a protocol is said to be secure if no environment can distinguish between the real and ideal worlds. Notice that this in particular means that the adversary cannot distinguish between the two worlds as otherwise the adversary could inform the environment which world is currently being executed, but, even if the two executions are indistinguishable to the adversary, the environment can still make use of the inputs it provided to the computation together with the outputs received to attempt to distinguish. If, even after this, the two executions are still indistinguishable, then it is because not only the distributions look similar to the adversary but also the outputs in the real world follow the same distribution as in the ideal world with respect to the inputs provided, which corresponds precisely to the correct results of the computation.

In the UC model, the environment and the adversary are essentially "one and the same", which is modeled by the fact that these two agents have a shared port that enables the environment to fully control the adversary, in essentially the same way as the adversary can fully control the honest parties in the case of an active corruption. Given that these two entities, including also the corrupt parties, are so entangled, in this work *we merge the environment, the adversary and the corrupt parties*, using the term environment/adversary indistinctively to refer to the resulting interactive agent. This entity is in charge of (1) playing the role of the corrupt parties and (2) sending inputs to the honest parties and receiving output from these.[7]

Simulator. Finally, as we have already discussed in Section 1.2.1, the simulator is in charge of acting as an "interface" between the adversary and the desired functionality in the ideal world. This is formalized by means of an interactive agent that connects to the adversary/environment in the ideal world through the same ports as the corrupt parties do in the real world, and also connects to the functionality under consideration, "on behalf" of the corrupt parties. This way, the simulator can send inputs to and receive

[7] More generally, the environment, as the name implies, gets to see all the "execution setting", which, on top of inputs and outputs, also involves other "metadata" such as information about when a party sends a message to another, the sizes of these, etc. This is formalized in [34] by means of *leakage ports*, which provide the environment with this type of information. The environment is also in charge of "scheduling" the execution of the protocol. We refer the reader to [34] for details.

outputs from the functionality, which constitutes the simulator's main tool to create an indistinguishable scenario towards the environment with respect to the real world.

1.2.3 Interactive Systems

An *interactive system* is simply a collection of interactive agents. As an example, a collection of parties is an interactive system, which we refer to as a *protocol*. We first review some notions that will be important for our discussions.

Open/closed ports. Recall that a communication port is simply a "channel" that different interactive agents have access to in order to send a receive messages. For example, each party has shared ports with the functionalities used in the protocol execution, which enables them to send and receive messages to/from it. An interactive system, being a collection of interactive agents, contains several ports. Many of them will involve at least two interactive agents, like the ports used between each party and a functionality. However, in an interactive system some ports may only involve one interactive agent. For example, the environment is in charge of sending inputs to and receiving output from the honest parties, plus it can send instructions to and receive information from the corrupt parties, which means that the parties have ports to communicate with the environment. Given this, in an interactive system such as a protocol, which does not contain the environment, these ports only involve one interactive agent (or, in other word, these ports are "open-ended"). Other open ports in a protocol are these that the parties use to communicate with the different functionalities.

Ports involving at least two interactive agents are known as *closed ports*, while ports involving only one interactive agent are called *open ports*.

Open/closed interactive systems. An interactive system with open ports is referred to as an *open interactive system*, and an interactive system that only has closed ports is known as a *closed interactive system*. For example, a protocol is an open interactive system, given that it has the open ports corresponding to the interaction between the environment and the parties, as well as between the parties and the different functionalities used in the real world.

Open interactive systems cannot in principle be run, as they may miss some data that should be written into the open ports. For example, a protocol cannot be run, given that it misses at least one functionality the parties can use for communication, and it also misses the inputs to be used have to be provided (by the environment) into the open ports (plus, the environment is also in charge of scheduling the execution itself). On the other hand, if we consider a larger interactive system consisting of the protocol (which is the set of parties), the different functionalities to be used, and the environment, now we obtain a closed interactive system. This system can be run, as the environment can now provide inputs to the parties, execute the protocol, obtain results (and in fact, it can do this multiple times).

Composition of interactive systems. Given two interactive systems $\mathcal{I}_1$ and $\mathcal{I}_2$, it is possible to obtain a bigger system from these two by considering the collection of all the interactive agents involved in these two sets, or, in other words, the union of the collections $\mathcal{I}_1$ and $\mathcal{I}_2$. This is denoted by $\mathcal{I} = \mathcal{I}_1 \diamondsuit \mathcal{I}_2$. For example, we considered above a closed system given by $\mathcal{Z} \diamondsuit \Pi \diamondsuit (\mathcal{F}_1 \diamondsuit \cdots \diamondsuit \mathcal{F}_\ell)$, where Π was the protocol under consideration and $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ the various functionalities used in the protocol execution.

1.2.3.1 Relevant Interactive Systems in the UC Framework

We already discussed an important interactive system, a protocol, which is simply a collection of parties. Now we consider the two main interactive systems in the UC framework: the real and ideal worlds. Recall that, in the real world, is where the actual execution of the given protocol takes place, while in the ideal world the parties make use of a trusted third party, modeled as a functionality, to compute the function securely. These ideas are easily formalized via the notion of an interactive system.

Real world. Intuitively, the real world is where the actual execution of the secure multiparty computation protocol at hand takes place. We formalize this via the following interactive system. Let $\Pi = \{P_1, \ldots, P_n\}$ be the protocol and let $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ be the functionalities to be used in the execution of the protocol. The *real world* is defined as the interactive system given by $\text{Real} := \Pi \diamondsuit (\mathcal{F}_1 \diamondsuit \cdots \diamondsuit \mathcal{F}_\ell)$. Notice that this is an open system, as it requires the environment to provide inputs and schedule the protocol execution.

Ideal world. At a high level we have considered the ideal world as where the parties send their inputs to a third trusted party, and receive outputs afterwards. This had to be refined to include a simulator $\mathcal{S}$, that acts as the interface between the adversary/environment, and the third trusted party.

Let $\mathcal{F}$ be the functionality that models the desired computation to be carried out securely (i.e. the third trusted party), and let $\mathcal{S}$ be a simulator. The *ideal world* is defined as the interactive system given by $\text{Ideal} := \mathcal{S} \diamondsuit \mathcal{F}$. Once again, this is an open system, and in fact it has the same open ports as the interactive system Real: the simulator contains open ports for the environment to connect, as if it were connecting to the corrupt parties in the real world, and the functionality $\mathcal{F}$ has open ports for the environment to provide input to and receive output from the honest parties. In particular, the interactive systems $\mathcal{Z} \diamondsuit \text{Real}$ and $\mathcal{Z} \diamondsuit \text{Ideal}$ are both closed.

1.2.3.2 Parameterized Interactive Systems

Finally, before we dive into the actual security definitions we will consider in this work, we remark that some interactive agents (and hence, interactive systems) can contain several external tweakable parameters. For example, a protocol typically allows for computation over different algebraic structure (e.g. say fields, but of different sizes), or a functionality

may be parameterizable according to the length of the messages it accepts, to cite some examples. These are all *external parameters*, meaning that they have to be set before considering an execution of these interactive agents. For illustration, they can be thought to be analogous to compile-time parameters in compiled programming languages.

A very important external parameter is the *security parameter*. Intuitively, this is a natural number that, as it grows larger, the protocol becomes "more secure". This will become clearer in Section 1.2.4 where we consider different notions of security. For now, it suffices to recall that, among all the different external parameters that the various interactive agents under consideration have, one we make explicit mention to is the security parameter, denoted by κ. To make this explicit we may sometimes write $\mathcal{I}(\kappa)$, where $\mathcal{I}$ is an interactive system/agent that is parameterized by κ.

1.2.4 Security Definition

Having defined the different interactive agents involved in our framework, we now turn out attention to defining security. As we have already mentioned, this will be achieved by requiring that no environment can distinguish between the real and ideal executions. In this section we approach in more detail the task of properly defining "indistinguishability".

We begin by introducing some minor preliminaries. First, we present the definition of a negligible function.

Definition 1.1 (Negligible functions)**.** *A function $\mu : \mathbb{N} \mapsto [0, \infty)$ is negligible if, for every $c \in \mathbb{N}$, there exists $\kappa_c \in \mathbb{N}$ such that, for every $\kappa \geq \kappa_c$, it holds that $\mu(\kappa) \leq \kappa^{-c}$. Alternatively, μ is negligible if, for every polynomial $p(X)$, there exists $\kappa_{p(X)} \in \mathbb{N}$ such that, for every $\kappa \geq \kappa_{p(X)}$, it holds that $\mu(\kappa) \leq p(\kappa)$.*

An example of a negligible function is $\mu(\kappa) = 2^{-\kappa}$. Intuitively, a negligible function is a function whose inverse, asymptotically, grows faster than any possible polynomial. These functions are widely used throughout cryptography to represent very small quantities.

The second consideration we must take care of before approaching our formal definitions, is that we include additional semantic notion to the environment. This interactive agent is in charge of distinguishing the real from the ideal execution, and it does so by interacting with either of these worlds, and outputting a bit,[8] that is, either 0 or 1, that represents which world the environment considers it is interacting with. As we will see, the assignments between these bits and the two worlds is irrelevant. Whenever the environment $\mathcal{Z}$ interacts with an interactive system $\mathcal{I}$, and produces output b, we denoted this by $b \leftarrow \mathcal{Z} \diamondsuit \mathcal{I}$. Notice that this is a random variable, given that the whole computation carried out by $\mathcal{Z}$ is potentially randomized.

Below we consider a setting in which a protocol Π is used to securely compute a functionality $\mathcal{F}$, while making use of the functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$. We remark that all the

[8] An interactive agent, being an enhance Turing machine, can produce output simply by writing it to a special tape and halting.

notions below are set with respect to a given adversarial structure, which, as discussed in Section 1.1, dictates the possible sets that can be corrupted.

1.2.4.1 Perfect Security

First we define the idea of perfect security, which reflects a protocol whose security cannot be broken even by unboundedly powerful environments/adversaries.

Definition 1.2 (Perfect security.)**.** *We say that a protocol Π securely instantiates a functionality $\mathcal{F}$ in the $(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$-hybrid model with **perfect security** if there exists a simulator $\mathcal{S}$ such that, for any environment $\mathcal{Z}$ and for every $\kappa \in \mathbb{N}$,*

$$\Pr[1 \leftarrow (\mathcal{Z} \diamondsuit \mathsf{Real})(\kappa)] = \Pr[1 \leftarrow (\mathcal{Z} \diamondsuit \mathsf{Ideal})(\kappa)],$$

where $\mathsf{Real} = \Pi \diamondsuit \mathcal{F}_1 \diamondsuit \cdots \diamondsuit \mathcal{F}_\ell$ *and* $\mathsf{Ideal} = \mathcal{S} \diamondsuit \mathcal{F}$.

Let us analyze the definition above in detail. First, perfectly secure protocols typically do not rely on the parameter κ, so we can remove it from the definition (it is included for the sake of maintaining certain "uniformity" in the notation with respect to the other notions of security described below). Now, the security definition above states that, there must exist a simulator $\mathcal{S}$ such that $\mathcal{Z}$ outputs 1 when interacting with the system Real with exactly the same probability that $\mathcal{Z}$ would output 1 when interacting with the system Ideal. This means precisely that $\mathcal{Z}$ cannot distinguish between the two worlds since, if it could, it could choose for example to output 1 only in the real world, while outputting 0 in the ideal world (so $\Pr[1 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Real}] = 1$ and $\Pr[1 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Ideal}] = 0$).

Notice that there is nothing special about the output 1. The same definition could have been considered with the output 0, given that $\Pr[0 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Real}] = 1 - \Pr[1 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Real}]$ and $\Pr[0 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Ideal}] = 1 - \Pr[1 \leftarrow \mathcal{Z} \diamondsuit \mathsf{Ideal}]$. This remark also holds for the other security notions below.

Finally, the quantity $|\Pr[1 \leftarrow (\mathcal{Z} \diamondsuit \mathsf{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z} \diamondsuit \mathsf{Ideal})(\kappa)]|$ is typically referred to as the *statistical advantage* of $\mathcal{Z}$, and it is essentially a measure of how well $\mathcal{Z}$ can distinguish between the real and ideal worlds. We see that, in the setting of perfect security, the advantage of any environment is 0.

1.2.4.2 Statistical Security

Now we consider a more flexible definition that allows certain small distinguishing advantage.

Definition 1.3 (Statistical security.)**.** *We say that a protocol Π securely instantiates a functionality $\mathcal{F}$ in the $(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$-hybrid model with **statistical security** if there exists a neg-*

ligible function $\mu(\kappa)$ *such that, for any environment* $\mathcal{Z}$,[9]

$$|\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Ideal})(\kappa)]| \leq \mu(\kappa),$$

where Real $= \Pi\diamondsuit\mathcal{F}_1\diamondsuit\cdots\diamondsuit\mathcal{F}_\ell$ *and* Ideal $= \mathcal{S}\diamondsuit\mathcal{F}$.

In this case, $\mathcal{Z}$ might be able to distinguish the two worlds "a little", which is reflected in the case that $\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Real})(\kappa)]$ and $\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Ideal})(\kappa)]$ may not be equal. In fact, it could be the case that, for some values of κ, the environment might distinguish the two worlds very well (for instance it can happen that $\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Real})(\kappa)] = 1$ and $\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Ideal})(\kappa)] = 0$ for some values of κ). However, the definition requires that, as κ grows, this distinguishing advantage shrinks at a good rate. For example, if $\mu(\kappa) = 2^{-\kappa}$, then choosing $\kappa = 1$ may be too bad since this means that the advantage that the environment has to distinguish between the two worlds is only $1/2$, but if $\kappa = 40$, then this is reduced to 2^{-40}, which is much more acceptable (in fact, 2^{-40} is a very common value to aim for when designing statistically secure protocols).

1.2.4.3 Computational Security

Finally, we consider the "weakest" of the security notions regarding secure multiparty computation protocols. In this case, the environment has a small distinguishing advantage, but this only holds if the environment is computationally bounded, meaning that it runs in polynomial time. In terms of practical meaning, this notion is good enough given that in an actual MPC deployment all parties involved will use a bounded amount of computational resources. Furthermore, as we will see in Section 1.3, some secure multiparty computation scenarios do not allow for any of the previous notions, and require computational security instead.

Definition 1.4 (Computational security.)**.** *We say that a protocol* Π *securely instantiates a functionality* $\mathcal{F}$ *in the* $(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$*-hybrid model with* ***computational security*** *if, for any* efficient *environment* $\mathcal{Z}$,[10] *there exists a negligible function* $\mu_{\mathcal{Z}}(\kappa)$ *such that*

$$|\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\text{Ideal})(\kappa)]| \leq \mu_{\mathcal{Z}}(\kappa),$$

where Real $= \Pi\diamondsuit\mathcal{F}_1\diamondsuit\cdots\diamondsuit\mathcal{F}_\ell$ *and* Ideal $= \mathcal{S}\diamondsuit\mathcal{F}$.

The first thing to notice with the definition above is that, unlike Definition 1.3, there is not a single negligible function $\mu(\kappa)$ that bounds the advantage of every possible environment $\mathcal{Z}$ when attempting to distinguish the real and the ideal worlds. This is not

[9] Here we must slightly limit the environment with respect to these from Definition 1.2, which did not have any limitation. In this case, we must assume that, although $\mathcal{Z}$ might be computationally unbounded, it only makes a polynomial (in κ) number of "calls" to either Real or Ideal. Otherwise, the notion cannot be achieved, since by interacting with one of the two worlds a super-polynomial number of times the distinguishing probability can be arbitrarily improved.

[10] An efficient environment is one that, at a high level, runs in polynomial time with respect to its parameters. However, there are several details that must be taken care of when properly defining this idea, given that the environment, and in general, any interactive agent, exchanges messages with other agents and can make "calls" to these. We refer the reader to [34] for details.

possible to achieve in general since there can be a series of environments $\mathcal{Z}_1, \mathcal{Z}_2, ...$ with each $\mathcal{Z}_c$ running in time κ^c (which is polynomial), so for a fixed κ_0 these environments have running times $\kappa_0^1, \kappa_0^2, ...$, which is unbounded. Eventually, with enough running time it would be possible to break the fixed bound on the advantage of $\mu(\kappa_0)$.

As a result, the best that can be hoped for is that, for every single environment $\mathcal{Z}$, its distinguishing advantage can be upper bounded by a negligible function $\mu_{\mathcal{Z}}(\kappa)$ that depends on this environment. A practical interpretation of this can be the following. After a careful analysis of the construction at hand, we consider the most efficient known attack on the protocol, derive an environment $\mathcal{Z}$ from this, determine the associated negligible function $\mu_{\mathcal{Z}}(\kappa)$ and choose κ so that the advantage of this environment in particular is below certain threshold (e.g. 2^{-80}). Given our observations above, it could be the case that this choice of κ is not sufficient to ensure a low distinguishing advantage for other environments, but at least it rules out the best one that is currently known.

We remark that, in this thesis, even though we consider settings in which only computational security is achievable, we only deal with perfect and statistical security in our actual security proofs. This is because, for the settings not admitting this type of security, we consider an offline/online paradigm that enables computation with perfect or statistical security with the help of certain functionalities.

1.2.5 The Composition Theorem

Consider a protocol $\Pi_{\mathcal{F}}$ that securely instantiates a functionality $\mathcal{F}$ with the help of some other functionality $\mathcal{R}$, that is, in the $\mathcal{R}$-hybrid model. In the real world, this functionality $\mathcal{R}$ acts as some kind of third trusted party that the parties can use to aid them in the task of securely computing $\mathcal{F}$; however, in practice, this functionality $\mathcal{R}$ must somehow be instantiated. For example, if $\mathcal{R}$ represents peer-to-peer encrypted and authenticated channels then a protocol like TLS must be executed to set these up. Formally, this would mean that a new protocol $\Pi_{\mathcal{R}}$ that instantiates $\mathcal{R}$, perhaps in some $\mathcal{T}$-hybrid model, is used, and a natural question is then the following: what types of formal security guarantees can the new protocol achieve? With the "new protocol", we mean protocol $\Pi_{\mathcal{F}}$ but replacing the interaction with the functionality $\mathcal{F}_{\mathrm{R}}$ by executions of the protocol $\Pi_{\mathcal{R}}$, which instantiates this functionality.

The core result of the UC framework, or universal-composability framework, is that, precisely, the resulting *composed* protocol inherits the properties of the two protocols involved, $\Pi_{\mathcal{F}}$ and $\Pi_{\mathcal{R}}$. In particular, this protocol still instantiates $\mathcal{F}$, but instead of doing it in the $\mathcal{R}$-hybrid model, it does it in the $\mathcal{T}$-hybrid one, which is the functionality needed by the protocol $\Pi_{\mathcal{R}}$. It will typically be the case that $\mathcal{T}$ is much simpler than $\mathcal{R}$, which implies that progress has been achieved towards instantiating $\mathcal{F}$ securely.

Before we discuss the theorem in detail, we discuss some of the consequences of the above high-level description of the result. First, the composition theorem enables a modular description of highly complex protocols by breaking them into pieces and then proving the security of each fragment separately, an approach that we make extensive use of throughout this thesis. For example, in a large and complex protocol Π it might

be the case that certain piece or pattern is repeated in several places of the protocol execution. This part can be isolated as a protocol Π' on its own, instantiating certain functionality $\mathcal{R}$, and the protocol Π could possibly be expressed in a much simpler way in terms of this functionality. Now, to prove security, we do not need to provide a proof of the big "monolithic" protocol Π, but rather, we can prove that its simpler variant, which is set in the $\mathcal{R}$-hybrid model, instantiates the desired functionality, and then we can focus only in proving that the protocol Π' indeed instantiates $\mathcal{R}$. For illustration purposes, it is useful to think of the approach above as splitting complex functions in a programming language into simpler constructions that make calls to other functions. As we will see in Part II of this thesis, this approach enables clear and modular proofs and protocol descriptions, and, in the author's opinion, is one of the key factors that has enabled such a rich and fruitful body of research in the area of secure multiparty computation.

The composition theorem is not only useful as a pedagogic tool. In practice, secure multiparty computation protocols are deployed in large and complex distributed systems that are possibly running, concurrently, many other protocols to achieve other tasks. For example, keys must be negotiated, random values must be sampled, inputs must be provided, etc. The composition theorem ensures that, even if several protocols are executed simultaneously, as long as each of them can be proven secure, then the resulting group of protocols is also secure. This is a crucial observation that favors the UC framework with respect to other formal models, such as the stand-alone one, that does not accept such flexible concurrent composition.

Composing protocols. In order to properly state the composition theorem, it is important to clearly and explicit define the different interactive agents and systems involved.

Consider a protocol $\Pi_{\mathcal{F}} = \{P_1, \dots, P_n\}$ that instantiates a functionality $\mathcal{F}$ in the $\mathcal{R}$-hybrid model, and consider another protocol $\Pi_{\mathcal{R}}$ which has different parties $\{Q_1, \dots, Q_n\}$[11] and instantiates the functionality $\mathcal{R}$ in the $\mathcal{T}$-hybrid model. Composing the protocols $\Pi_{\mathcal{F}}$ and $\Pi_{\mathcal{R}}$ amounts to simply composing them as interactive systems, which is denoted by $\Pi_{\mathcal{F}} \diamondsuit \Pi_{\mathcal{R}}$.

To make more sense of this notion, recall that the parties $Q_1, \dots, Q_n$ have ports to communicate with the environment, while the parties $P_1, \dots, P_n$ have ports to communicate with the functionality $\mathcal{R}$. These ports are one and the same: messages sent from P_i to $\mathcal{R}$ are received by Q_i as coming from the environment, and similarly in the opposite direction. This way, the interactive system $\{P_i, Q_i\}$ acts as one single party, interacting with the environment (through P_i's ports) and also with the functionality $\mathcal{T}$ (through Q_i's ports). With this new interpretation, we see that the composition of two "compatible" protocols is again a protocol, where the new parties might be interactive systems that behave just like an interactive agent. For more details we refer the reader to Section 4.2.7 in [34].

[11] Recall that, formally, a party carries the "code" of the protocol that is executed, so different protocols involve different parties.

The main theorem. With the notation above at hand, we are ready to properly state the composition theorem, which is fully proven in [34].

Theorem 1.1 (Composition Theorem, Thm 4.20 in [34])**.** *Let $\Pi_{\mathcal{F}}$ be a protocol instantiating a functionality $\mathcal{F}$ in the $\mathcal{R}$-hybrid model with perfect/statistical/computational security, and let $\Pi_{\mathcal{R}}$ be a protocol instantiating $\mathcal{R}$ in the $\mathcal{T}$-hybrid model with the same type of security. Then, the composed protocol $\Pi_{\mathcal{F}} \diamondsuit \Pi_{\mathcal{R}}$ securely instantiates the functionality $\mathcal{F}$ in the $\mathcal{T}$-hybrid model, with the same type of security.*

1.2.6 Some Basic Functionalities

We end this chapter with a description of some functionalities we use throughout this thesis. This includes the basic communication resource that the parties use to interact with each other, and the functionality used to model the task of general purpose secure computation, with a variant to account for security with abort.

1.2.6.1 Underlying Communication Resource

As a starting point, we assume that the parties communicate through the following functionality.

Functionality $\mathcal{F}_{\text{P2P+BC}}$

The functionality proceeds as follows:

- On input (message, j, m) from party P_i, send (message, i, m) to P_j.
- On input (broadcast, m) from party P_i, send (broadcast, i, m) to all parties.

The functionality above models a *peer-to-peer encrypted and authenticated network* in which the parties can send messages to each other confidentially, and the adversary cannot modified their contents when the sender is not an actively corrupt party. In addition to this, in includes a *broadcast channel*, in which a sender with a given message can distribute this data to the other parties in such a way that all parties are guaranteed to receive the exact same value.

All of our protocols assume $\mathcal{F}_{\text{P2P+BC}}$ as a basis, so we do not write that a given instantiation is "in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model". Only in Section 1.3, where we present several fundamental results, we sometimes consider a functionality $\mathcal{F}_{\text{P2P}}$ that does not include the broadcast channel.

1.2.6.2 Arithmetic Black Box Model

Recall that, in general-purpose secure multiparty computation, our aim is to securely compute any possible function, written as an arithmetic circuit over certain algebraic

structure. In this work, we model such computations in two different ways. First, we consider standard arithmetic circuits as defined in Section 1.1.4.3. These a directed acyclic graphs with input, operation (addition and multiplication) and output gates, and they are better suited for modeling non-reactive computation. Naturally, such type of computation can be easily described as a functionality that receives inputs from the parties, computes the give arithmetic circuit, and returns the output.

In some other places, however, we consider a more flexible functionality that is better suited for reactive computation, which, as defined in Section 1.1.4.2, enables the parties to obtain partial results, learn them, and continue the computation possibly depending on these values. The model we make use of to represent such behavior is the *arithmetic black box* model. At a high level, this allows the parties to access a "storage box" that can keep values sent by the parties, but it also allows for additions and multiplications to be carried out on stored values, saving the results. Finally, it enables the parties to read any stored value at any time, which effectively models the setting of reactive computation. The formal functionality is described below.

Functionality $\mathcal{F}_{\text{ABB}}$: Arithmetic Black Box

The functionality proceeds as follows.

- On input $(\text{input}, \text{id}, i)$ from the honest parties, send $(\text{input}, \text{id}, i)$ to the adversary, wait for input $(\text{value}, \text{id}, x)$ from party P_i, where $x \in \mathbb{Z}/2^k\mathbb{Z}$, and then store (id, x) in memory.
- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$ from the honest parties, retrieve (id_i, x_i) for $i \in [\ell]$ from memory and store $(\text{id}_{\ell+1}, z)$, where $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2^k$. Then send $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$ to the adversary.
- On input $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ from the honest parties, retrieve (id_1, x) and (id_2, y) from memory and store (id_3, z), where $z = x \cdot y$. Then send $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ to the adversary.
- On input (open, id) from the honest parties, retrieve (id, x) from memory and send x to all the parties. Then send (open, id) to the adversary.

Formalizing security with abort. We have discussed in Section 1.1 three different notions regarding the guarantees the honest parties have with respect to the output of the computation: guaranteed output delivery, where all parties will receive output, fairness, where honest parties receive output if the corrupt parties do so as well, and security with abort, where the adversary may cause the honest parties to abort, perhaps not obtaining any output, while the adversary may be able to learn the result nevertheless.

In this thesis *we focus solely on security with abort* (except in Section 3.4 where we discuss the general idea for obtaining guaranteed output delivery in the $t < n/3$ setting). We capture this in the UC framework by endowing all functionalities with the following behavior: at any point of the execution, the adversary can input a special signal abort to the given functionality, which sends abort to all honest parties. Upon receiving such message, each honest party immediately produces abort as output, and halts.

In the real world, whenever we say that "the parties abort", it means that they produce

abort as output, and halt. In some cases, the event triggering an abort is only seen by one party (e.g. some party receives incorrect data). When we say that "party P_i aborts", we implicitly assume that this party sends abort through the broadcast channel, so all parties abort too.[12]

1.3 Fundamental Results

We now proceed to presenting some of the most fundamental results in the theory of secure multiparty computation, regarding the feasibility or impossibility of MPC in certain contexts. Below, we consider different results in the context in which the adversarial structure is a threshold structure with threshold t, categorized by whether $t < n/3$, $t < n/2$ or $t < n$. As we will comment in each relevant section, most of the results for $t < n/2$ and $t < n/3$ carry over to the case of $Q2$ and $Q3$ general adversarial structures, respectively.

1.3.1 Results for $t < n/3$

The context in which the adversary corrupts at most one third of the parties is particularly relevant as it allows for the strongest level of simulation-based security, namely, perfect security.

Positive results. We begin with the following crucial result, which shows that the most desired notions of perfect security and guaranteed output delivery can be achieved if $t < n/3$, even if the adversary is active.

Theorem 1.2. *There exists a protocol instantiating $\mathcal{F}_{\text{ABB}}$ with perfect security and guaranteed output delivery in the $\mathcal{F}_{\text{P2P}}$-hybrid model,[13] secure against an active adversary corrupting at most $t < n/3$ of the parties.*

The proof of this result can be found for example in [17,29]. We present a protocol of this kind in Section 2.4, except it does not achieve guaranteed output delivery.

Remark 1.2. *Notice that, in Theorem 1.2, the basic communication resource is $\mathcal{F}_{\text{P2P}}$, which represents encrypted and authenticated peer-to-peer communication,* without a broadcast channel. *Most constructions will still make use of a broadcast channel, but this is possible to construct from plain peer-to-peer channels with perfect security if $t < n/3$, as shown in [75].*

[12] This is the crucial difference between *selective abort* and *unanimous abort*. In the former, it can happen that only some honest parties abort while the others remain in the computation. Through a broadcast channel, as shown in [56] and as used here, we can ensure unanimous abort by asking aborting parties to announce their status through the broadcast channel.

[13] As mentioned in Section 1.2.6.1, our security statements later in the thesis are all set in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model and we do not write this explicitly.

Negative results. An interesting fact is that, if the adversary breaks the $t < n/3$ condition, that is, if the adversary corrupts more than one third of the parties, then Theorem 1.2 does not hold anymore. More precisely, the "perfect security" part of the theorem cannot be fulfilled, which is summarized in the following theorem.

Theorem 1.3. *No protocol can instantiate $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{P2P}}$-hybrid model against an active adversary corrupting at least $n/3$ parties with perfect security.*

To show that this theorem holds, it suffices to exhibit a particular function that cannot be instantiated with perfect security in the $\mathcal{F}_{\text{P2P}}$-hybrid model, if an active adversary corrupts at least $n/3$ parties. This invalidates the possibility of $\mathcal{F}_{\text{ABB}}$ being instantiable, given that $\mathcal{F}_{\text{ABB}}$ can be used to trivially instantiate any other functionality. There are several functions that, if $t \geq n/3$, cannot be securely computed in the $\mathcal{F}_{\text{P2P}}$-hybrid model. A typical example being the broadcast functionality. This result can be found in [75].

Finally, although there are some functions such as broadcast that cannot be instantiated with perfect security in the $\mathcal{F}_{\text{P2P}}$-hybrid model if $t \geq n/3$, it is natural to ask whether, even if we add broadcast as a basis, that is, if we work in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model, there are still some functions that cannot be instantiated with perfect security if $t \geq n/3$. This turns out to be the case, as shown for example in [34] (see Theorem 5.12 in the reference).

Theorem 1.4. *No protocol can instantiate $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model against an active adversary corrupting at least $n/3$ parties with perfect security.*

1.3.2 Results for $t < n/2$

Now we turn our attention to the setting in which the adversary corrupts at most one half of the parties. In this setting, although general-purpose secure computation with perfect security against an active adversary is not possible (as illustrated in Theorem 1.4), several other properties are still attainable.

1.3.2.1 The Case of a Passive Adversary

Positive results. First we discuss what happens when the adversary is passive. In this case, it turns out that a protocol with perfect security can be designed, as expressed by the following theorem.

Theorem 1.5. *There exists a protocol instantiating $\mathcal{F}_{\text{ABB}}$ with perfect security in the $\mathcal{F}_{\text{P2P}}$-hybrid model, secure against a passive adversary corrupting at most $t < n/2$ of the parties.*

Notice that this theorem is similar to Theorem 1.2, except that this time the adversary is passive and the corruption threshold is at most $n/2$, instead of being upper bounded by

$n/3$. Protocols that illustrate the validity of Theorem 1.5 are presented in [17, 29], and we include one of such constructions in Section 2.3.

Negative results. A protocol with perfect security to compute arbitrary functionalities cannot exist if the condition $t < n/2$ is broken. In fact, such protocol cannot exist even if we loosen the security notion to statistical security. This is shown in the following theorem.

Theorem 1.6. *No protocol can instantiate $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{P2P}}$-hybrid model against a passive adversary corrupting at least $n/2$ parties with statistical security.*

A proof of this result can be found, for example, in [72].

1.3.2.2 The Case of an Active Adversary

Positive results. Now we focus on the case in which the adversary corrupts a subset of the parties actively. As shown in Theorem 1.3, in the case in which $t < n/2$, given that in principle it could hold that $t \geq n/3$, it is not possible to instantiate $\mathcal{F}_{\text{ABB}}$ with perfect security in the $\mathcal{F}_{\text{P2P}}$-hybrid model (or even in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model from Theorem 1.4). However, it turns out that, if we relax the security requirement to statistical security rather than perfect security, an instantiation can be realized. Furthermore, the strongest output notion of guaranteed output delivery can be attained. This is summarized below.

Theorem 1.7. *There exists a protocol instantiating $\mathcal{F}_{\text{ABB}}$ with statistical security and guaranteed output delivery in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model, secure against an active adversary corrupting at most $t < n/2$ of the parties.*

Protocols proving this theorem include [72], or the more recent results of [57] which improve over the communication complexity of the previous ones.

Negative results. Theorem 1.6 shows that, if the bound $t < n/2$ is violated, then no protocol can instantiate $\mathcal{F}_{\text{ABB}}$ with statistical security in the $\mathcal{F}_{\text{P2P}}$-hybrid model, even if the adversary is assumed to be passive. In particular, this impossibility extends ("with even more reason") if the adversary is active.

On the other hand, another natural question is whether the strong notion of guaranteed output delivery, achievable if $t < n/2$, is still attainable if $t \geq n/2$. There is a negative answer to this question, and in fact, not even the weaker notion of fairness can be realized if $t \geq n/2$. This is captured in the following theorem.

Theorem 1.8. *No protocol can achieve fairness when instantiating $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{P2P+BC}}$-hybrid model against an active adversary corrupting at least $n/2$ parties.*

A proof of this result can be found in [31].

1.3.3 Positive Results for $t < n$

Finally, we discuss what results are possible in the most general case in which the adversary can, in principle, corrupt all but one party, in contrast to the previous settings, in which the adversary was assumed to corrupt at most a $1/2$ or even a $1/3$ proportion of the parties. In terms of negative results, we can infer from Theorem 1.6 that information-theoretic security (that is, either perfect or statistical security) is out of the picture in this case, and from Theorem 1.8 we rule out the possibility of achieving fairness. As a result, protocols in this setting must make use of computational assumptions (even if the adversary is passive), or, in other words, they must involve cryptographic constructions whose security depends on the hardness of certain underlying problem, and they have to settle for security with abort.

Fortunately, in terms of positive results, it can be shown that protocols with the properties described above indeed exist. In this case the instantiation can be done over $\mathcal{F}_{\text{P2P}}$ rather than $\mathcal{F}_{\text{P2P+BC}}$ since it is possible to obtain broadcast by making use of computational assumptions.

Theorem 1.9. *There exists a protocol instantiating $\mathcal{F}_{\text{ABB}}$ with computational security in the $\mathcal{F}_{\text{P2P}}$-hybrid model, secure against a passive adversary corrupting possibly all but one of the parties.*

Several protocols of this type have been achieved in the literature, such as [19,41,43,62,63]. In Sections 2.6 and 2.7 we include constructions in this setting with passive and active security, respectively. These constructions satisfy perfect and statistical security, which contradicts the impossibility results discussed above. This is because, as we will see in the relevant sections, these protocols are not set in the $\mathcal{F}_{\text{P2P}}$-hybrid model, but instead, they are built making use of a stronger functionality that distributes certain preprocessed data among the parties.

1.3.4 Summary of Main Results

The following table summarizes the results we have seen so far in the section. A check mark (✓) represents that a construction in the given setting can be obtained, while an X mark (✗) indicates that it is not possible in general to instantiate $\mathcal{F}_{\text{ABB}}$ in the scenario under consideration. Additionally, marks in black indicate results that can be trivially derived from the marks in red, and the numbers in parentheses after the latter type of marks represent the number of the theorem in previous sections associated to that result. Finally, this table omits certain details regarding the possibility/impossibility of broadcast, or more precisely, when $\mathcal{F}_{\text{P2P}}$ or $\mathcal{F}_{\text{P2P+BC}}$ is needed, in the $t < n/2$ row containing the asterisk ($\star$), so we refer the reader to the relevant section above for details.

		Privacy guarantees			Output guarantees		
		perf.	stat.	comp.	GOD	fair	abort
Active	$t < n$	✗	✗	✓ (1.9)	✗	✗ (1.8)	✓ (1.9)
	$t < n/2$ *	✗ (1.6)	✓ (1.7)	✓	✓ (1.7)	✓	✓
	$t < n/3$	✓ (1.2)	✓	✓	✓	✓	✓
Passive	$t < n$	✗	✗ (1.6)	✓	✓	✓	✓
	$t < n/2$	✓ (1.5)	✓	✓	✓	✓	✓
	$t < n/3$	✓	✓	✓	✓	✓	✓

Chapter 2

Some Essential MPC Constructions

The goal of this chapter is to introduce the reader to several essential techniques used in secure multiparty computation *over fields*. This serves two main purposes. First, some of these techniques have a close resemblance to the ones we present in Part II of this thesis, where we discuss secure multiparty computation over rings of the form $\mathbb{Z}/2^k\mathbb{Z}$. This way, the reader will be able to identify the core techniques that, as a result of the contributions of this thesis, enable computation over this domain. Second, this chapter works as a short yet comprehensive resource for readers interested in learning some of the core techniques in (secret-sharing-based) MPC. Typically, the only sources for this type of information are the original papers where these techniques are introduced, which can be overwhelming to read in some cases given their necessary formalism and targeted audience. Furthermore, in some cases, some techniques are the result of a series of works rather than a single reference, and this chapter, having appropriate references to the techniques presented, also serves as a starting point when getting into these topics.

Given that the focus of this section is simply to provide an overview of major existing techniques for secure multiparty computation over fields, we omit formal proofs in the simulation based model from Section 1.2, and content ourselves with including more intuitive and simple arguments about why the different techniques and protocols satisfy the different properties they intend to. In Part II, where the actual contributions of this thesis are presented, formal simulation-based proofs will be presented.

This chapter is organized as follows. First, we discuss in Section 2.1 a general "template" to design secure multiparty computation protocols, which constitutes the type of constructions we focus on in this work. Followed by this, in Section 2.2 we present Shamir secret-sharing, a popular and very useful secret-sharing scheme that underlies a wide range of constructions in the literature. Then in Section 2.3 we present a passively secure protocol with perfect security tolerating $t < n/2$ corruptions, and in Section 2.4 we extend it to active and still perfect security, but now tolerating $t < n/3$ corruptions. In Section 2.5 we improve this to $t < n/2$ corruptions in detriment of achieving statistical security only. All of these protocols make use of Shamir secret-sharing as explained in section 2.2. Finally, in Sections 2.6 and 2.7 we discuss protocols in the dishonest majority setting (that is, $t < n$) with passive and active security respectively. These protocols require computational assumptions that heavily increase protocol complexity, meaning that not only the resulting protocols tend to be more inefficient than the ones using Shamir secret-sharing, but a full description of these protocols is much more complicated. We avoid this complication by focusing on secure multiparty computation in the

preprocessing model, where we assume certain data among the parties is pre-distributed by a trusted party. Details are given in the corresponding section.

2.1 Secret-Sharing-Based MPC

Essentially all existing approaches to secure multiparty computation in the literature begin by representing the function to be computed as an arithmetic circuit, which were described in Section 1.1.4.3. However, once this circuit can be established, the specific method used to securely evaluate such circuit tends to vary from work to work. In spite of this, it is still possible to identify some general patterns, and, although a few works may not properly fall within any of the categories below, these are inclusive enough to fit a large portion of the literature in the field of (general-purpose) secure multiparty computation.

First, some constructions make use of *homomorphic encryption* techniques to homomorphically evaluate the given circuit (or at least certain portions of it), typically without involving a lot of interaction. These techniques are mostly theoretical (at least these involving large encrypted computations) as the overhead in terms of computational complexity is typically too large. However, as the field of fully homomorphic encryption progresses, this approach becomes more and more practical and, as a result, it may eventually turn into a more practical solution for large and complex computation, at least if used in a partial and clever way (that is, instead of simply evaluating the entire function in one go using homomorphic encryption).

A different approach consists of somehow obtaining a "hidden" version of the circuit that then can be evaluated only on the set of inputs provided by the parties. This turns out to be the approach initially proposed by Yao [80] when the concept of secure multiparty computation was itself born, and a rich and extensive body of works has taken care of enhancing and improving this method, which is known as *garbled circuits*. This technique has several benefits in terms of efficiency as it is typically the case that, after the circuit has been "hidden" (or *garbled*), which requires interaction in a constant number of rounds, the evaluation of the circuit itself can happen with little to none communication. This makes this technique ideal for settings in which the parties are widely distributed and latency is high, so minimizing round trips becomes relevant. Unfortunately, a big downside of the garbled circuits approach is that the process of garbling the circuit, even though happens in a constant number of rounds, tends to involve a large amount of data, which ends up in consuming a lot of bandwidth, up to the extent that for certain applications this becomes a serious bottleneck. Furthermore, the garbled circuit technique is generally better suited for binary circuits (that is, circuits defined over $\mathbb{F}_2 = \{0, 1\}$), with a handful of (mostly theoretical) constructions considering more general arithmetic circuits (e.g. [8, 13]).

The alternative approach to securely evaluate an arithmetic circuit, which constitutes the focus of this thesis, is based on a tool called *secret-sharing*. In a secret-sharing scheme (a concept that we define more precisely in Section 2.1.1 below), a given value can be distributed among several parties so that each of these participants now holds a "share", which satisfy the following: certain sets of shares do not leak anything about

the underlying value, while some other sets of shares completely determine it. In secret-sharing-based MPC the goal is then to obtain a secret-shared representation of the inputs to the computation, using a secret-sharing scheme that ensures that any possible set in the adversarial structure (that is, any set of parties that could be corrupted) cannot learn anything about the underlying secret. This is followed by methods to obtain a secret-shared representation of all intermediate values in the computation, until the output is reached.

We discuss the secret-sharing-based MPC approach in more detail in the sections below. Before we dive into it we remark, however, that there are several works in the literature that, although they make use of secret-sharing techniques, they do not adhere exactly to the template provided here. For example, it is very common to mix secret-sharing with garbled circuits (e.g. [45]), or even with homomorphic encryption techniques.

2.1.1 Linear Secret-Sharing Schemes

We begin our discussion on secret-sharing-based secure multiparty computation protocols, which is the general template to which all of the constructions considered in this thesis adhere, by first discussing the concept of a secret-sharing scheme itself.

For the purpose of this section we consider a finite field $\mathbb{F}$. Furthermore, we restrict to *threshold* secret-sharing schemes, which only protect the secret if less than certain amount of shares is known, and completely leak the value otherwise. Finally, we also remark that our description here is entirely informal (plus it makes several simplifications) and does not constitute in any way a formal nor precise treatment of linear secret-sharing schemes (and, in fact, such treatment is not necessary for the contributions of this thesis in Part II). For a more concrete mathematical presentation on these tools we refer the reader to [34].

Let $s \in \mathbb{F}$. At an intuitive level, a secret-sharing scheme for n parties with threshold t provides methods for, on input s, computing a set of values $(s_1, \dots, s_n) \in \mathbb{F}^n$ such that

1. For any set $A \subseteq [n]$ with $|A| \leq t$, the set of shares $\{s_i\}_{i \in A}$ does not leak anything about the value s;

2. For any set $B \subseteq [n]$ with $|B| \geq t + 1$, the value s can be completely reconstructed from the set of shares $\{s_i\}_{i \in B}$.

When a value $s \in \mathbb{F}$ is secret-shared as above, it is common to denote this by $[\![s]\!] := (s_1, \dots, s_n)$. In a distributed setting with n parties it is typically assumed implicitly in the notation above that each party P_i has the share i, for $i \in [n]$.

A secret-sharing scheme as above is *linear* if, in words, each party can locally add/subtract their shares of different values to obtain shares of the corresponding operation on the secrets. A bit more precisely, it must hold that, if $[\![x]\!] = (x_1, \dots, x_n)$ and $[\![y]\!] = (y_1, \dots, y_n)$, then $[\![x \pm y]\!] = (x_1 \pm y_1, \dots, x_n \pm y_n)$. Now we discuss some simple examples of linear secret-sharing schemes

Example 2.1 (Additive secret-sharing)**.** *The following is a construction of a linear secret-sharing scheme for n parties with threshold $n-1$, which means that every set of at least $(n-1)+1=n$ shares can reconstruct the secret while any smaller set remains oblivious to this value. Notice that there is only one possible set of n shares, which is the set of all the shares.*

To secret-share a value $s \in \mathbb{F}$, a tuple $(s_1, \ldots, s_n) \in \mathbb{F}^n$ is sampled uniformly at random constrained to $s_1 + \cdots + s_n = s$. This can be done, for example, by sampling $n-1$ random values $s_1, \ldots, s_{n-1}$ and defining $s_n = s - (s_1 + \cdots + s_{n-1})$. More generally, any set of $n-1$ shares can be sampled uniformly at random while the last one is defined as the secret subtracted with the sum of the other shares. The set of shares is then $(s_1, \ldots, s_n)$.

To analyze the required properties by a linear secret-sharing scheme, we observe the following:

- *Any set of at most $n-1$ shares follows the uniform distribution, so in particular it does not reveal anything about the secret s.*
- *Given all the shares $s_1, \ldots, s_n$, the secret s can be fully determined as $s_1 + \cdots + s_n = s$. This, together with the point above, shows that this construction is a secret-sharing scheme.*
- *Given two shared values $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$, that is, $x = x_1 + \cdots + x_n$ and $y = y_1 + \cdots + y_n$, since $x \pm y = (x_1 \pm y_1) + \cdots + (x_n \pm y_n)$, it holds that $[\![x \pm y]\!] = (x_1 \pm y_1, \ldots, x_n \pm y_n)$. This shows that the proposed method constitutes a linear secret-sharing scheme.*

Example 2.2 (Replicated secret-sharing [59])**.** *The following is a construction of a linear secret-sharing scheme for n parties with a more general threshold $t < n$. To secret-share a value $s \in \mathbb{F}$, first a set of values $\{s_A\}_{A \subseteq [n], |A|=t} \subseteq \mathbb{F}$ is sampled uniformly at random, constrained to $\sum_{A \subseteq [n], |A|=t} s_A = s$. Each share s_i for $i \in [n]$ is defined to be a vector itself, which is given by $\mathbf{s}_i = (s_A)_{A \subseteq [n], i \notin A}$.*

Now we analyze the required properties by a linear secret-sharing scheme.

- *Given any set $B \subseteq [n]$ with $|B| \leq t$, the collection of shares $\{\mathbf{s}_i\}_{i \in B}$ miss the value $s_{B'}$ for any $B \subseteq B' \subseteq [n]$. As a result, these shares together do not have enough information to reconstruct s, since all of the "additive values" $\{s_A\}_{A \subseteq [n], |A|=t}$ are needed to do so.*
- *Given any set $B \subseteq [n]$ with $|B| \geq t+1$, the collection of shares $\{\mathbf{s}_i\}_{i \in B}$ contains all the summands $\{s_A\}_{A \subseteq [n], |A|=t}$, which enables the reconstruction of the secret s. To see this, let $A \subseteq [n]$ with $|A| = t$. The summand s_A is included in all $\mathbf{s}_i$ for $i \notin A$, and since $|A| = t < t+1 \leq |B|$, there is at least one such indexes i in the set B.*
- *The fact that the construction above constitutes a* linear *secret-sharing scheme is straightforward to see.*

2.1.2 MPC based on Linear Secret-Sharing Schemes

Consider a linear secret-sharing scheme $[\![\cdot]\!]$. As we know, thanks to the linearity properties of $[\![\cdot]\!]$, it is possible for the parties to obtain $[\![x+y]\!]$ given two shared values $[\![x]\!]$ and $[\![y]\!]$. Now, suppose that the parties count on a method to obtain, not only the addition (and subtraction), but also the product of two shared values. More precisely, suppose the parties can obtain $[\![x \cdot y]\!]$ from two shared values $[\![x]\!]$ and $[\![y]\!]$, possibly by performing some interaction. With this at hand, the parties can easily compute the given arithmetic circuit by following this procedure:

1. Each party P_i, having input x_i, distributes shares of this value to the other parties, so the parties obtain $[\![x_i]\!]$.

2. For each operation gate in the circuit where the inputs x and y are secret-shared as $[\![x]\!]$ and $[\![y]\!]$, the parties proceed as follows:

   - If the gate is an *addition gate*, then the parties use the linear property of the secret-sharing scheme to obtain $[\![x+y]\!]$ without any interaction.

   - If the gate is an *multiplication gate*, then the parties use the assumed method to obtain $[\![x \cdot y]\!]$, potentially with interaction.

   - Eventually, the parties get shares of the result of the computation $[\![z]\!]$. At this point, $t+1$ of the parties announce their shares to all the others so that the parties, having at least $t+1$ shares, can reconstruct the output z.

From the template above, we see that, to design a secure multiparty computation protocol, it suffices to consider a linear secret-sharing scheme with the same threshold as the upper bound on corrupted parties, together with a method to obtain $[\![x \cdot y]\!]$ from $[\![x]\!]$ and $[\![y]\!]$. As we will see throughout this thesis, this general template is highly effective for building efficient protocols in a wide variety of settings, and, most of the time, the major complications appear not in the secret-sharing scheme itself, but in the procedure to multiply secret-shared values.

2.1.2.1 The Case of an Active Adversary

The general template above works well if the adversary is passive, but, when the corruptions are active, care must be taken in some parts of the protocol. First, naturally, the assumed method to securely compute multiplications must be actively secure, since otherwise an active adversary can attack the whole protocol by simply attacking multiplication gates. However, the phase where shares of the inputs are distributed must also be revisited, since, as we will see in subsequent sections, there are several secret-sharing schemes where, if the party distributing shares behaves maliciously, sending perhaps "incorrect" shares, the protocol can be rendered insecure due to "inconsistencies" created among the parties.

To prevent actively corrupt parties from secret-sharing their inputs incorrectly, a typical

approach consists of somehow reducing this to the broadcast channel by using a random shared value $[\![r]\!]$, where the secret t is known by the party providing input. If the input is x, this party simply needs to use the broadcast channel to announce $e = x - r$, which leaks nothing about x since r is uniformly random and only known to the party sending the message, and then the other parties can locally compute $[\![x]\!] = [\![r]\!] + e$, which leads to shares of the input x since $r + e = x$. This method has the advantage that, assuming that $[\![r]\!]$ was distributed "correctly", the resulting shares of x will also be correct.

Finally, another place where the adversary can attack the protocol is in the last step, where the shares of certain parties are announced in order to reconstruct the result of the computation. In this case, an actively corrupt party can simply lie about his own share, and it is not clear what would happen in such scenario. Indeed, this is a problem we will need to deal with throughout this thesis, and such attack, unless countered, tends to lead to the parties reconstructing a wrong result.

This type of behavior is addressed by enhancing the secret-sharing scheme with some method to check that the announced shares are somehow "correct", and have not been modified. In some settings, especially the $t < n/2$ and $t < n/3$ scenarios, this can be achieved without modifying the sharing procedure itself, while in some other cases, like in dishonest majority, some additional information that aids at ensuring integrity must be added. Details on all these problems and different approaches to solve them in various settings are given throughout the thesis, although the first relevant sections where such techniques are encountered are Sections 2.4, 2.5 and 2.7.

2.1.2.2 Offline-Online Paradigm

In several cases, part of the interaction involved during the execution of a given secure multiparty computation protocols is independent of the inputs from the parties. For example, when we discussed in the previous section the general idea to obtain secret-shared inputs correctly, we made use of a secret-shared value $[\![r]\!]$ where r is known by the input provider. This type of data has to be produced by certain interaction in the MPC protocol, but it is independent of any of the inputs to the computation.

It is common in the field to refer to the steps in a given protocol that *do not* depend on the inputs to the computation as the *preprocessing* or *offline* phase, while the part of the protocol that requires the parties to know their inputs is typically called the *online* phase. The main motivation behind this terminology is that the preprocessing phase, being independent of the inputs, can be executed at the very start of the protocol execution, and after it is over the parties become ready to provide inputs and "actually compute" the function.

The distinction between an offline and an online phase is not only relevant at the language level. Consider two protocols having roughly the same performance overall, except that one has a very efficient online phase when compared to the other. Even though both protocols perform similarly, the total latency since the moment the inputs are provided until the output is obtained is smaller in the protocol with a fast online phase. Imagine a setting where the parties are idle before running the computation, which is scheduled

in advance. This time can be then used to execute the offline phase, so that the parties are ready to run the efficient online phase when the inputs are known.

As we will see in Sections 2.6 and 2.7, and also in some of the constructions from Part II in this thesis, this offline/online paradigm takes even more relevance in the dishonest majority, since in this case, as we saw in Section 1.3, protocols must make use of heavy cryptographic tools to operate. Modern constructions, such as the ones we consider here, push all the complexities and inefficiencies of these mechanisms to the offline phase, while leaving a relatively simpler and much more efficient online phase (which, in addition, typically enjoys information-theoretic security).

2.2 Shamir Secret-Sharing

We begin by presenting the construction and properties of a very popular and widely used secret-sharing scheme, namely Shamir secret-sharing scheme. This was proposed by Adi Shamir in [74], and it is one of the most widely known and used examples of a linear secret-sharing scheme over a field.

Let $\mathbb{F}$ be a field of size q, where q is a power of a prime. Assume that $q > n$, and let $\alpha_0, \alpha_1, \dots, \alpha_n \in \mathbb{F}$ be different points in $\mathbb{F}$. We denote by $\mathbb{F}_{\leq d}[\mathtt{X}]$ the $\mathbb{F}$-module of univariate polynomials over $\mathbb{F}$ of degree at most d in the variable $\mathtt{X}$. Let $\mathbb{F}^{u\times v}$ denote the set of matrices with dimensions $u \times v$.

Given $\beta_1, \dots, \beta_u \in \mathbb{F}$, let $\mathrm{Van}^{u\times v}(\beta_1, \dots, \beta_u) \in \mathbb{F}^{u\times v}$ be the matrix given by

$$\mathrm{Van}^{u\times v}(\beta_1, \dots, \beta_u) \in \mathbb{F}^{u\times v} := \begin{pmatrix} 1 & \beta_1^1 & \beta_1^2 & \cdots & \beta_1^{v-1} \\ 1 & \beta_2^1 & \beta_2^2 & \cdots & \beta_2^{v-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta_u^1 & \beta_u^2 & \cdots & \beta_u^{v-1} \end{pmatrix}.$$

When the β's are clear from context we denote this matrix simply by $\mathrm{Van}^{u\times v}$. This is called a *Vandermonde matrix*, and it is well known that if $u = v$ (so the matrix is square) its determinant is equal to $\prod_{i<j}(\beta_i - \beta_j)$. In particular, this determinant is non-zero (and hence the matrix is invertible) if and only if all β's are different. Let $d \geq 0$ and let $\beta_0, \dots, \beta_d \in \mathbb{F}$ be all different. Since given a polynomial $f(\mathtt{X}) = \sum_{i=0}^{d} c_i \mathtt{X}^i \in \mathbb{F}_{\leq d}[\mathtt{X}]$ it holds that

$$(f(\beta_0), \dots, f(\beta_d))^\intercal = \mathrm{Van}^{(d+1)\times(d+1)}(\beta_0, \dots, \beta_d) \cdot (c_0, \dots, c_d)^\intercal,$$

this shows that every polynomial of degree at most d is determined by its evaluation at any $d+1$ distinct points.

To secret-share a value $s \in \mathbb{F}$ using Shamir secret-sharing, the dealer samples a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq t}[\mathtt{X}]$ at random, restricted only to $f(\alpha_0) = s$. The share corresponding to party P_i is then $f(\alpha_i)$. As an example, if $\alpha_0 = 0$, it is easier to see more explicitly how such sampling could be done: the dealer samples $c_1, \dots, c_t \in_R \mathbb{F}$ and sets $f(\mathtt{X}) = s + \sum_{i=1}^{t} c_i \mathtt{X}^i$, but for a general α_0 the process is slightly more complex, as we describe below.

Shamir Secret-Sharing

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \dots, P_n$ as follows.

1. Sample $s_1, \dots, s_t \in_R \mathbb{F}$ and define

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{pmatrix} = \begin{pmatrix} 1 & \alpha_0^1 & \alpha_0^2 & \cdots & \alpha_0^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_d^1 & \alpha_d^2 & \cdots & \alpha_d^d \end{pmatrix}^{-1} \cdot \begin{pmatrix} s \\ s_1 \\ \vdots \\ s_d \end{pmatrix}.$$

   Let $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ be given by $f(\mathtt{X}) = \sum_{i=0}^{t} c_i \mathtt{X}^i$.

2. For each $i = 1, \dots, n$, the dealer distributes the share $f(\alpha_i)$ to party P_i

---

Reconstruction from any $t+1$ shares

Given shares $\{f(\alpha_i)\}_{i \in \mathcal{A}}$ for some subset $\mathcal{A} \subseteq [n]$ with $|\mathcal{A}| = t+1$, the secret is reconstructed as follows.

1. Fix some ordering in $\mathcal{A}$ and let us denote $(\alpha_i)_{i \in \mathcal{A}} = (\alpha_{a_1}, \dots, \alpha_{a_{t+1}})$. Let

$$(\lambda_{a_1}^{\mathcal{A}}, \dots, \lambda_{a_{t+1}}^{\mathcal{A}}) = (1, \alpha_0, \dots, \alpha_0^t) \cdot \mathrm{Van}^{(t+1)\times(t+1)}(\alpha_{a_1}, \dots, \alpha_{a_{t+1}})^{-1}.$$

2. The secret is computed as $s = \sum_{i=1}^{t+1} \lambda_{a_i}^{\mathcal{A}} \cdot f(\alpha_{a_i})$.

Definition 2.1. *Given $\mathcal{A} \subseteq \{1, \dots, n\}$ with $|\mathcal{A}| = t+1$, we call the values above $\{\lambda_i^{\mathcal{A}}\}_{i \in \mathcal{A}}$* Lagrange coefficients. *These can be alternatively computed as*

$$\lambda_i^{\mathcal{A}} = \prod_{j \in \mathcal{A},\, j \neq i} \frac{\alpha_0 - \alpha_j}{\alpha_i - \alpha_j}.$$

The reconstruction of a secret distributed with Shamir secret-sharing from any $t+1$ shares is done by using these shares, which are evaluations of a polynomial of degree at most t, to recover such polynomial, followed by its evaluation at α_0. This is written much more explicitly in the description of the protocol above, which in particular shows that the secret is computed as a linear combination of the shares involved, a fact that will be used later in one of our described protocols.

Privacy of this secret-sharing scheme, that is, the fact that any set of t shares does not leak anything about the secret s, is also easy to see, as these are in a 1-1 correspondence with the randomness used by the dealer. To see this, consider for example the case in which the given set of shares is $\{f(\alpha_i)\}_{i=1}^{t}$. In the way that the secret-sharing scheme is defined, these t values constitute the seed used by the dealer, which are taken completely at random and independently from the secret s. In the general case it can be shown that there is a bijective affine transformation between the randomness used by the dealer and any set of t shares, which shows that these shares are uniformly random and independent of the secret. We omit the proof of this result since it follows in a straightforward manner from the properties introduced so far, and it is only heavy in notation.

2.2.1 Secret-Sharing and d-Consistency.

In our protocols, the parties will hold shares of different values, and the various parties will play the role of the dealer in Shamir secret-sharing to distribute certain secrets. We begin with the following definition.

Definition 2.2. *Let $\beta_1, \dots, \beta_\ell \in \mathbb{F}$ be all different. Given $\mathbf{s} = (s_1, \dots, s_\ell) \in \mathbb{F}^\ell$, we say that $\mathbf{s}$ is d-*consistent *if there exists $f(X) \in \mathbb{F}_{\leq d}[X]$ such that $s_i = f(\beta_i)$ for $i = 1, \dots, \ell$. Observe that if $\ell \leq d+1$, then every vector $\mathbf{s} \in \mathbb{F}^\ell$ is d-consistent, but if $d+1 < \ell$, then the set of d-consistent vectors constitutes a strict $\mathbb{F}$-vector subspace of $\mathbb{F}^\ell$.*

If the parties hold d-consistent Shamir shares $(s_1, \dots, s_n)$ of a value s, we denote this by $[\![s]\!]_d = (s_1, \dots, s_n)$. This definition makes sense in the context of a passive adversary: dealers will always follow the protocol so they will distribute valid shares, and the (passively) corrupt parties will always use the correct shares they hold when required. However, in the context of an active adversary, the following can happen:

- An actively corrupt dealer distributes shares $(s_1, \dots, s_n)$ that are not d-consistent.
- Even if a dealer distributes d-consistent shares $(s_1, \dots, s_n)$, an actively corrupt party P_i can modify its own share from s_i to any value s_i' at any given point.

Given these issues, it does not make too much sense to say that the parties *hold* a vector of shares $(s_1, \dots, s_n)$ (again, since the actively corrupt parties can change their shares). To address this, we expand the definition of $[\![s]\!]_d$ and d-consistency to the setting of an active adversary, as follows.

Definition 2.3. *Consider the setting of an active adversary. Let $\mathcal{H}, \mathcal{C} \subseteq [n]$ be the sets of indexes corresponding to honest and corrupt parties, respectively. We say that the parties hold d-*consistent *shares of a secret $s \in \mathbb{F}$ if there exists a polynomial $f(X)$ of degree at most d such that:*

1. *$s = f(\alpha_0)$;*
2. *Each honest party P_i for $i \in \mathcal{H}$ has $s_i = f(\alpha_i)$ (i.e. the honest parties' shares are d-consistent);*
3. *The adversary knows $s_j = f(\alpha_j)$ for $j \in \mathcal{C}$.*

Furthermore, when this holds, we write $[\![s]\!]_d = (s_1, \dots, s_n)$.[1]

One aspect that is not very formal from the definition above is what it means for the adversary to "know" a given value. This is formalized in the context of simulation-based

[1] We clarify that sometimes the notation $[\![s]\!]$ instead of $[\![s]\!]_d$ will be used when the degree d is clear from context.

proofs by requiring that the simulator is able to "extract" the given value from the adversary, after interacting with it on the emulated protocol execution. Details are left to the relevant sections that make use of this concept, such as Section 2.5. However, some intuition on this notion can be provided. Consider for example a setting in which an honest dealer sends d-consistent shares $(s_1, \dots, s_n)$ of a secret s to the parties, with $s_i = f(\alpha_i)$ for some $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$, but the corrupt parties P_i change their shares from s_i to s'_i. Even though the resulting vector may not be d-consistent anymore, the parties still have d-consistent shares $[\![s]\!]_d$: (1) the honest parties' shares are d-consistent, and (2) the adversary *knows* the "real" values s_i corresponding the corrupt parties P_i.

Remark 2.1. *Let $\mathcal{H} \subseteq [n]$ be the set of indexes corresponding to honest parties. Let $[\![s]\!]_d = (s_1, \dots, s_n)$ be a d-consistent sharing, which means that there exists a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $s = f(\alpha_0)$, $s_i = f(\alpha_i)$ for $i \in \mathcal{H}$, and the adversary knows $s_j = f(\alpha_j)$ for $j \in [n] \setminus \mathcal{H}$. If $|\mathcal{H}| = n - t \geq d + 1$, then the polynomial $f(\mathtt{X})$ is unique, and in particular, so is the secret s. However, if $n - t \leq d$ then, for any secret $s' \in \mathbb{F}$, there exists a polynomial $f_{s'}(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $f_{s'}(\alpha_0) = s$ and $f_{s'}(\alpha_i) = s_i$ for $i \in \mathcal{H}$, so if the adversary knows $s'_j = f_{s'}(\alpha_j)$ for $j \in [n] \setminus \mathcal{H}$, the parties would simultaneously "hold" shares $[\![s']\!]_d$ of any secret s', or, in other words, there is not a well-defined secret from the honest shares alone. This problem will appear in Section 2.5 where $n/3 \leq t < n/2$, and $d = 2t$ in certain points.*

Finally, notice that if $t \geq n/2$ and $d = t$, then $n - t \leq d$. This is the reason why Shamir secret-sharing is typically only used if $t < n/2$: in this setting there are at least $t + 1$ honest parties, so their honest shares, if t-valid, define a unique secret.

Homomorphisms. Consider two shared values $[\![x]\!]_d$ and $[\![y]\!]_d$, using polynomials $f(\mathtt{X}), g(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$, respectively (that is, P_i's share of x is $f(\alpha_i)$ and the one of y is $g(\alpha_i)$). If each party adds their shares together, that is, each P_i computes $f(\alpha_i) + g(\alpha_i)$, they obtain shares of $x + y$ under the polynomial $f(\mathtt{X}) + g(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$. This is denoted by $[\![x + y]\!]_d \leftarrow [\![x]\!]_d + [\![y]\!]_d$. On the other hand, if the parties locally multiply their shares, they obtain shares of $x \cdot y$ under the polynomial $f(\mathtt{X}) \cdot g(\mathtt{X}) \in \mathbb{F}_{\leq 2d}[\mathtt{X}]$. This is denoted by $[\![x \cdot y]\!]_{2d} \leftarrow [\![x]\!]_d \cdot [\![y]\!]_d$. Note that the degree of the polynomial increases from d to $2d$.

Finally, observe that subtraction can be performed locally in a similar way as addition, as well as multiplying by any constant (that is, a value known by all parties). Furthermore, the parties can also locally add a constant, that is, obtain $[\![x + c]\!]$ from $[\![x]\!]$ and c, by each party adding this constant to their share.

2.2.2 Error Detection/Correction

As it has been already mentioned, Shamir secret-sharing is used, in the context of MPC, in order to distribute the inputs of the computation, as well as the intermediate values and the output, among the different parties, in such a way that the adversary does not learn anything about the underlying secrets. In the process of securely computing the given function, it will be necessary for the parties to *reconstruct*, or *open*, some secret-shared

values. This is the case, for example, for obtaining the output of the computation, which is in secret-shared form and must be learned by all the parties. Additionally, processing certain operations like multiplications during the course of the computation requires the parties to learn certain secret-shared data.

Reconstructing a secret-shared value $[\![s]\!]$ can be achieved, for example, by asking $t+1$ parties to send their shares to all the other parties.[2] After a party receives at least $t+1$ shares (including possibly its own), this party can reconstruct the polynomial of degree at most t that interpolate these shares, hence obtaining the secret.

Unfortunately, when the parties are actively corrupt, they can misbehave in the reconstruction of a secret-shared value by sending incorrect shares. For example, a secret reconstructed from the shares $(s_1, \dots, s_{t+1})$ would be computed as $s = \sum_{i=1}^{t+1} \lambda_i^{[t+1]} \cdot s_i$, but if party P_1 announces a different share $s_1 + \delta_1$, then the reconstructed secret would be $s + \delta$, with $\delta = \lambda_1^{[t+1]}\delta_1$. Furthermore, the parties do not have a way of detecting that a change has been introduced, given that $(s_1 + \delta_1, s_2, \dots, s_{t+1})$ look like legitimate shares of $s + \delta$.

The reason why the parties cannot detect that an error has been introduced is because any set of $t+1$ values $(s'_1, \dots, s'_{t+1})$ is consistent with a polynomial of degree at most t. However, if the parties use, say, $t+2$ shares $(s_1, \dots, s_{t+1}, s_{t+2})$ to first check the existence of a polynomial $f(\mathrm{X})$ such that $f(\alpha_i) = s_i$ for $i = 1, \dots, t+2$, then it is "less likely" that the error δ_1 that P_1 introduced preserves the existence of this polynomial. This is because a vector of $t+2$ shares is not necessarily consistent with a polynomial of degree at most t, and in fact, we can show that if $\delta_1 \neq 0$ then there is no way in which $(s_1 + \delta_1, \dots, s_{t+1}, s_{t+2})$ is consistent with a polynomial of degree at most t.

Unfortunately, the method above of checking consistency using $t+2$ shares is insufficient to prevent an attack that modifies the reconstructed secret, given that the adversary has the power to modify not only P_1's share, but the shares of at most t parties. The following example shows that, if the parties only check that $2t$ shares are consistent with a polynomial of degree at most t, then the adversary, who controls t parties actively, can cause the reconstructed secret to be incorrect, without being detected. This is related to Remark 2.1, where it is mentioned that if $n' - t \leq d$, where n' is the total number of shares (so $2t$ for the purpose of this example, where also $d = t$), then the secret is not well-defined by the honest shares alone, and the corrupt parties can modify theirs to result in t-valid sharings of any secret.

Example 2.3. *Let $[\![s]\!]_t = (s_1, \dots, s_n)$, where the underlying polynomial is $f(\mathrm{X})$, that is, $f(\alpha_0) = s$ and $f(\alpha_i) = s_i$ for $i \in [n]$. Let $\mathcal{A} \subseteq [n]$ be the set of the t indexes corresponding to corrupt parties, and let $\mathcal{B} \supseteq \mathcal{A}$ such that $|\mathcal{B}| \leq 2t$. Suppose that at reconstruction time the parties check that the announced shares corresponding to indexes in $\mathcal{B}$ are consistent with a polynomial $h(\mathrm{X})$ of degree at most t, and output $h(0)$ if this is the case. Then the adversary can cause the parties to reconstruct a wrong secret as follows.*

1. *The adversary samples a polynomial $g(\mathrm{X})$ of degree at most t such that $g(\alpha_i) = 0$*

[2] This incurs in a total communication complexity of $\approx t \cdot n$. This can be improved to $O(n)$, as described in Section 2.2.4.

for $i \in \mathcal{B} \setminus \mathcal{A}$ and $g(\alpha_0) = \delta$ for some $\delta \neq 0$ of the adversary's choice. Observe this is possible since $|(\mathcal{B} \setminus \mathcal{A}) \cup \{\alpha_0\}| \leq t + 1$.

2. *The corrupt parties P_i for $i \in \mathcal{A}$ modify their share s_i as $s_i' = s_i + \delta_i$, with $\delta_i = g(\alpha_i)$.*

At reconstruction time the parties check if there is a polynomial $h(\mathtt{X})$ of degree at most t such that $h(\alpha_i) = s_i$ for $i \in \mathcal{B} \setminus \mathcal{A}$, and $g(\alpha_i) = s_i'$ for $i \in \mathcal{A}$, and if this is the case, they output $h(0)$ as the reconstructed secret. Such polynomial indeed exists, namely $h(\mathtt{X}) = f(\mathtt{X}) + g(\mathtt{X})$. Indeed, if $i \in \mathcal{B} \setminus \mathcal{A}$, then $h(\alpha_i) = f(\alpha_i) + g(\alpha_i) = s_i + 0$, and if $i \in \mathcal{A}$ then $h(\alpha_i) = f(\alpha_i) + g(\alpha_i) = s_i + \delta_i = s_i'$. However, the reconstructed secret is equal to $h(\alpha_0) = f(\alpha_0) + g(\alpha_0) = s + \delta$.

In what follows we will see that the parties can check that the announced shares are correct if they use at least $2t + 1$ shares. In fact, we will see that they can identify the incorrect shares, remove them, and therefore reconstruct the right secret, if they use at least $3t + 1$ shares. These results will be presented in a more general way, using terminology that resembles that in the field of error correcting codes, and later in Section 2.2.3 we will interpret what these results imply in our setting.

The results in this section are more general, since they are phrased in the context of error-correcting codes. Let ℓ and d be non-negative integers, and let $\beta_1, \ldots, \beta_\ell \in \mathbb{F}$ be all different. Let $\mathbf{s} = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ be a d-consistent vector, which, recalling from Definition 2.2, means that there exists $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $s_i = f(\beta_i)$ for $i = 1, \ldots, \ell$. Suppose that this vector is modified as $\mathbf{s} + \boldsymbol{\delta}$, for some error vector $\boldsymbol{\delta} \in \mathbb{F}^\ell$. In the context of *error-correction*, the goal is to recover the polynomial f such that $f(\beta_i) = s_i$ for $i = 1, \ldots, \ell$ from the corrupted vector $\mathbf{s} + \boldsymbol{\delta}$. In *error-detection* we are only interested in determining whether $\boldsymbol{\delta}$ is non-zero.

2.2.2.1 Error Detection

First, notice that being d-consistent is an $\mathbb{F}$-linear property, so $\boldsymbol{\delta}$ is d-consistent if and only if $\mathbf{s} + \boldsymbol{\delta}$ is d-consistent as well. If these conditions hold, then any hope of error-correction or detection is lost given that $\mathbf{s} + \boldsymbol{\delta}$ will look as "legitimate" as $\mathbf{s}$. Given the above, we begin by looking at some conditions under which $\boldsymbol{\delta}$ can be d-consistent. Let e be an upper bound on the number of non-zero entries of δ. Observe that if $e < \ell - d$, then $\boldsymbol{\delta}$ cannot be d-consistent, unless it is the zero vector. This is because, if $\boldsymbol{\delta}$ was d-consistent, its $\ell - e > d$ zero entries would be enough to determine the underlying polynomial, which has to be the zero polynomial. On the other hand, if $e \geq \ell - d$, then it is easy to check that $\boldsymbol{\delta}$ could be d-consistent.[3]

We see then that, if $e < \ell - d$, the only way in which $\mathbf{s} + \boldsymbol{\delta}$ can be d-consistent is if $\boldsymbol{\delta} = \mathbf{0}$, so if no error was introduced. As a result, we can *detect* whether $\boldsymbol{\delta}$ is zero by checking if $\mathbf{s} + \boldsymbol{\delta}$ is d-consistent.

[3] In fact, this is the reason why the attack in Example 2.3 works. There we have $\ell \leq 2t$, $d = t$ and $e = t$, so $e \geq \ell - d$. This is also the reason why, if $n - t \leq d$, a degree-d secret-shared value is not well defined if the adversary is active, as pointed out in Remark 2.1.

A more intuitive view. Another way to see the result displayed above is the following. Suppose that $e < \ell - d$, and that $\mathbf{s} + \boldsymbol{\delta}$ happens to be d-consistent. Then, the underlying polynomial could be recovered from any set of $d + 1$ entries, and in particular, it could be recovered from the $\ell - e \geq d + 1$ entries that were not affected by $\boldsymbol{\delta}$, which shows that the underlying polynomial of $\mathbf{s} + \boldsymbol{\delta}$ has to be the same as that of $\mathbf{s}$.

2.2.2.2 Error Correction

Unfortunately, even if $e < \ell - d$, knowing that $\mathbf{s} + \boldsymbol{\delta}$ is not enough to find the polynomial $f(\mathtt{X})$ underlying $\mathbf{s}$. For example, if $e = \ell - d - 1$, a pair of non-zero error vectors $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2 \in \mathbb{F}^\ell$ having at most e non-zero entries can be found such that $\mathbf{s}_2 = \boldsymbol{\delta}_1 - \boldsymbol{\delta}_2$ is d-consistent, but this is a problem since the modified vectors $\mathbf{s}_1 + \boldsymbol{\delta}_1$ and $\mathbf{s}_2 + \boldsymbol{\delta}_2$, where $\mathbf{s}_1 = \mathbf{0}$ (which is d-consistent), are the same, so given only this vector it is not possible to know if it is a modified version of $\mathbf{s}_1$ or of $\mathbf{s}_2$.

If the bound e is zero, then obviously we can always find f from $\mathbf{s} + \boldsymbol{\delta}$, so there must be a point in which the amount of errors in $\boldsymbol{\delta}$ is so small, that error-correction is possible. This point is reached when $e < (\ell - d)/2$, and moreover, this is optimal in the sense that, for $(\ell - d)/2 \leq e < \ell - d$, we can always build examples as the one suggested above that show that finding the original polynomial for $\mathbf{s}$ is not possible.

We claim that if $e < (\ell - d)/2$, if $\mathbf{s}_1, \mathbf{s}_2 \in \mathbb{F}^\ell$ are two different d-consistent vectors, and if $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2 \in \mathbb{F}^\ell$ are error vectors with at most e non-zero entries each, then $\mathbf{s}_1 + \boldsymbol{\delta}_1 = \mathbf{s}_2 + \boldsymbol{\delta}_2$ cannot hold. This would enable error correction of $\mathbf{s} + \boldsymbol{\delta}$ by looking through all possible error vectors with at most e non-zero entries, subtracting it from $\mathbf{s} + \boldsymbol{\delta}$, and checking if the result is d-consistent vector. To show the claim above simply notice that $\mathbf{s}_1 + \boldsymbol{\delta}_1 = \mathbf{s}_2 + \boldsymbol{\delta}_2$ implies that $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1 = \mathbf{s}_1 - \mathbf{s}_2$ would be a d-consistent vector, but this cannot happen since $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1$ has at most $2e < \ell - d$ non-zero entries, but we just showed above that a vector with strictly less than $\ell - d$ non-zero entries cannot be d-consistent unless it is the zero vector, which would imply that $\mathbf{s}_1 = \mathbf{s}_2$.

Efficient error-correction. Above, we said that to error correct $\mathbf{s} + \boldsymbol{\delta}$, one would have to go over all possible vectors $\boldsymbol{\delta}$ could be equal to, which is of course very inefficient. This could be optimized slightly by looping over all possible subsets of e coordinates, and checking if the remaining coordinates of $\mathbf{s} + \boldsymbol{\delta}$ form a d-consistent vector (of dimension $\ell - e$). Because $(\ell - e) - d > e$, the results from before show that this can only happen if this "sub"-vector does not have any errors in it, which means that the guessed coordinates contain all the possible errors. This, unfortunately, is still too inefficient.

Instead, in practice we would recur to *error-correction algorithms*, also known as *decoders*, which achieve the task of identifying the error locations very efficiently. For the case at hand, we could use for example the Berlekamp-Welch algorithm [79], which is an efficient algorithm to solve the decoding problem. A generalization of this method to other algebraic structured beyond fields is presented in Section 3.1.4.

A more intuitive view. The process of error-correction can be also thought of as follows. Suppose that after looping through all subsets of coordinates of $\mathbf{s} + \boldsymbol{\delta}$ of size $\ell - e$, we find one that is d-consistent. Then, any set of $d + 1$ coordinates among these determine all the others. Furthermore, we know that the chosen sub-vector has at most e errors, so it has at least $(\ell - e) - e = \ell - 2e \geq d + 1$ non-modified entries, which determine the polynomial completely. Since these entries are the same as in $\mathbf{s}$, we see that the polynomial underlying the chosen sub-vector is the same as the one for the original vector $\mathbf{s}$.

2.2.3 Error Correction/Detection in the Context of MPC

As motivated at the beginning of Section 2.2.2, the purpose of the theory of error detection/correction in the context of MPC is to allow parties to reconstruct secret-shared values correctly, in spite of the actively corrupt parties announcing incorrect shares. More precisely, the parties have a secret-shared value $[\![s]\!]_d = \mathbf{s} = (s_1, \ldots, s_n)$, and in order to learn s, each party P_i announces its share s_i to the other parties. Actively corrupt parties may announce an incorrect $s_i + \delta_i$ for some error δ_i, which means the secret must be reconstructed from the shares $\mathbf{s} + \boldsymbol{\delta}$, where $\boldsymbol{\delta} \in \mathbb{F}^n$ has as its i-th entry 0 if i is an index corresponding to an honest party, and δ_i otherwise. Since there are t corrupt parties, there are at most t non-zero entries in $\boldsymbol{\delta}$. This puts us in the context of error correction/detection studied before with $\ell = n$ and $e = t$.

The results from the previous sections can be summarized as follows:

No error detection. If $e \geq \ell - d$, then the adversary can choose $\boldsymbol{\delta}$ so that $\mathbf{s} + \boldsymbol{\delta}$ is d-consistent, and the reconstructed secret will be $s + \delta$ for some δ of the adversary's choice.

Error detection. If $e < \ell - d$, then $\mathbf{s} + \boldsymbol{\delta}$ is d-consistent if and only if $\boldsymbol{\delta} = 0$.

Error correction. If $2e < \ell - d$, then there exist efficient algorithms to recover $\mathbf{s}$ from $\mathbf{s} + \boldsymbol{\delta}$.

As we will see in subsequent sections, the values of d that we will need to make use of in our protocols are $d = t$ and $d = 2t$. Furthermore, the two main settings in which we will make use of these techniques are the honest majority setting, in which $t < n/2$ (see Section 2.5), and two-thirds honest majority where $t < n/3$ (see Section 2.4). The following table summarizes, for the cases of $t < n/2$ and $t < n/3$, and for the different degrees $d = t$ and $d = 2t$, when error detection/correction is possible. These results will be used later on in Sections 2.4 and 2.5 when we construct actively secure protocols with $t < n/3$ and $t < n/2$ respectively.

		error corr.	error det.
$t < n/2$	degree t	✓	✓
	degree $2t$	✗	✓
$t < n/3$	degree t	✗	✓
	degree $2t$	✗	✗

2.2.4 Reconstructing Secret-Shared Values Efficiently

The previous section addressed the question of ensuring the adversary does not fool the parties into reconstructing a secret-shared value $[\![s]\!]_d$ incorrectly as $s+\delta$. However, doing this naively would require each party to send its share to all other parties, which incurs in a total communication complexity of at least $n \cdot t$ field elements being transmitted. To alleviate this issue, an alternative is to let the parties send their shares to a single "intermediate" receiver, who reconstruct the secret and then informs all the parties of this result. This is in fact the approach taken in the protocol from Section 2.3.2.1 to reconstruct $[\![a]\!]_{2t}$: the parties send their shares to P_1, who sends the result back to all the other parties.

Unfortunately, the issue that arises with this approach is that an actively corrupt intermediate receiver may choose to lie about the reconstructed value, which would ultimately lead to the parties learning an incorrect secret. Designing a solution to this problem while still achieving linear communication complexity is far from trivial, and the one we will present below was introduced in [42].

Suppose that the parties are not reconstructing one value $[\![s]\!]_d$, but many of these $[\![s_0]\!]_d, [\![s_1]\!]_d, \ldots, [\![s_d]\!]_d$. Consider the polynomial $f(\mathtt{X}) = \sum_{j=0}^{d} s_j \mathtt{X}^j$. The parties can compute $[\![f(\alpha_i)]\!]_d = \sum_{j=0}^{t} [\![s_j]\!]_d \, \alpha_j^i$ for $i \in [n]$ (observe that each $f(\alpha_i)$ can seen as a degree-d share of $f(\alpha_0)$). Then, the parties can reconstruct each of these shares towards the corresponding parties, which leads the parties to obtain shares $[\![f(\alpha_0)]\!]_d = (f(\alpha_1), \ldots, f(\alpha_n))$. At this point they can reconstruct this new secret using the naive approach from above: each party P_i sends its share $f(\alpha_i)$ to *all* other parties. This enables each party to error correct/detect in order to recover the secret $f(\alpha_0)$, but, what is more important, is that each party will not only recover the secret, but the polynomial $f(\mathtt{X})$ itself, whose coefficients are $s_0, s_1, \ldots, s_d$.

Regarding communication complexity, the solution above involves $\Theta(d \cdot n)$ field elements. However, since $d+1$ secret-shared values are reconstructed, the amortized complexity per secret is $\Theta(n)$, as required. The protocol is summarized below.

$\Pi_{\text{PublicRec}}$: Efficient Public Reconstruction

Input: Secret-shared values $[\![s_0]\!]_d, \ldots, [\![s_d]\!]_d$
Output: All the parties learn $s_0, \ldots, s_d$.
Protocol: The parties proceed as follows

1. Let $f(\mathtt{X}) = \sum_{j=0}^{d} s_j \mathtt{X}^j$. The parties locally compute $[\![f(\alpha_i)]\!]_d = \sum_{j=0}^{d} [\![s_j]\!]_d \, \alpha_i^j$ for $i \in [n]$.

2. Each party P_k for $k \in [n]$ sends its share of $[\![f(\alpha_i)]\!]$ to P_i, for $i \in [n]$.
3. Upon receiving the shares of $[\![f(\alpha_i)]\!]$, each P_i for $i \in [n]$ does the following:
   - If $n > d + 2t$, then perform error correction to recover $f(\alpha_i)$.
   - If $n > d + t$, then perform error detection to either recover $f(\alpha_i)$, or fail at reconstruction. If the latter happens then the party aborts.
4. If no abort was produced in the previous step, each P_i for $i \in [n]$ sends the reconstructed $f(\alpha_i)$ to each other party P_j.
5. Upon receiving the shares, each party P_j proceeds as follows:
   - If $n > d + 2t$, then perform error correction to recover the polynomial $f(X)$, and output its coefficients $s_0, \ldots, s_t$.
   - If $n > d + t$, then perform error detection to either recover the polynomial $f(X)$, outputting its coefficients, or fail at reconstruction. If the latter happens then abort.

To see why the protocol works as intended we proceed as follows. First, by the results from Section 2.2.3, each party P_i for $k \in [n]$, upon receiving the shares of $[\![f(\alpha_i)]\!]_d$ from the other parties, is able to perform error correction if $n > d + 2t$ to recover $f(\alpha_i)$, or alternatively it can perform error detection if $n > d + t$.

Now notice the polynomial $f(X)$ has degree at most d, and at this point if no abort has happened the parties hold the evaluation points $(f(\alpha_1), \ldots, f(\alpha_n))$. Again by the results from Section 2.2.3 we have that, when these shares are announced, the parties can perform error correction if $n > d + 2t$ to recover not only the "secret" $f(\alpha_0)$, but the polynomial $f(X) = \sum_{j=0}^{d} s_j X^j$, and if $n > d + t$, error detection can be performed. As a result, if no party aborts, the parties finish the protocol reconstructing the correct original secret-shared values.

Remark 2.2. *The protocol $\Pi_{\text{PublicRec}}$ requires the parties to reconstruct several secret-shared values $[\![s_0]\!]_d, \ldots, [\![s_d]\!]_d$ in order to benefit from the improved efficiency. However, it can be the case during certain steps of a protocol execution only a few secret-shared values must be reconstructed. For example, this is the case if the function being computed only has one output, which is obtained in secret-shared form and must be reconstructed. In this case, instead of using the protocol $\Pi_{\text{PublicRec}}$, the parties can simply use the more naive approach of sending the shares to each other in one single round, with each party error correcting on its own to get the output. This involves quadratic communication complexity, but this is acceptable since this is only called once at the end of the protocol execution.*

2.3 Passive and Perfect Security for Honest Majority

With the tools that have been described so far we are now ready to describe a perfectly secure protocol that can withstand a passive adversary corrupting t parties where $t < n/2$. Recall from Section 1.3.2 that the combination of passive and perfect security with honest majority cannot be enhanced in any way, in the sense that improving any of

the three properties degrades the others. For example, shifting from passive to active security requires us to either switch from perfect to statistical simulation, or from honest majority to two-thirds honest majority. Also, for instance, if the threshold $t < n/2$ does not hold, that is $t \geq n/2$, then even if we settle with passive security, the simulation would have to be computational.

Notice that, since in this first protocol the adversary is assumed to be passive, the theory on error correction/detection developed in Section 2.2.2 does not play a role here given that the corrupt parties do not modify their shares when reconstructing a secret-shared value. This theory will be used in Sections 2.4 and 2.5 when we consider active adversaries.

2.3.1 A First Protocol

Let us begin with a protocol that is conceptually very simple, which is taken from [17,54].

Recall from Section 2.2 that, for any subset of indexes $\mathcal{A} \subseteq [n]$ of size $d+1$, there exists coefficients $\lambda_1^{\mathcal{A}}, \ldots, \lambda_{d+1}^{\mathcal{A}}$ such that, whenever the parties have a shared secret $[\![s]\!]_d = (s_1, \ldots, s_n)$, this value can be computed as $s = \sum_{i \in \mathcal{A}} \lambda_i^{\mathcal{A}} \cdot s_i$. This will be used in the protocol below.

Passively secure protocol with perfect security

Setting: Each party P_i has input $x_i \in \mathbb{F}$.

Input phase: Each party P_i acts as the dealer in Shamir secret-sharing to distribute shares of its input x_i using polynomials of degree t. The parties obtain $[\![x_i]\!]_t$.

Addition gates: For each addition gate with secret-shared inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties locally compute $[\![x+y]\!]_t \leftarrow [\![x]\!]_t + [\![y]\!]_t$.

Multiplication gates: For each multiplication gate with secret-shared inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties proceed as follows:

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$.
2. Let us write $[\![x \cdot y]\!]_{2t} = (z_1, \ldots, z_n)$. Let $\lambda_1, \ldots, \lambda_{2t+1} \in \mathbb{F}$ be the (publicly known) coefficients such that $x \cdot y = \sum_{i=1}^{2t+1} \lambda_i \cdot z_i$. Each party P_i for $i = 1, \ldots, 2t+1$ acts as the dealer in Shamir secret-sharing to distribute shares $[\![z_i]\!]_t$.
3. The parties compute locally $[\![x \cdot y]\!]_t = \sum_{i=1}^{2t+1} \lambda_i \cdot [\![z_i]\!]_t$

Output phase: Let $[\![z]\!]_t$ be the output of the computation.

1. Each P_i for $i = 1, \ldots, t+1$ sends its own share of z to all the other parties.
2. After each party P_i receives $t+1$ shares (possibly counting its own), it reconstruct the output z.

Correctness and privacy of the protocol should be clear given that the adversary is passive: after the input phase the parties have shares of each input to the protocol, which does not leak information to the adversary due to the privacy properties of Shamir secret-sharing scheme since the adversary knows only t shares coming from the corrupt parties,

which are not enough to determine a $[\![\cdot]\!]_t$-shared secret. After the input phase, all parties proceed in a "gate-by-gate" fashion, computing shares of each intermediate value, or wire, in the computation. Addition gates are clearly correct since these make use of the homomorphic properties of Shamir secret-sharing. For multiplication gates, correctness follows from inspection, and privacy follows again from the fact that each party P_i is distributing information only in secret-shared form, which does not leak information towards the adversary.

Intuition for the security proof

As explained at the beginning of the chapter, we will not include formal security proofs for now, as these are delayed until Part II of this work, where the actual contributions are presented. However, it is still a fruitful exercise to provide some intuition as to how such proof would work in the protocol we have at hand here.

In order to obtain a proof of the security of the protocol described above, we would have to define a simulator $\mathcal{S}$ that interacts with the adversary corrupting t parties, in such a way that the adversary does not know if he is interacting in a real-world execution of the protocol, where the actual honest parties are running the protocol at hand, or in a ideal-world execution, where the honest parties only provide their inputs to an ideal functionality that is in charge of computing the function on the given inputs and returning only the output.

For simplicity in the notation, let us assume that the corrupt parties are $P_1, \dots, P_t$. The simulator $\mathcal{S}$ can interact with the t corrupt parties, and also with the functionality that computes the given function ideally by providing inputs to it on behalf of the corrupt parties. $\mathcal{S}$ needs to "fool" the adversary into believing that he is interacting with real honest parties, so to do this $\mathcal{S}$ emulates a set of honest parties $\overline{P}_{t+1}, \dots, \overline{P}_n$ that will interact with the corrupt parties in what should look like a genuine execution of the protocol. Notice that the trick here is that the simulator does not know what the actual inputs from the honest parties are!

Let us begin by analyzing how the input phase could be simulated. In the protocol description, all parties need to secret-share their input. $\mathcal{S}$ does not know the inputs from the honest parties, so it cannot distribute towards the corrupt parties shares of these inputs. However, this does not matter: the adversary corrupts only t parties, and recall from the properties of Shamir secret-sharing that any set of t shares looks completely random. Hence, the emulated honest parties can simply send random values to the t corrupt parties as the shares of their inputs, without actually knowing what their inputs are. Therefore, so far, the adversary cannot tell whether he is interacting with the actual honest parties in the real world, or with the simulator in the ideal world.

On the other hand, as part of the input phase, the emulated honest parties $\overline{P}_{t+1}, \dots, \overline{P}_n$ will receive shares of the inputs $x_1, \dots, x_t$ from the corrupt parties $P_1, \dots, P_t$, and since there are at least $t+1$ parties among the parties emulated by $\mathcal{S}$ (given that $t < n/2$), this enables him to reconstruct these inputs $x_1, \dots, x_t$. Furthermore, this also enables $\mathcal{S}$ to reconstruct not only the inputs $x_1, \dots, x_t$, but also the shares that the corrupt parties

have of these inputs (since any $t+1$ shares not only determine the secret, but the polynomial that was used to distribute it, so in particular they also determine all of the other shares).

At this point the corrupt parties have "shares" of all the inputs, with the quotes serving the purpose of emphasizing that the shares corresponding to the inputs of honest parties are just random values. On the other hand, the emulated honest parties do not really have any share. In a sense, the only parties that hold shares are the corrupt ones, but there are only t of them so these shares do not actually determine any secret, which enables the computation to "take place", even though in the ideal world only the inputs and the outputs exist. This will be made clearer in subsequent paragraphs.

Observe that $\mathcal{S}$ knows all the shares that the corrupt parties have, which is crucial as we will show towards the end. The next steps of the computation involve proceeding gate-by-gate, obtaining shares of their outputs from shares of the inputs. Addition gates are handled in a simple way as they require no interaction: the corrupt parties simply add their shares together (notice in particular that the simulator still knows the shares held by the corrupt parties if it knew the ones for the input summands).

Multiplication gates require a bit more care. Suppose that the shares held by the corrupt parties $P_1, \dots, P_t$ corresponding to the inputs of the multiplication are $(x_1, \dots, x_t)$ and $(y_1, \dots, y_t)$, which are known to $\mathcal{S}$. According to the description of the protocol, each corrupt party P_i will send to the other parties degree-t shares of $x_i \cdot y_i$. The simulator receives through the emulated honest parties at least $t+1$ shares of each $x_i \cdot y_i$, which enables him to determine the shares that the other corrupt parties received. Also, the protocol requires parties $\overline{P}_{t+1}, \dots, \overline{P}_{2t+1}$ to similarly secret-share the product of their shares of the inputs, but this is not possible since, again, the simulator does not know this information. This is again not a problem since these parties can simply send random shares to the t corrupt parties without actually knowing what value is being secret-shared. The adversary cannot distinguish this from what happens in the real execution.

Finally, the protocol reaches the output phase. The corrupt parties have shares of this output, and the simulator knows what these shares are. Recall that $\mathcal{S}$ learned the inputs $x_1, \dots, x_t$ from the corrupt parties in the input phase. The simulator can interact with the functionality to obtain the output z of the computation, using the *real* inputs from the actual honest parties. This is the exact same output that would have been computed if the adversary was involved instead in a real-world execution, where the actual honest parties participate in the protocol. So far the adversary cannot tell the difference.

Since $\mathcal{S}$ knows the output z, and he also knows the t shares of this value that the corrupt parties have, these $t+1$ points enables $\mathcal{S}$ to compute what the share corresponding to $\overline{P}_{t+1}$ should be so that it look consistent with the corrupt parties' shares and the given output z. Once this is done, $\overline{P}_{t+1}$ can easily play the output phase of the protocol by sending to the corrupt parties the computed share. The adversary ends up reconstructing z, since this is how the share of $\overline{P}_{t+1}$ was computed, and this is exactly what would have happened in a real execution.

2.3.2 A More Efficient Protocol

The protocol described above is conceptually very simple. However, its main disadvantage is the amount of data the parties have to communicate measured in terms of n, the number of parties. Let M denote the number of multiplication gates in the circuit. For each of these gates, the protocol requires the parties $P_1, \dots, P_{2t+1}$ to secret-share a value towards the other parties, which in turn requires each of these parties to send a share to each other party. This amounts to $n-1$ field elements sent by each of the parties in $\{P_1, \dots, P_{2t+1}\}$, which gives a total of $\Theta(t \cdot n)$ field elements transmitted over the network, per multiplication gate. Alternatively, this can be written as $\Theta(t \cdot n \cdot M)$ for the whole computation.[4]

As a practical observation, it is natural to increase t as n increases, given that this value determines the amount of parties that the adversary needs to corrupt in order to break the privacy of the protocol, and it can be argued that the more parties that participate in the protocol, the "easier" it becomes for the adversary to corrupt a large portion of these. Furthermore, it is typical to consider the maximal case for which $t < n/2$, namely $t = \lfloor (n-1)/2 \rfloor$, an in this case the total communication complexity of the protocol above becomes $\Theta(n^2 M)$, which is also referred to as *quadratic communication complexity*. It would be much more ideal if we had a protocol with *linear communication complexity*, that is, $\Theta(nM)$. A protocol with such communication complexity has the property that, in average, the communication required by each party, which is $\frac{1}{n}\Theta(nM) = \Theta(M)$, is not affected by how many parties participate in the protocol. This can be phrased as follows: even if more parties join the computation, the amount of data each party has to send remains, in average, constant.

The goal of this section is to present a perfectly secure MPC protocol in the honest majority setting against a passive adversary that achieves linear communication complexity. The protocol is taken from [42], and it follows a similar template to the one described in Section 2.3.1: each party first secret-shares its input, and then the parties proceed in a gate-by-gate fashion, obtaining shares of the output of each gate, until they reach the final output of the computation, whose shares are exchanged so that the parties can reconstruct the result in the clear. The main difference lies in the way multiplication gates are handled, which is the main source of inefficiency in the previous protocol. The multiplication protocol from the previous section can be seen as having the following structure: The parties locally multiply the shares of the inputs, obtaining degree-$2t$ shares of the product of the underlying secrets, and then they perform an action that converts these shares from degree-$2t$ to degree-t. In the previous protocol this conversion was achieved via *resharing*: each party secret-shares its own degree-$2t$ share using degree-t sharings, and since reconstruction is linear, these shares can be combined in an appropriate manner to obtain degree-t sharings of the original secret. Instead of using resharing, the protocol we will discuss next uses the so-called *double-sharings* to reduce the $2t \to t$ conversion to the task of simply reconstructing certain shared value, which is much more efficient to achieve as we will see.

[4] This ignores the communication involved in other steps of the protocol such as the input and output phases. This is, however, reasonable, as in typical applications the "inner" complexity of the function (measured by the amount of multiplications in our case) is much bigger than the amount of inputs and outputs.

2.3.2.1 Using Double-Sharings for Secure Multiplication

Our protocol relies heavily on the concept of double-sharings, which constitute a special type of preprocessing material that enables the parties to handle multiplications securely. A double-sharing is a pair of the form $([\![r]\!]_t, [\![r]\!]_{2t})$, where $r \in \mathbb{F}$ is uniformly random and unknown to the adversary.[5] These double-sharings constitute *preprocessing material*, since they do not depend in any way on the inputs to the multiplication protocol, or in general, the inputs to the function under consideration.

Secure multiplication protocol via double-sharings

Preprocessing: A double-sharing $([\![r]\!]_t, [\![r]\!]_{2t})$.
Input: $[\![x]\!]_t$ and $[\![y]\!]_t$ two secret-shared values.
Output: $[\![z]\!]_t$, where $z = x \cdot y$
Protocol: The parties execute the following

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
2. The parties $P_2, \ldots, P_{2t+1}$ send their shares of $[\![a]\!]_{2t}$ to P_1 who, together with his own share, reconstructs a.
3. P_1 sends a to all the other parties, so this value becomes publicly known.
4. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

First, notice that the protocol achieves linear communication complexity: in step 2, $2t + 1$ parties send a single field element to only one party, P_1, who sends in step 3 one field element to all other parties. This yields a total communication complexity of $\Theta(n)$. Naturally, this only holds assuming that the parties can get the double sharing with linear communication complexity too, which is discussed in Section 2.3.2.2 below.

A small optimization. Instead of sending a to all parties, P_1 can secret-share this value, so that the parties get $[\![a]\!]_t$. The rest of the protocol remains the same, changing $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$ with $[\![z]\!]_t \leftarrow [\![r]\!]_t + [\![a]\!]_t$. The advantage of doing this is that, since a does not need to be kept private, t of the shares can be fixed to be 0, and the remaining shares can be computed from these together with the "secret" a. This means that P_1 only needs to communicate the shares to $n - t$ parties, since t of the parties know already that their share of a will be 0. This optimization was introduced in [57].

2.3.2.2 Producing Double-Sharings Efficiently

The task for this section is to describe a protocol in which the parties can compute double-sharings. This protocol could be used in a preprocessing stage, before the inputs of the parties are known.

To get started, let us consider the following simple protocol:

[5] This is formalized as a *functionality* that samples r internally and acts as the dealer in Shamir secret-sharing, distributing the appropriate shares to the parties. However, we stress that this chapter is not concerned with the formalisms of the protocols, so we omit this.

1. Each P_i for $i \in [t+1]$ samples $r_i \in_R \mathbb{F}$ and secret-shares this value towards the parties twice: using degree-t and degree-$2t$ polynomials. The parties obtain $[\![r_i]\!]_t$ and $[\![r_i]\!]_{2t}$.

2. The parties produce the double-sharing $([\![r]\!]_t, [\![r]\!]_{2t})$, where $[\![r]\!]_t = \sum_{i=1}^{t+1} [\![r_i]\!]_t$ and $[\![r]\!]_{2t} = \sum_{i=1}^{t+1} [\![r_i]\!]_{2t}$.

Since the adversary corrupts t parties, there is at least one honest party among $P_1, \ldots, P_{t+1}$, which implies that the value of r looks uniformly random to the adversary who knows all but one of the random summands. Unfortunately, this approach, although simple, does not suffice for our purposes since it has quadratic communication complexity (each party P_i for $i \in [t+1]$ needs to send shares to all other parties).

The following approach, proposed in [42], enables the parties to produce, using quadratic communication, a total of $\Theta(n)$ double-sharings. As a result, the amortized communication cost *per* double-sharing is linear. The protocol works as follows. Let $\mathbf{M} = \mathrm{Van}^{n\times(n-t)}(\beta_1, \ldots, \beta_n)$, where $\beta_1, \ldots, \beta_n$ are mutually-different elements of $\mathbb{F}$.

Preprocessing double-sharings

Output: A set of double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$.

2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_{n-t}]\!]_t \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_{n-1}]\!]_t \\ [\![s_n]\!]_t \end{pmatrix}, \qquad \begin{pmatrix} [\![r_1]\!]_{2t} \\ [\![r_2]\!]_{2t} \\ \vdots \\ [\![r_{n-t}]\!]_{2t} \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_{2t} \\ [\![s_2]\!]_{2t} \\ \vdots \\ [\![s_{n-1}]\!]_{2t} \\ [\![s_n]\!]_{2t} \end{pmatrix}.$$

3. The parties output the double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$.

Let us analyze that the protocol produces correct double-sharings, for which it suffices to show that the values $r_1, \ldots, r_{n-t}$ produced by the protocol look uniformly random to the adversary. We claim that these values are in a 1-1 correspondence with the values s_i sampled by the $n-t$ honest parties, which is enough to reach the desired conclusion as these are uniformly random and unknown to the adversary. To see that the claim holds, assume for simplicity that the first $n-t$ parties, $P_1, \ldots, P_{n-t}$, are honest. Let $\mathbf{M}' = \mathrm{Van}^{(n-t)\times(n-t)}(\beta_1, \ldots, \beta_{n-t})$ and $\mathbf{M}'' = \mathrm{Van}^{(n-t)\times t}(\beta_{n-t+1}, \ldots, \beta_n)$, then $\mathbf{M}^\intercal$ can be written in block form as $\mathbf{M}^\intercal = [\mathbf{M}'^\intercal | \mathbf{M}''^\intercal]$, so

$$\begin{pmatrix} r_1 \\ \vdots \\ r_{n-t} \end{pmatrix} = \mathbf{M}'^\intercal \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_{n-t} \end{pmatrix} + \mathbf{M}''^\intercal \cdot \begin{pmatrix} s_{n-t+1} \\ \vdots \\ s_n \end{pmatrix}.$$

Since $\mathbf{M}'$ is invertible, we obtain the desired result. The general case in which the honest parties may not be $P_1, \ldots, P_{n-t}$ is handled in a similar way by taking the appropriate $(n-t)\times(n-t)$ submatrix of $\mathbf{M}$.

Regarding communication complexity, observe that in the first step, which is the only step involving interaction, each party sends a share to each other party, which leads to a communication complexity of $\Theta(n^2)$. Since $n - t$ double-sharings are produced, we conclude that the amortized communication complexity of generating each double-share is $\Theta(n^2/(n-t))$, which is linear in n since $n - t > n/2$.

2.4 Active and Perfect Security for Two-Thirds Honest Majority

In the previous section we studied a perfectly secure protocol that is secure against a passive adversary corrupting t parties where $t < n/2$. The goal now is to extend this to active security. As mentioned before, this requires us to either lower the threshold from $t < n/2$ to $t < n/3$, or consider statistical instead of perfect security. In this section we take the first route, that is, we consider two-thirds honest majority and maintain the requirement on perfect security. The second approach, active and statistical security in the honest majority setting, is discussed in Section 2.5.

Before we get into the description of our protocol, recall that, as shown in Section 2.2.3, in the setting under consideration, $t < n/3$, the parties can reconstruct sharings $[\![s]\!]_d$ with error-correction (i.e. the parties are guaranteed to learn the correct secret) if $d = t$, and with error-detection (i.e. either the parties reconstruct the right secret, or the presence of errors is detected and the parties abort) if $d = 2t$. Furthermore, it is described in Section 2.2.4 how to do this efficiently via the protocol $\Pi_{\text{PublicRec}}$.

2.4.1 Actively Secure Multiplication for $t < n/3$

The tools developed in the previous sections are essential for the construction of an actively secure version of the multiplication protocol described in Section 2.3.2.1. The main issue with that protocol when ported to the actively secure scenario lies in opening, or reconstructing, the secret-shared value $[\![a]\!]_{2t}$. To this end we can use the reconstruction techniques from Section 2.2.4. In this case, $d = 2t$, and since we assume that $t < n/3$, we can take $\ell = n$ and ensure that the error detection bound $\ell > d + t$ holds (if $d = t$, then the error correction bound can be achieved, a fact that will be useful later on). The resulting protocol is described below. It assumes several simultaneous multiplications are to be processed, given that the reconstruction protocol from the previous section requires $t+1$ values to be opened to operate efficiently. Furthermore, as the preprocessing protocol from Section 2.3.2.2, the actively secure method to compute double-sharings we will discuss in Section 2.4.2 also operates in batches.

Actively secure multiplication protocol via double-sharings

Preprocessing: A double-sharing $([\![r]\!]_t, [\![r]\!]_{2t})$.
Input: Secret-shared values $[\![x]\!]_t$ and $[\![y]\!]_t$.
Output: $[\![z = x \cdot y]\!]_t$.
Protocol: The parties execute the following

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
2. The parties call the protocol $\Pi_{\text{PublicRec}}$ from Section 2.2.4 to learn a.[a]
3. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

[a] Recall that this protocol operates in batches of secret-shared data so this would be called once for many simultaneous secure multiplications.

2.4.2 Instantiating the Offline Phase

The protocol from above required as preprocessing material double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$. Unfortunately, the protocol from Section 2.3.2.2 to achieve such task cannot be used directly in the actively secure setting, with the main reason being the fact that, when a corrupt party P_i is asked with distributing shares $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$, it may not do this *consistently*. More precisely, the underlying secrets in the degree-t and degree-$2t$ must be equal according to the protocol specification, but P_i may choose them to be different. Furthermore, what is worse is that P_i can send shares that are not $t/2t$-consistent, that is, they may not be the result of evaluating a polynomial of the appropriate degree on the points $\alpha_1, \ldots, \alpha_n$. This is very sensitive, since the theory of error detection and correction that we developed in Section 2.2.2 relies heavily on the fact that the shares that the parties had were consistent.

The protocol we will consider to deal with this situation is taken from [16], and it makes use of the so-called hyper-invertible matrices in order to guarantee that the sharings distributed by each party satisfy the necessary consistency requirements. These are defined below, and used to generate double-sharings in Section 2.4.2.2.

2.4.2.1 Hyper-Invertible Matrices

A matrix $\mathbf{M} \in \mathbb{F}^{k \times \ell}$ is said to be *hyper-invertible* if every square sub-matrix obtained by taking subsets of the rows and columns of $\mathbf{M}$ is invertible.

An example of a hyper-invertible matrix is the following. Let $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_\ell \in \mathbb{F}$ be all different field elements, and let $M_{uv} = \prod_{i \in [\ell] \setminus \{v\}} \frac{\beta_u - \alpha_i}{\alpha_v - \alpha_i}$ for $u \in [k], v \in [\ell]$. As shown in [16], the matrix $\mathbf{M} \in \mathbb{F}^{k \times \ell}$ whose (u, v) entry is given by M_{uv} is hyper-invertible.[6]

2.4.2.2 Generating Double-Sharings

The protocol to generate the necessary double-sharings using hyper-invertible matrices is presented below. We let $\mathbf{M} \in \mathbb{F}^{n \times n}$ be a hyper-invertible matrix.

[6] A generalization of this construction and a full proof of why it is a hyper-invertible matrix can be found in Section 3.3.1.1.

Preprocessing double-sharings with active security

Output: A set of double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=2t+1}^{n}$

Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$, but observe that corrupt parties may distribute shares *inconsistently*.
2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_n]\!]_t \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_n]\!]_t \end{pmatrix}, \qquad \begin{pmatrix} [\![r_1]\!]_{2t} \\ [\![r_2]\!]_{2t} \\ \vdots \\ [\![r_n]\!]_{2t} \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} [\![s_1]\!]_{2t} \\ [\![s_2]\!]_{2t} \\ \vdots \\ [\![s_n]\!]_{2t} \end{pmatrix}.$$

3. For each $i \in [2t]$, all the parties send their shares of $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$ to P_i.
4. Upon receiving these shares, each P_i for $i \in [2t]$ checks that the received sharings of $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$ are t and $2t$-consistent, respectively. If any of the sharings is not consistent, or if both are but the reconstructed value is not equal in both cases, P_i sends abort to all parties and halts.
5. If no party sends an abort message in the previous step, then the parties output the double-sharings $([\![r_i]\!]_t, [\![r_i]\!]_{2t})$ for $i \in \{2t+1, \ldots, n\}$.

To analyze the protocol let us assume without loss of generality that the corrupt parties are $P_1, \ldots, P_t$. We claim that if there are no abort messages, then the following holds:

1. For each $i \in \{2t+1, \ldots, n\}$ the sharings $[\![r_i]\!]_t$ and $[\![r_i]\!]_{2t}$ held by the honest parties are t and $2t$-consistent, respectively, and their underlying secrets match.

2. For each $i \in \{2t+1, \ldots, n\}$, the secret r_i looks uniformly random and unknown to the adversary.

For the first claim we use the fact that no party among $P_1, \ldots, P_{2t}$ sent an abort message in step 4. Since $P_1, \ldots, P_t$ are actively corrupt, they may refrain from sending such message when they were actually supposed to. However, $P_{t+1}, \ldots, P_{2t}$ are all honest, so if none of these parties sent an abort message it is because the sharings they received, $([\![s_i]\!]_t, [\![s_i]\!]_{2t})$ for $i \in \{t+1, \ldots, 2t\}$, pass the check these parties perform. This means these sharings are consistent and their underlying secrets match.

Now, let us partition $\mathbf{M}$ in block form as follows:

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} & \mathbf{I} \end{pmatrix},$$

where $\mathbf{A}, \mathbf{B}, \mathbf{D}, \mathbf{E} \in \mathbb{F}^{t \times t}$, $\mathbf{C}, \mathbf{F} \in \mathbb{F}^{t \times (n-2t)}$, $\mathbf{G}, \mathbf{H} \in \mathbb{F}^{(n-2t) \times t}$ and $\mathbf{I} \in \mathbb{F}^{(n-2t) \times (n-2t)}$. Given

this partition, we see that, for $d = t, 2t$, it holds that

$$\begin{pmatrix} [\![r_{t+1}]\!]_d \\ [\![r_{t+2}]\!]_d \\ \vdots \\ [\![r_{2t}]\!]_d \end{pmatrix} = \mathbf{D} \cdot \begin{pmatrix} [\![s_1]\!]_d \\ [\![s_2]\!]_d \\ \vdots \\ [\![s_t]\!]_d \end{pmatrix} + \mathbf{E} \cdot \begin{pmatrix} [\![s_{t+1}]\!]_d \\ [\![s_{t+2}]\!]_d \\ \vdots \\ [\![s_{2t}]\!]_d \end{pmatrix} + \mathbf{F} \cdot \begin{pmatrix} [\![s_{2t+1}]\!]_d \\ [\![s_{2t+2}]\!]_d \\ \vdots \\ [\![s_n]\!]_d \end{pmatrix}.$$

Since $\mathbf{M}$ is hyper-invertible, the square submatrix $\mathbf{D}$ is invertible, which means that we can rewrite the equation above as

$$\begin{pmatrix} [\![s_1]\!]_d \\ [\![s_2]\!]_d \\ \vdots \\ [\![s_t]\!]_d \end{pmatrix} = \mathbf{D}^{-1} \cdot \begin{pmatrix} [\![r_{t+1}]\!]_d \\ [\![r_{t+2}]\!]_d \\ \vdots \\ [\![r_{2t}]\!]_d \end{pmatrix} - \mathbf{D}^{-1}\mathbf{E} \cdot \begin{pmatrix} [\![s_{t+1}]\!]_d \\ [\![s_{t+2}]\!]_d \\ \vdots \\ [\![s_{2t}]\!]_d \end{pmatrix} - \mathbf{D}^{-1}\mathbf{F} \cdot \begin{pmatrix} [\![s_{2t+1}]\!]_d \\ [\![s_{2t+2}]\!]_d \\ \vdots \\ [\![s_n]\!]_d \end{pmatrix}.$$

Observe that all the sharings that appear in the right-hand side of the equation above are d-consistent for $d = t, 2t$, and their underlying secrets are the same: we already argued this for $([\![r_{t+1}]\!]_d, \ldots, [\![r_{2t}]\!]_d)$, and for the remaining shares this holds since these were distributed by honest parties. As a result, since these properties are preserved under linear combinations, we see that the shares on the left-hand side also satisfy said properties. This shows that *all* sharings provided by the parties, $\{([\![s_i]\!]_t, [\![s_i]\!]_{2t})\}_{i=1}^n$, are t and $2t$-degree consistent and the underlying secrets match, which implies that the same holds for the final double-sharings produced by the protocol, $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=2t+1}^n$, since these are obtained as linear combinations of the ones above. This proves the first claim.

To prove the second claim we observe that we can write

$$\begin{pmatrix} r_{2t+1} \\ r_{2t+2} \\ \vdots \\ r_n \end{pmatrix} = \mathbf{G} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_t \end{pmatrix} + \mathbf{H} \cdot \begin{pmatrix} s_{t+1} \\ s_{t+2} \\ \vdots \\ s_{2t} \end{pmatrix} + \mathbf{I} \cdot \begin{pmatrix} s_{2t+1} \\ s_{2t+2} \\ \vdots \\ s_n \end{pmatrix}.$$

Since $\mathbf{I}$ is invertible, we see that $(r_{2t+1}, \ldots, r_n)$ is in a 1-1 correspondence with the vector $(s_{2t+1}, \ldots, s_n)$, but the latter is chosen at random by the honest parties and is unknown to the adversary, so $(r_{2t+1}, \ldots, r_n)$ will inherit such properties as well.

Finally, it is easy to see that the communication complexity of the protocol is $\Theta(n^2)$. However, since $n - 2t > n/3$ double-sharings are produced per execution, the amortized complexity per double-sharing is $\Theta(n)$, as required.

2.4.3 Actively Secure Input Phase

So far we have discussed how to deal with multiplications and output reconstruction when the adversary is behaving actively. The only missing step to completely port the protocol from Section 2.3.2 to the actively secure setting is the input phase in which each party distributes shares of its own input. The problem here when the adversary is active

is the same problem we have already encountered before: corrupt parties may distribute inconsistent shares.

Consistency is enforced by means of the following protocol. It requires as preprocessing material a secret-shared value $[\![r]\!]_t$, where $r \in \mathbb{F}$ is random only known by the party P_i who will provide input. This can be generated by taking a double-sharing $([\![r]\!]_t\, [\![r]\!]_{2t})$, discarding the degree-$2t$ part (or alternatively ignoring it from the start when generating the double-sharing), and letting the parties send their shares of $[\![r]\!]_t$ to P_i, which enables P_i to error correct to learn r. The protocol also requires the broadcast primitive that is part of the communication channel we assume, as discussed in Section 1.2.6.1. However, as discussed in Section 1.3.1, in the case in which $t < n/3$ and the corruption is active, which is the setting in this chapter, a protocol with perfect security instantiating this broadcast primitive exists.

Distributing shares of a given input

Input: Party P_i has an input x.
Preprocessing: A secret-shared value $[\![r]\!]_t$ where $r \in \mathbb{F}$ is random and only known by P_i.
Output: Parties get consistent shares $[\![x]\!]_t$. If P_i is corrupt then the underlying secret may not be equal to x.

Protocol: The parties proceed as follows:

1. P_i *broadcasts* $e = x - r$.
2. The parties locally compute $[\![x]\!]_t = [\![r]\!]_t + e$ as the final shares of the input x.

If the sender is honest then its input is kept private since the only information revealed is $e = x - r$, and since r is uniformly random and unknown to the adversary, this does not leak anything about x. Furthermore, in case P_i is corrupt, the resulting shares are still consistent since the they are obtained by adding a publicly known value e to an already consistently-shared value $[\![r]\!]$. Observe that this assumes that e is known by everyone, which implicitly means that all parties know the *same* value. This may not be the case if the rogue P_i sends different values for e to different parties. However, this is easy to enforce by means of a *broadcast protocol*, as described below.

2.5 Active and Statistical Security for Honest Majority

In this section we study active security in the honest majority setting, that is, where the number of corrupted parties t is strictly less than $n/2$. As discussed in Section 1.3.2, the best security notion achievable with this threshold is statistical security, which is the type of security we aim at in this section.

The protocol we will consider here follows a similar approach as the protocol from the previous section: in a preprocessing phase the parties generate double sharings which are then used in an online phase to compute multiplications securely. However, the main issue that will appear in the $t < n/2$ case is, as we will see, that the adversary can inject certain errors in the online phase that may cause the computation to be incorrect and, moreover, this may lead to leakage of sensitive information about the honest parties'

inputs. This is dealt with by executing a check before the output phase that is intended to verify that no errors were introduced during the computation.

Throughout this section we will assume that $n = 2t+1$. This is not only for simplicity: our protocol is designed to tolerate *exactly* t corruptions while assuming that there are $t+1$ honest parties. As mentioned in Remark 1.1, contrary to intuition, it is not generally true that a protocol that has been designed to withstand t corruptions is also secure against less than t corruptions, and the one presented in this section is an example of that. We will discuss this issue in detail towards the end of this section.

2.5.1 Reconstructing Secret-Shared Values

As in the previous protocols, an operation that the parties will need to execute several times lies in the reconstruction of a secret-shared value $[\![s]\!]_d$, where the degree d is either equal to t or $2t$. From Section 2.2.2, we see that in our current setting a party receiving n shares can error-detect if $d = t$, and moreover, reconstruction of secret-shared values in this case can be done with a communication complexity of $O(n)$ field elements with the help of the protocol $\Pi_{\text{PublicRec}}$ from Section 2.2.4. However, as discussed in Section 2.2.3, if $d = 2t$, then the adversary can cause the parties to reconstruct an incorrect secret $s + \delta$ for some (potentially non-zero) chosen δ. Fortunately, sharings of degree $2t$ are only used to compute multiplications securely, and, as we will soon see, cheating in this opening leads to incorrect multiplications which can be verified using different techniques. As a result, in spite of the adversary being able to cheat in the multiplications, leading to incorrect results, the validity of these can be checked by the parties.

We will make use of the protocol $\Pi_{\text{PublicRec}}$ from Section 2.2.4. However, this protocol does not consider the case $d = 2t$ in which the adversary can cause reconstruction to result in incorrect values. A secret-shared value $[\![s]\!]_{2t}$ can be reconstructed with the same communication complexity of $O(n)$ field elements as follows:

1. The parties send their shares of $[\![s]\!]_{2t}$ to P_1;
2. P_1 uses the first $2t+1$ shares $s_1, \ldots, s_{2t+1}$[7] to interpolate the unique polynomial $f(X)$ of degree at most $2t$ such that $f(\alpha_i) = s_i$ for $i \in [2t+1]$, and sets $s = f(\alpha_0)$;
3. P_1 sends s to all the parties.

As expected, after the execution of this protocol the adversary can cause the parties to reconstruct $s + \delta$ for some chosen error δ. This can happen if P_1 is corrupt and adds this

[7] Since we assume that $n = 2t+1$, these $2t+1$ shares constitute *all* the shares, but this method also works for $2t+1 < n$.

error when sending the result to the parties[8], or it can also occur even if P_1 is honest if the actively corrupt parties send wrong shares to P_1.

2.5.2 Preprocessing Phase

As we have already mentioned, the protocol we will use for the setting $t < n/2$ resembles a lot the protocol from Section 2.4 in which parties produce double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$ in the preprocessing phase, which are then used to obtain shares of a product $[\![xy]\!]_t$ from two shared values $[\![x]\!]_t$ and $[\![y]\!]_t$. This is done by letting the parties locally obtain $[\![xy]\!]_{2t}$, then open $a \leftarrow [\![xy]\!]_{2t} - [\![r]\!]_{2t}$ and later compute $[\![xy]\!]_t \leftarrow [\![r]\!]_t + a$ non-interactively.

We first discuss how the parties can obtain the necessary preprocessing material $([\![r]\!]_t, [\![r]\!]_{2t})$. In Section 2.4.2 we presented a protocol based on the so-called hyper-invertible matrices (HIM) to obtain this type of correlation for the case in which $t < n/3$. Unfortunately, this protocol is not suitable for our setting in which $t < n/2$, which can be seen by thoroughly inspecting the construction. However, in Section 2.3.2.2 we presented a *passively secure* protocol for generating double-sharing in the honest majority setting, and this method will serve as the basis for the actively secure mechanism to produce double-sharings we need in this section.

Let us begin by recalling briefly how the passively secure protocol from Section 2.3.2.2 works. It begins by asking each party P_i to sample $s_i \in_R \mathbb{F}$, and then secret-share this value twice using thresholds t and $2t$ as $([\![s_i]\!]_t, [\![s_i]\!]_{2t})$. Then the parties locally apply to these sharings a matrix that acts as a randomness extractor in order to obtain the final double-shares.

As mentioned in Section 2.4.2, this protocol is not actively secure, mainly because an actively corrupt party P_i may cheat when asked to secret-share the value s_i. This cheating may take place in different ways:

- P_i does not distribute $[\![s_i]\!]_t$ consistently, that is, the shares $(s_{i1}, \ldots, s_{in})$ of $[\![s_i]\!]_t$ sent by P_i to the other parties are not t-consistent.
- P_i does not distribute $[\![s_i]\!]_{2t}$ consistently, that is, the shares $(s'_{i1}, \ldots, s'_{in})$ of $[\![s_i]\!]_{2t}$ sent by P_i to the other parties are not $2t$-consistent.
- The shares $(s_{i1}, \ldots, s_{in})$ and $(s'_{i1}, \ldots, s'_{in})$ are t and $2t$-consistent, respectively, but the underlying secrets are not the same.

Although these issues, at a high level, seem harmful for the protocol, we can show that the ultimate effect they have on the execution is that a multiplication of two shared values $[\![x]\!]_t$ and $[\![y]\!]_t$ may result in $[\![x \cdot y + \delta]\!]_t$ for some adversarially-chosen error $\delta \in$

[8] If P_1 is actively corrupt then he can even perhaps add different errors to the value sent to different parties, which results in the parties learning different values. For simplicity in the presentation we assume this is not the case, that is, the honest parties obtain the *same* value $s + \delta$. This can be achieved by asking P_1 to use a broadcast channel to send this value. However, this is not necessary as the protocol still works even if P_1 distributes different values, as shown in Section 3.3.2, although as we said we assume this does not happen for the sake of presentation.

$\mathbb{F}$. This is of course a problem for the correctness of the protocol, since an adversary can cause intermediate values to be computed incorrectly. Fortunately, it is possible to check, quite efficiently, that the multiplications have been computed correctly, which is explained in Section 2.5.4.

From the observation above, the protocol the parties use in order to generate double-sharings efficiently is essentially the same as the protocol from Section 2.3.2.2. The protocol is described explicitly below for the sake of completeness. Let $\mathbf{M} = \mathrm{Van}^{n\times(n-t)}(\beta_1, \ldots, \beta_n)$, where $\beta_1, \ldots, \beta_n$ are different elements of $\mathbb{F}$.

Preprocessing double-sharings

Output: A set of double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$.
2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_{n-t}]\!]_t \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_{n-1}]\!]_t \\ [\![s_n]\!]_t \end{pmatrix}, \qquad \begin{pmatrix} [\![r_1]\!]_{2t} \\ [\![r_2]\!]_{2t} \\ \vdots \\ [\![r_{n-t}]\!]_{2t} \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_{2t} \\ [\![s_2]\!]_{2t} \\ \vdots \\ [\![s_{n-1}]\!]_{2t} \\ [\![s_n]\!]_{2t} \end{pmatrix}.$$

3. The parties output the double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$.

As mentioned before, even though this protocol is in principle not actively secure (in the sense that there are seemingly a lot of places where the adversary can cheat to cause a potentially harmful outcome), we will be able to show that, when this protocol is used in conjunction with the protocol for multiplying two shared values from Section 2.5.3 below, the end result is that the adversary is able to inject additive errors to the result of a multiplication. Fortunately, this type of attack can be prevented as described in Section 2.5.4.

Instead of analyzing in detail the security guarantees of this protocol on their own, we postpone the analysis to Section 2.5.3 below where we analyze the properties of the multiplication protocol that aims to produce $[\![xy]\!]_t$ from $[\![x]\!]_t$ and $[\![y]\!]_t$. However, before we move into that, we provide in this section a bit of intuition about why is it the case that none of the attacks proposed above is relevant. In a nutshell, the reason lies in the fact that, the notion of d-consistency from Definition 2.3 in Section 2.2.1, is only concerned with the consistency of the shares held by honest parties.

Distributing shares of degree t inconsistently. An actively corrupt party can misbehave when acting as a dealer in Shamir secret-sharing, and can choose to send arbitrary values $(s_1, \ldots, s_n)$ that are not t-consistent to the parties. This is not a problem nonetheless, or rather, an adversary can cause the exact same effect even if the dealer is honest, as we now show.

For simplicity in the notation, assume that the corrupt parties are $P_1, \dots, P_t$. Since $n = 2t+1$, there are exactly $t+1$ honest parties $P_{t+1}, \dots, P_n$. Let $f(\mathtt{X}) \in \mathbb{F}_{\leq t}[\mathtt{X}]$ be the unique polynomial of degree at most t such that $f(\alpha_i) = s_i$ for $i = t+1, \dots, n$. Since the dealer is actively corrupt, the adversary knows $f(\mathtt{X})$ and therefore it knows $s'_i = f(\alpha_i)$ for $i = 1, \dots, t$. In particular, from Definition 2.3, the parties hold t-consistent shares $[\![s]\!]_t = (s'_1, \dots, s'_t, s_{t+1}, \dots, s_n)$, where $s = f(\alpha_0)$.

In conclusion, even if the dealer is actively corrupt, any set of shares it sends will be by definition t-consistent since there are exactly $t+1$ parties and the shares these parties receive uniquely define a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq t}[\mathtt{X}]$.

Distributing shares of degree $2\mathtt{t}$ inconsistently. Assume for simplicity in the notation that the corrupt parties are $P_1, \dots, P_t$. As before, an actively corrupt dealer can misbehave and choose to send arbitrary values $(s_{t+1}, \dots, s_n)$ to the honest parties. Since $n = 2t+1$ (this also works for $n \geq 2t+1$), for any secret s there exists a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq 2t}[\mathtt{X}]$ such that $f(\alpha_0) = s$ and $f(\alpha_i) = s_i$ for $i = t+1, \dots, n$. Since the adversary knows $f(\mathtt{X})$, it knows in particular $f(\alpha_j)$ for $j = 1, \dots, t$. This means, according to Definition 2.3, that the parties *trivially* hold $2t$-consistent shares of *any* secret.

As in Remark 2.1, this is not a good thing, since it means the adversary can change the corrupt parties' shares in order to obtain sharings of different values. However, as pointed out before, this will be acceptable in our protocol: the concrete effect that this type of attack will have in the overall protocol is that the adversary will be able to add errors to the output of secure multiplications, but the correctness of these will be verified with a simple protocol as described in Section 2.5.4.

Shares of degree $\mathtt{t}$ and $2\mathtt{t}$ having different secrets. From the two notes above we see that the adversary cannot, by definition, distribute shares t or $2t$-inconsistently. The last attack it could carry out then in the protocol for preprocessing double-sharings is that the secret s in the shares of degree t is not the same as the secret s' in the shares of degree $2t$. Once again, although the shares of degree t uniquely define a secret s, the shares of degree $2t$ are consistent with any possible secret, so there is not even an s' defined. As mentioned before, this gap will result in a concrete attack in the multiplication protocol from Section 2.5.3 below, which can be prevented using certain checks after the multiplication has been performed as explained in Section 2.5.4.

2.5.3 Online Phase

We now move to the description of the online phase of our protocol. Recall that the function to be evaluated, $F : \mathbb{F}^n \to \mathbb{F}$, is given by an arithmetic circuit over $\mathbb{F}$. Let $x_i \in \mathbb{F}$ be the input of party P_i. As in previous sections, the protocol consists of having the parties obtain shares $[\![x_1]\!]_t, \dots, [\![x_n]\!]_t$ of their inputs, followed by methods to obtain from two given shared values $[\![x]\!]_t$ and $[\![y]\!]_t$, shares of $[\![x+y]\!]_t$ and $[\![x \cdot y]\!]_t$. This allows the parties to obtain shares of all intermediate values of the computation, until shares of

the output $[\![z]\!]_t$, with $z = F(x_1, \ldots, x_n)$, are produced. At this point the parties can simply reconstruct this result to learn the output of the computation.

The input phase in which the parties obtain shares of their inputs, is handled in exactly the same way as in Section 2.4.3, that is, for a party P_i to provide input x_i, we assume the parties have a random shared value $[\![r]\!]_t$ where $r \in_R \mathbb{F}$ is only known by P_i. This could be easily adapted from the protocol to obtain double-shares. Then P_i uses the broadcast channel to send $e = x_i - r$ to all the parties, who define $[\![x_i]\!] = [\![r]\!] + e$ as their shares of x_i.

Given $[\![x]\!]_t$ and $[\![y]\!]_t$, it is straightforward for the parties to obtain $[\![x + y]\!]_t$ given the linearity properties of Shamir secret-sharing. On the other hand, to obtain $[\![x \cdot y]\!]$, the parties first execute the following protocol, which is exactly the same as the one presented in Section 2.3.2.1.

Actively secure multiplication protocol via double-sharings

Preprocessing: Double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$.
Input: Secret-shared values $[\![x]\!]_t$ and $[\![y]\!]_t$.
Output: $[\![z = x \cdot y]\!]_t$.

Protocol: The parties execute the following

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
2. The parties call the protocol $\Pi_{\text{PublicRec}}$ from Section 2.5.1 to learn a.
3. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

The main difference of this protocol with respect to the one from Section 2.4.1 is the set of guarantees this one provides. In the protocol from Section 2.4.1, we could prove that the parties obtain the correct $[\![x \cdot y]\!]_t$ at the end of the protocol execution. In our case here, we will not be able to prove this. This is because, as has been mentioned before, in our case where $t < n/2$, opening degree-$2t$ shares cannot be done while ensuring the integrity of the underlying secret, which is not the case when $t < n/3$. Here, we show the following:

Proposition 2.1. *Let $[\![x]\!]_t$ and $[\![y]\!]_t$ be inputs to the multiplication protocol above. Then, at the end of the protocol execution, the parties get shares $[\![xy + \delta]\!]_t$, where $\delta \in \mathbb{F}$ is a value known to the adversary.*

Proof. The result of the call to protocol $\Pi_{\text{PublicRec}}$ is $a + \delta = (xy - r) + \delta$, so the parties compute $[\![zy + \delta]\!] \leftarrow [\![r]\!] + (xy - r) + \delta$ in the last step of the protocol. □

2.5.4 Verification Phase

As we showed in Proposition 2.1, an active adversary can inject errors to the result of secure multiplications. This is of course a problem since correctness of the computation is not guaranteed anymore. Furthermore, it can lead to concrete privacy leakage attacks.

For example, if $n = 3$ and $F(x_1, x_2, x_3) = (x_1 \cdot x_2) \cdot x_3$, a corrupt party could add a non-zero error in the first multiplication so that the output of the computation is $(x_1 \cdot x_2 + \delta) \cdot x_3 = F(x_1, x_2, x_3) + \delta \cdot x_3$. If the adversary corrupts P_1 and sets $x_1 = 0$, then the correct output is always 0 regardless of the inputs of the other parties, so the adversary should not be able to learn anything about these according to the security definition of MPC. However, with the attack above, the result becomes $0 + \delta \cdot x_3$, which in particular means that the adversary can learn x_3 by multiplying the result with δ^{-1}.

Given this, it is imperative that, before the parties reconstruct the final result, they check that no errors have been introduced in any of the multiplications involved in the computation. The more concrete setting is the following. The parties have (t-consistent) shares $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$, where z is supposed to be equal to $x \cdot y$. However, due to adversarial behavior, it is actually the case that $z = x \cdot y + \delta$ for some adversarially chosen value $\delta \in \mathbb{F}$, and the parties want to check that $\delta = 0$. At a high level, the method we will present to address this issue consists of the following. First, the parties generate a triple such as the one above $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$, where $c = a \cdot b + \epsilon$ for some adversarially chosen value ϵ, and in addition, $a, b \in \mathbb{F}$ are uniformly random and unknown to the adversary. Then, the parties will make use of this triple of shared values to check the correctness of z.

Generating the tuple $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$ is straightforward given the tools we have presented thus far: getting $[\![a]\!]_t$ and $[\![b]\!]_t$ can be done by a simple modification of the protocol from Section 2.5.2 to obtain double-sharings, without considering the degree-$2t$ part, and $[\![c]\!]_t$ can be obtained from $[\![a]\!]_t$ and $[\![b]\!]_t$ by applying the multiplication protocol from Section 2.5.3. Now, using such tuple to check the correctness of z is done with the following protocol. Below, we let $\mathcal{F}_{\text{Coin}}$ denote a functionality that returns public random values to all the parties.

Verifying secure multiplications

Preprocessing: A tuple $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$, where $c = a \cdot b + \epsilon$ for some value $\epsilon \in \mathbb{F}$ known by the adversary, and $a, b \in \mathbb{F}$ are uniformly random and unknown to the adversary.
Input: Secret-shared values $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$, where $z = x \cdot y + \delta$ for some value $\delta \in \mathbb{F}$ known by the adversary.
Output: A signal `pass`/`fail`.

Protocol: The parties execute the following

1. The parties call $s \leftarrow \mathcal{F}_{\text{Coin}}$;
2. The parties compute locally $[\![d]\!]_t \leftarrow [\![x]\!]_t - s \cdot [\![a]\!]_t$ and $[\![e]\!]_t \leftarrow [\![y]\!]_t - [\![b]\!]_t$
3. The parties call the protocol $\Pi_{\text{PublicRec}}$ from Section 2.5.1 to reconstruct d and e.
4. The parties compute locally $[\![w]\!] \leftarrow s \cdot e \cdot [\![a]\!] + d \cdot [\![b]\!] + s \cdot [\![c]\!] + d \cdot e - [\![z]\!]$.
5. The parties call the protocol $\Pi_{\text{PublicRec}}$ to reconstruct w, and check that $w = 0$. If this is the case, output `pass`. Else, output `fail`.

Proposition 2.2. *Let $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$ with $z = xy + \delta$ be an input to the protocol above. Then, if $\delta \neq 0$, the probability that the protocol results in the parties outputting `pass` is at most $1/|\mathbb{F}|$. Furthermore, nothing about x or y is learned by the adversary after the execution of the protocol.*

Proof. Since $d = x - s \cdot a$, $e = y - b$, $z = x \cdot y + \delta$ and $c = a \cdot b + \epsilon$, we have that

$$\begin{aligned} w =& s \cdot e \cdot a + d \cdot b + s \cdot c + d \cdot e - z \\ =& s \cdot (y - b) \cdot a + (x - s \cdot a) \cdot b + s \cdot (a \cdot b + \epsilon) + (x - s \cdot a)(y - b) - (x \cdot y + \delta) \\ =& sya - sba + xb - sab + sab + s\epsilon + xy - xb - say + sab - xy - \delta \\ =& s \cdot \epsilon - \delta. \end{aligned}$$

From this, we see that $w = 0$ if and only if $s \cdot \epsilon - \delta = 0$, or $s \cdot \epsilon = \delta$. Assume that $\delta \neq 0$ and nevertheless $w = 0$. Then $\epsilon \neq 0$ since otherwise $\delta = s \cdot 0 = 0$, but this implies that $s = \delta/\epsilon$, which happens with probability $1/|\mathbb{F}|$ since s is uniformly random and sampled independently of δ and ϵ. □

Remark 2.3. *The verification step above can be improved so that, when many checks are performed simultaneously (as expected in an actual secure computation scenario), the overhead in communication by performing this check is very small. More concretely, this overhead can be made* sub-linear *in the number of multiplications being checked thanks to the novel techniques presented in [57].*

2.6 Passive Security for Dishonest Majority

All the protocols we have seen so far assume that the adversary corrupts strictly less than $n/2$ or $n/3$ parties. However, if such assumption is violated, privacy would break, which can be seen from the fact that if the adversary corrupts more parties than the threshold used for Shamir secret sharing then the underlying secret is revealed.

It would be ideal if we could design protocols where, from the view of *each single party*, their input is kept private even if *all* of the other parties collude against the single party. In other words, we would like to guarantee security under an adversary corrupting t parties, even if t grows as large as $n - 1$, leaving only one honest party. This setting, where the only bound on t is $t < n$, is called *dishonest majority* since in principle a majority of the parties could be corrupt; this is in contrast to the case in which $t < n/2$ where the majority of parties are guaranteed to be honest.

In this section we explore an MPC protocol in the dishonest majority setting with passive security, which means that each party's input is secure even if all the other parties collude, as long as these parties follow the protocol specification. In Section 2.7 we explore the case of active security, which ensures privacy even if the other parties misbehave. This is the strongest possible setting, but it is also, naturally, the most expensive.

Another important aspect of the dishonest majority setting is that it includes the relevant case in which $n = 2$ and $t = 1$, since in this case none of the bounds $t < n/2$ nor $t < n/3$ hold. This particular scenario appears in many different applications, so it must be considered as well.

2.6.1 Additive Secret-Sharing

We assume that $t = n - 1$. Unlike the results from Section 2.5, assuming t reaches the maximum possible bound is only done for the sake of clarity in the notation, instead of being a security feature. All of our results carry over if $t < n - 1$.

Additive secret-sharing

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \ldots, P_n$ as follows.

1. Sample $s_1, \ldots, s_{n-1} \in_R \mathbb{F}$ and define $s_n = s - (s_1 + \cdots + s_{n-1})$.
2. Distribute the value s_i to party P_i, for $i \in [n]$.

In other words, the tuple $(s_1, \ldots, s_n)$ is uniformly random in $(\mathbb{Z}/2^k\mathbb{Z})^n$, constrained to $s = s_1 + \cdots + s_n$. As a result, for every set $A \subseteq [n]$ with $|A| \leq n - 1$, the distribution of the shares $\{s_i\}_{i \in A}$ is uniformly random, and in particular, it is independent of the secret s.

When the parties have shares as above, we denote $[\![s]\!] = (s_1, \ldots, s_n)$. Notice that, given two shared values $[\![x]\!]$ and $[\![y]\!]$, the parties can locally add/subtract their shares of these values to obtain $[\![x \pm y]\!]$. Furthermore, given a value $c \in \mathbb{F}$ known to all the parties, the parties can locally obtain $c\,[\![x]\!]$ by multiplying c to every share, and they can obtain $[\![x \pm c]\!]$ by asking only one party, say P_1, to add/subtract the value of c to its share.

2.6.2 Protocols for Secure Multiplication

As in previous sections, we obtain a secure computation protocol by asking the parties to distribute shares of their inputs (which is trivial since, if x is known to party P_i, the parties can non-interactively obtain $[\![x]\!]$ by writing $x = x_1 + \cdots + x_n$ where $x_j = 0$ if $j \neq i$, and $x_i = x$), followed by the parties executing different subprotocols to obtain $[\![x + y]\!]$ and $[\![xy]\!]$ from shared values $[\![x]\!]$ and $[\![y]\!]$. As we mentioned before, the case of addition can be easily handled by the parties adding their shares locally. However, obtaining $[\![xy]\!]$ is, as usual, a much harder task. Furthermore, what complicates matters in the dishonest majority scenario is that, as we have mentioned already in Section 1.3.3, this setting requires the use of tools from the public-key cryptography domain, which are, in their nature, much more expensive than the information-theoretic techniques we have been making use of so far.

2.6.2.1 Product-to-Sum Conversion

We begin by reducing the problem of obtaining $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$ to a simpler problem. Write $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$, so

$$xy = (x_1 + \cdots + x_n)(y_1 + \cdots + y_n) = \sum_{i=1}^{n} x_i y_i + \sum_{i,j \in [n], i \neq j} x_i y_j.$$

The goal is to obtain shares of each of these summands, which can in turn be added locally to obtain shares of $x \cdot y$. Since each term of the form $x_i y_i$ is known by party P_i, the parties can obtain $[\![x_i y_i]\!]$ trivially by setting P_i's share to be $x_i y_i$, and the setting the other shares to be 0. The main challenge lies on the terms of the form $x_i y_j$ for $i \neq j$, since one factor is known by one party P_i, and the other factor is known by a different party P_j.

Assume the existence of a protocol for *product-to-sum conversion*, in which P_i inputs x_i, P_j inputs y_j, and P_i and P_j receive z_i and z_j respectively, with these values being uniformly random constrained to $x_i y_j = z_i + z_j$. With such a tool, the parties can obtain $[\![x_i y_j]\!]$ by letting P_i and P_j execute the product-to-sum protocol, obtaining z_i and z_j, and defining the other parties' shares to be zero.

From our observations above, the parties can locally compute $[\![x_i y_i]\!]$ for $i \in [n]$, and, with the help of a product-to-sum conversion protocol, they can also compute $[\![x_i y_j]\!]$ for $i, j \in [n]$ and $i \neq j$. As a result, they can compute shares of x and y as follows.

$$[\![xy]\!] = \sum_{i=1}^{n} [\![x_i y_i]\!] + \sum_{i,j \in [n], i \neq j} [\![x_i y_j]\!] .$$

2.6.2.2 Product-to-Sum Conversion Based on Homomorphic Encryption

From the previous section, we see that, in order for the parties to compute $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$, it suffices to design a two-party protocol for product-to-sum conversion. Recall that, in such protocol two parties, which we denote by P_1 and P_2, each input a value x_1 and x_2, and they receive z_1 and z_2, which are uniformly random values constrained to $x_1 x_2 = z_1 + z_2$. In this section we show how such primitive can be instantiated making use of Additively Homomorphic Encryption, or AHE for short. We remark that our aim is simply to provide intuition on how this can be done, so we do not provide a lot of details nor present a lot of formalism.

An encryption scheme consists of an encryption and decryption algorithms $\mathsf{Enc}_{\mathsf{pk}}(\cdot)$ and $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$ such that, intuitively:

1. $\mathsf{Enc}_{\mathsf{pk}}(m)$ does not leak anything about the message m if the key pair $(\mathsf{sk}, \mathsf{pk})$ is sampled by a sampling algorithm.

2. If $c = \mathsf{Enc}_{\mathsf{pk}}(m)$, then $m = \mathsf{Dec}_{\mathsf{sk}}(c)$.

In an *additively homomorphic encryption scheme* (AHE), in addition, there is a way to "add/subtract" the ciphertexts to obtain encryptions of the respective operations on the plaintexts, that is, given $c = \mathsf{Enc}_k(x)$ and $d = \mathsf{Enc}_k(y)$, it is possible to compute $d \pm e = \mathsf{Enc}_k(x \pm y)$. An example of an additively homomorphic encryption scheme is Paillier's [70]. Such an encryption scheme can be used to instantiate the product-to-sum conversion primitive as follows.

Secure multiplication

Input: Party P_i has input x_i, for $i \in \{1, 2\}$
Setup: Key pair (sk, pk) with pk known by P_1 and P_2, and sk known by P_1. **Output:** P_i gets z_i for $i \in \{1, 2\}$, where (z_1, z_2) is uniformly random constrained to $z = z_1 + z_2$.
Protocol:

1. P_1 sends $c = \mathsf{Enc}_{\mathsf{pk}}(x_1)$ to P_2.
2. P_2 samples z_2 and sends $d = x_2 \cdot c - \mathsf{Enc}_{\mathsf{pk}}(z_2)$ to P_1
3. P_1 computes $z_1 = \mathsf{Dec}_{\mathsf{sk}}(d)$

We see that P_2 does not learn anything about x_1 since it only receives $c = \mathsf{Enc}_{\mathsf{pk}}(x_1)$, which by the properties of the encryption scheme, completely hides x_1. On the other hand, P_1 decrypts $z_1 = x_2 \cdot x_1 - z_2$, as desired.

2.6.3 Preprocessing Model

The tools we have described so far enable the parties to securely compute any arithmetic circuit comprised of additions and multiplications over the finite field $\mathbb{F}$: the parties hold additive shares of the inputs to the computation, addition gates can be processed non-interactively, and multiplication gates make use of the method from Section 2.6.2, which relies on an AHE scheme. Unfortunately, this approach would provide poor efficiency when compared to the other MPC protocols we have explored in precious sections. These protocols, to process multiplication gates, only required simple arithmetic over $\mathbb{F}$, while the use of AHE techniques, and in general, the different tools used in practice to perform secure multiplication in the dishonest majority settings, is considerably more expensive.

Unfortunately, the use of these techniques is unavoidable when the adversary corrupts more than a majority of the parties, even if the corruption is passive. Given this limitation, an standard approach to limit its effect in practice is to split the computation in two phases: an offline phase, also called preprocessing phase, which is independent of the inputs of the computation, and an online phase, which now makes use of the inputs. The online phase is designed to be much more efficient than the protocol we have sketched so far. In fact, this phase typically achieves information-theoretic security and only makes use of simple arithmetic operations, so it achieves high efficiency. This way, by pushing the offline phase to a much earlier time before the parties set their inputs, say, when the parties are idle, the execution of the MPC protocol is much more efficient from a practical perspective, counting the latency from the time the parties provide input to the time they produce the output.

2.6.4 Offline Phase

To accelerate the computation of secure multiplications in the online phase, the parties will need to produce a set of multiplication triples. A multiplication triple, also called Beaver triple, is a tuple of the form $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, where a, b are uniformly random in

$\mathbb{F}$ and unknown to any party, and $c = a \cdot b$. In the offline phase, the parties generate one such tuple for every multiplication gate expected in the arithmetic circuit under consideration. Notice that these tuples only contain random data and do not make use of the inputs of the computation, which are used later in the online phase. As we have mentioned, this is crucial for the preprocessing paradigm to provide any benefit since, as we will see, it is in the production of these tuples where the parties spent most of their computational resources, in order to be able to compute the online phase in a much more efficient way.

To generate a multiplication triple, the parties can proceed as follows.

Generating multiplication triples with passive security

Output: A multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$, where a, b are uniformly random in $\mathbb{F}$ and unknown to the adversary, and $c = a \cdot b$.
Protocol:

1. Each party P_i samples $a_i, b_i \in_R \mathbb{F}$. This leads to sharings $[\![a]\!] = (a_1, \dots, a_n)$ and $[\![b]\!] = (b_1, \dots, b_n)$.
2. The parties execute a multiplication protocol to obtain $[\![c]\!]$, where $c = ab$.
3. The parties output the shares $([\![a]\!], [\![b]\!], [\![c]\!])$.

Notice that $a = a_1 + \cdots + a_n$ and $b = b_1 + \cdots + b_n$ look uniformly random and unknown to the adversary since there is at least one honest party contributing with a uniformly random summand in each of these expressions.

2.6.5 Online Phase

Now we show how the parties can make use of a multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ to securely obtain in the online phase $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$, in a much more efficiently way than simply using a multiplication protocol like the one from Section 2.6.2.

Multiplication based on multiplication triples

Input: Shared values $[\![x]\!]$ and $[\![y]\!]$.
Output: Shared value $[\![z]\!]$, where $z = xy$.
Preprocessing: Multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$
Protocol:

1. The parties compute locally $[\![d]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![e]\!] \leftarrow [\![y]\!] - [\![b]\!]$
2. The parties send their shares of $[\![d]\!]$ and $[\![e]\!]$ to each other to learn d and e.[a]
3. The parties compute locally $[\![z]\!] \leftarrow d\,[\![b]\!] + e\,[\![a]\!] + [\![c]\!] + de$.

[a] This can be optimized by asking parties to send their shares to one single party, say P_1, who reconstructs d and e and announces these values to the parties.

To see that the protocol works as intended, observe first that, given that $d = x - a$, $e = y - b$, and $c = ab$, it holds that $db + ea + c + de = xy$, so the protocol indeed produces shares of $x \cdot y$. On the other hand, nothing is leaked about x or y since the only

values that are opened during the protocol execution are d and e, which look uniformly random to the adversary given that a and b are random and unknown to the adversary. In particular, notice that the protocol is perfectly secure. Furthermore, the protocol shines from its simplicity, involving only the reconstruction of two secret-shared values and simple local arithmetic.

2.7 Active Security for Dishonest Majority

In Section 2.6 we described a protocol for secure computation in the dishonest majority setting, assuming the corruption is passive. However, that protocol is not secure if the corrupted parties behave maliciously. As an example of what goes wrong when the corruption is active, consider a secret-shared value $[\![x]\!] = (x_1, \ldots, x_n)$. Suppose that, at reconstruction time, the corrupt parties P_i for $i \in \mathcal{C}$ change their share from x_i to $x_i' = x_i + \delta_i$ for some $\delta_i \in \mathbb{F}$. Since there is no method for the honest parties to detect this change, the reconstructed value would be $\sum_{i\in\mathcal{C}} x_i' + \sum_{i\in\mathcal{H}} x_i = \sum_{i=1}^{n} x_i + \sum_{i\in\mathcal{C}} \delta_i = x + \delta$, where $\delta = \sum_{i\in\mathcal{C}} \delta_i$. In particular, the adversary can cause the reconstruction to lead to an incorrect value, similar to what happened in Section 2.5.1 when shares of degree $2t$ had to be reconstructed. This was fixed in the protocol for $t < n/2$ by noticing that the ultimate effect that this attack has is that the adversary can affect the result of secure multiplications, which can be checked with a verification protocol. In our current dishonest majority setting this "share-modification" attack is much more devastating, since it does not only affect multiplications but every single step that requires an opening.

To fix the issue of the adversary modifying the corrupt parties' shares, we need to add certain "redundancy" to the sharings, comparable to the redundancy present in Shamir shares for $t < n/2$. This comes in the form of a tool called *Message Authentication Codes*, or MACs for short. This term is taken from the symmetric key cryptography literature, and in general, we use it to represent a primitive that guarantees *data integrity*, that is, that an adversary cannot modify certain piece of data without being detected. This is precisely the type of tool we need in our current context to disallow the adversary from modifying the shares of the corrupt parties.

MACs are used in order to authenticate the parties' shares so that they cannot change them at a later point, and the way these are used is divided into two. In the first approach, described in Section 2.7.1, each single party has an extra piece of information to check the integrity of every other party's share at reconstruction time. The second approach, described in Section 2.7.2, consists of all the parties jointly having a way of checking not the integrity of each individual share, but rather the integrity of the reconstructed *secret*. This works better for a large number of parties since it is not necessary for every party to hold authentication information of the share held by each other party.

Both of these methods are based on the following basic idea for ensuring integrity. Given a piece of data $m \in \mathbb{F}$, compute a random value $\alpha \in_R \mathbb{F}$, and let $\tau = \alpha \cdot m$. Integrity is checked by verifying that m, multiplied by the value α, results in τ. If m is modified as $m' = m + \delta$ and τ is modified as $\tau' = \tau + \epsilon$, then the only way in which $\alpha m'$ can equal τ' is if $\alpha\delta = \epsilon$. If $\delta \neq 0$, that is, if the data m was indeed modified to a different m', then this equation translates to $\alpha = \epsilon/\delta$. If we somehow guarantee that ϵ and δ are independent

of α, then, given that α is uniformly random in $\mathbb{F}$, this equation can only be satisfied with probability $1/|\mathbb{F}|$.

2.7.1 Integrity via Pairwise MACs

We begin by presenting the construction in which each party has a way to check the share announced by each other party. This construction was proposed initially in [19].

Additive secret-sharing with pairwise MACs

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \ldots, P_n$ with pairwise MACs as follows.

1. Sample $s_1, \ldots, s_n \in \mathbb{F}$ uniformly at random constrained to $s = s_1 + \cdots + s_n$.
2. For each $i, j \in [n]$, the dealer does the following:
   - Sample $(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)}) \in_R \mathbb{F}$.
   - Compute $\tau_{ji}^{(s)} = \alpha_{ij}^{(s)} s_j + \beta_{ij}^{(s)}$.
3. Distribute the tuple $\mathbf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$ to party P_i, for $i \in [n]$.

Throughout this subsection we will denote by $\langle s \rangle$ the situation in which the parties have shares $(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ of s, with the additional redundancy, as above. Notice that this type of sharing does not leak anything about s to the adversary.

2.7.1.1 Reconstructing secret-shared values

Now, assume the parties have shares $(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ of some value s, with $\mathbf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$. At reconstruction time, each party P_i announces $(s_i', \{\tau_{ij}'^{(s)}\}_{j\in[n]})$, where $(s_i', \{\tau_{ij}'^{(s)}\}_{j\in[n]}) = (s_i, \{\tau_{ij}^{(s)}\}_{j\in[n]})$ for at least one $i \in [n]$, which corresponds to the indexes of the honest parties who announce their shares correctly. To check the validity of these values, each party P_i executes the following.

Reconstructing shared values with pairwise MACs

Given $\langle s \rangle = (\mathbf{s}_1, \ldots, \mathbf{s}_n)$, with $\mathbf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$, the parties reconstruct s as follows:

1. At reconstruction time, each party P_i sends s_i to all the other parties and $\tau_{ij}^{(s)}$ to party P_j.
2. Each party P_i checks if for all $j \in [n]$ it holds that $\tau_{ji}^{(s)} = \alpha_{ij}^{(s)} s_j + \beta_{ij}^{(s)}$. If so then P_i reconstructs the value $s = s_1 + \cdots + s_n$. Else, the parties abort.

Proposition 2.3. *If the protocol above does not result in abort, then its output is the correct s with probability at least $1 - 1/|\mathbb{F}|$.*

Proof. Assume the parties did not abort, and let $i_0 \in \mathcal{H}$. Let $(s'_j, \tau'^{(s)}_{ji_0})$ for $j \in \mathcal{C}$ be the actual values sent by the corrupt parties to P_{i_0}. Since P_{i_0} did not abort, it holds that, for all $j \in \mathcal{C}$, $\tau'^{(s)}_{ji_0} = \alpha^{(s)}_{i_0 j} s'_j + \beta^{(s)}_{i_0 j}$. Let us write $s'_j = s_j + \delta_j$ and $\tau'^{(s)}_{ji_0} = \tau^{(s)}_{ji_0} + \epsilon_j$ for $j \in \mathcal{C}$. Recalling that $\tau^{(s)}_{ji_0} = \alpha^{(s)}_{i_0 j} s_j + \beta^{(s)}_{i_0 j}$, the equations above are equivalent to $\epsilon_j = \alpha^{(s)}_{i_0 j} \delta_j$ for $j \in \mathcal{C}$.

Now, suppose that $\delta_{j_0} \neq 0$ for some $j_0 \in \mathcal{C}$. From the above we have that $\alpha^{(s)}_{i_0 j_0} = \frac{\epsilon_{j_0}}{\delta_{j_0}}$. It is easy to see that the view of the adversary before the execution of the reconstruction protocol is independent of $\alpha_{i_0 j}$ for $j \in \mathcal{C}$ since the adversary only sees $\tau_{ji_0} = \alpha^{(s)}_{i_0 j} \cdot s_j + \beta_{i_0 j}$, but the uniformly random value $\beta_{i_0 j}$ is unknown to the adversary, which perfectly hides the term $\alpha^{(s)}_{i_0 j} \cdot s_j$. From this, we see that the errors δ_{j_0} and ϵ_{j_0} added by the adversary are independent of the uniformly random value $\alpha^{(s)}_{i_0 j_0}$, so the equation $\alpha^{(s)}_{i_0 j_0} = \frac{\epsilon_{j_0}}{\delta_{j_0}}$ from above can only be satisfied with probability $1/|\mathbb{F}|$.

We obtain that, except with probability $1/|\mathbb{F}|$, it holds that $\delta_j = 0$ for all $j \in \mathcal{C}$, so the announced shares s_i for $i \in [n]$ are all correct and therefore the reconstructed value is correct as well. □

2.7.1.2 Local Operations

Finally, to show that our secret-sharing scheme is suitable for secure computation, we need to show that basic operations can be handled locally by the parties. This is shown below.

Addition/Subtraction. Assume the parties have two shares values $\langle x \rangle = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ and $\langle y \rangle = (\mathtt{y}_1, \ldots, \mathtt{y}_n)$, with $\mathtt{x}_i = (x_i, \{(\alpha^{(x)}_{ij}, \beta^{(x)}_{ij})\}_{j\in[n]}, \{\tau^{(x)}_{ij}\}_{j\in[n]})$ and $\mathtt{y}_i = (y_i, \{(\alpha^{(y)}_{ij}, \beta^{(y)}_{ij})\}_{j\in[n]}, \{\tau^{(y)}_{ij}\}_{j\in[n]})$. According to our description of the sharing procedure, the values $\alpha^{(x)}_{ij}$ and $\alpha^{(y)}_{ij}$ are sampled separately when sharing the value x and y. However, to make this scheme compatible with local addition and subtraction, we need to assume that $\alpha^{(x)}_{ij}, \alpha^{(y)}_{ij}$ for $i, j \in [n]$ are sampled uniformly at random with $\alpha_{ij} := \alpha^{(x)}_{ij} = \alpha^{(y)}_{ij}$, or, more precisely, that the dealer samples and distributes $\alpha_{ij} \in_R \mathbb{F}$ once, and uses these to compute the values $\{\tau^{(z)}_{ij}\}_{j\in[n]}$ for every new secret-shared value z.

To obtain $\langle x + y \rangle$, each party P_i defines $\mathtt{z}_i$ as $(z_i, \{(\alpha_{ij}, \beta^{(z)}_{ij})\}_{j\in[n]}, \{\tau^{(z)}_{ij}\}_{j\in[n]})$, where:

$$\begin{cases} z_i = x_i + y_i \\ \beta^{(z)}_{ij} = \beta^{(x)}_{ij} + \beta^{(y)}_{ij}, \ j \in [n] \\ \tau^{(z)}_{ij} = \tau^{(x)}_{ij} + \tau^{(y)}_{ij}, \ j \in [n]. \end{cases}$$

Subtraction works in a similar fashion.

Canonical shares of public values. Let c be a publicly known value, that is, a value known by all the parties. The parties can obtain shares $\langle c \rangle$ by defining $c_i = (c_i, \{(\alpha_{ij}, \beta_{ij}^{(c)})\}_{j\in[n]}, \{\tau_{ij}^{(c)}\}_{j\in[n]})$ as follows.

$$c_i = \begin{cases} 0 & \text{for } i \in [n] \setminus \{1\} \\ c & \text{for } i \in \{1\} \end{cases}$$
$$\beta_{ij}^{(c)} = \begin{cases} 0 & \text{for } i \in [n], j \in [n] \setminus \{1\} \\ -\alpha_{ij} \cdot c & \text{for } i \in [n], j \in \{1\} \end{cases}$$
$$\tau_{ij}^{(c)} = \{0 \quad \text{for } i, j \in [n] \ .$$

Each party P_i can compute its share c_i locally, and, moreover, it can be easily checked that $\tau_{ij}^{(c)} = \alpha_{ji} c_i + \beta_{ij}^{(c)}$ for $i, j \in [n]$, as required by the syntax of the secret-sharing scheme.

2.7.2 Integrity via Global MACs

In the previous method from Section 2.7.1 to add integrity to the basic additive secret-sharing scheme $[\![s]\!] = (s_1, \ldots, s_n)$, for each (ordered) pair of parties (P_i, P_j), P_i could verify the correctness of P_j's share s_j by means of a key $(\alpha_{ij}, \beta_{ij}^{(s)})$ and a *tag* $\tau_{ji}^{(s)}$ held by s. This way, if any of the announced shares is incorrect, the check the honest party performs would fail, which results in the parties aborting. However, the main drawback with this approach is that each party P_i must hold a key and tag with respect to every other single party P_j, which ultimately means that the size of each party's share grows linearly with n (more concretely, each party's share consists of $1 + 3n$ elements in $\mathbb{F}$.) This may matter little if n is relatively small, which is the case in the relevant setting of two-party computation, for example. However, for a large number of parties, a share size that grows as $\Omega(n)$ may be just too large.

Given the above, we present in this section a different method to ensure integrity that only adds a small overhead to the size of each share with respect to the basic additive secret-sharing scheme. This was first proposed in the work of [43]. To provide intuition on how this method works, recall that the main goal is to add some redundant information to a given additively-shared value $[\![s]\!] = (s_1, \ldots, s_n)$ so that, when the parties announce their shares $(s'_1, \ldots, s'_n)$, the parties can verify that the reconstructed value $s' = s'_1 + \cdots + s'_n$ is indeed correct. The method from Section 2.7.1 ensures this by providing the parties with a method for checking that each party's share s'_i is announced correctly, that is, $s_i = s'_i$, which implies that $s' = s$. However, a crucial observation is that, ultimately, what is desired is that $s' = s$, which can happen even if $s'_i = s_i$ for some values of i. Hence, the core idea is to add integrity not to each individual share, but rather to the shared value s itself. This is described in detail below. Notice that each party's share is now made of only 3 elements in $\mathbb{F}$.

Additive secret-sharing with global MACs

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \ldots, P_n$ with a global MAC as follows.

1. Sample $s_1, \ldots, s_n \in \mathbb{F}$ uniformly at random constrained to $s = s_1 + \cdots + s_n$.
2. Sample $\alpha_1^{(s)}, \ldots, \alpha_n^{(s)} \in_R \mathbb{F}$, and let $\alpha^{(s)} = \sum_{i=1}^n \alpha_i^{(s)}$.
3. Sample $\gamma_1^{(s)}, \ldots, \gamma_n^{(s)} \in \mathbb{F}$ uniformly at random constrained to $\alpha^{(s)} \cdot s = \sum_{i=1}^n \gamma_i^{(s)}$.
4. Distribute the tuple $\mathbf{s}_i = (s_i, \alpha_i^{(s)}, \gamma_i^{(s)})$ to party P_i, for $i \in [n]$.

For the sake of this subsection, we denote by $\langle s \rangle$ the situation in which the parties have shares $(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ of s, with the additional redundancy, as above. Intuitively, we may write $\langle s \rangle = (\llbracket s \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot s \rrbracket)$.

2.7.2.1 Reconstructing Secret-Shared Values

Partial openings. Assume now that the parties have additive shares $\langle s \rangle = (\mathbf{s}_1, \ldots, \mathbf{s}_n)$ of some value s. By *partially opening* $\langle s \rangle$, we mean the following:

1. Each party P_i sends their additive share s_i of $\llbracket s \rrbracket$ to P_1.
2. P_1 computes $s = s_1 + \cdots + s_n$ and then he broadcasts P_1 to all parties.

This basic opening does not ensure the correct value is reconstructed, hence the name partial. In this case the adversary can cause the reconstruction to be $s + \delta$, where, furthermore, δ can depend on s if P_1 is corrupted.

Commit-and-open. Before we describe the mechanism for the parties to reconstruct values correctly, we describe another type of opening that does not necessarily ensure that the reconstructed value is correct, but at least guarantees that the added adversarial error is independent of the secret. For this construction we will need to make use of a cryptographic tool known as a *commitment*. For a formal treatment on these see for example [38]. Intuitively, a commitment scheme is a pair (Commit, Open) where Commit(m, r) allows a participant to "commit" to a value m using a uniformly random "key" r, and Open(m, r, c) checks whether the commitment c corresponds to m and r. The basic properties of such a scheme are (1) Commit(m, r), for a uniformly random r, does not reveal anything about m and (2) given $c =$ Commit(m, r), it is not possible to find m' and r' with $m \neq m'$ such that Open(m', r', c) reports that the commitment c corresponds to m', r'. An effective construction of such a scheme consists of Commit$(m, r) = \mathrm{H}(m \| r)$, where H is a cryptographic hash function.

With this tool at hand, the parties commit-and-open to a shared value $\llbracket z \rrbracket = (z_1, \ldots, z_n)$ as follows.

1. Each party P_i samples r_i and computes the commitment $c_i = \text{Commit}(z_i, r_i)$. Then P_i broadcasts c_i.

2. After all these values are broadcast, each party P_i broadcasts (z_i, r_i).

3. The parties check that, for all $i \in [n]$, $\text{Open}(z_i', r_i', c_i')$ accepts, where c_i' and (z_i', r_i') are the values broadcast by P_i in the previous two steps. If $\text{Open}(z_i', r_i', c_i')$ rejects, then the parties abort.

A corrupt party P_i may still lie about its own share by broadcasting $z_i + \delta_i$, so the resulting reconstructed value may still be incorrect. However, P_i only broadcasts $z_i + \delta_i$ after he has broadcasted the commitment c_i, and by the properties of the commitment scheme sketched above, this party cannot announce a different share than the one it has committed to, which means that P_i has chosen δ_i based on the information sent in the first part of the protocol. At this stage only commitments to the shares have been sent, which leak nothing about the shares themselves, so the possible errors δ_i are independent of the other shares and hence independent of the secret, as desired.

Reconstruction. Let $\langle s \rangle = (\mathfrak{s}_1, \ldots, \mathfrak{s}_n)$ be a value shared by the parties. Now we show how to put together the different reconstruction methods from above so that the parties learn s correctly.

Reconstructing shared values with global MACs

Given a shared value $\langle s \rangle = (\llbracket s \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot s \rrbracket)$, the parties reconstruct s as follows:

1. The parties partially open $s' \leftarrow \llbracket s \rrbracket = (s_1, \ldots, s_n)$.
2. The parties compute locally $\llbracket \mu \rrbracket \leftarrow \llbracket \alpha \cdot s \rrbracket - s' \llbracket \alpha \rrbracket$.
3. The parties commit-and-open $\mu' \leftarrow \llbracket \mu \rrbracket$. If $\mu' = 0$, then the parties accept s' as the opened value. Else, the parties abort.

Proposition 2.4. *If the protocol above does not result in abort, then each party outputs s with probability at least $1 - 1/|\mathbb{F}|$.*

Proof. Let us write $s' = s + \delta$, where δ is an additive error introduced by the adversary that might depend on s. Also, let us write $\mu' = \mu + \epsilon$, where ϵ is also an additive error introduced by the adversary, but this, unlike δ, does not depend on the value of the secret μ. We have that

$$\mu' = \mu + \epsilon = (\alpha s - s'\alpha) + \epsilon = \alpha s - (s + \delta)\alpha + \epsilon = \epsilon - \alpha \cdot \delta,$$

so $\mu' = 0$ if and only if $\epsilon - \alpha \cdot \delta = 0$.

Now, assume that $\delta \neq 0$, we would have that $\alpha = \epsilon/\delta$, and since ϵ is chosen independently of α, which is uniformly random, this equation can only be satisfied with probability $1/|\mathbb{F}|$. From this we see that, if the protocol does not result in abort, $\delta = 0$ except with probability $1/|\mathbb{F}|$, which means that the reconstructed value is $s' = s$. □

2.7.2.2 Local Operations

Addition/Subtraction. Given two shared values $\langle x \rangle = (\mathtt{x}_1, \dots, \mathtt{x}_n)$ and $\langle y \rangle = (\mathtt{y}_1, \dots, \mathtt{y}_n)$, with $\mathtt{x}_i = (x_i, \alpha_i^{(x)}, \gamma_i^{(x)})$ and $\mathtt{y}_i = (y_i, \alpha_i^{(y)}, \gamma_i^{(y)})$ for $i \in [n]$, it is possible for the parties to locally obtain shares $\langle x + y \rangle$. This requires that $\alpha_i := \alpha_i^{(x)} = \alpha_i^{(y)}$, that is, the dealer samples $\alpha_1, \dots, \alpha_n \in_R \mathbb{F}$ once, and uses $\alpha = \sum_{i=1}^{n} \alpha_i$ to define the shares of all subsequent values. This way, if $\langle x \rangle = (\llbracket x \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot x \rrbracket)$ and $\langle y \rangle = (\llbracket y \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot y \rrbracket)$, $\langle x + y \rangle$ may be computed locally exploiting the homomorphic properties of basic additive secret-sharing as $(\llbracket x \rrbracket + \llbracket y \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot x \rrbracket + \llbracket \alpha \cdot y \rrbracket)$. More precisely, the shares $(\mathtt{z}_1, \dots, \mathtt{z}_n)$ are defined as $\mathtt{z}_i = (z_i, \alpha_i, \gamma_i^{(z)})$, where $z_i = x_i + y_i$ and $\gamma_i^{(z)} = \gamma_i^{(x)} + \gamma_i^{(y)}$ for $i \in [n]$.

Canonical shares of public values. Given a value $c \in \mathbb{F}$ known by all the parties, the parties can obtain shares $\langle c \rangle = (\mathtt{c}_1, \dots, \mathtt{c}_n)$ by defining $\mathtt{c}_i = (c_i, \alpha_i, \gamma_i^{(c)})$, with $c_i = 0$ for $i \in [n] \setminus \{1\}$, $c_i = c$ for $i = 1$, and $\gamma_i^{(c)} = \alpha_i \cdot c$ for $i \in [n]$.

2.7.3 Online Phase

Let $\langle \cdot \rangle$ denote "authenticated" sharings as the ones from either Section 2.7.1 or Section 2.7.2. These have in common that local addition/subtraction of shared values, together with local multiplication and addition/subtraction of publicly known values, is possible. Furthermore, when reconstructing secret-shared values, although the adversary may initially cause the parties to open a shared-value incorrectly, the parties can execute a verification step that ensures that, with high probability, the opened value under consideration is reconstructed correctly.

For our protocol we assume that the extra data required for the authenticated secret-sharing scheme $\langle \cdot \rangle$, like the necessary keys and tags, or their shares, depending on whether the authentication mechanism chosen is pairwise or global MACs, is computed by the parties in a preprocessing phase. During this phase the parties also obtain a multiplication triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ with $a, b \in_R \mathbb{F}$ uniformly random and unknown to the adversary and $c = a \cdot b$, for every multiplication gate in the circuit under consideration. Furthermore, for every input gate corresponding to party P_i, the parties have $\langle r \rangle$, where $r \in \mathbb{F}$ is uniformly random and known only to P_i.

With these tools at hand, the parties can securely compute the given arithmetic circuit in a similar way as in Section 2.6.5. Multiplication gates are processed in essentially the same way: the parties reconstruct a masked version of the inputs to the multiplication gate, making use of a multiplication triple. However, since the secret-sharing scheme $\langle \cdot \rangle$ is more complex than simple additive secret-sharing $\llbracket \cdot \rrbracket$, it is not possible for the parties to obtain shares of the inputs to the computation non-interactively. This is achieved in a similar way as the protocol from Section 2.4.3, by letting each party broadcast a masked version of their inputs using random values that are secret-shared, and then the parties add this publicly known value to these shares. This is detailed below.

Secure computation based on authenticated secret-sharing

Offline phase: The parties obtain from the preprocessing phase:

- The necessary keys/shares for the secret-sharing scheme $\langle \cdot \rangle$
- A multiplication triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ for every multiplication gate
- For every input gate with owner P_i, a uniformly random shared value $\langle r \rangle$ only known to P_i.

Online phase: The parties execute the following.

Input gates. For every party P_i holding input x, the parties execute the following. Let $\langle r \rangle$ be a random shared value, where P_i knows r.

1. P_i broadcasts $e = x - r$.
2. The parties compute the sharings $\langle x \rangle \leftarrow \langle r \rangle + e$.

Addition gates. These are handled locally by using the properties of the secret-sharing scheme $\langle \cdot \rangle$.

Multiplication gates. Given two shared values $\langle x \rangle$ and $\langle y \rangle$, the parties obtain $\langle xy \rangle$ as follows. Let $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ be a multiplication triple.

1. The parties compute locally $\langle d \rangle \leftarrow \langle x \rangle - \langle a \rangle$ and $\langle e \rangle \leftarrow \langle y \rangle - \langle b \rangle$
2. The parties reconstruct $d \leftarrow \langle d \rangle$ and $e \leftarrow \langle e \rangle$.
3. The parties compute locally $\langle z \rangle = d \langle b \rangle + e \langle a \rangle + \langle c \rangle + de$.

Output gates. The parties reconstruct $\langle z \rangle$ for every output shared value.

As with the protocol from Section 2.6, privacy is guaranteed in the input phase since the input x is masked with the random value r when P_i broadcasts $e = x - r$, and similarly for the reconstructed values $d = x - a$ and $e = y - b$ in the multiplication. Correctness follows from the fact that, if $d = x - a$, $e = y - b$ and $c = ab$, it holds that $db + ea + c + de$ is equal to xy.

Part II

MPC Techniques over $\mathbb{Z}/2^k\mathbb{Z}$

Chapter 3

Two-Thirds Honest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$

In this chapter we begin with the first contribution of this thesis, which consists in the design of an MPC protocol to securely compute functions $F : (\mathbb{Z}/2^k\mathbb{Z})^n \to \mathbb{Z}/2^k\mathbb{Z}$ expressed as an arithmetic circuit over $\mathbb{Z}/2^k\mathbb{Z}$, ensuring perfect security against an active adversary corrupting t parties, where $t < n/3$. We focus on security with abort, which is captured in the functionality by allowing the adversary to cause the parties to abort, as mentioned in Section 1.2.6.2. Our protocol can be enhanced to achieve guaranteed output delivery using standard techniques, and this is in fact what is achieved in the original work of [2]. We provide a more detailed discussion on this topic in Section 3.4.

In Section 2.4 we studied one protocol in the setting of active and perfect security with $t < n/3$ for the case in which the arithmetic circuit is expressed over a field $\mathbb{F}$. Furthermore, this field needs to satisfy $|\mathbb{F}| \geq n + 1$ for Shamir secret-sharing to work, as explained in Section 2.2, and this restriction must be strengthened to $|\mathbb{F}| \geq 2n$ for the construction of hyper-invertible matrices from Section 2.4.2.1 to work. As a first step towards obtaining a protocol over $\mathbb{Z}/2^k\mathbb{Z}$, we generalize in Section 3.1 the construction of Shamir secret-sharing from $\mathbb{F}$ to any commutative ring $\mathcal{R}$ satisfying certain property that generalizes the notion of $|\mathbb{F}|$ being large enough. This is based on the results in the original work of [51]

Once this is set, we can use the template of the protocol from Section 2.4 to obtain a protocol for computation over $\mathbb{Z}/2^k\mathbb{Z}$. Unfortunately, the ring $\mathbb{Z}/2^k\mathbb{Z}$ does not satisfy the necessary condition to admit the construction of Shamir secret-sharing. To alleviate this issue, we define the so-called Galois rings in Section 3.2, which are a generalization of the ring $\mathbb{Z}/2^k\mathbb{Z}$ and allow for the construction of Shamir secret-sharing from Section 3.1. With these tools, we finally discuss in Section 3.3 a construction of a perfectly secure MPC protocol with $t < n/3$ over any sub-Galois ring of a larger Galois ring, which includes in particular the case of $\mathbb{Z}/2^k\mathbb{Z}$. As we will see, although our protocol resembles a lot the one described in previous sections over fields, several optimizations are needed to help mitigate the overhead of using the Galois extension rings.

The contents of this chapter are mostly based on the original work of [2], although ideas from [51] are used in Section 3.1.

3.1 Shamir Secret-Sharing over Arbitrary Commutative Rings

An interesting and relevant result is that Shamir secret-sharing, as presented in Section 2.2, is not restricted to fields only. For example, in the original work of [2] it is shown that Shamir secret-sharing is also possible over the so-called Galois rings, which are generalizations of the ring $\mathbb{Z}/2^k\mathbb{Z}$, and are described in detail in Section 3.2. Then, in the same work, it was shown that this can be used as the basis for a perfectly secure protocol over $\mathbb{Z}/2^k\mathbb{Z}$ with $t < n/3$, in a flavor that resembles the protocol over fields from Section 2.4.

In this section, instead of presenting the construction of Shamir secret-sharing for Galois rings from [2], we present a more general result that shows that Shamir secret-sharing is possible over any commutative finite ring, as long as the ring satisfies certain condition. This result is introduced in the original work of [51]. The advantage of considering this general result is that it illustrates the minimum requirements on the given algebraic structure for efficient and homomorphic secret-sharing to be possible.

About the original work of [51]. We remark that, while the work of [51] introduces Shamir secret-sharing over arbitrary finite rings (even non-commutative ones), its main contribution lies in using this result as a tool to obtain information-theoretic MPC over these rings. This includes in particular $\mathbb{Z}/2^k\mathbb{Z}$, the ring of interest to this thesis. However, for this thesis we chose to only use the construction of Shamir secret-sharing from [51], while targeting the construction of our MPC protocol in Section 3.3 to the concrete case of Galois rings, which ultimately lead to MPC over $\mathbb{Z}/2^k\mathbb{Z}$, our ring of interest. This is because, due to the fact that [51] assumes almost no algebraic structure on the given ring, the protocols in [51] are conceptually complex and relatively inefficient. As an example, the rings considered in [51] could be non-commutative (like, for example, a ring of matrices, which is a relevant instantiation in practice), which heavily complicates their constructions. Focusing on Galois rings allow us to simplify the exposition drastically, and it allows us to obtain much more efficient constructions.

In summary, we only take from [51] the construction of Shamir secret-sharing for arbitrary finite rings, adapted to the commutative case, and leave out the constructions of MPC protocols for arbitrary finite rings in [51] with the goal of including more efficient MPC constructions tailored to the more concrete ring $\mathbb{Z}/2^k\mathbb{Z}$ like the ones from the original work [2].

3.1.1 Algebraic Preliminaries

Let $\mathcal{R}$ be a finite commutative ring. These are more general than fields, as these may have non-zero zero divisors, that is, non-zero elements $x \in \mathcal{R}$ such that $x \cdot y = 0$ for some $y \in \mathcal{R} \setminus \{0\}$.[1] For example, in $\mathbb{Z}/2^k\mathbb{Z}$ the value 2^{k-1} is a non-zero zero divisor since $2^{k-1} \cdot 2 \equiv 0 \bmod 2^k$.

[1] It can be shown that, over a finite ring, an element is either invertible or a zero-divisor. Hence, non-field finite rings are precisely these with non-zero zero divisors.

Given that, in general, commutative rings do not enjoy the property of fields that every non-zero element is invertible, it is natural to ask what results from the ones studied in Section 2.2 carry over to more general rings. It turns out that, fortunately, by restricting to a "nice enough" subset of the ring, most of the results from the case of fields in Section 2.2 carry over to the more general setting of an arbitrary finite commutative ring. This type of subsets is captured in the following definition.

Definition 3.1. *Let $A = \{a_1, \dots, a_\ell\} \subset \mathcal{R}$. We say that A is an* exceptional set *if, for all $a_i, a_j \in A$ with $a_i \neq a_j$, it holds that $a_i - a_j \in \mathcal{R}^*$. We define the* Lenstra constant *of $\mathcal{R}$ to be the maximum size of an exceptional set in $\mathcal{R}$.*

Example 3.1. *Let $\mathbb{F}$ be a field. Since any non-zero element of $\mathbb{F}$ is invertible, for any $x, y \in \mathbb{F}$ with $x \neq y$ it holds that $x - y \in \mathbb{F}^*$, which implies that $\mathbb{F}$ is itself an exceptional set. In fact, it is easy to see that if a ring $\mathcal{R}$ is an exceptional set then $\mathcal{R}$ is itself a field.*

Given $\beta_1, \dots, \beta_u \in \mathcal{R}$, let $\mathrm{Van}^{u\times v}(\beta_1, \dots, \beta_u) \in \mathcal{R}^{u\times v}$ be the matrix given by

$$\begin{pmatrix} 1 & \beta_1^1 & \beta_1^2 & \cdots & \beta_1^{v-1} \\ 1 & \beta_2^1 & \beta_2^2 & \cdots & \beta_2^{v-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta_u^1 & \beta_u^2 & \cdots & \beta_u^{v-1} \end{pmatrix}.$$

The following is a standard result in linear algebra.

Theorem 3.1. *Assume that $u = v$. Then the determinant of the matrix $\mathrm{Van}^{u\times v}(\beta_1, \dots, \beta_u) \in \mathcal{R}^{u\times v}$ is $\prod_{i<j}(\beta_i - \beta_j)$.*

Corollary 3.1. *Let $\mathcal{A} = \{a_1, \dots, a_\ell\} \subseteq \mathcal{R}$ be an exceptional set. Then $\mathrm{Van}^{\ell\times\ell}(a_1, \dots, a_\ell) \in \mathcal{R}^{\ell\times\ell}$ is invertible.*

From this we obtain the following results for polynomials over $\mathcal{R}$.

Proposition 3.1. *Let $A = \{a_0, \dots, a_d\} \subset \mathcal{R}$ be an exceptional set. Let $e_0, \dots, e_d \in \mathcal{R}$. Then there exists a unique $f(X) \in \mathcal{R}_{\leq d}[X]$ such that $f(a_i) = e_i$ for $i \in \{0, \dots, d\}$.*

Proof. Write $f(X) = \sum_{i=0}^{d} c_i X^i$, where the $c_0, \dots, c_d$ are unknowns. Let $\mathbf{M} = \mathrm{Van}^{(d+1)\times(d+1)}(a_0, \dots, a_d)$. It is easy to see that $f(a_i) = e_i$ for $i \in \{0, \dots, d\}$ if and only if

$$(e_0, \dots, e_d)^\intercal = \mathbf{M} \cdot (c_0, \dots, c_d)^\intercal.$$

From Corollary 3.1, $\mathbf{M}$ is invertible, so this system of equation has the unique solution $(c_0, \dots, c_d)^\intercal = \mathbf{M}^{-1} \cdot (e_0, \dots, e_d)^\intercal$. □

Definition 3.2. *Let $A = \{a_0, \dots, a_d\} \subset \mathcal{R}$ be an exceptional set. Let $e_0, \dots, e_d \in \mathcal{R}$. We denote by* $\text{Interpolate}_d(\{(a_i, e_i)\}_{i=0}^d)$ *the unique polynomial $f(\mathtt{X}) \in \mathcal{R}_{\leq d}[\mathtt{X}]$ such that $f(a_i) = e_i$ for $i \in \{0, \dots, d\}$.*

Finally, the following definition will be useful when we discuss Shamir secret-sharing in the following section.

Definition 3.3 (Definition 2.2 over $\mathcal{R}$)**.** *Let $\{\beta_1, \dots, \beta_\ell\} \subseteq \mathcal{R}$ be an exceptional set. Given $\mathbf{s} = (s_1, \dots, s_\ell) \in \mathcal{R}^\ell$, we say that $\mathbf{s}$ is* d-consistent *if there exists $f(\mathtt{X}) \in \mathcal{R}_{\leq d}[\mathtt{X}]$ such that $s_i = f(\beta_i)$ for $i = 1, \dots, \ell$.*

Observe that d-consistency is an $\mathcal{R}$-linear property, that is, the set of d-consistent vectors constitutes an $\mathcal{R}$-submodule of $\mathcal{R}^\ell$. Also, from Proposition 3.1, if $\ell \leq d + 1$ then every vector $\mathbf{s} \in \mathcal{R}^\ell$ is d-consistent, if $\ell = d + 1$ then every vector is d-consistent with a unique polynomial, and if $d + 1 < \ell$ then not every vector is d-consistent.

3.1.2 Secret-Sharing and Reconstruction

Let $\mathcal{R}$ be a finite commutative ring, and let $\lambda + 1$ be its Lenstra constant. Let $\mathcal{A} = \{\alpha_0, \alpha_1, \dots, \alpha_\lambda\} \subseteq \mathcal{R}$ be an exceptional set, and assume that $\lambda \geq n$.

Shamir secret-sharing is defined by sampling random polynomials of a given degree whose evaluation at α_0 is the desired secret, and letting the shares be the evaluations at the point α_i for $i \in [n]$. We describe below this method in full detail. We assume that $d + 1 \leq n$.

Definition 3.4 (Sharing Procedure)**.** *Let $s \in \mathcal{R}$, and let $\mathcal{A} \subseteq [n]$ be a set with $|\mathcal{A}| = \ell \leq d$ (ℓ could be zero, in which case $\mathcal{A}$ is the empty set). Let $\mathcal{B}$ be any set with $\mathcal{B} \subseteq [n] \setminus \mathcal{A}$ and $|\mathcal{B}| = d - \ell$. Let $\{s_i\}_{i \in \mathcal{A}} \subseteq \mathcal{R}$. We define* $\text{Share}_d(s, \{s_i\}_{i \in \mathcal{A}})$ *as follows.*

1. *Sample $s_i \in_R \mathcal{R}$ for $i \in \mathcal{B}$;*
2. *Let $f(\mathtt{X}) \leftarrow \text{Interpolate}_d(\{(\alpha_0, s)\} \cup \{(\alpha_i, s_i)\}_{i \in \mathcal{A} \cup \mathcal{B}})$.*
3. *Output $(f(\alpha_1), \dots, f(\alpha_n))$.*

Now we characterize the distribution of $\text{Share}_d(s)$.

Lemma 3.1. *Let $s \in \mathcal{R}$, and let $\mathcal{A} \subseteq [n]$ be a set with $|\mathcal{A}| = \ell \leq d$. Let $\mathcal{B}$ be any set with $\mathcal{B} \subseteq [n] \setminus \mathcal{A}$ and $|\mathcal{B}| = d - \ell$. Let $\{s_i\}_{i \in \mathcal{A}} \subseteq \mathcal{R}$. Then the distribution of $(s_1, \dots, s_n) \leftarrow \text{Share}_d(s, \{s_i\}_{i \in \mathcal{A}})$ is uniformly random over $\mathcal{R}^n$, constrained to the following:*

- *$(s_1, \dots, s_n)$ is d-consistent.*

- $s = f(\alpha_0)$ *with* $f(\mathtt{X}) \leftarrow \mathsf{Interpolate}_d(\{(\alpha_i, s_i)\}_{i=1}^n)$.

In particular, the distribution is independent of the chosen set $\mathcal{B}$.

Proof. Assume for now that $\ell < d$; we handle the case $\ell = d$ towards the end of the proof. Consider the mapping $(s_i)_{i\in\mathcal{B}} \mapsto \mathsf{Interpolate}_d(\{(\alpha_0, s)\} \cup \{(\alpha_i, s_i)\}_{i\in\mathcal{A}\cup\mathcal{B}})$ (recall that s and $\{s_i\}_{i\in\mathcal{A}} \subseteq \mathcal{R}$ are fixed). Observe that the output of $\mathsf{Share}_d(s, \{s_i\}_{i\in\mathcal{A}})$ is obtained by sampling a random input to this function and returning its output. It is a direct consequence of Proposition 3.1 that the mapping above is a bijection between $\mathcal{R}^{d-\ell}$ and the set of vectors $(s_1, \ldots, s_n) \subseteq \mathcal{R}^n$ satisfying the constraints from the lemma, which concludes the proof for the case in which $\ell < d$.

If $\ell = d$, it follows from Proposition 3.1 that there is only one vector $(s_1, \ldots, s_n)$ satisfying the constraints of the lemma, and it is easy to see that this is precisely the vector output by $\mathsf{Share}_d(s, \{s_i\}_{i\in\mathcal{A}})$. □

From the lemma above we obtain the following simple but crucial theorem. In terms of notation, if the set $\mathcal{A}$ in Definition 3.4 is empty then the method $\mathsf{Share}_d(s, \emptyset)$ is simply denoted by $\mathsf{Share}_d(s)$.

Theorem 3.2 (Privacy of Shamir Secret-Sharing)**.** *Let* $s \in \mathcal{R}$. *Let* $(s_1, \ldots, s_n) \leftarrow \mathsf{Share}_d(s)$. *Let* $\mathcal{D} \subseteq [n]$ *with* $|\mathcal{D}| \leq d$. *Then the values* $\{s_i\}_{i\in\mathcal{D}}$ *follow the uniform distribution. In particular, their distribution is independent of the secret* s.

Proof. Without loss of generality assume that $|\mathcal{B}| = d$. Then take the set $\mathcal{B} = \mathcal{D}$ in the computation of $\mathsf{Share}_d(s)$. □

Given a secret s that is secret-shared as $(s_1, \ldots, s_n) \leftarrow \mathsf{Share}_d(s)$, we denote the vector $(s_1, \ldots, s_n)$ by $[\![s]\!]_d$. Furthermore, in the context of secure computation, we use the notation $[\![s]\!]_d$ not only to represent the vector of shares, but also to represent the situation in which each party P_i has the share s_i. This is an informal statement that will become clear in the construction of our protocols.

We define the following procedures for reconstruction.

Definition 3.5 (Reconstruction Procedures)**.** *Let* $s \in \mathcal{R}$, *and let* $\mathcal{A} \subseteq [n]$ *be a set with* $|\mathcal{A}| = \ell \geq d + 1$. *Let* $(s_i)_{i\in\mathcal{A}} \in \mathcal{R}^\ell$ *be a* d*-consistent vector. Let* $f(\mathtt{X})$ *be the (unique, from Proposition 3.1, since* $\ell \geq d + 1$*) polynomial of degree at most* d *such that* $f(\alpha_i) = s_i$ *for* $i \in \mathcal{A}$.

- $\mathsf{RecPoly}_d(\{s_i\}_{i\in\mathcal{A}})$ *is defined as* $f(\mathtt{X})$;[2]
- $\mathsf{RecSecret}_d(\{s_i\}_{i\in\mathcal{A}})$ *is defined as* $f(\alpha_0)$;
- $\mathsf{RecShares}_d(\{s_i\}_{i\in\mathcal{A}})$ *is defined as* $(f(\alpha_1), \ldots, f(\alpha_n))$.

[2] This was already considered for the case in which $\ell = d + 1$ in Definition 3.2.

Homomorphisms. Given two shared values $[\![x]\!]_d = (x_1, \dots, x_n)$ and $[\![y]\!]_d = (y_1, \dots, y_n)$, it is easy to see that:

- $[\![x]\!]_d + [\![y]\!]_d = (x_1 + y_1, \dots, x_n + y_n)$ are d-consistent shares of $x + y$. In the context of secure computation where each party P_i has (x_i, y_i), we denote by $[\![x + y]\!]_d \leftarrow [\![x]\!]_d + [\![y]\!]_d$ the protocol in which each party P_i locally adds $x_i + y_i$.

- $[\![x]\!]_d \star [\![y]\!]_d = (x_1 \cdot y_1, \dots, x_n \cdot y_n)$ are $2d$-consistent shares of $x \cdot y$. In the context of secure computation where each party P_i has (x_i, y_i), we denote by $[\![x \cdot y]\!]_{2d} \leftarrow [\![x]\!]_d \cdot [\![y]\!]_d$ the protocol in which each party P_i locally multiplies $x_i \cdot y_i$.

3.1.3 Error Detection/Correction

Let ℓ and d be non-negative integers, and let $\{\beta_1, \dots, \beta_\ell\} \subseteq \mathcal{R}$ be an exceptional set. Let $\mathbf{s} = (s_1, \dots, s_\ell) \in \mathcal{R}^\ell$ be a d-consistent vector, so there exists $f(\mathtt{X}) \in \mathcal{R}_{\leq d}[\mathtt{X}]$ such that $s_i = f(\beta_i)$ for $i = 1, \dots, \ell$. Let $\boldsymbol{\delta} \in \mathcal{R}^\ell$ be a vector with at most e non-zero entries.

Theorem 3.3. *The following holds:*

Error detection. *Suppose that $e < \ell - d$. If $\boldsymbol{\delta} \neq \mathbf{0}$, then $\mathbf{s} + \boldsymbol{\delta}$ cannot be d-consistent.*

Error correction. *Suppose that $e < (\ell - d)/2$. Let $\mathbf{s}_1, \mathbf{s}_2 \in \mathcal{R}^\ell$ be d-consistent vectors and let $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2 \in \mathcal{R}^\ell$ be vectors each with at most e non-zero entries. If $\mathbf{s}_1 + \boldsymbol{\delta}_1 = \mathbf{s}_2 + \boldsymbol{\delta}_2$, then $\mathbf{s}_1 = \mathbf{s}_2$.*

Proof. **Error detection.** Note that $\mathbf{s} + \boldsymbol{\delta}$ is d-consistent if and only if $\boldsymbol{\delta}$ is d-consistent. However, a vector with at least $\ell - e$ zero entries with $d < \ell - e$, such as δ, cannot be d-consistent, unless it is the zero vector. This is because, if this were the case, then there would exist a polynomial $f(\mathtt{X}) \in \mathcal{R}_{\leq d}[\mathtt{X}]$ such that $f(\beta_i) = 0$ for at least $\ell - e \geq d + 1$ indexes, which, from Proposition 3.1, implies that $f(\mathtt{X})$ is the zero polynomial. In particular, all the remaining entries $f(\alpha_j)$ of the claimed vector are zero.

Error correction. Assume that $e < (\ell - d)/2$. If $\mathbf{s}_1 + \boldsymbol{\delta}_1 = \mathbf{s}_2 + \boldsymbol{\delta}_2$, then $\mathbf{s}_1 - \mathbf{s}_2 = \boldsymbol{\delta}_2 - \boldsymbol{\delta}_1$, so $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1$ would be d-consistent. However, this vector has at most $2e$ non-zero entries, so at least $\ell - 2e \geq d + 1$ entries. As shown in the previous paragraph, this cannot happen unless $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1 = \mathbf{0}$, which implies $\mathbf{s}_1 = \mathbf{s}_2$.

□

From the error correction part of Theorem 3.3, we see that, if $e < (\ell - d)/2$, if $\mathbf{s} \in \mathcal{R}^\ell$ is d-consistent and $\boldsymbol{\delta} \in \mathcal{R}^\ell$ has at most e non-zero entries, and if $\mathbf{c} = \mathbf{s} + \boldsymbol{\delta}$, then $\mathbf{s}$ is the only d-consistent vector that satisfies that $\mathbf{c} - \mathbf{s}$ has at most e non-zero entries. As a result, $\mathbf{s}$ can be recovered from $\mathbf{c}$ by trying all possible vectors $\boldsymbol{\delta}' \in \mathcal{R}^\ell$ with at most e non-zero entries, and checking if $\mathbf{c} - \boldsymbol{\delta}'$ is d-consistent. When this holds, then $\mathbf{s} = \mathbf{c} - \boldsymbol{\delta}'$ (and $\boldsymbol{\delta} = \boldsymbol{\delta}'$).

Unfortunately, for our parameters of interest, there are exponentially-many vectors $\boldsymbol{\delta}'$ having at most e non-zero entries. More efficient *decoders*, that is, algorithms for finding $(\mathbf{s}, \boldsymbol{\delta})$ from $\mathbf{c}$ as above, are designed in Sections 3.1.4 and 3.2.4.

Definition 3.6. *Let $s \in \mathcal{R}$, and let $\mathcal{A} = \{a_1, \ldots, a_\ell\} \subseteq [n]$ be a set. Let $(s_{a_1}, \ldots, s_{a_\ell}) \in \mathcal{R}^\ell$ be a d-consistent vector. We extend Definition 3.5 to the case in which the input to the reconstruction procedures might not be d-consistent because of a perturbation in at most e entries as follows.*

Let $\mathbf{c} = \mathbf{s} + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ is guaranteed to have at most e non-zero entries.

$e < \ell - d$ ***(error detection).*** *In this case, the reconstruction procedures are defined as follows:*

- *If $(c_{a_1}, \ldots, c_{a_\ell})$ is not d-consistent, then $\mathsf{RecPoly}_d(\{c_i\}_{i\in\mathcal{A}})$, $\mathsf{RecSecret}_d(\{c_i\}_{i\in\mathcal{A}})$ and $\mathsf{RecShares}_d(\{c_i\}_{i\in\mathcal{A}})$ all output a special symbol $\perp$.*
- *If $(c_{a_1}, \ldots, c_{a_\ell})$ is d-consistent, then let $f(\mathtt{X})$ be the unique polynomial in $\mathcal{R}_{\leq d}[\mathtt{X}]$ with $f(\alpha_i) = c_i$ for $i \in \mathcal{A}$. Then:*
  - *$\mathsf{RecPoly}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $f(\mathtt{X})$*
  - *$\mathsf{RecSecret}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $f(\alpha_0)$*
  - *$\mathsf{RecShares}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $(f(\alpha_1), \ldots, f(\alpha_n))$*

$e < (\ell - d)/2$ ***(error correction).*** *Use a* decoding algorithm *(see Sections 3.1.4 and 3.2.4) to recover $\mathbf{s}$ from $\mathbf{c}$. Let $f(\mathtt{X})$ be the unique polynomial in $\mathcal{R}_{\leq d}[\mathtt{X}]$ with $f(\alpha_i) = s_i$ for $i \in \mathcal{A}$. Then:*

- *$\mathsf{RecPoly}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $f(\mathtt{X})$*
- *$\mathsf{RecSecret}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $f(\alpha_0)$*
- *$\mathsf{RecShares}_d(\{c_i\}_{i\in\mathcal{A}})$ is defined as $(f(\alpha_1), \ldots, f(\alpha_n))$.*

Observe that the procedures above are not defined if $\ell - d \leq e$.

3.1.4 Adaptation of the Berlekamp-Welch Decoding Algorithm

In this section we present an adaptation of the Berlekamp-Welch decoder [79], which solves efficiently the task of recovering $\mathbf{s}$ from $\mathbf{c}$ as above when $\mathcal{R}$ is a field, to the case in which $\mathcal{R}$ is an arbitrary commutative ring. This was achieved in the original work of [51], and our presentation here is a simplification of that result, making explicit use of the fact that our ring $\mathcal{R}$ is assumed to be commutative.

Computational task. The concrete setting is the following. Assume that $e < (\ell - d)/2$. Let $f(\mathtt{X}) \in \mathcal{R}_{\leq d}[\mathtt{X}]$, $\mathbf{s} = (f(\beta_1), \ldots, f(\beta_\ell))$, and let $\boldsymbol{\delta} \in \mathcal{R}^\ell$ be a vector with at most e non-zero entries. The task is to find $\mathbf{s}$ from $\mathbf{c} = \mathbf{s} + \boldsymbol{\delta}$.

The BW-conditions. We begin with the following definition.

Definition 3.7. *We say that two polynomials $p(\mathtt{X}), q(\mathtt{X}) \in \mathcal{R}[\mathtt{X}]$ satisfy the BW-conditions if*

1. $\deg(p) = e$;
2. $\deg(q) \leq d + e$;
3. $p(\mathtt{X})$ *is monic;*
4. *For all $i \in [\ell]$, it holds that $c_i \cdot p(\beta_i) = q(\beta_i)$.*

Claim 3.1. *There exists a pair $p(\mathtt{X}), q(\mathtt{X}) \in \mathcal{R}[\mathtt{X}]$ that satisfies the BW-conditions from Definition 3.7 above.*

Proof. Define $p(\mathtt{X}) = \prod_{\delta_i \neq 0}(\mathtt{X} - \beta_i)$ and $q(\mathtt{X}) = f(\mathtt{X})p(\mathtt{X})$. Clearly $\deg(p) = e$, $\deg(q) \leq d + e$ and $p(\mathtt{X})$ is monic. To check the last condition, let $i \in [\ell]$:

- If $\delta_i \neq 0$ then $p(\beta_i) = 0$ and $q(\beta_i) = f(\beta_i)p(\beta_i) = 0$, so $c_i \cdot p(\beta_i) = q(\beta_i)$.
- If $\delta_i = 0$ then $f(\beta_i) = c_i$, so $q(\beta_i) = f(\beta_i)p(\beta_i) = c_i \cdot p(\beta_i)$.

□

The goal of our decoding algorithm is to find a pair $p(\mathtt{X}), q(\mathtt{X})$ that satisfies the BW-conditions. However, this pair may not be unique. The next claim shows that any other pair satisfying the BW-conditions is as good as the one guaranteed from the previous claim for the purpose of recovering $f(\mathtt{X})$.

Claim 3.2. *Let $p(\mathtt{X}) = \prod_{\delta_i \neq 0}(\mathtt{X} - \beta_i)$ and $q(\mathtt{X}) = f(\mathtt{X})p(\mathtt{X})$, and suppose that $\hat{p}(\mathtt{X}), \hat{q}(\mathtt{X})$ satisfy the BW-conditions. Then $\hat{p}(\mathtt{X})$ divides $\hat{q}(\mathtt{X})$ and $\hat{q}(\mathtt{X})/\hat{p}(\mathtt{X}) = f(\mathtt{X})$.*

Proof. Consider the polynomial $r(\mathtt{X}) = \hat{q}(\mathtt{X})p(\mathtt{X}) - q(\mathtt{X})\hat{p}(\mathtt{X})$. For every $i \in [\ell]$, it holds that

$$r(\beta_i) = \hat{q}(\beta_i)p(\beta_i) - q(\beta_i)\hat{p}(\beta_i) = c_i\hat{p}(\beta_i)p(\beta_i) - c_i(\beta_i)\hat{p}(\beta_i) = 0.$$

Observe that in the last equality we have used the fact that $\hat{p}(\alpha_i)p(\alpha_i) = p(\alpha_i)\hat{p}(\alpha_i)$. Since $\deg(r) \leq d + 2e < \ell$, it follows from Proposition 3.1 that $r(\mathtt{X})$ is the zero polynomial, which shows that $\hat{q}(\mathtt{X})p(\mathtt{X}) = q(\mathtt{X})\hat{p}(\mathtt{X})$. Given that $q(\mathtt{X}) = f(\mathtt{X})p(\mathtt{X})$, we have that $\hat{q}(\mathtt{X})p(\mathtt{X}) = f(\mathtt{X})p(\mathtt{X})\hat{p}(\mathtt{X})$, which implies $(\hat{q}(\mathtt{X}) - f(\mathtt{X})\hat{p}(\mathtt{X})) \cdot p(\mathtt{X}) = 0$.

If $\mathcal{R}$ was a field, then $\deg(a \cdot b) = \deg(a) + \deg(b)$ for any two polynomials $a(X)$, $b(X)$, which would allow us to conclude that either $\hat{q}(X) - f(X)\hat{p}(X)$ or $p(X)$ is the zero polynomial, and since $p(X)$ is not zero, it must be the case that $\hat{q}(X) - f(X)\hat{p}(X) = 0$, obtaining the desired result. However, $\mathcal{R}$ may have non-zero zero divisors, which imply that $\deg(a \cdot b) \leq \deg(a) + \deg(b)$. In particular, there exist non-zero polynomials whose product is zero.

To prove that $\hat{q}(X) - f(X)\hat{p}(X) = 0$, we follow a different approach: we show that this polynomial evaluates to 0 in at least $d + e + 1$ points in an exceptional set, which implies the desired equality in light of Proposition 3.1. To see this, consider the evaluation of this polynomial at β_i for all i such that $\delta_i = 0$. Observe that there are at least $\ell - e \geq d + e + 1$ such evaluation points. Since $\{\beta_1, \ldots, \beta_\ell\}$ is an exceptional set, $p(\beta_i)$ is invertible, so $r(\beta_i) = (\hat{q}(\beta_i) - f(\beta_i)\hat{p}(\beta_i)) \cdot p(\beta_i) = 0$ implies that $\hat{q}(\beta_i) - f(\beta_i)\hat{p}(\beta_i) = 0$, as required. At this point we see that $\hat{q}(X) = f(X)\hat{p}(X)$, which concludes the proof of the main claim. □

From Claim 3.2 above, we see that, to recover **s** from **c**, it suffices to

1. Find $p(X)$ and $q(X)$ that satisfy the BW-conditions from Definition 3.7, and let $f(X) = q(X)/p(X)$
2. Return $(f(\beta_1), \ldots, f(\beta_\ell))$.

We discuss how to solve for the BW-conditions below.

Solving for the BW-conditions. To find $p(X)$ and $q(X)$ that satisfy the BW-conditions, we write $p(X) = X^e + \sum_{i=0}^{e-1} p_i \cdot X^i$ and $q(X) = \sum_{i=0}^{d+e} q_i \cdot X^i$ for some unknowns $p_0, \ldots, p_{e-1}, q_0, \ldots, q_{d+e}$. The last item in the BW-conditions can be phrased as $\sum_{j=0}^{d+e} q_i \cdot \beta_i^j - \sum_{j=0}^{e} p_i \cdot (c_i \beta_i^j) = 0$ for $i \in [\ell]$. This is an overdetermined homogeneous system of ℓ equations in $d + 2e + 1 \leq \ell$ variables, and from Claim 3.1, this system has at least one non-trivial solution. This solution can be found by using algorithms for solving linear equations over $\mathcal{R}$.

Remark 3.1. *We are not aware of generic algorithms for solving linear equations over an* arbitrary *finite commutative ring $\mathcal{R}$, so the last step above highly depends on the given ring. For the case in which $\mathcal{R}$ is a Galois ring, which is the case we will deal with ultimately, efficient algorithms for solving systems of linear equations can be devised, and these are discussed in Section 3.2.4.1.*

Alternatively, we also discuss in Section 3.2.4.2 how to turn any decoding algorithm over fields into a decoding algorithm over a Galois ring, and this algorithm can be used in order to solve the decoding problem over a Galois ring instead of using the adaptation of the Berlekamp-Welch algorithm from this section.

3.1.5 Reconstructing Secret-Shared Values Efficiently

At different points throughout the execution of our protocol, the parties will need to reconstruct several secret-shared values. This is captured by the following functionality.

Functionality $\mathcal{F}_{\text{PublicRec}}(d)$

Receive s_i from each honest party P_i. $\{s_i\}_{i\in\mathcal{H}}$ is guaranteed to be d-consistent.

1. Let $(s_1, \ldots, s_n) \leftarrow \text{RecShares}_d(\{s_i\}_{i\in\mathcal{H}})$. Send $\{s_i\}_{i\in[n]}$ to the adversary.
2. Depending on the value of d:
   - **If $t < n - d$ (error detection).** If the adversary sends abort then make the honest parties abort. Else, if the adversary sends continue, then send s with $s \leftarrow \text{RecSecret}_d(\{s_i\}_{i\in\mathcal{H}})$ to the honest parties.
   - **If $t < (n-d)/2$ (error correction).** Compute s as above and send this value to the honest parties. Abort signals are ignored.

Observe that the only difference between the setting of reconstruction with error detection and error correction is that in the former case the adversary is allowed to cause the parties to abort, whereas in the latter this is not the case.

Functionality $\mathcal{F}_{\text{PublicRec}}$ can be instantiated by the protocol $\Pi_{\text{PublicRec}}$ below. This protocol reconstructs several values simultaneously. This is an adaptation to the arbitrary-ring setting of the corresponding protocol from [42].

Protocol $\Pi_{\text{PublicRec}}(d)$

Input: Secret-shared values $[\![s_0]\!]_d, \ldots, [\![s_t]\!]_d$.
Output: All the parties learn $s_0, \ldots, s_t$.
Protocol: The parties proceed as follows

1. Let $f(\mathrm{X}) = \sum_{j=0}^{t} s_j \mathrm{X}^j$. The parties locally compute $[\![z_i]\!]_d = \sum_{j=0}^{t} [\![s_j]\!]_d \, \alpha_i^j$ for $i \in [n]$. Let us denote these shares by $(z_{i1}, \ldots, z_{in})$.
2. Each party P_k for $k \in [n]$ sends z_{ik} to P_i, for $i \in [n]$.
3. Upon receiving $(z_{i1}, \ldots, z_{in})$, each P_i for $i \in [n]$ computes $z_i \leftarrow \text{RecSecret}_d(z_{i1}, \ldots, z_{in})$. If this output is $\perp$, the parties abort.
4. For $i, j \in [n]$, P_i sends the reconstructed z_i to P_j.
5. Upon receiving $(z_1, \ldots, z_n)$, each party P_j computes $f(\mathrm{X}) \leftarrow \text{RecPoly}_d(z_1, \ldots, z_n)$.
   - If the output of the call to RecPoly_d above results in $\perp$, then the parties abort.
   - Else, let $f(\mathrm{X}) = \sum_{i=0}^{d} s_i \mathrm{X}^i$. P_j outputs $(s_0, \ldots, s_d)$.

Below we present the formal security proof that shows that Protocol $\Pi_{\text{PublicRec}}(d)$ instantiates $\mathcal{F}_{\text{PublicRec}}(d)$. We present a complete full-fledged simulation-based proof, and we will do so with many of the constructions that follow in this thesis. As a convention that we will adhere to in this work, simulators are described into separate boxes, and the indistinguishability arguments between the real and ideal worlds is done in a "line by line" approach where every single step of the protocol is analyzed in both worlds, arguing why the environment cannot distinguish both executions at the given "instruction". Finally, given some character that denotes a value in the real world protocol, say x, we write the

same character with a line on the top, so $\overline{x}$ in this case, to denote the corresponding value in the ideal world.

Theorem 3.4. *Protocol $\Pi_{\text{PublicRec}}(d)$ instantiates functionality $\mathcal{F}_{\text{PublicRec}}(d)$ with perfect security against an active adversary corrupting $t < n/3$ parties.*

Proof. We begin by defining the simulator $\mathcal{S}$, who emulates virtual honest parties $\overline{P}_i$ for $i \in \mathcal{H}$.

The simulator first obtains $\{s_{ij}\}_{j\in[n]}$ for $i = 0, \dots, t$ from $\mathcal{F}_{\text{PublicRec}}$. Let $s_i = \text{RecSecret}_d(\{s_{ij}\}_{j\in[n]})$ for $i = 0, \dots, t$. Let $f_j(\mathtt{X}) = \sum_{\ell=0}^{t} s_{\ell j}\mathtt{X}^\ell$ for $j \in [n]$, and $f(\mathtt{X}) = \sum_{\ell=0}^{t} s_\ell \mathtt{X}^\ell$. Observe that for every $i \in [n]$, $(f_1(\alpha_i), \dots, f_n(\alpha_i))$ is d-consistent with secret $f(\alpha_i)$.

1. $\mathcal{S}$ computes $f_j(\alpha_i) = \sum_{\ell=0}^{t} s_{\ell j}\alpha_i^\ell$ for $i, j \in [n]$.
2. For every $i \in \mathcal{C}$ and $k \in \mathcal{H}$:
   - $\overline{P}_k$ sends $f_j(\alpha_i)$ to P_i.
   - $\overline{P}_k$ receives $\overline{z}_{ki}$ from P_i.
3. Depending on the value of t:
   $t < n - d$ **(error detection).** If there exists $i \in \mathcal{C}$ such that $\overline{z}_{ki} \neq f_j(\alpha_i)$, then send abort to $\mathcal{F}_{\text{PublicRec}}$. Else, send continue to $\mathcal{F}_{\text{PublicRec}}$.
   $t < (n - d)/2$ **(error correction).** Do nothing.
4. (If no abort was produced) every party $\overline{P}_j$ for $j \in \mathcal{H}$ sends $f(\alpha_j)$ to every corrupt party. Let $\overline{z}_i^{(j)}$ be the value sent by P_i to $\overline{P}_j$, with $i \in \mathcal{C}$ and $j \in \mathcal{H}$.
5. Depending on the value of t:
   $t < n - d$ **(error detection).** If there exists $i \in \mathcal{C}$ such that $\overline{z}_i^j \neq f(\alpha_i)$, then send abort to $\mathcal{F}_{\text{PublicRec}}$. Else, send continue to $\mathcal{F}_{\text{PublicRec}}$.
   $t < (n - d)/2$ **(error correction).** Do nothing.

We argue indistinguishability between the real and ideal worlds.

Real world	**Ideal world**
1. N/A (local computation)	1. N/A (local computation)
2. The adversary receives $\{z_{ij}\}_{i\in\mathcal{C}, j\in\mathcal{H}}$ from the honest parties.	2. By definition $z_{ij} = f_j(\alpha_i)$, so the adversary receives the exact same values $\{f_j(\alpha_i)\}_{i\in\mathcal{C}, j\in\mathcal{H}}$ as in the real world from the emulated honest parties.
3. Let z'_{ij} be the value sent by P_j to P_i, for $j \in \mathcal{C}$ and $i \in \mathcal{H}$. If $t < n - d$, the error detection property of RecSecret_d ensures that the output of $\text{RecSecret}_d(\{z_{ij} = f_j(\alpha_i)\}_{j\in\mathcal{H}} \cup \{z'_{ij}\}_{j\in\mathcal{C}})$ is $f(\alpha_i)$ if $z'_{ij} = z_{ij}$ for $j \in \mathcal{C}$, and $\perp$ otherwise, in which case the parties abort. If $t < (n - d)/2$, the error correction property of RecSecret_d ensures that the output of $\text{RecSecret}_d(\{z_{ij}\}_{j\in\mathcal{H}} \cup \{z'_{ij}\}_{j\in\mathcal{C}})$ is $f(\alpha_i)$.	3. $\overline{z}_{ij}$ is the value sent by P_j to P_i for $j \in \mathcal{C}$ and $i \in \mathcal{H}$. If $t < n - d$, then $\mathcal{S}$ instructs $\mathcal{F}_{\text{PublicRec}}$ to abort the (real) honest parties if $\overline{z}_{ij} \neq f_j(\alpha_i)$ for some $j \in \mathcal{C}$, as in the real world. If $t < (n - d)/2$, then emulated parties continue, as in the real world.

From the previous step, honest parties abort in the real world if and only if the (real) honest parties in the ideal world abort. In what follows we assume the parties do not abort.

4. The corrupt parties receive $\{z_i\}_{j\in\mathcal{H}}$. Also, each honest party P_j receives $\{z_i^{(j)}\}_{i\in\mathcal{C}}$ from the corrupt parties.	4. z_i is equal to $f(\alpha_i)$ for $i \in \mathcal{H}$, so the corrupt parties receive the exact same values $\{f(\alpha_i)\}_{i\in\mathcal{H}}$ from the emulated honest parties. Each emulated honest party $\overline{P}_j$ receives $\{\overline{z}_i^{(j)}\}_{i\in\mathcal{C}}$ from the honest parties.
5. If $t < n - d$, the error detection property of RecPoly_d ensures that the output of $\text{RecPoly}_d(\{z_i^{(j)}\}_{i\in\mathcal{C}} \cup \{z_i = f(\alpha_i)\}_{i\in\mathcal{H}})$ is $f(\mathbf{x})$ if $z_i^{(j)} = f(\alpha_i)$ for $i \in \mathcal{C}$, and $\perp$ otherwise, in which case the parties abort. If $t < (n-d)/2$, the error correction property of RecPoly_d ensures that the output of $\text{RecPoly}_d(\{z_i^{(j)}\}_{i\in\mathcal{C}} \cup \{z_i\}_{i\in\mathcal{H}})$ is $f(\mathbf{x})$.	5. If $t < n - d$, then $\mathcal{S}$ instructs $\mathcal{F}_{\text{PublicRec}}$ to abort the (real) honest parties if $\overline{z}_i^{(j)} \neq f(\alpha_i)$ for some $j \in \mathcal{C}$, as in the real world. If $t < (n-d)/2$, then emulated parties continue, as in the real world.
Honest parties abort in the real world if and only if the (real) honest parties in the ideal world abort. In what follows we assume the parties do not abort.	
The honest parties output $(s_0, \ldots, s_t)$.	The (real) honest parties receive the exact same $(s_0, \ldots, s_t)$ from $\mathcal{F}_{\text{PublicRec}}$ and output these values.

□

Communication complexity of $\Pi_{\text{PublicRec}}(d)$. Protocol $\Pi_{\text{PublicRec}}(d)$ first requires all parties to one share to each other party, which results in n^2 ring elements communicated. Then, once again, each party needs to send one share to each other party, leading to other n^2 ring elements of communication. In total, the communication complexity is $2n^2$ ring elements. However, since a total of $t+1$ secret-shared values are reconstructed, the *amortized* cost per shared value is $\frac{2n^2}{1+t}$. For the case in which $n = 3t+1$ this is the same as $\frac{6n^2}{n+2} \approx 6n$.

3.2 Galois Rings

In this section we introduce a particular type of commutative rings, the so-called Galois rings, that will be useful for our ultimate task of designing MPC protocols over $\mathbb{Z}/2^k\mathbb{Z}$. The usefulness of these rings lies, essentially, in the fact that they are a generalization of $\mathbb{Z}/2^k\mathbb{Z}$, and they posses suitable properties to be able to use Shamir secret-sharing on them. The use of these rings in the context of secure multiparty computation is first documented in the original work of [2].

In this section we introduce Galois rings, and we study some of their properties that are most relevant in our context. For a more in-depth introduction to Galois rings, their structure and advanced properties, we refer the reader to [78]. Throughout this section we let p be a prime number, and k, τ be positive integers.

Definition 3.8. *Let $h(\mathtt{X}) \in \mathbb{Z}/p^k\mathbb{Z}[\mathtt{X}]$ be a monic polynomial of degree τ such that $h \bmod p \in GF(p)[\mathtt{X}]$ is irreducible. The Galois ring of degree τ and modulo p^k, denoted by $GR(p^k, \tau)$, is defined as the quotient ring*

$$\frac{(\mathbb{Z}/p^k\mathbb{Z})[\mathtt{X}]}{h(\mathtt{X})(\mathbb{Z}/p^k\mathbb{Z})[\mathtt{X}]}.$$

Let $h(\mathtt{X}) = \sum_{i=0}^{\tau} h_i \mathtt{X}^i$ with $h_\tau = 1$. From its definition, $\mathrm{GR}(p^k, \tau)$ can be seen as $(\mathbb{Z}/p^k\mathbb{Z})[\xi]$, where $h(\xi) = 0$. Since $\xi^\tau = -\sum_{i=0}^{\tau-1} h_i \xi^i$, every polynomial in $(\mathbb{Z}/p^k\mathbb{Z})[\xi]$ can be rewritten so that its maximum power is $\xi^{\tau-1}$. Furthermore, it can be shown that this representation is unique. As a result, we have that

$$\mathrm{GR}(p^k, \tau) = \{c_0 + c_1\xi + \cdots + c_{\tau-1}\xi^{\tau-1} : c_i \in \mathbb{Z}/p^k\mathbb{Z}\}.$$

From this, we see that, by setting $c_1, \ldots, c_{\tau-1} = 0$, $\mathbb{Z}/p^k\mathbb{Z}$ is a subring of $\mathrm{GR}(p^k, \tau)$. It can also be seen that $\mathrm{GR}(p^k, \tau)$ is isomorphic to $(\mathbb{Z}/p^k\mathbb{Z})^\tau$ as $\mathbb{Z}/p^k\mathbb{Z}$-modules. We list these properties, in addition to some others, in the following theorem.

Theorem 3.5 (See [78])**.** *A Galois ring has the following properties.*

- *$GR(p^k, \tau)$ is unique up to ring isomorphisms.*
- *A Galois ring can be also characterized as a finite ring $\mathcal{R}$ such that its set of zero divisors forms a principal ideal $p1 \cdot \mathcal{R}$ for some prime number $p \in \mathbb{Z}$, where $p1$ means the identity 1 added to itself p times.*
- *$\mathbb{Z}/p^k\mathbb{Z}$ is a subring of $GR(p^k, \tau)$.*
- *$GR(p^k, \tau)$ is isomorphic to $(\mathbb{Z}/p^k\mathbb{Z})^\tau$ as $\mathbb{Z}/p^k\mathbb{Z}$-modules.*
- *$GR(p, \tau)$ is equal to $GF(p^\tau)$.*

Recall that every element in $\mathrm{GR}(p^k, \tau)$ can be seen as a polynomial over $\mathbb{Z}/p^k\mathbb{Z}$ of degree at most $\tau-1$. This representation, which we will refer to as the *polynomial representation*, is particularly useful when we want to interpret $\mathrm{GR}(p^k, \tau)$ as $(\mathbb{Z}/p^k\mathbb{Z})^\tau$. However, another representation that we will make extensive use of in this work is given in the following theorem. This representation is called the *p-adic representation*.

Theorem 3.6 (Theorem 14.8 in [78])**.** *There exists $\xi \in GR(p^k, \tau)$ of order $p^\tau - 1$ with $h(\xi) = 0$, such that every element in $a \in GR(p^k, \tau)$ can be uniquely written as $a = a_0 + a_1 p + \cdots + a_{k-1}p^{k-1}$, where $a_i \in \mathcal{T}$, with $\mathcal{T} = \{0, 1, \xi, \ldots, \xi^{p^\tau - 2}\}$. Furthermore, $a \in \mathcal{R}^*$ if and only if $a_0 \neq 0$.*

In a similar way as taking modulo p^s of an element in $\mathbb{Z}/p^k\mathbb{Z}$ leads to an element in $\mathbb{Z}/p^s\mathbb{Z}$, we can take modulo p^s of elements in $\mathrm{GR}(p^k, \tau)$ to obtain elements in $\mathrm{GR}(p^s, \tau)$. This is captured in the following definition.

Definition 3.9 (Modular reduction in $\mathrm{GR}(p^k, \tau)$)**.** *Let $s \leq k$, and let $a \in GR(p^k, \tau)$. We define $a \bmod p^s$ to be the element of $GR(p^s, \tau)$ defined as follows:*

- *Writing $a = a_0 + a_1\xi + \cdots + a_{\tau-1}\xi^{\tau-1}$ in polynomial representation,*

$$a \bmod p^s := (a_0 \bmod p^s) + (a_1 \bmod p^s)\xi + \cdots + (a_{\tau-1} \bmod p^s)\xi^{\tau-1}.$$

- *Writing $a = a_0 + a_1 p + \cdots + a_{k-1}p^{k-1}$ in p-adic representation,*

$$a \bmod p^s := a_0 + a_1 p + \cdots + a_{s-1}p^{s-1}.$$

Notice that $a \mapsto a \bmod p^s$ is a ring homomorphism $GR(p^k, \tau) \to GR(p^s, \tau)$.

Observe that, from the definition above and from Theorem 3.6, $a \in \mathrm{GR}(p^k, \tau)^*$ if and only if $a \bmod p \neq 0$. The following lemma is easy to see.

Lemma 3.2. *The image of $GR(p^k, \tau)$ under the homomorphism $a \mapsto a \bmod p^s$ is $GR(p^s, \tau)$. In particular, the image of $GR(p^k, \tau)$ under the homomorphism $a \mapsto a \bmod p$ is $GF(p^\tau)$.*

3.2.1 Galois Ring Extensions

Definition 3.10. *Let $h(\mathtt{X}) \in \mathbb{Z}/p^k\mathbb{Z}$ be a monic polynomial such that $h(\mathtt{X}) \bmod p$ is irreducible, and let $\tau' = \deg(h)$. The quotient ring $\frac{GR(p^k,\tau)[\mathtt{X}]}{h(\mathtt{X})GR(p^k,\tau)[\mathtt{X}]}$ is called a* Galois ring extension *of $GR(p^k, \tau)$ of degree τ'.*

In particular, from the definition above we see that $\mathrm{GR}(p^k, \tau)$ is itself a Galois ring extension of $\mathbb{Z}/p^k\mathbb{Z} = \mathrm{GR}(p^k, 1)$ of degree τ. The following theorem shows that this is no coincidence: every Galois ring extension is itself a Galois ring.

Theorem 3.7 (Theorem 14.23 in [78])**.** *Let $h(\mathtt{X})$ be a monic polynomial such that $h(\mathtt{X}) \bmod p$ is irreducible, and let $\tau' = \deg(h)$. Then*

$$\frac{GR(p^k, \tau)[\mathtt{X}]}{h(\mathtt{X})GR(p^k, \tau)[\mathtt{X}]} \cong GR(p^k, \tau \cdot \tau').$$

3.2.2 Lenstra Constant of a Galois Ring

Now we analyze the Lenstra constant, defined in Definition 3.1, of a Galois ring. We begin with the following example.

Example 3.2. *The Lenstra constant of $\mathbb{Z}/p^k\mathbb{Z}$ is p, given that, from the pigeonhole principle, given any $p+1$ different integers $\alpha_1, \ldots, \alpha_{p+1}$, p must divide at least one of $\{\alpha_{p+1} - \alpha_i\}_{i \in [p]}$, and invertible elements in $\mathbb{Z}/p^k\mathbb{Z}$ are precisely these that are not divisible by p. In particular, the Lenstra constant of $\mathbb{Z}/2^k\mathbb{Z}$ is only 2. This means that the construction of Shamir secret-sharing from Section 3.1, when instantiated with $\mathcal{R} = \mathbb{Z}/2^k\mathbb{Z}$, only allows for $n = 1$ (since the Lenstra constant has to be greater than $n+1$), which unfortunately does not make any sense in the context of secure multiparty computation. This is why Shamir secret-sharing does not directly work over $\mathbb{Z}/2^k\mathbb{Z}$.*

Over fields, the requirement for Shamir secret-sharing to work is that $|\mathbb{F}| > n$. If the field under consideration is too small, one can consider a field extension of suitable size. With the theory we have developed so far we can see that the small size of $\mathbb{F}$ is not the real problem that prevents Shamir secret-sharing from working over $\mathbb{F}$, and instead, the real problem lies in $\mathbb{F}$ having a small Lenstra constant, which is increased by taking a field extension.

It turns out that a similar approach can be taken for Galois rings. As we just saw, the Lenstra constant of $\mathbb{Z}/2^k\mathbb{Z}$ is too small for Shamir secret-sharing to work, but, by taking a Galois ring extension $\mathsf{GR}(2^k, \tau)$ of $\mathbb{Z}/2^k\mathbb{Z}$, the Lenstra constant can be increased. This enables the construction of Shamir secret-sharing over $\mathsf{GR}(2^k, \tau)$, which ultimately enables Shamir secret-sharing over $\mathbb{Z}/2^k\mathbb{Z}$ since this ring is a subring of $\mathsf{GR}(2^k, \tau)$. This is shown in the following theorem.

Theorem 3.8. *The Lenstra constant of $\mathsf{GR}(p^k, \tau)$ is p^τ.*

Proof. Recall that $a \in \mathsf{GR}(p^k, \tau)$ is invertible if and only if $a \bmod p \neq 0$. Hence, the Lenstra constant of $\mathsf{GR}(p^k, \tau)$ is the same as that of $\mathsf{GF}(p^\tau)$, which is $|\mathsf{GF}(p^\tau)| = p^\tau$. □

3.2.3 Efficient Computation over Galois Rings

Computing over $\mathsf{GR}(p^k, \tau)$ for $k > 1$ is not a very well studied computational task, at least when compared to the case of computing over $\mathsf{GR}(p^k, \tau)$ for $k = 1$, which corresponds the field extension $\mathsf{GF}(p^\tau)$. However, in our setting, the degree of the extension τ is approximately equal to $\log(n)$, where n is the desired number of parties. In practical applications n is not expected to be very large. For example, $n = 100$ is already a very large parameter and, since secure multiparty computation is a distributed application, the overhead of coordinating such large number of parties and the communication complexity is likely to be a barrier in practice.

Even for $n = 100$, an extension of degree $\tau = 8$ suffices. Since this degree is relatively small, computation over $\mathsf{GR}(p^k, \tau)$ can be implemented by using the polynomial representation $a = a_0 + a_1\xi + \cdots + a_{\tau-1}\xi^{\tau-1}$, using efficient algorithms for polynomial multiplication, and using a lookup table with the values of ξ^i for $i \geq \tau$. This approach was taken in the original work of [4], which, to the best of our knowledge, constitutes the first implementation of Galois ring arithmetic *in the context of MPC*.

Unfortunately, this approach does not scale well with τ. For a large extension degree, more efficient algorithms must be devised. To the best of our knowledge, the only existing library that addresses computation over this type of rings is the C library ZEN (zenfact.sourceforge.net), and the multiplication method is based on Karatsuba's algorithm. Unfortunately, this method does not perform very well for very large τ. For example, in the original work of [36], some experiments using the ZEN library were run, in order to determine the feasibility of computing over these rings. The results show that, for $k = 64$ and $\tau = 46$, this library can compute less than 22,000 multiplications per second on a single core of a 2.8 GHz i7 processor, which is too small for practical secure multiparty computation.

The above is a problem when extensions of a degree that is equal to a statistical security parameter, which is typically above 40, are needed. This is the case for many recent works that claim to offer concretely practical secure multiparty computation, yet neglect this efficiency issue [22, 65, 71].

3.2.4 Error Correction over a Galois Ring

We present some additional results regarding error correction in the context where the ring $\mathcal{R}$ is a Galois ring. In Section 3.1.4 we saw an algorithm to efficiently solve the task of error correction, where a d-consistent vector $\mathbf{s} \in \mathcal{R}^\ell$ is perturbed in at most e entries, and the goal is to recover $\mathbf{s}$, assuming that $e < (\ell - d)/2$. This method works for any finite commutative ring $\mathcal{R}$, assuming that solving systems of linear equations over this ring is possible.

First, in Section 3.2.4.1 we will show that, indeed, solving systems of linear equations over a Galois ring can be done efficiently. This result implies that the adaptation of the Berlekamp-Welch decoding algorithm from Section 3.1.4 works "from beginning to end" when $\mathcal{R}$ is a Galois ring.

Finally, as an additional contribution, we will show in Section 3.2.4.2 that an efficient decoding algorithm over $\mathrm{GR}(p^k, \tau)$ can be constructed from *any* efficient decoding algorithm over $\mathrm{GR}(p, \tau) = \mathrm{GF}(p^\tau)$. This result was shown in the original work of [2], and it is of major significance as it implies that improvements in the task of error correction over fields, which is a very well studied field, carry over to the setting of a Galois ring.

3.2.4.1 Solving Systems of Linear Equations over a Galois Ring

First, observe that the task of solving a system of u equations in v variables over $\mathrm{GR}(p^k, \tau)$ can be seen as the task of inverting a $\mathrm{GR}(p^k, \tau)$-linear map $M : \mathrm{GR}(p^k, \tau)^v \to \mathrm{GR}(p^k, \tau)^u$. Using the fact that for every $\ell > 0$ there exists a $\mathbb{Z}/p^k\mathbb{Z}$-module isomorphism $\phi_\ell : \mathrm{GR}(p^k, \tau)^\ell \to (\mathbb{Z}/p^k\mathbb{Z})^{\tau\ell}$, we see that the task of inverting ϕ reduces to the fact of inverting the $\mathbb{Z}/p^k\mathbb{Z}$ linear function given by $\phi_u \circ M \circ \phi_v^{-1} : (\mathbb{Z}/p^k\mathbb{Z})^v \to (\mathbb{Z}/p^k\mathbb{Z})^u$. In other words, the task of solving a system of u equations in v variables over $\mathrm{GR}(p^k, \tau)$ can be

reduced to the task of solving a system of $u \cdot \tau$ equations in $v \cdot \tau$ variables over $\mathbb{Z}/p^k\mathbb{Z}$ seen as the task of inverting a $\mathrm{GR}(p^k, \tau)$-linear map $M : \mathrm{GR}(p^k, \tau)^v \to \mathrm{GR}(p^k, \tau)^u$.

In what follows we focus on the task of solving linear equations over $\mathbb{Z}/p^k\mathbb{Z}$. This was addressed in the original work of [51]. We begin with the following simple observation:

Proposition 3.2 (Proposition 1 in [10])**.** *Let A be an $m \times n$ integer matrix, b be an m integer column vector, and p be a prime and k a positive integer. The system of linear equations $Ax \equiv b \bmod p^k$ is feasible (in the finite ring $\mathbb{Z}/p^k\mathbb{Z}$) if and only if $Ax + p^k y = b$ has a solution in $\mathbb{Z}$.*

Observe that the equation $Ax + p^k y = b$ can be written as $[\, A \mid p^k \mathbb{I} \,](\, x \mid y \,)^\intercal = b$, so the task of solving a linear system over $\mathbb{Z}_{p^k}$ is reduced to the task of solving a linear system over $\mathbb{Z}$. It has been shown in [53, 77] that such task admits a polynomial time algorithm, which completes the analysis.

Finally, we end with a more efficient algorithm for the case in which the number of variables equal the number of equations, and the matrix defining the system of linear equations is invertible modulo p. This is captured in the proof of the following result, which is inspired by the proof of Hensel's lemma.

Theorem 3.9. *Let $A \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$. For every $0 < e \leq k$, if A is invertible modulo p^e, then A is invertible modulo p^{e+1}. Furthermore, the inverse of A modulo p^{e+1} is efficiently computable from the inverse modulo p^e.*

Proof. Assume that A is invertible modulo p^e, so there exists $A' \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$ such that $A \cdot A' \equiv \mathbb{I} \bmod p^e$. Let us write $A \cdot A' = \mathbb{I} + p \cdot Q$ for some $Q \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$. Define $P \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$ as $P = -A' \cdot Q$. It holds then that $A \cdot P = -A \cdot A' \cdot Q$, so $A \cdot P \equiv -Q \bmod p^e$, and in particular $p \cdot A \cdot P \equiv -p \cdot Q \bmod p^{e+1}$.

Now let $A'' \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$ be defined as $A'' = A' + p \cdot P$. We have that $A \cdot A'' = A \cdot A' + p \cdot A \cdot P = \mathbb{I} + p \cdot Q + p \cdot A \cdot P$, which modulo p^{e+1} becomes $A \cdot A'' \equiv \mathbb{I} + p \cdot Q - p \cdot Q \equiv \mathbb{I} \bmod p^{e+1}$, which shows that A is invertible modulo p^{e+1}. We also see that the inverse of A modulo p^{e+1}, A'', is efficiently computable from A'. □

From this theorem we get the following corollary.

Corollary 3.2. *Let $A \in (\mathbb{Z}/p^k\mathbb{Z})^{r \times r}$. If A is invertible modulo p, then it is invertible modulo p^k and its inverse is efficiently computable.*

Unfortunately, this approach does not naturally extend to the case in which A is not invertible. As an example, take $r = 1$, $A = p$ and let $y = p$. Modulo p both A and y become zero, so any x satisfies the equation $Ax \equiv y \bmod p$. However, not all solutions mod p can be extended to mod p^k since, if $Ax \equiv y \bmod p^k$ then $x \equiv 1 \bmod p^{k-1}$.

3.2.4.2 Error Correction over $\mathrm{GR}(p^k, \tau)$ from Error Correction over $\mathrm{GF}(p^\tau)$

Let e, ℓ, d be positive integers with $e < (\ell - d)/2$.

Definition 3.11. *A decoding algorithm over $GR(p^k, \tau)$ takes as input $\mathbf{c}$, where $\mathbf{c} = \mathbf{s} + \boldsymbol{\delta}$ for some d-consistent vector $\mathbf{s} \in GR(p^k, \tau)^\ell$ and a vector $\boldsymbol{\delta} \in GR(p^k, \tau)^\ell$ with at most e non-zero entries, and returns $(\mathbf{s}, \boldsymbol{\delta})$.*

Let $(\mathbf{s}, \boldsymbol{\delta}) \leftarrow \mathrm{Decode}_1(\mathbf{c})$ be a decoding algorithm over $\mathrm{GR}(p, \tau) = \mathrm{GF}(p^\tau)$. We will show how to obtain, by calling Decode_1 k times, a decoding algorithm Decode_k over $\mathrm{GR}(p^k, \tau)$.

Algorithm Decode_k

Input: $\mathbf{c} \in \mathrm{GR}(p^k, \tau)^\ell$, where $\mathbf{c} = \mathbf{s} + \boldsymbol{\delta}$ for some d-consistent vector $\mathbf{s} \in \mathrm{GR}(p^k, \tau)^\ell$ and a vector $\boldsymbol{\delta} \in \mathrm{GR}(p^k, \tau)^\ell$ with at most e non-zero entries
Output: The vector $\mathbf{s}$.

The algorithm proceeds as follows.

- Let $\boldsymbol{\gamma}_0 = \mathbf{c}$. For $i = 0, \ldots, k-1$:
  1. Compute $(\boldsymbol{\sigma}_i, \boldsymbol{\epsilon}_i) = \mathrm{Decode}_1(\boldsymbol{\gamma}_i \bmod p)$
  2. Let $\boldsymbol{\gamma}_{i+1} = (\boldsymbol{\gamma}_i - (\boldsymbol{\sigma}_i + \boldsymbol{\epsilon}_i))/p$.[a]
- Output $(\boldsymbol{\sigma}, \boldsymbol{\epsilon})$ with $\boldsymbol{\sigma} = \sum_{i=0}^{k-1} \boldsymbol{\sigma}_i p^i$ and $\boldsymbol{\epsilon} = \sum_{i=0}^{k-1} \boldsymbol{\epsilon}_i p^i$.

[a] It will be the case that $\boldsymbol{\gamma}_i - (\boldsymbol{\sigma}_i + \boldsymbol{\epsilon}_i)$ is divisible by p.

Theorem 3.10. *Decode_k is a correct decoding algorithm.*

Proof. Let us write $\mathbf{s} = \sum_{i=0}^{k-1} \mathbf{s}_i p^i$ and $\boldsymbol{\delta} = \sum_{i=0}^{k-1} \boldsymbol{\delta}_i p^i$, using the p-adic representation from Theorem 3.6. Let $\mathbf{c}_i = \mathbf{s}_i + \boldsymbol{\delta}_i$, so $\mathbf{c} = \sum_{i=0}^{k-1} \mathbf{c}_i p^i$ We begin with the following claim:

Claim 3.3. *Algorithm Decode_k satisfies the following invariant: for every $i = 0, \ldots, k-1$: $\boldsymbol{\gamma}_i = \sum_{j=i}^{k-1} \mathbf{c}_j p^{j-i}$, $\boldsymbol{\sigma}_i = \mathbf{s}_i$, and $\boldsymbol{\epsilon}_i = \boldsymbol{\delta}_i$.*

Proof. To prove the claim we proceed by induction.

($i = 0$). Clearly $\boldsymbol{\gamma}_0 = \mathbf{c} = \sum_{j=0}^{k-1} \mathbf{c}_j p^j$. Also, $(\boldsymbol{\gamma}_0 \bmod p) = \mathbf{c}_0 = \mathbf{s}_0 + \boldsymbol{\delta}_0$, so the output of $\mathrm{Decode}_1(\boldsymbol{\gamma}_0 \bmod p)$ is $\boldsymbol{\sigma}_0 = \mathbf{s}_0$ and $\boldsymbol{\epsilon}_0 = \boldsymbol{\delta}_0$.

($i \implies i+1$). Now, assume the claim holds for some i, that is, $\boldsymbol{\gamma}_i = \sum_{j=i}^{k-1} \mathbf{c}_j p^{j-i}$, and let us show that the claim still holds for $i+1$. Since $(\boldsymbol{\gamma}_i \bmod p) = \mathbf{c}_i = \mathbf{s}_i + \boldsymbol{\delta}_i$, we have that $\boldsymbol{\sigma}_i = \mathbf{s}_i$ and $\boldsymbol{\epsilon}_i = \boldsymbol{\delta}_i$, so $\boldsymbol{\gamma}_{i+1} = (\boldsymbol{\gamma}_i - (\mathbf{s}_i + \boldsymbol{\delta}_i))/p = (\sum_{j=i}^{k-1} \mathbf{c}_j p^{j-i} - (\mathbf{s}_i + \boldsymbol{\delta}_i))/p = (\sum_{j=i+1}^{k-1} \mathbf{c}_j p^{j-i})/p = \sum_{j=i+1}^{k-1} \mathbf{c}_j p^{j-(i+1)}$.

□

From the invariant above we see that $\boldsymbol{\sigma} = \sum_{i=0}^{k-1} \mathbf{s}_i p^i$ and $\boldsymbol{\epsilon} = \sum_{i=0}^{k-1} \boldsymbol{\delta}_i p^i$, as required. □

3.2.5 Concatenating Secret-Sharing

Finally, before we describe our secure computation protocols, we describe a tool that will be useful in Section 3.3.3 when we present a protocol for generating shares $[\![s]\!]_t$ over a Galois ring $\mathrm{GR}(2^k, \tau)$, where the secret s is guaranteed to lie in a smaller Galois ring $\mathrm{GR}(2^k, \tau')$ for some $\tau' | \tau$.

Let $\mathrm{GR}(p^k, \tau\ell)$ be a Galois ring extension of $\mathrm{GR}(p^k, \tau)$ of degree ℓ, and represent elements $a \in \mathrm{GR}(p^k, \tau\ell)$ using the polynomial representation as $a = a_0 + a_1\xi + \cdots + a_{\ell-1}\xi^{\ell-1}$ where $a_i \in \mathrm{GR}(p^k, \tau)$. Let $\phi : \mathrm{GR}(p^k, \tau)^\ell \to \mathrm{GR}(p^k, \tau\ell)$ be the $\mathrm{GR}(p^k, \tau)$-module isomorphism given by $(a_0, \ldots, a_{\ell-1}) \mapsto a_0 + a_1\xi + \cdots + a_{\ell-1}\xi^{\ell-1}$.

Proposition 3.3. *Let $[\![s_i]\!]_d = (s_{i1}, \ldots, s_{in})$ for $i \in [\ell]$ be a series of secret-shared values over $\mathrm{GR}(p^k, \tau)$. Then $(s_1, \ldots, s_n)$ are d-consistent sharings of s, where $s_j = \phi\left((s_{ij})_{i=1}^{\ell}\right)$ and $s = \phi\left((s_i)_{i=1}^{\ell}\right)$.*

Proof. By assumption, there exist polynomials $f_i(\mathtt{X}) \in (\mathrm{GR}(p^k, \tau))_{\leq d}[\mathtt{X}]$ for $i \in [\ell]$ such that $f_i(\alpha_j) = s_{ij}$ for $j \in [n]$ and $f(\alpha_0) = s_i$. Now, let $f(\mathtt{X}) \in (\mathrm{GR}(p^k, \tau))_{\leq d}[\mathtt{X}]$ be defined as $f(\mathtt{X}) = \sum_{i=1}^{\ell} f_i(\mathtt{X}) \cdot \xi^{i-1}$. For every $i \in [\ell]$ and $j \in [n]$, we have the following:

$$f(\alpha_j) = \sum_{i=1}^{\ell} f_i(\alpha_j) \cdot \xi^{i-1} = \sum_{i=1}^{\ell} s_{ij} \cdot \xi^{i-1} = \phi\left((s_{ij})_{i=1}^{\ell}\right) = s_j,$$

and similarly $f(\alpha_0) = s$. □

From the above proposition, we see that, in words, given a series $([\![s_1]\!]_d, \ldots, [\![s_\ell]\!]_d)$ of secret-shared values over $\mathrm{GR}(p^k, \tau)$, the parties can concatenate their shares and apply ϕ to the result to obtain shares of $s = \phi(s_1, \ldots, s_\ell)$ over $\mathrm{GR}(p^k, \tau\ell)$. This local procedure is denoted by $[\![s]\!]_d \leftarrow \phi([\![s_1]\!]_d, \ldots, [\![s_\ell]\!]_d)$. In a similar way, the parties can compute $([\![s_1]\!]_d, \ldots, [\![s_\ell]\!]_d) \leftarrow \phi^{-1}([\![s]\!]_d)$.

For clarity, in some places we will make use of a superscript that explicitly denotes the ring over which the sharing is done. For example, the local procedure from above is sometimes denoted as $[\![s]\!]_d^{\mathrm{GR}(p^k,\tau\ell)} \leftarrow \phi([\![s_1]\!]_d^{\mathrm{GR}(p^k,\tau)}, \ldots, [\![s_\ell]\!]_d^{\mathrm{GR}(p^k,\tau)})$. Finally, since $\mathrm{GR}(p^k, \tau)$ is naturally embedded into $\mathrm{GR}(p^k, \tau\ell)$, shares over the former ring can also be naturally regarded as shares over the latter ring.

3.3 MPC over GR$(2^k, \tau)$

In this section we present a secure multiparty computation protocol for arithmetic circuits defined over a Galois ring $\mathcal{R} = \mathrm{GR}(2^k, \tau)$.[3] Our ultimate goal is to securely compute an arithmetic circuit $F : (\mathbb{Z}/2^k\mathbb{Z})^{I_F} \to (\mathbb{Z}/2^k\mathbb{Z})^{O_F}$. To achieve this, first we interpret this circuit as an arithmetic circuit $F' : \mathcal{R}^{I_F} \to \mathcal{R}^{O_F}$ over $\mathcal{R}$, for which we can obtain secure computation protocols, and observe that $F' \mid_{\mathbb{Z}/2^k\mathbb{Z}} = F$. As a result, to compute F, the parties can compute F' over $\mathcal{R}$, ensuring the inputs are in $\mathbb{Z}/2^k\mathbb{Z}$.

Throughout the rest of this chapter, let $\mathcal{A} = \mathrm{GR}(2^k, \rho)$, with $\rho|\tau$. Let χ be such that $\rho \cdot \chi = \tau$. Our protocol from this section will enable secure computation of an arithmetic circuit $F' : \mathcal{R}^{I_F} \to \mathcal{R}^{O_F}$, while ensuring that the inputs lie in $\mathcal{A}$. As a result, the arithmetic circuit that is actually computed is $F : \mathcal{A}^{I_F} \to \mathcal{A}^{O_F}$ given by $F = F' \mid_{\mathcal{A}}$. This approach is necessary since secure computation over $\mathcal{A}$ may not be directly possible, simply because of the fact that the Lenstra constant of $\mathcal{A}$ may not be large enough to allow for Shamir secret-sharing. By performing computation over $\mathcal{R}$, which is possible since τ will be chosen so that Shamir secret-sharing works over $\mathcal{R}$, and by enforcing inputs to lie in $\mathcal{A}$, we get secure computation of the desired circuit over $\mathcal{A}$.

Assume that $t < n/3$. As usual, we let $\mathcal{H}, \mathcal{C} \subseteq [n]$ be the set of indexes corresponding to honest and corrupt parties, respectively. Through the rest of this section we let $\mathcal{R} = \mathrm{GR}(2^k, \tau)$ with $2^\tau \geq n + 1$. In particular, from Theorem 3.8 in Section 3.2.2, we see that the construction of Shamir secret-sharing from Section 3.1 can be instantiated over the ring $\mathcal{R}$ with n parties. Later in Section 3.3.1.1 we will strengthen this to $2^\tau \geq 2n$ so that the construction of hyper-invertible matrices from that section works over $\mathcal{R}$.

This section is organized as follows. First, in Section 3.3.1 we show how to generate the so-called double-sharings, which is the necessary preprocessing material to handle multiplications securely, which we show how to do in Section 3.3.2. Then, in Section 3.3.3 we show how to get shares of random values in the subring $\mathcal{A}$, which will be particularly useful for enforcing inputs to lie in this set, as we show in Section 3.3.4. Finally, in Section 3.3.5 we present the final secure computation protocol.

3.3.1 Double-Sharings

We begin by presenting the functionality for generating double-sharings, which is a pair $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$ where r is uniformly random and unknown to the adversary. These are necessary to process multiplications securely. This is formalized by the following functionality.

[3] The results are still true over $\mathrm{GR}(p^k, d)$ but we focus on $\mathrm{GR}(2^k, d)$ since we are ultimately interested in computation modulo powers of two.

Functionality $\mathcal{F}_{\text{D.Shar}}$

- Sample $r \in_R \mathcal{R}$.
- Receive $\{(r_j, r'_j)\}_{j\in\mathcal{C}}$ from the adversary.
- Run $(r_1, \dots, r_n) \leftarrow \text{Share}_t(r, \{r_j\}_{j\in\mathcal{C}})$ and $(r'_1, \dots, r'_n) \leftarrow \text{Share}_{2t}(r, \{r'_j\}_{j\in\mathcal{C}})$.
- For every $j \in \mathcal{H}$, send (r_j, r'_j) to P_j.

3.3.1.1 Hyper-Invertible Matrices

Now we consider hyper-invertible matrices, which constitute a necessary construction for our protocol to preprocess double-sharings.

Definition 3.12. *Let $\mathcal{R}$ be any commutative ring. A matrix $\mathbf{M} \in \mathcal{R}^{u\times v}$ is said to be* hyper-invertible *if every square sub-matrix obtained by taking subsets of the rows and columns of $\mathbf{M}$ is invertible.*

Example 3.3. *An example of a hyper-invertible matrix is the following. Let $\{\alpha_1, \dots, \alpha_v, \beta_1, \dots, \beta_u\} \subseteq \mathcal{R}$ be an exceptional set. Let $\mathbf{A} = \text{Van}^{v\times v}(\alpha_1, \dots, \alpha_v)$ and $\mathbf{B} = \text{Van}^{u\times v}(\beta_1, \dots, \beta_u)$, and let $\mathbf{M} = \mathbf{B}\mathbf{A}^{-1} \in \mathcal{R}^{u\times v}$. We claim that $\mathbf{M}$ is a hyper-invertible matrix. To see this, let $R \subseteq [u]$ and $C \subseteq [v]$ with $|R| = |C| = \ell$. Assume without loss of generality that $R = C = \{1, \dots, \ell\}$.*

To show that $\mathbf{M}$ is hyper-invertible we need to show that $\mathbf{M}[R, C]$ is invertible. Let $\mathbf{G} \in \mathcal{R}^{v\times\ell}$ be defined as

$$\mathbf{G} = \begin{pmatrix} \text{Van}^{(v-\ell)\times v}(\alpha_{\ell+1}, \dots, \alpha_v) \\ \text{Van}^{\ell\times v}(\beta_1, \dots, \beta_\ell) \end{pmatrix}^{-1} \cdot \begin{pmatrix} \mathbf{0}_{(v-\ell)\times\ell} \\ \mathbb{I}_{\ell\times\ell} \end{pmatrix},$$

and let $\mathbf{W} = \mathbf{A}[C, \cdot] \cdot \mathbf{G} \in \mathcal{R}^{\ell\times\ell}$. We claim that $(\mathbf{M}[R, C])^{-1} = \mathbf{W}$. To be able to prove this, we first show a series of claims.

Claim 3.4. $\mathbf{A}[C^c, \cdot] \cdot \mathbf{G} = \mathbf{0}_{(v-\ell)\times\ell}$

Proof. From definition over $\mathbf{G}$, we see that

$$\begin{pmatrix} \text{Van}^{(v-\ell)\times v}(\alpha_{\ell+1}, \dots, \alpha_v) \\ \text{Van}^{\ell\times v}(\beta_1, \dots, \beta_\ell) \end{pmatrix} \cdot \mathbf{G} = \begin{pmatrix} \mathbf{0}_{(v-\ell)\times\ell} \\ \mathbb{I}_{\ell\times\ell} \end{pmatrix},$$

and since $\text{Van}^{(v-\ell)\times v}(\alpha_{\ell+1}, \dots, \alpha_v) = \mathbf{A}[C^c, \cdot]$, we obtain that $\mathbf{A}[C^c, \cdot] \cdot \mathbf{G} = \mathbf{0}_{(v-\ell)\times\ell}$. □

Claim 3.5. $\mathbf{B}[R, \cdot] \cdot \mathbf{G} = \mathbb{I}_{\ell\times\ell}$

Proof. From definition over $\mathbf{G}$, we see that

$$\begin{pmatrix} \mathsf{Van}^{(v-\ell)\times v}(\alpha_{\ell+1}, \ldots, \alpha_v) \\ \mathsf{Van}^{\ell\times v}(\beta_1, \ldots, \beta_\ell) \end{pmatrix} \cdot \mathbf{G} = \begin{pmatrix} \mathbf{0}_{(v-\ell)\times\ell} \\ \mathbb{I}_{\ell\times\ell} \end{pmatrix},$$

and since $\mathsf{Van}^{\ell\times v}(\beta_1, \ldots, \beta_\ell) = \mathbf{B}[R, \cdot]$, we obtain that $\mathbf{B}[R, \cdot] \cdot \mathbf{G} = \mathbb{I}_{\ell\times\ell}$. □

Claim 3.6. $\mathbf{A} \cdot \mathbf{G} = \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix}$

Proof. Making use of claim 3.4:

$$\mathbf{A} \cdot \mathbf{G} = \begin{pmatrix} \mathbf{A}[C, \cdot] \\ \mathbf{A}[C^{\mathsf{c}}, \cdot] \end{pmatrix} \cdot \mathbf{G} = \begin{pmatrix} \mathbf{A}[C, \cdot] \cdot \mathbf{G} \\ \mathbf{A}[C^{\mathsf{c}}, \cdot] \cdot \mathbf{G} \end{pmatrix} = \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix}$$

□

Claim 3.7. $\mathbf{A}^{-1}[\cdot, C] \cdot \mathbf{W} = \mathbf{A}^{-1} \cdot \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix}$

Proof.

$$\mathbf{A}^{-1} \cdot \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix} = (\mathbf{A}^{-1}[\cdot, C] \mid \mathbf{A}^{-1}[\cdot, C^{\mathsf{c}}]) \cdot \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix} = \mathbf{A}^{-1}[\cdot, C] \cdot \mathbf{W}.$$

□

Now, to see the main claim that $(\mathbf{M}[R, C])^{-1} = \mathbf{W}$*, we proceed as follows:*

$$\begin{aligned} \mathbf{M}[R, C] \cdot \mathbf{W} &= (\mathbf{B}[R, \cdot] \cdot \mathbf{A}^{-1}[\cdot, C]) \cdot \mathbf{W} \\ &= \mathbf{B}[R, \cdot] \cdot \mathbf{A}^{-1} \cdot \begin{pmatrix} \mathbf{W} \\ \mathbf{0}_{(v-\ell)\times\ell} \end{pmatrix} && \text{from Claim 3.7} \\ &= \mathbf{B}[R, \cdot] \cdot \mathbf{A}^{-1} \cdot \mathbf{A} \cdot \mathbf{G} && \text{from Claim 3.6} \\ &= \mathbf{B}[R, \cdot] \cdot \mathbf{G} \\ &= \mathbb{I}_{\ell\times\ell} && \text{from Claim 3.5} \end{aligned}$$

From now on we require $2^\tau \geq 2n$ to allow the construction of a hyper-invertible matrix from Example 3.3 with $u = v = n$. Let $\mathbf{M} \in \mathcal{R}^{n\times n}$ be a hyper-invertible matrix.

3.3.1.2 Generating Double-Sharings

The following protocol $\Pi_{\text{D.Shar}}$ can be used to generated double-sharings, or more precisely, to instantiate Functionality $\mathcal{F}_{\text{D.Shar}}$. This is a direct adaptation of the corresponding protocol in [42] over fields. Notice that the protocol produces a set of $n - 2t$ double-sharings, rather than a single one, as done in Functionality $\mathcal{F}_{\text{D.Shar}}$. The effect of this

is that, when we present the simulation-based proof that Protocol $\Pi_{\text{D.Shar}}$ instantiates $\mathcal{F}_{\text{D.Shar}}$ in Theorem 3.11 below, multiple calls to $\mathcal{F}_{\text{D.Shar}}$ will be done by the simulator and the honest parties.

Protocol $\Pi_{\text{D.Shar}}$

Output: A set of double sharings $\{(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})\}_{i=2t+1}^{n}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathcal{R}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$, but observe that corrupt parties may distribute shares *inconsistently*.
2. The parties compute locally the following shares:
$$\begin{pmatrix} \llbracket r_1 \rrbracket_t \\ \llbracket r_2 \rrbracket_t \\ \vdots \\ \llbracket r_n \rrbracket_t \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} \llbracket s_1 \rrbracket_t \\ \llbracket s_2 \rrbracket_t \\ \vdots \\ \llbracket s_n \rrbracket_t \end{pmatrix}, \qquad \begin{pmatrix} \llbracket r'_1 \rrbracket_{2t} \\ \llbracket r'_2 \rrbracket_{2t} \\ \vdots \\ \llbracket r'_n \rrbracket_{2t} \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} \llbracket s'_1 \rrbracket_{2t} \\ \llbracket s'_2 \rrbracket_{2t} \\ \vdots \\ \llbracket s'_n \rrbracket_{2t} \end{pmatrix}.$$
3. For each $i \in [2t]$, all the parties send their shares of $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$ to P_i.
4. Upon receiving these shares, each P_i for $i = 1, \ldots, 2t$ checks that the received sharings of $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$ are t and $2t$-consistent, respectively. If any of the sharings is not consistent, or if both are but the reconstructed value is not equal in both cases, P_i broadcasts abort to all parties and halts.
5. If no party sends an abort message in the previous step, then the parties output the double-sharings $(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})$ for $i = 2t+1, \ldots, n$.

Theorem 3.11. *Protocol $\Pi_{\text{D.Shar}}$ instantiates functionality $\mathcal{F}_{\text{D.Shar}}$ with perfect security against an active adversary corrupting $t < n/3$ parties.*

Proof. We define the simulator $\mathcal{S}$ below.

1. - The virtual honest parties $\overline{P}_i$ sample $\overline{s}_i \in_R \mathcal{R}$ and call $(\overline{s}_{i1}, \ldots, \overline{s}_{in}) \leftarrow \text{Share}_t(\overline{s}_i)$ and $(\overline{s}'_{i1}, \ldots, \overline{s}'_{in}) \leftarrow \text{Share}_{2t}(\overline{s}_i)$. Then $\overline{P}_i$ sends $(\overline{s}_{ij}, \overline{s}'_{ij})$ to P_j for $j \in \mathcal{C}$.
   - Every simulated party $\overline{P}_i$ receives a pair $(\overline{s}_{ji}, \overline{s}'_{ji})$ as their share of $\llbracket \overline{s}_j \rrbracket_t$ and $\llbracket \overline{s}_j \rrbracket_{2t}$ from each P_j with $j \in \mathcal{C}$.
2. Each emulated party $\overline{P}_i$ computes
$$\begin{pmatrix} \overline{r}_{1i} \\ \overline{r}_{2i} \\ \vdots \\ \overline{r}_{ni} \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} \overline{s}_{1i} \\ \overline{s}_{2i} \\ \vdots \\ \overline{s}_{ni} \end{pmatrix}, \qquad \begin{pmatrix} \overline{r}'_{1i} \\ \overline{r}'_{2i} \\ \vdots \\ \overline{r}'_{ni} \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} \overline{s}'_{1i} \\ \overline{s}'_{2i} \\ \vdots \\ \overline{s}'_{ni} \end{pmatrix}.$$
3. - Every emulated party $\overline{P}_j$ sends $(\overline{r}_{ij}, \overline{r}'_{ij})$ to P_i for $i \in \mathcal{C} \cap [2t]$.
   - Every emulated party $\overline{P}_i$ with $i \in [2t]$ receives $(\overline{r}_{ij}, \overline{r}'_{ij})$ from P_j for $j \in \mathcal{C}$.
4. Every emulated $\overline{P}_i$ with $i \in [2t]$ checks if $(\overline{r}_{i1}, \ldots, \overline{r}_{in})$ and $(\overline{r}'_{i1}, \ldots, \overline{r}'_{in})$ are t and $2t$-consistent, respectively, and that their underlying secrets are the same. If this does not hold then $\overline{P}_i$ aborts.

5. If no abort was produced in the previous step, $\mathcal{S}$ computes $(\overline{r}_{i1}, \dots, \overline{r}_{in}) \leftarrow \text{RecShares}((\overline{r}_{ij})_{j\in\mathcal{H}})$ for $i \in \{2t+1, \dots, n\}$. Then $\mathcal{S}$ sends $\{(\overline{r}_{ij}, \overline{r}'_{ij})\}_{j\in\mathcal{C}}$ for $i \in \{2t+1, \dots, n\}$ to $\mathcal{F}_{\text{D.Shar}}$.

To argue indistinguishability we proceed by describing the view of the adversary in each of the two worlds as the computation progresses in the following diagram.

Real world	**Ideal world**
1. From Thm. 3.2, the adversary's shares $\{s_{ij}\}_{j\in\mathcal{C}}$ for $i \in \mathcal{H}$ look uniformly random.	1. From Thm. 3.2, the adversary's shares $\{\overline{s}_{ij}\}_{j\in\mathcal{C}}$ for $i \in \mathcal{H}$ also look uniformly random.
2. Local operations	2. Local operations
3. The adversary gets values $\{r_{ij}\}_{j\in\mathcal{H}}$ and $\{r'_{ij}\}_{j\in\mathcal{H}}$ for $i \in \mathcal{C} \cap [2t]$.	3. The adversary gets values $\{\overline{r}_{ij}\}_{j\in\mathcal{H}}$ and $\{\overline{r}'_{ij}\}_{j\in\mathcal{H}}$ computed in the same way as in the real world, for $i \in \mathcal{C} \cap [2t]$.
4. The parties perform the check	4. The emulated parties perform the same check.
Honest parties may abort. However, since the two worlds are indistinguishable up to this point, honest parties abort in the real world if and only if emulated honest parties abort in the ideal world, and in this case $\mathcal{S}$ instructs $\mathcal{F}_{\text{D.Shar}}$ to make the real honest parties abort. In what follows we assume the parties do not abort.	
5. The honest parties P_j output $\{(r_{ij}, r'_{ij})\}_{i=2t+1}^{n}$.	5. The (real) honest parties P_j output uniformly random values $\{(\overline{\overline{r}}_{ij}, \overline{\overline{r}}'_{ij})\}_{i=2t+1}^{n}$, where $\{\overline{\overline{r}}_{ij}\}_{j\in\mathcal{H}} \cup \{\overline{r}_{ij}\}_{j\in\mathcal{C}}$ is t-consistent and $\{\overline{\overline{r}}'_{ij}\}_{j\in\mathcal{H}} \cup \{\overline{r}'_{ij}\}_{j\in\mathcal{C}}$ is $2t$-consistent, for $i = 2t+1, \dots, n$, and the two underlying secrets are uniformly random and are the same.

To complete the proof of indistinguishability, we only need to show that the values output by the honest parties in the real execution are indistinguishable from these output by (real) honest parties in the ideal execution. For this, we make use of the following claims. Below, we let $\mathcal{I} = \{2t+1, \dots, n\}$.

Claim 3.8. *Assume that the parties do not abort in the protocol execution. Let $\mathcal{I} = \{2t+1, \dots, n\}$. Then, the values $\{(r_{ij}, r'_{ij})\}_{i=2t+1}^{n}$ output by the honest parties P_j with $j \in \mathcal{H}$ satisfy the following:*

- *For $i \in \mathcal{I}$, $\{r_{ij}\}_{j\in\mathcal{H}}$ is t-consistent with the values $\{r_{ij}\}_{j\in\mathcal{C}}$ held by the adversary. Let r_i be the underlying secret.*
- *For $i \in \mathcal{I}$, $\{r'_{ij}\}_{j\in\mathcal{H}}$ is $2t$-consistent with the values $\{r'_{ij}\}_{j\in\mathcal{C}}$ held by the adversary. Let r'_i be the underlying secret.*
- *For $i \in \mathcal{I}$ it holds that $r_i = r'_i$, and r_i is uniformly random.*

Proof. Let $\mathcal{A} \subseteq \mathcal{H} \cap [2t]$ with $|\mathcal{A}| = |\mathcal{C}| = t$. Notice that this set exists since there are at least $2t - t = t$ honest parties with indexes in $[2t]$. Now, using block-decomposition of

the hyper-invertible matrix $\mathbf{M}$, we see that, for every $j \in [n]$:

$$(r_{ij})_{i\in\mathcal{A}} = \mathbf{M}[\mathcal{A},\mathcal{C}] \cdot (s_{ij})_{i\in\mathcal{C}} + \mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_{ij})_{i\in\mathcal{C}^c}.$$

Since $\mathbf{M}$ is hyper-invertible, $\mathbf{M}[\mathcal{A},\mathcal{C}] \in \mathcal{R}^{t\times t}$ is invertible, so we can write

$$(s_{ij})_{i\in\mathcal{C}} = \mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r_{ij})_{i\in\mathcal{A}} - \mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_{ij})_{i\in\mathcal{C}^c}.$$

The above holds similarly for the shares r'_{ij}/s'_{ij}.

Now, since the parties did not abort, we have that the check performed by the *honest* parties in $[2t]$ passes, so, for every $i \in \mathcal{A}$, $(r_{ij})_{j=1}^n$ is a t-consistent vector with secret r_i, and $(r'_{ij})_{j=1}^n$ is $2t$-consistent with the same secret r_i. In particular, because of the linearity of d-consistency, $\left(\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r_{ij})_{i\in\mathcal{A}}\right)_{j=1}^n$ is t-consistent with secret $\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r_i)_{i\in\mathcal{A}}$, and $\left(\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r'_{ij})_{i\in\mathcal{A}}\right)_{j=1}^n$ is $2t$-consistent with the same secret $\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r_i)_{i\in\mathcal{A}}$.

Similarly, since the parties with indexes in $\mathcal{C}^c$ are honest, they follow the protocol specification faithfully and therefore, for $i \in \mathcal{C}^c$, it holds that the shares $(s_{ij})_{j=1}^n$ are t-consistent with secret s_i, and the shares $(s'_{ij})_{j=1}^n$ are t-consistent with the same secret s_i. From this, it holds that the shares $\left(\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_{ij})_{i\in\mathcal{C}^c}\right)_{j=1}^n$ are t-consistent with a secret $\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_i)_{i\in\mathcal{C}^c}$, and the shares $\left(\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s'_{ij})_{i\in\mathcal{C}^c}\right)_{j=1}^n$ are $2t$-consistent with the same secret $\mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_i)_{i\in\mathcal{C}^c}$.

Putting together the observations above, we see that, for each $i \in \mathcal{C}$, $(s_{ij})_{j=1}^n$ is t-consistent with the secret $s_i = \mathbf{M}[\mathcal{A},\mathcal{C}]^{-1} \cdot (r_i)_{i\in\mathcal{A}} - \mathbf{M}[\mathcal{A},\mathcal{C}]^{-1}\mathbf{M}[\mathcal{A},\mathcal{C}^c] \cdot (s_i)_{i\in\mathcal{C}^c}$, and $(s'_{ij})_{j=1}^n$ is $2t$-consistent with the same secret s_i. Since, for $j \in [n]$:

$$(r_{ij})_{i\in\mathcal{I}} = \mathbf{M}[\mathcal{I},\mathcal{C}] \cdot (s_{ij})_{i\in\mathcal{C}} + \mathbf{M}[\mathcal{I},\mathcal{C}^c] \cdot (s_{ij})_{i\in\mathcal{C}^c},$$

and similarly for r'_{ij}, we see that, for $i \in \mathcal{I}$, $(r_{ij})_{j=1}^n$ is t-consistent, $(r'_{ij})_{j=1}^n$ is $2t$-consistent, and the two secrets are the same, as required by the claim. □

Claim 3.9. *Assume that the parties do not abort in the protocol execution. For $i \in \mathcal{I}$, let r_i be the underlying secret of the t-consistent values $(r_{ij})_{j=1}^n$ (which is the same as the secret of the $2t$-consistent values $(r'_{ij})_{j=1}^n$). Then, each r_i is uniformly random.*

Proof. Let $\mathcal{A} \subseteq \mathcal{C}^c$ with $|\mathcal{A}| = n - 2t$, which exists since $|\mathcal{C}^c| = n - t \geq n - 2t$. Observe that $(r_i)_{i\in\mathcal{I}} = \mathbf{M}[\mathcal{I},\mathcal{A}^c] \cdot (s_i)_{i\in\mathcal{A}^c} + \mathbf{M}[\mathcal{I},\mathcal{A}] \cdot (s_i)_{i\in\mathcal{A}}$. Since $\mathbf{M}$ is hyper-invertible, $\mathbf{M}[\mathcal{I},\mathcal{A}] \in \mathcal{R}^{(n-2t)\times(n-2t)}$ is invertible, which means that $(r_i)_{i\in\mathcal{I}}$ is in a one-to-one correspondence with $(s_i)_{i\in\mathcal{A}}$. The latter values are uniformly random values since they are sampled by honest parties according to the protocol description, so the same holds for $(r_i)_{i\in\mathcal{I}}$. □

From the claim above, we see that the output produced in the real execution follows exactly the same distribution as in the ideal world, which completes the proof. □

Communication complexity of $\Pi_{\text{D.Shar}}$. Protocol $\Pi_{\text{D.Shar}}$ first requires each party P_i to send $2n$ ring elements to the other parties,[4] which amounts to a total of $2n^2$ messages sent. This is followed by the parties sending a pair of shares to $2t$ of the parties, which amounts to a total of $4nt$ ring elements being sent. In total, we see that the protocol communicates $2n^2 + 4nt$ ring elements. However, since the protocol produces $n - 2t$ sharings, we see that the *amortized* communication per double-share is $\frac{2n^2+4nt}{n-2t}$ ring elements. For the case in which $n = 3t + 1$, the expression above becomes $\frac{6n^2+4n(n-1)}{n+2} \approx 10n$.

3.3.2 Secure Multiplication

In this section we show how to make use of the double-sharings protocol from Section 3.3.1 to securely compute multiplications, that is, obtain $[\![xy]\!]_t$ from $[\![x]\!]_t$ and $[\![y]\!]_t$. This is captured formally by the following functionality. Observe that the functionality receives the (t-consistent) shares of the two factors from the honest parties, and based on this computes the shares corresponding to the corrupt parties and send these to the adversary. This is necessary for the simulation, since, although the actual corrupt parties may already know their shares from previous parts of the protocol execution, the simulator does not know these.

Functionality $\mathcal{F}_{\text{Mult}}$

1. Receive (x_i, y_i) from each honest party P_i.
2. Call $x \leftarrow \text{RecSecret}_t(\{x_i\}_{i\in\mathcal{H}})$, $(x_1, \ldots, x_n) \leftarrow \text{RecShares}_t(\{x_i\}_{i\in\mathcal{H}})$, $y \leftarrow \text{RecSecret}_t(\{y_i\}_{i\in\mathcal{H}})$ and $(y_1, \ldots, y_n) \leftarrow \text{RecShares}_t(\{y_i\}_{i\in\mathcal{H}})$
3. Send $\{(x_i, y_i)\}_{i\in\mathcal{C}}$ to the adversary.
4. Wait for $\{z_i\}_{i\in\mathcal{C}}$ from the adversary.
5. Run $(z_1, \ldots, z_n) \leftarrow \text{Share}_t(x \cdot y, \{z_i\}_{i\in\mathcal{C}})$.
6. For every $j \in \mathcal{H}$, send z_j to P_j.

The protocol to instantiate the functionality $\mathcal{F}_{\text{Mult}}$ is described below. It makes use of the functionality $\mathcal{F}_{\text{PublicRec}}$ from Section 3.1.5.

Protocol Π_{Mult}

Input: Secret-shared values $[\![x]\!]_t$ and $[\![y]\!]_t$.
Output: $[\![z = x \cdot y]\!]_t$.
Functionalities: $\mathcal{F}_{\text{D.Shar}}$ and $\mathcal{F}_{\text{PublicRec}}(2t)$.
Protocol: The parties execute the following

1. The parties $\mathcal{F}_{\text{D.Shar}}$ to get $([\![r]\!]_t, [\![r]\!]_{2t})$;
2. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
3. The parties call the functionality $\mathcal{F}_{\text{PublicRec}}(2t)$ to either learn a or abort.
4. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

[4] For simplicity, we also count messages from a party "to itself".

Theorem 3.12. *Protocol Π_{Mult} instantiates functionality $\mathcal{F}_{\text{Mult}}$ with perfect security in the $(\mathcal{F}_{\text{D.Shar}}, \mathcal{F}_{\text{PublicRec}}(2t))$-hybrid model against an active adversary corrupting $t < n/3$ parties.*

Proof. We define the simulator $\mathcal{S}$ as follows.

Before interacting with the adversary the simulator receives $\{(x_i, y_i)\}_{i\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$. Then, the simulation of the protocol proceeds as follows:

1. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{D.Shar}}$ by receiving values $\{(\overline{r}_i, \overline{r}'_i)\}_{i\in\mathcal{C}}$ from the adversary.
2. N/A (local computation)
3. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{PublicRec}}(2t)$:
   - $\mathcal{S}$ sets $\overline{a}_i = x_i \cdot y_i - \overline{r}'_i$ for $i \in \mathcal{C}$, samples $\overline{a} \in_R \mathcal{R}$ and calls $(\overline{a}_1, \dots, \overline{a}_n) \leftarrow \text{Share}_{2t}(\overline{a}, \{\overline{a}_i\}_{i\in\mathcal{C}})$. Then $\mathcal{S}$ sends $(\overline{a}_1, \dots, \overline{a}_n)$ to the adversary as the emulation of $\mathcal{F}_{\text{PublicRec}}(2t)$.
   - If the adversary sets the parties to abort in the emulated $\mathcal{F}_{\text{PublicRec}}(2t)$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{Mult}}$.
4. For $i \in \mathcal{C}$ let $\overline{z}_i = \overline{a}_i + \overline{r}_i$. $\mathcal{S}$ sends $\{\overline{z}_i\}_{i\in\mathcal{C}}$ to the functionality $\mathcal{F}_{\text{Mult}}$.

We argue indistinguishability.

Real world	**Ideal world**
1. The adversary sends $\{(r_i, r'_i)\}_{i\in\mathcal{C}}$ to $\mathcal{F}_{\text{D.Shar}}$.	1. The adversary sends $\{(\overline{r}_i, \overline{r}'_i)\}_{i\in\mathcal{C}}$ to (the emulation of) $\mathcal{F}_{\text{D.Shar}}$. These follow the same distribution since it is the first interaction of the adversary in the protocol execution.
2. N/A (local computation)	2. N/A (local computation)
3. The adversary either sets the parties to abort in the call to $\mathcal{F}_{\text{PublicRec}}(2t)$, or all the parties learn $a = x \cdot y - r$, which is uniformly random as r is.	3. The adversary either sets the parties to abort (with the same probability as in the real world) in the call to the emulated $\mathcal{F}_{\text{PublicRec}}(2t)$, or all the parties learn the uniformly random value $\overline{a}$.
4. The honest parties output $\{z_i = a + r_i\}_{i\in\mathcal{H}}$.	4. The (real) honest parties output $\{\overline{z}_i\}_{i\in\mathcal{H}}$ such that $\{\overline{z}_i\}_{i\in\mathcal{H}} \cup \{\overline{z}_i\}_{i\in\mathcal{C}}$ is t-consistent and the underlying secret is $x \cdot y$.

To finish the proof of indistinguishability, we only need to show that the output in the two worlds is equally distributed. This is done in the following claim.

Claim 3.10. *If no abort is produced, the output $\{z_i = a + r_i\}_{i\in\mathcal{H}}$ of the honest parties in the real world is uniformly random constrained to:*

- *$\{z_i\}_{i\in\mathcal{H}} \cup \{z_i = a + r_i\}_{i\in\mathcal{C}}$ is t-consistent*
- *The underlying secret is $x \cdot y$.*

Proof. $\{z_i\}_{i=1}^n$ is trivially t-consistent since $\{r_i\}_{i=1}^n$ is, by the properties of the $\mathcal{F}_{\text{D.Shar}}$ functionality. Furthermore, from the properties of $\mathcal{F}_{\text{PublicRec}}(2t)$, if no abort is produced then the reconstructed value a corresponds to $xy - r$, so the underlying secret of $\{z_i\}_{i=1}^n$ is $r + a = r + (xy - r) = xy$. □

□

Communication complexity of Π_{Mult}. To conclude, we study the communication complexity of the multiplication protocol. This amounts to the communication complexity of generating one double-share with $\mathcal{F}_{\text{D.Shar}}$, and opening one secret-shared value with $\mathcal{F}_{\text{PublicRec}}(2t)$. Using the instantiations of these functionalities from Sections 3.3.1 and 3.1.5, respectively, we obtain a total communication complexity of $10n + 6n = 16n$ for the case in which $n = 3t + 1$.

3.3.3 Shares of Random Values

Another important functionality that we will need for our final protocol is $\mathcal{F}_{\text{Rand}}(\mathcal{A})$, which, for a Galois ring $\mathcal{A} = \text{GR}(2^k, \rho) \subseteq \mathcal{R}$, distributes random t-consistent shares to the parties with the constraint that the underlying secret lies in $\mathcal{A}$. This is formalized below.

Functionality $\mathcal{F}_{\text{Rand}}(\mathcal{A})$

- Sample $r \in_R \mathcal{A}$.
- Receive $\{r_j\}_{j\in\mathcal{C}}$ from the adversary.
- Run $(r_1, \dots, r_n) \leftarrow \text{Share}_t(r, \{r_j\}_{j\in\mathcal{C}})$.
- For every $j \in \mathcal{H}$, send r_j to P_j.

Notice that the functionality is similar to $\mathcal{F}_{\text{D.Shar}}$, except it does not consider shares of degree $2t$, and it produces only one shared value in one call, contrary to $\mathcal{F}_{\text{D.Shar}}$, that produces batches of $n - 2t$ shared values. This is just for notational convenience.

Recall that $\mathcal{A} = \text{GR}(2^k, \rho)$, with $\rho|\tau$, and χ is such that $\rho \cdot \chi = \tau$. For simplicity, when $\mathcal{A} = \mathcal{R}$ (i.e. $\rho = \tau$), we denote $\mathcal{F}_{\text{Rand}}(\mathcal{A})$ by simply $\mathcal{F}_{\text{Rand}}$. The protocol to instantiate this functionality is presented below.

Protocol $\Pi_{\text{Rand}}(\mathcal{A})$

Output: A set of sharings $\{[\![r_{ij}]\!]_t : i \in \{2t+1, \dots, n\},\ j \in [\chi]\}$ where $r_{ij} \in_R \mathcal{A}$.
Protocol: The parties proceed as follows

1. Each party P_i samples $s_{i1}, \dots, s_{i\chi} \in_R \mathcal{A}$ and secret-shares it over $\mathcal{R}$ using a degree-t polynomial. The parties obtain $[\![s_{ij}]\!]_t^{\text{GR}(2^k,\tau)}$ for $j \in [\chi]$.
2. Following Section 3.2.5, the parties locally compute $[\![s_i]\!]^{\text{GR}(2^k,\tau\chi)}$ ←

$\phi([\![s_{i1}]\!]_t^{\mathsf{GR}(2^k,\tau)}, \ldots [\![s_{i\chi}]\!]_t^{\mathsf{GR}(2^k,\tau)})$ over $\mathsf{GR}(2^k, \tau\chi)$, where $s_i = \phi(s_{i1}, \ldots, s_{i\chi})$, with $\phi : \mathsf{GR}(2^k, \tau)^\chi \to \mathsf{GR}(2^k, \tau\chi)$ the mapping defined in that section.

3. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \\ [\![r_2]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \\ \vdots \\ [\![r_n]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} [\![s_1]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \\ [\![s_2]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \\ \vdots \\ [\![s_n]\!]_t^{\mathsf{GR}(2^k,\tau\chi)} \end{pmatrix}.$$

4. For each $i \in [2t]$, all the parties send their shares of $[\![s_i]\!]_t^{\mathsf{GR}(2^k,\tau\chi)}$ to P_i.
5. Upon receiving these shares, each P_i for $i \in [2t]$ checks that the received sharings of $[\![r_i]\!]_t$ are t-consistent and that the underlying secrets lie in $\phi(\mathcal{A}^\chi)$. If this does not hold then the parties abort.
6. If no abort was produced, then the parties output the sharings $([\![r_{i1}]\!]_t^{\mathsf{GR}(2^k,\tau)}, \ldots, [\![r_{i\chi}]\!]^{\mathsf{GR}(2^k,\tau)}) \leftarrow \phi^{-1}([\![r_i]\!]^{\mathsf{GR}(2^k,\tau\chi)})$ for $i \in \{2t+1, \ldots, n\}$.

Theorem 3.13. *Protocol $\Pi_{\mathsf{Rand}}(\mathcal{A})$ instantiates functionality $\mathcal{F}_{\mathsf{Mult}}(\mathcal{A})$ with perfect security against an active adversary corrupting $t < n/3$ parties.*

The full simulation-based proof of this theorem is very close to the one from Theorem 3.11, and therefore it is omitted. The main difference lies in the fact that the check performed is different, namely, the parties check that the shares of $[\![r_i]\!]_t$ (over $\mathsf{GR}(2^k, \tau\xi)$) are t-consistent and that the underlying secrets lie in $\phi(\mathcal{A}^\chi)$.

To make the proof from Theorem 3.11 work in this setting, it must be the case that the property being checked is "$\mathcal{R}$-linear" so that it is preserved after multiplying by the matrix $\mathbf{M}$. In this case, the property is that the secret of given sharings over $\mathsf{GR}(2^k, \tau\chi)$ lies not in this ring but in the subring $\phi(\mathcal{A}^\chi)$. It is easy to see that this property is $\mathcal{R}$-linear since ϕ is an $\mathcal{R}$-module homomorphism and $\mathcal{A}^\chi$, isomorphic to $\mathcal{R}$, is an $\mathcal{R}$-module.

Communication complexity of $\Pi_{\mathsf{Rand}}(\mathcal{A})$. Following an analysis similar to the one from Section 3.3.1, the communication complexity of protocol $\Pi_{\mathsf{Rand}}(\mathcal{A})$ is χn^2 elements in $\mathcal{R}$ plus $2tn$ elements in $\mathsf{GR}(2^k, \tau\chi)$, which correspond to $2tn\chi$ elements in $\mathcal{R}$. Since $\chi(n-2t)$ shared values are produced, the amortized cost per shared value is $\frac{\chi(n^2+2nt)}{\chi(n-2t)} = \frac{n^2+2nt}{n-2t}$. For the case in which $n = 3t+1$, this becomes $\approx 5n$.

Remark 3.2. *If $\mathcal{A} = \mathcal{R}$, then a much simpler protocol to instantiate $\mathcal{F}_{\mathsf{Rand}}(\mathcal{A})$ can be devised. In this case, the parties can simply run the protocol for generating double-sharings, modified to remove the degree-$2t$ parts. The communication complexity would be exactly the same as the one from the analysis above, except that the protocol would produce $n - 2t$ shared values instead of $\chi(n-2t)$ for some $\chi > 1$.*

3.3.4 Secret-Sharing Inputs

As a building block, we also require a functionality that receives an input from a given party and distributes shares of this input to the other parties. Furthermore, as mentioned at the beginning of Section 3.3, our ultimate goal is to obtain a secure computation protocol over $\mathcal{A} = \mathsf{GR}(2^k, \rho)$ by first designing a secure computation protocol over $\mathcal{R} = \mathsf{GR}(2^k, \tau)$, followed by restricting this protocol so that the inputs lie in $\mathcal{A}$. Given this, our functionality for distributing shares of inputs must also guarantee that the shared inputs, which in principle could lie in $\mathcal{R}$, actually belong to $\mathcal{A}$. This is formalized by the functionality below.

Functionality $\mathcal{F}_{\mathsf{Input}}(\mathcal{A})$

Let $s \in [n]$ be the index of the party providing input.

- Receive (input, x) from P_s.
  - If $x \in \mathcal{A}$ then store (P_s, x) in memory.
  - Else send abort to the honest parties.
- On input $\{\overline{x}_i\}_{i\in\mathcal{C}}$ from the adversary retrieve (P_s, x) from memory and do the following:
  1. Run $(x_1, \dots, x_n) \leftarrow \mathsf{Share}_t(x, \{\overline{x}_i\}_{i\in\mathcal{C}})$.
  2. Send x_j to each party $P_j \in \mathcal{H}$.

For simplicity, when $\mathcal{A} = \mathcal{R}$, we denote $\mathcal{F}_{\mathsf{Input}}(\mathcal{A})$ by simply $\mathcal{F}_{\mathsf{Input}}$. The functionality $\mathcal{F}_{\mathsf{Input}}(\mathcal{A})$ can be instantiated by the following protocol.

Protocol $\Pi_{\mathsf{Input}}(\mathcal{A})$

Input: Party P_s has input $x \in \mathcal{A}$, where $\mathcal{A}$ is a subring of $\mathcal{R}$.
Output: The parties get t-consistent shares $[\![x]\!]_t$.
Functionalities: $\mathcal{F}_{\mathsf{Rand}}(\mathcal{A})$.
Protocol: The parties proceed as follows:

1. The parties call $\mathcal{F}_{\mathsf{Rand}}(\mathcal{A})$ to get $[\![r]\!]_t = (r_1, \dots, r_n)$, with $r \in \mathcal{A}$.
2. The parties send their shares of $[\![r]\!]_t$ to P_s.
3. P_s, upon receiving shares $(r_1, \dots, r_n)$, executes $r \leftarrow \mathsf{RecSecret}_t(r_1, \dots, r_n)$
4. P_s broadcasts $a = x - r$ to all the parties.
5. Upon receiving a from the broadcast channel, the parties do the following:
   - If $a \notin \mathcal{A}$, then the parties abort.
   - Else, they compute $[\![x]\!]_t = a + [\![r]\!]_t$.

Theorem 3.14. *Protocol $\Pi_{\mathsf{Input}}(\mathcal{A})$ instantiates the functionality $\mathcal{F}_{\mathsf{Input}}(\mathcal{A})$ in the $\mathcal{F}_{\mathsf{Rand}}(\mathcal{A})$-hybrid model with perfect security against an active adversary corrupting $t < n/3$ parties.*

Proof. Let P_s the sender. We define a simulator $\mathcal{S}$ as follows. $\mathcal{S}$ emulates the honest parties and also the functionality $\mathcal{F}_{\mathsf{Rand}}(\mathcal{A})$.

We divide the description of the simulator in the cases that P_s is honest and corrupt. If P_s is corrupt, $\mathcal{S}$ emulates the steps in the protocol as below.

1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Rand}}(\mathcal{A})$ by sampling $\overline{r} \in_R \mathcal{A}$, computing $(\overline{r}_1, \ldots, \overline{r}_n) \leftarrow \text{Share}_t(\overline{r})$, and sending $\overline{r}_j$ to each corrupt party P_j (including P_s).
2. The virtual honest parties $\overline{P}_j$ send their shares $\overline{r}_j$ of $\overline{r}$ to P_s.
3. N/A (only P_s acts locally in this step)
4. $\mathcal{S}$, who emulates the broadcast channel, receives $\overline{a}$ from P_s. If $\overline{a} \notin \mathcal{A}$, then $\mathcal{S}$ instructs $\mathcal{F}_{\text{Input}}(\mathcal{A})$ to make the parties abort. Else, the simulator sets $\overline{x} := \overline{a} + \overline{r}$ and calls $\mathcal{F}_{\text{Input}}(\mathcal{A})$ on behalf of P_s on input $(\text{input}, \overline{x})$.
5. $\mathcal{S}$ calls $\mathcal{F}_{\text{Input}}(\mathcal{A})$ on input $\{\overline{r}_i + \overline{a}\}_{i \in \mathcal{C}}$.

To argue indistinguishability we proceed as follows.

Real world	**Ideal world**
1. Call to $\mathcal{F}_{\text{Rand}}(\mathcal{A})$.	1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Rand}}(\mathcal{A})$ as in the real world.
2. From Thm. 3.2, the adversary's shares $\{r_j\}_{j \in \mathcal{C}}$ look uniformly random.	2. From Thm. 3.2, the adversary's shares $\{\overline{r}_j\}_{j \in \mathcal{C}}$ look uniformly random.
3. P_s error corrects a uniformly random r	3. P_s error corrects a uniformly random $\overline{r}$
4. P_s sends some a to the broadcast channel	4. P_s sends some $\overline{a}$ to the broadcast channel. At this point $\mathcal{F}_{\text{Input}}$ has set $\overline{x} = \overline{a} + \overline{r}$ as the input from P_s.
5. The honest parties output the shares $\{r_j + a\}_{j \in \mathcal{H}}$, which are t-consistent with the secret $x = a + r$ and with the shares $\{r_i + a\}_{i \in \mathcal{C}}$ held by the corrupt parties.	5. The honest parties output shares that are t-consistent with the secret $\overline{x}$ and with the shares $\{\overline{r}_i + a\}_{i \in \mathcal{C}}$ held by the corrupt parties.

We see that the two worlds follow the exact same distribution through all the steps.

The case in which P_s is honest is handled in a similar but simpler way, and we leave it out of the proof. □

Communication complexity of $\Pi_{\text{Input}}(\mathcal{A})$. The total communication complexity of protocol $\Pi_{\text{Input}}(\mathcal{A})$ amounts to one call to $\mathcal{F}_{\text{Rand}}(\mathcal{A})$, plus the parties sending shares to P_s, so n ring elements, and then P_s sending one ring element in the broadcast channel. Using the instantiation of $\mathcal{F}_{\text{Rand}}(\mathcal{A})$ from Section 3.3.3, this leads to a total of $\approx 5n + n + \text{BC}_{\mathcal{R}}(1) = 6n + \text{BC}_{\mathcal{R}}(1)$ for the case in which $n = 3t + 1$.[5] It is very important to notice the following, however. Since the input phase is only called once, at the beginning, during the execution of a secure computation protocol, its communication complexity, as the circuit being computed grows, becomes less relevant.

[5] $\text{BC}_{\mathcal{R}}(\ell)$ is defined as the communication complexity of broadcasting ℓ elements in $\mathcal{R}$.

3.3.5 Final MPC Protocol

Now we put together the different tools explored in previous sections to obtain a protocol that instantiates $\mathcal{F}_{\text{MPC}}(\mathcal{A})$ with perfect security.

Protocol $\Pi_{\text{MPC}}(\mathcal{A})$

Input: The parties have inputs in $\mathcal{A}$ for F.
Output: The parties learn the evaluation of F on these inputs.
Functionalities: $\mathcal{F}_{\text{Input}}(\mathcal{A})$, $\mathcal{F}_{\text{Mult}}$, $\mathcal{F}_{\text{PublicRec}}(t)$
Protocol: The parties proceed as follows

1. For each $i \in [n]$ and each input $x \in \mathcal{A}$ held by P_i, the parties call $\mathcal{F}_{\text{Input}}(\mathcal{A})$ to get $[\![x]\!]_t$.
2. - For every addition operation with inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties locally compute $[\![x+y]\!]_t \leftarrow [\![x]\!]_t + [\![y]\!]_t$.
   - For every multiplication operation with inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties call $\mathcal{F}_{\text{Mult}}$ to get $[\![xy]\!]_t$.
3. For each secret-shared output value $[\![z]\!]_t$, the parties call $\mathcal{F}_{\text{PublicRec}}(t)$ to learn z.

The following theorem is a direct consequence of the previous results.

Theorem 3.15. *Protocol $\Pi_{\text{MPC}}(\mathcal{A})$ instantiates the functionality $\mathcal{F}_{\text{MPC}}(\mathcal{A})$ in the $(\mathcal{F}_{\text{Input}}(\mathcal{A}), \mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{PublicRec}}(t))$-hybrid model with perfect security against an active adversary corrupting $t < n/3$ parties.*

By taking $\mathcal{A} = \mathbb{Z}/2^k\mathbb{Z}$, we obtain the following.

Corollary 3.3. *Protocol $\Pi_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$ instantiates the functionality $\mathcal{F}_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$ in the $(\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z}), \mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{PublicRec}}(t))$-hybrid model with perfect security against an active adversary corrupting $t < n/3$ parties.*

Communication complexity of $\Pi_{\text{MPC}}(\mathcal{A})$. Let I_F, O_F and M_F be the number of input, output and multiplication gates of the arithmetic circuit F, respectively. Putting together the complexity analysis from each subsection above, we obtain for the case in which $n = 3t+1$ that the total communication complexity required to securely compute F is, in terms of elements of $\mathcal{R}$,

$$I_F \cdot (6n + \text{BC}_{\mathcal{R}}(1)) + M_F \cdot (16n) + O_F \cdot (6n).$$

The following observations are important.

- Recall that $\tau \approx \log(2n)$, so, in terms of *bits*, the communication complexity from above, which is measured in elements of $\mathcal{R} = \text{GR}(2^k, \tau)$, gets multiplied by $k\tau = k \cdot \log(2n)$. In particular, it is *not* linear in n.

- Observe that this communication complexity is independent of ρ, the degree of the Galois ring $\mathcal{A} = \mathsf{GR}(2^k, \rho)$. In particular, even if ρ is very small, like $\rho = 1$ for the case in which $\mathcal{A} = \mathbb{Z}/2^k\mathbb{Z}$, there are no communication savings with respect to the case in which $\rho = \tau$.

- Some of the protocols presented in this section, specifically $\Pi_{\mathsf{PublicRec}}(d)$, $\Pi_{\mathsf{D.Shar}}$ and $\Pi_{\mathsf{Rand}}(\mathcal{A})$, operate in *batches*, meaning that they produce not only one but $\Omega(t)$ outputs in one call. This is acceptable for $\Pi_{\mathsf{D.Shar}}$ and $\Pi_{\mathsf{Rand}}(\mathcal{A})$ since these protocols are called in the preprocessing phase and, unless the circuit has too few input and multiplication gates, these calls can be batched as required. However, most of the calls to $\Pi_{\mathsf{PublicRec}}(d)$ happen in the online phase and can only be parallelized for multiplication gates of the same depth. If the circuit is too "narrow", that is, if there are layers with too few multiplication gates, then $\Pi_{\mathsf{PublicRec}}(d)$ presents no savings in terms of amortized communication complexity.

3.4 Guaranteed Output Delivery

In this thesis we have decided to focus on the notion of security with abort to keep the exposition of the protocols simple. This is also motivated on the fact that most of the protocols presented in this work can be enhanced to achieve the stronger notion of guaranteed output delivery by making use of standard techniques in the literature that are essentially agnostic to the underlying algebraic structure over which the arithmetic circuit is defined.

This is precisely the approach taken in the original work of [2], where Shamir secret-sharing, plus different tools for secure multiparty computation over $\mathbb{Z}/2^k\mathbb{Z}$, are introduced, together with indications about how to obtain from these techniques a protocol with guaranteed output delivery. For this part they follow the general template of [16], where a perfectly secure protocol over *fields* tolerating $t < n/3$ active corruptions and satisfying guaranteed output delivery is designed.

In this section we sketch how to use the techniques from [16] to endow our protocol from Section 3.3 with the stronger property of guaranteed output delivery. The main technique is a tool called *player elimination*, which aims at identifying a pair of parties $\{P_i, P_j\}$ where at least one of them is guaranteed to be corrupt. Such pair of parties is called a *semi-corrupt pair*. After this is done, the values $n' = n - 2$ and $t' = t - 1$ are computed, and the parties, without the identified pair, run again an instance of the protocol with n' parties and threshold t'.[6] Notice that it still holds that $t' < n'/3$, which enables the use of this protocol.

[6] To avoid running the protocol from scratch, potentially wasting a lot of computation, the execution is done in *segments* which are checked regularly and, if a semi-corrupt pair is identified, only the given segment is restarted. We will not consider this in this section.

3.4.1 Different Locations Where the Protocol can Abort

Given the general approach to guaranteed output delivery outlined above, our focus in this section is to enhance the protocol from Section 3.3 so that a semi-corrupt pair is identified at the places in the protocol execution where an abort can happen. To this end, we first identify precisely these locations where abort can take place.

- Whenever any of the reconstruction protocols from Definition 3.6 in Section 3.1.3 is called with a degree d such that $(n-d)/2 \leq t < n-d$. This is the case for $d = 2t$, which is used in Π_{Mult} and $\Pi_{\text{D.Shar}}$.

- In the check that the secrets belong to the subring $\phi(\mathcal{A}^\chi)$ performed in step 5 in $\Pi_{\text{Rand}}(\mathcal{A})$.

- In the check that the two shared secrets coincide in $\Pi_{\text{D.Shar}}$.

- In $\Pi_{\text{Input}}(\mathcal{A})$ if the broadcasted value a is not in $\mathcal{A}$.

3.4.2 General Strategy to Identify Semi-Corrupt Pairs

In order to identify a semi-corrupt pair in any of the situations above, the general strategy is the following. Suppose that the parties are instructed to abort in the protocol (by any of the reasons stated in the previous subsection). The parties execute the following:

1. For $i, j \in [n]$, let M_{ij} be the set of messages that party P_i has sent to party P_j as part of the execution of the protocol step that led to an abort. Each party P_i broadcasts $\{M_{ij}\}_{j\in[n]}$

2. Each party P_j broadcasts (complain, P_i) if the broadcasted message M'_{ij} from P_i does not coincide the set of messages M_{ij} that P_j received from this party. The parties output the semi-corrupt pair $\{P_{j_0}, P_{i_0}\}$, where $j_0 \in [n]$ is the smallest index of the party that sent a message of the form (complain, P_i), and i_0 is the smallest index appearing in such messages.

3. If no message of the form (complain, P_i) is ever broadcasted, then the parties scan the messages $\{M_{ij}\}_{i,j\in[n]}$ looking for a message $M_{i_0 j_0}$ that deviates from the protocol specification, and output $\{P_{i_0}\}$ as a corrupt party.

Since no honest party P_i would broadcast an incorrect set $\{M_{ij}\}_{j\in[n]}$, and no honest party P_j would incorrectly complain about another party broadcasting a wrong set, the only way in which a party P_j would broadcast (complain, P_i) is if either P_i or P_j is corrupt, so the set $\{P_{i_0}, P_{j_0}\}$ produced in step 2 above is indeed a semi-corrupt pair. On the other hand, if no complaint message was broadcasted by any party, then the broadcasted messages $\{M_{ij}\}_{i\in\mathcal{C}, j\in\mathcal{H}}$ correctly reflect the messages sent by corrupt parties to honest parties. If these messages all follow the protocol execution, then no abort signal would have been generated in a first place. As a result, there has to be at least one M_{ij} that does not

follow the protocol execution, corresponding to a corrupt party P_i. Hence, the party P_{i_0} produced in step 3 above is guaranteed to be corrupt.

3.4.3 Removing the Possibility of Abort from the Online Phase

Unfortunately, although the generic approach above does help in identifying a semi-corrupt pair, it requires all parties to reveal all the messages sent so far in the protocol execution and therefore exposes the sensitive information such as the private inputs from the honest parties or any intermediate computation values derived from these.

Recall that the protocols $\Pi_{\text{D.Shar}}$ and $\Pi_{\text{Rand}}(\mathcal{A})$ are called in a *preprocessing* phase before all inputs are distributed. Let us begin by identifying which of the abort locations discussed in Section 3.4.1 contain sensitive information.

- Calling $\Pi_{\text{PublicRec}}(2t)$:
  - In $\Pi_{\text{D.Shar}}$: this is not a problem since this happens in the preprocessing phase, so all messages sent so far are independent of the inputs.
  - In Π_{Mult}: this is a problem since this happens in the online phase where the inputs have been already distributed.
- In the check in $\Pi_{\text{Rand}}(\mathcal{A})$: this is part of the preprocessing.
- In the check in $\Pi_{\text{D.Shar}}$: this is also part of the preprocessing.
- In Π_{Input}: an abort can only be caused if the party sharing the input is corrupt, so the parties do not even need to execute the method from Section 3.4.2 to identify a semi-corrupt pair, they can simply flag the sending party as corrupt.

Given the above, we see that the only problematic cause of abort is in the execution of Π_{Mult} when opening shares of degree $2t$, which is likely to involve sensitive information. Therefore, our goal now is to transform the multiplication protocol so that no abort can occur while executing it. This is achieved as follows.

Preprocessing phase. The parties compute the following preprocessing material:

1. The parties call $\Pi_{\text{Rand}}(\mathcal{R})$ to get $[\![a]\!]_t$ and $[\![b]\!]_t$.
2. The parties call Π_{Mult} to obtain $[\![c]\!]_t$ from $[\![a]\!]_t$ and $[\![b]\!]_t$, where $c = ab$.

Online phase. To obtain $[\![xy]\!]_t$ from $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties proceed as follows

1. Let $[\![d]\!]_t \leftarrow [\![x]\!]_t - [\![a]\!]_t$ and $[\![e]\!]_t \leftarrow [\![y]\!]_t - [\![b]\!]_t$. Call $\Pi_{\text{PublicRec}}(t)$ to reconstruct d and e.
2. Compute locally $[\![xy]\!]_t = d\,[\![b]\!]_t + e\,[\![a]\!]_t + [\![c]\!]_t + de$.

Correctness of the protocol can be checked by inspection, and privacy holds from the fact that the only opened values are d and e, which are uniformly random since a and b are. Furthermore, what is more important for the main goal of this section is that the online phase of this protocol only involves opening sharings of degree t instead of degree $2t$, which falls into the error correction regime and therefore can be done without possibility of abort.

Chapter 4

Honest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$

The goal of this chapter is to design a statistically secure multiparty computation protocol with abort for arithmetic circuits over $\mathbb{Z}/2^k\mathbb{Z}$ that tolerates $t < n/2$ active corruptions. Fortunately, with the tools described in Chapter 3, this task becomes relatively simple. In a nutshell, our approach consists of following a similar template as the protocol from Chapter 3: use Shamir secret-sharing to distribute shares of the intermediate values, use its additively homomorphic properties to process addition gates, and preprocess double-shares to handle multiplication gates securely.

In this chapter we assume that $n = 2t+1$. As in Section 2.5, this is *not* for simplicity, since the existence of more than $t+1$ honest parties can lead to inconsistency of distributed values. This can be handled by the parties executing a consistency check for values distributed by the parties, which is described in [2]. We avoid this complication in this thesis by simply assuming that $n = 2t+1$.

Throughout the rest of this section, let $\mathcal{R} = \mathrm{GR}(2^k, \tau)$ and $\mathcal{A} \in \{\mathcal{R}, \mathbb{Z}/2^k\mathbb{Z}\}$.

4.1 Preliminaries

In this chapter we make use of the concepts and results on Shamir secret-sharing and Galois rings from Section 3.1 and 3.2.

Let $\lambda_1, \ldots, \lambda_n \in \mathcal{R}$ be the scalars such that $\mathrm{RecSecret}_{2t}(\{s_i\}_{i\in[n]}) = \lambda_1 \cdot s_1 + \cdots + \lambda_n \cdot s_n$.

We also introduce additional results below.

Proposition 4.1. *Consider a Galois ring $GR(p^k, u)$. Let $f(X) \in GR(p^k, u)[X]$ be a non-zero polynomial of degree at most d. Then, the probability that $f(\alpha) = 0$ for $\alpha \in_R GR(p^k, u)$ is at most d/p^u.*

Proof. Let p^ℓ with $\ell < k$ be the largest power of p simultaneously dividing all the coefficients of $f(X)$. We can divide by p^ℓ to obtain the polynomial $p^{-\ell}f(X) \in \mathrm{GR}(p^{k-\ell}, u)$. Consider the non-zero polynomial $g(X) = (p^{-\ell}f(X)) \bmod p$ in $\mathrm{GF}(p^u)$, which has at most d roots. If $f(\alpha) = 0$ then $g(\alpha \bmod p) = 0$, so in particular $\alpha \bmod p$ has to be one of the at

most d roots of $g(X)$. Since $\alpha \bmod p$ is uniformly random over $\mathrm{GF}(p^u)$ for $\alpha \in_R \mathrm{GR}(p^k, u)$, we get that the probability of $\alpha \bmod p$ being one of these at most d roots is upper bounded by d/p^u. □

Proposition 4.2. *Consider a Galois ring $GR(p^k, u)$ and a Galois ring extension of it $GR(p^k, uv)$. Let $f(X) \in (GR(p^k, uv)[X]) \setminus (GR(p^k, u)[X])$ be a polynomial of degree at most d. Then, the probability that $f(\alpha) \in GR(p^k, u)$ for $\alpha \in_R GR(p^k, u)$ is at most d/p^u.*

Proof. Using the polynomial representation of $\mathrm{GR}(p^k, uv)$ over $\mathrm{GR}(p^k, u)$, write $f(X) = \sum_{i=1}^{v} \xi^{i-1} f_i(X)$, where $f_i(X) \in \mathrm{GR}(p^k, u)[X]$. Since $f(X) \notin \mathrm{GR}(p^k, u)[X]$, we have that $f_{i_0}(X) \neq 0$ for some $i_0 \in [v] \setminus \{1\}$. Now, given that $f(\alpha) \in \mathrm{GR}(p^k, u)$ if and only if $f_i(\alpha) = 0$ for $i \in [v] \setminus \{1\}$, we would have that $f_{i_0}(\alpha) = 0$. From Proposition 4.1 above, this happens with probability at most d/p^u. □

4.1.1 Public Reconstruction of Secret-Shared Values

Similar to the protocol from Chapter 3, a crucial tool that we will need throughout our protocol from this chapter is the ability for the parties to reconstruct a secret-shared value $[\![s]\!]_d$ efficiently. To this end, we introduced in Section 3.1.5 a family of functionalities $\mathcal{F}_{\mathrm{PublicRec}}(d)$, parameterized by the degree d, that modeled the task of the parties reconstructing the secret-shared value $[\![s]\!]_d$. The description of the functionality depended on the value of d: if $t < n - d$ (error detection), then the functionality allowed the adversary to make the parties abort, but the correct secret s was reconstructed if an abort did not happen; if $t < (n - d)/2$ (error correction), then the functionality would always reconstruct the correct secret and would not allow the adversary to cause an abort. If $t \geq n - d$, however, then the functionality $\mathcal{F}_{\mathrm{PublicRec}}(d)$ is not defined.

In our setting the values that d will take are t and $2t$. Recall that $n = 2t + 1$, so, for $d = t$, the bound $t < n - d$ holds but not $t < (n - d)/2$, so $\mathcal{F}_{\mathrm{PublicRec}}(t)$ is well defined and in fact it can be instantiated with the protocol $\Pi_{\mathrm{PublicRec}}(t)$ from Section 3.1.5. This is summarized in the following theorem, whose proof is a small variant of the one from Theorem 3.4.

Theorem 4.1. *Protocol $\Pi_{\mathrm{PublicRec}}(t)$ instantiates functionality $\mathcal{F}_{\mathrm{PublicRec}}(t)$ with perfect security against an active adversary corrupting $t < n/2$ parties.*

Communication complexity of $\Pi_{\mathrm{PublicRec}}(t)$. An analysis similar to the one from Section 3.1.5 shows that the communication complexity of reconstructing *one* shared value is $\frac{2n^2}{t+1}$ elements in $\mathcal{R}$, which, for $n = 2t + 1$, leads to $\frac{4n^2}{n+1} \approx 4n$.

4.2 Secure Multiplication with Additive Errors

Unlike the protocol from Section 3.3.2, we will not be able to produce $[\![xy]\!]_t$ from $[\![x]\!]_t$ and $[\![y]\!]_t$. Instead, the parties will be able to obtain $[\![xy+\delta]\!]_t$, where $\delta \in \mathcal{R}$ is some value chosen by the adversary. In this section we show how to instantiate such functionality, which is done in two steps. First, in Section 4.2.1 we show how the parties can generate double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$, which is done in a simpler way than in Section 3.3.1 given that there is less redundancy in these sharings in the $t < n/2$ case. Then, in Section 4.2.2 it is shown how to use these pairs to obtain a multiplication with additive errors $[\![xy+\delta]\!]_t$.

As shown at the beginning of Section 2.5.4, the additive error attack is not only a problem for correctness: for certain circuits this type of attacks enables the adversary to learn sensitive information. Hence, to prevent an adversary from carrying out these attacks, a verification step is carried out by the parties in which they check that all the multiplications in the circuit computation are carried out correctly. This is done in Sections 4.4 and 4.5.

4.2.1 Double Sharings

We begin by presenting the functionality to generate double-sharings. This functionality has a crucial core difference with respect to its $t < n/3$ counterpart from Section 3.3.1: for our case in which $n = 2t+1$, the $n - t = t + 1$ honest shares are not enough to determine the secret underlying the shares $[\![r]\!]_{2t}$, or, in other words, the adversary can always choose to change the shares from the corrupt parties, hence changing the value of r.

This is captured in the functionality $\mathcal{F}_{\text{D.Shar}}$ as follows. For the degree-t sharings $[\![r]\!]_t$, the adversary is allowed to choose the shares of the corrupt parties $\{r_i\}_{i\in\mathcal{C}}$, and then the functionality will sample the remaining $t+1$ honest shares $\{r_j\}_{j\in\mathcal{H}}$ in such a way that $\{r_i\}_{i\in[n]}$ is t-consistent with the random secret r. This, so far, is no different than the functionality from Section 3.3.1. Now, for the degree-$2t$ sharings $[\![r]\!]_{2t}$, the functionality could allow the adversary to also choose the shares of the corrupt parties $\{r'_i\}_{i\in\mathcal{C}}$, and then the functionality would sample $\{r'_j\}_{j\in\mathcal{H}}$ so that $\{r'_i\}_{i\in[n]}$ is $2t$-consistent with r, which ultimately amounts to $\sum_{i=1}^n \lambda_i r'_i = r$. However, since the distribution of $\{r'_j\}_{j\in\mathcal{H}}$ would be the exact same for any other shares $\{r''_i\}_{i\in\mathcal{C}}$ satisfying $\sum_{i\in\mathcal{C}} \lambda_i r''_i = \sum_{i\in\mathcal{C}} \lambda_i r'_i = r$, the functionality does not receive $\{r'_i\}_{i\in\mathcal{C}}$ from the adversary, but instead, it receives the values $\Delta = \sum_{i\in\mathcal{C}} \lambda_i r'_i$, which is ultimately what dictates the distribution of the honest parties' shares $\{r'_j\}_{j\in\mathcal{H}}$.

Functionality $\mathcal{F}_{\text{D.Shar}}$

- Sample $r \in_R \mathcal{R}$.
- Receive $(\{r_j\}_{j\in\mathcal{C}}, \Delta)$ from the adversary.
- Run $(r_1, \dots, r_n) \leftarrow \text{Share}_t(r, \{r_j\}_{j\in\mathcal{C}})$.
- Sample $\{r'_i\}_{i\in\mathcal{H}} \subseteq \mathcal{R}$ uniformly at random subject to $\sum_{i\in\mathcal{H}} \lambda_i r'_i = r - \Delta$.

- For every $j \in \mathcal{H}$, send (r_j, r'_j) to P_j.

Let $\mathbf{M} = \mathrm{Van}^{n\times(n-t)}(\beta_1, \ldots, \beta_n)$, where $\beta_1, \ldots, \beta_n$ are different elements of $\mathcal{R}$.

Protocol $\Pi_{\text{D.Shar}}$

Output: A set of double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$.
2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_{n-t}]\!]_t \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_{n-1}]\!]_t \\ [\![s_n]\!]_t \end{pmatrix}, \qquad \begin{pmatrix} [\![r_1]\!]_{2t} \\ [\![r_2]\!]_{2t} \\ \vdots \\ [\![r_{n-t}]\!]_{2t} \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_{2t} \\ [\![s_2]\!]_{2t} \\ \vdots \\ [\![s_{n-1}]\!]_{2t} \\ [\![s_n]\!]_{2t} \end{pmatrix}.$$

3. The parties output the double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$.

Theorem 4.2. *Protocol $\Pi_{\text{D.Shar}}$ instantiates functionality $\mathcal{F}_{\text{D.Shar}}$ with perfect security against an active adversary corrupting $t < n/2$ parties.*

Proof. We define the simulator $\mathcal{S}$ below.

1. - For each $i \in \mathcal{H}$ and $j \in \mathcal{C}$, the virtual honest party $\overline{P}_i$ samples $\overline{s}_{ij}, \overline{s}'_{ij} \in_R \mathcal{R}$ and sends $(\overline{s}_{ij}, \overline{s}'_{ij})$ to P_j.
   - For each $i \in \mathcal{H}$ and $j \in \mathcal{C}$, the virtual honest party $\overline{P}_i$ receives a pair $(\overline{s}_{ji}, \overline{s}'_{ji})$. $\mathcal{S}$ computes $(\overline{s}_{j1}, \ldots, \overline{s}_{jn}) \leftarrow \mathrm{RecShares}_t(\{\overline{s}_{ji}\}_{i\in\mathcal{H}})$ and $\overline{s}_j \leftarrow \mathrm{RecSecret}_t(\{\overline{s}_{ji}\}_{i\in\mathcal{H}})$.[a]
2. N/A (local operations)
3. $\mathcal{S}$ computes the following quantities
   - $(\overline{r}_{1j}, \overline{r}_{2j}, \ldots, \overline{r}_{n-t,j})^\intercal = \mathbf{M}^\intercal \cdot (\overline{s}_{1j}, \overline{s}_{2j}, \ldots, \overline{s}_{n-1,j}, \overline{s}_{nj})^\intercal$ for $j \in \mathcal{C}$,
   - $\overline{\Gamma}_i = \overline{s}_i - \sum_{j\in\mathcal{H}} \lambda_j \overline{s}'_{ij}$ for $i \in \mathcal{C}$,
   - $\overline{\Delta}_i = \sum_{\ell\in\mathcal{C}} \mathbf{M}[i,\ell] \cdot \overline{\Gamma}_\ell$ for $i \in [n-t]$.

   Then, for each $i \in [n-t]$, $\mathcal{S}$ sends $(\{\overline{r}_{ij}\}_{j\in\mathcal{C}}, \overline{\Delta}_i)$ to $\mathcal{F}_{\text{D.Shar}}$.

[a] This is possible since $n = 2t+1$, so $|\mathcal{H}| = n - t = t+1$, which means that $\{\overline{s}_{ji}\}_{i\in\mathcal{H}}$ is t-consistent.

To argue indistinguishability we proceed by describing the view of the adversary in each of the two worlds as the computation progresses in the following diagram.

Real world	**Ideal world**
1. From Thm. 3.2, the shares the adversary receives $\{(s_{ij}, s'_{ij})\}_{j\in\mathcal{C}}$ for $i \in \mathcal{H}$ look uniformly random.	1. From Thm. 3.2, the adversary's shares $\{(\overline{s}_{ij}, \overline{s}'_{ij})\}_{j\in\mathcal{C}}$ for $i \in \mathcal{H}$ also look uniformly random.

2. N/A (local operations)	2. N/A (local operations)
3. Each honest party P_j with $j \in \mathcal{H}$ outputs $\{(r_{ij}, r'_{ij})\}_{i=1}^{n-t}$	3. The (real) honest parties P_j output uniformly random values $\{(\overline{r}_{ij}, \overline{r}'_{ij})\}_{i=1}^{n-t}$, constrained to $\{\overline{r}_{ij}\}_{j\in\mathcal{H}} \cup \{\overline{r}_{ij}\}_{j\in\mathcal{C}}$ being t-consistent with a uniformly random value $\overline{r}_i$ and $\sum_{j\in\mathcal{H}} \lambda_j \overline{r}'_{ij} = \overline{r}_i - \overline{\Delta}_i$, for $i \in [n-t]$.

To complete the proof of indistinguishability, we only need to show that the values output by the honest parties in the real execution are indistinguishable from these output by (real) honest parties in the ideal execution. To this end, we first define the counterparts in the real world of some of the values used by the simulator in the ideal world. For $(i,j) \in ([n] \times [n]) \setminus (\mathcal{C} \times \mathcal{C})$, let (s_{ij}, s'_{ij}) be the pair sent by P_i to P_j as the first part of the real protocol execution. We define the following:

- $(s_{i1}, \ldots, s_{in}) \leftarrow \mathsf{RecShares}_t(\{s_{ij}\}_{j\in\mathcal{H}})$ and $s_i \leftarrow \mathsf{RecSecret}_t(\{s_{ij}\}_{j\in\mathcal{H}})$ for $i \in [n]$. Note that, for $i \in \mathcal{H}$, s_i is the uniformly random value sampled by P_i in the protocol execution.

- $(r_{1j}, r_{2j}, \ldots, r_{n-t,j})^\intercal = \mathbf{M}^\intercal (s_{1j}, s_{2j}, \ldots, s_{nj})^\intercal$ for $j \in [n]$, and $(r'_{1j}, r'_{2j}, \ldots, r'_{n-t,j})^\intercal = \mathbf{M}^\intercal (s'_{1j}, s'_{2j}, \ldots, s'_{nj})^\intercal$ for $j \in \mathcal{H}$. Observe that $\{(r_{ij}, r'_{ij})\}_{i\in[n-t]}$ for $j \in \mathcal{H}$ is the output produced by the honest parties.

- $(r_1, r_2, \ldots, r_{n-t})^\intercal = \mathbf{M}^\intercal (s_1, s_2, \ldots, s_n)^\intercal$. Note that $r_i \leftarrow \mathsf{RecSecret}_t(\{r_{ij}\}_{j\in[n]})$ for $i \in [n-t]$.

- $\Gamma_i = s_i - \sum_{j\in\mathcal{H}} \lambda_j s'_{ij}$ for $i \in \mathcal{C}$, and $\Delta_i = \sum_{\ell\in\mathcal{C}} \mathbf{M}[i,\ell] \cdot \Gamma_\ell$ for $i \in [n-t]$. Notice that $\Delta_i = \sum_{\ell\in\mathcal{C}} \mathbf{M}[i,\ell] \cdot (s_\ell - \sum_{j\in\mathcal{H}} \lambda_j s'_{\ell j}) = r_i - \sum_{j\in\mathcal{H}} \lambda_j r'_{ij}$.

To show indistinguishability, it suffices to prove the following claim.

Claim 4.1. *The output $\{(r_{ij}, r'_{ij})\}_{i\in[n-t]}$ for the honest parties P_j for $j \in \mathcal{H}$ in the real world is uniformly random constrained to, for every $i \in [n-t]$:*

1. *r_i is uniformly random*
2. *$\{r_{ij}\}_{j\in\mathcal{H}} \cup \{r_{ij}\}_{j\in\mathcal{C}}$ is t-consistent*
3. *$\sum_{j\in\mathcal{H}} \lambda_j r'_{ij} = r_i - \Delta_i$.*

Proof. First we show that the values $r_1, \ldots, r_{n-t}$ are uniformly random. To this end, observe we can write $(r_i)_{i\in[n-t]} = \mathbf{M}^\intercal[\cdot, \mathcal{C}] \cdot (s_i)_{i\in\mathcal{C}} + \mathbf{M}^\intercal[\cdot, \mathcal{H}] \cdot (s_i)_{i\in\mathcal{H}}$. Since $\mathbf{M}^\intercal[\cdot, \mathcal{H}] = \mathsf{Van}^{(n-t)\times(n-t)}((\beta_i)_{i\in\mathcal{H}})^\intercal$, we have from Corollary 3.1 that $\mathbf{M}^\intercal[\cdot, \mathcal{H}]$ is invertible, so $(r_i)_{i\in[n-t]}$ is in one-to-one correspondence with $(s_i)_{i\in\mathcal{H}}$, which are uniformly random.

Now, in a similar way, it is easy to see that the mapping $\{\mathtt{s}_{ij}\}_{i,j\in\mathcal{H}} \mapsto \{\mathtt{r}_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$ given by $(\mathtt{r}_{ij})_{i\in[n-t]} = \mathbf{M}^\intercal[\cdot, \mathcal{C}] \cdot (s_{ij})_{i\in\mathcal{C}} + \mathbf{M}^\intercal[\cdot, \mathcal{H}] \cdot (\mathtt{s}_{ij})_{i\in\mathcal{H}}$ for $j \in \mathcal{H}$ is a one-to-one mapping between the values $\{\mathtt{s}_{ij}\}_{i,j\in\mathcal{H}}$ such that $\{\mathtt{s}_{ij}\}_{j\in\mathcal{H}} \cup \{s_{ij}\}_{j\in\mathcal{C}}$ are t-consistent with the secret

s_i for $i \in \mathcal{H}$, and the values $\{\mathbf{r}_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$ such that $\{\mathbf{r}_{ij}\}_{j\in\mathcal{H}} \cup \{r_{ij}\}_{j\in\mathcal{C}}$ are t-consistent with the secret r_i for $i \in [n-t]$. Since $\{s_{ij}\}_{i,j\in\mathcal{H}}$ constitute random values in the domain of this bijection, their image, $\{r_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$, constitute random values in the codomain, as desired.

In a similar way it is shown that $\{r'_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$ is uniformly random constrained to $\sum_{j\in\mathcal{H}} \lambda_j r'_{ij} = r_i - \Delta_i$. In a bit more detail, define the mapping $\{\mathbf{s}'_{ij}\}_{i,j\in\mathcal{H}} \mapsto \{\mathbf{r}'_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$ given by $(\mathbf{r}'_{ij})_{i\in[n-t]} = \mathbf{M}^\intercal[\cdot,\mathcal{C}] \cdot (s'_{ij})_{i\in\mathcal{C}} + \mathbf{M}^\intercal[\cdot,\mathcal{H}] \cdot (\mathbf{s}'_{ij})_{i\in\mathcal{H}}$ for $j \in \mathcal{H}$, where the domain is the set of values $\{\mathbf{s}'_{ij}\}_{i,j\in\mathcal{H}}$ such that $\sum_{j\in\mathcal{H}} \lambda_j \mathbf{s}'_{ij} + \Gamma_i = s_i$, and the codomain is the set of values $\{\mathbf{r}'_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$ such that $\sum_{j\in\mathcal{H}} \lambda_j \mathbf{r}'_{ij} + \Delta_i = r_i$. As before, this mapping is a bijection, so the output of the honest parties $\{r'_{ij}\}_{i\in[n-t],j\in\mathcal{H}}$, which is the image of $\{s'_{ij}\}_{i,j\in\mathcal{H}}$ under this mapping, looks uniformly random constrained to $\sum_{j\in\mathcal{H}} \lambda_j r'_{ij} = r_i - \Delta_i$, as desired. □

□

Communication complexity of $\Pi_{\text{D.Shar}}$. In this protocol, each party has to send 2 shares to each other single party, which leads to $2n^2$ ring elements being communicated. Since $n - t = t + 1 = (n+1)/2$ double-shares are produced, the amortized communication complexity per double-sharing is $\frac{4n^2}{n+1} \approx 4n$ elements in $\mathcal{R}$.

4.2.2 Secure Multiplication

We define a new multiplication functionality that accepts additive errors from the adversary.

Functionality $\mathcal{F}_{\text{Mult}}$

1. Receive (x_i, y_i) from each honest party P_i.
2. Call $x \leftarrow \text{RecSecret}_t(\{x_i\}_{i\in\mathcal{H}})$, $(x_1, \dots, x_n) \leftarrow \text{RecShares}_t(\{x_i\}_{i\in\mathcal{H}})$, $y \leftarrow \text{RecSecret}_t(\{y_i\}_{i\in\mathcal{H}})$ and $(y_1, \dots, y_n) \leftarrow \text{RecShares}_t(\{y_i\}_{i\in\mathcal{H}})$
3. Send $\{(x_i, y_i)\}_{i\in\mathcal{C}}$ to the adversary.
4. Wait for $(\{z_i\}_{i\in\mathcal{C}}, \delta)$ from the adversary.
5. Run $(z_1, \dots, z_n) \leftarrow \text{Share}_t(x \cdot y + \delta, \{z_i\}_{i\in\mathcal{C}})$.
6. For every $j \in \mathcal{H}$, send z_j to P_j.

The protocol to instantiate the functionality $\mathcal{F}_{\text{Mult}}$ is described below.

Protocol Π_{Mult}

Input: Secret-shared values $[\![x]\!]_t$ and $[\![y]\!]_t$.
Functionalities: $\mathcal{F}_{\text{D.Shar}}$.
Output: $[\![z = x \cdot y + \delta]\!]_t$ for some adversarially chosen value $\delta \in \mathcal{R}$.
Protocol: The parties execute the following

1. The parties call $\mathcal{F}_{\text{D.Shar}}$ to obtain $([\![r]\!]_t, [\![r]\!]_{2t})$
2. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
3. Each party P_i sends its share a_i of $[\![a]\!]_{2t}$ to P_1.
4. After receiving $(a_1, \ldots, a_n)$, P_1 computes $a = \sum_{i=1}^{n} \lambda_i a_i$ and sends a to all parties.
5. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

Theorem 4.3. *Protocol Π_{Mult} instantiates functionality $\mathcal{F}_{\text{Mult}}$ with perfect security in the $\mathcal{F}_{\text{D.Shar}}$-hybrid model against an active adversary corrupting $t < n/2$ parties.*

Proof. We define the simulator $\mathcal{S}$ as follows.

Before interacting with the adversary the simulator receives $\{(x_i, y_i)\}_{i\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$. Then, the simulation of the protocol proceeds as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{D.Shar}}$ by receiving $(\{\overline{r}_j\}_{j\in\mathcal{C}}, \overline{\Delta})$ from the adversary.
2. N/A (local operations)
3. $\mathcal{S}$ samples $\overline{a} \in_R \mathcal{R}$ and samples $\{\overline{a}_i\}_{i\in\mathcal{H}}$ uniformly at random, and let $\overline{a} = (-\overline{\Delta} + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i) + \sum_{i\in\mathcal{H}} \lambda_i \overline{a}_i$. Then:
   - If $1 \in \mathcal{C}$: $\mathcal{S}$ sends $\{\overline{a}_i\}_{i\in\mathcal{H}}$ to the corrupt party P_1.
   - If $1 \notin \mathcal{C}$: The virtual honest party $\overline{P}_1$ receives $\{\overline{a}'_i\}_{i\in\mathcal{C}}$ from the corrupt parties.
4. - If $1 \in \mathcal{C}$: Each virtual honest party $\overline{P}_j$ receives $\overline{a}^{(j)}$ from the corrupt P_1.
   - If $1 \notin \mathcal{C}$: The virtual honest party $\overline{P}_1$ sends $\overline{a}' = \sum_{i\in\mathcal{C}} \lambda_i \overline{a}'_i + \sum_{j\in\mathcal{H}} \lambda_j \overline{a}_j$ to the corrupt parties.
5. - If $1 \in \mathcal{C}$: $\mathcal{S}$ computes $(\overline{a}^{(1)}, \ldots, \overline{a}^{(n)}) \leftarrow \text{RecShares}_t(\{\overline{a}^{(j)}\}_{j\in\mathcal{H}})$, $\overline{a}' \leftarrow \text{RecSecret}_t(\{\overline{a}^{(j)}\}_{j\in\mathcal{H}})$ and $\overline{\delta} = \overline{a}' - \overline{a}$. Then $\mathcal{S}$ sets $\overline{z}_i = \overline{r}_i + \overline{a}^{(i)}$ for $i \in \mathcal{C}$ and sends $(\{\overline{z}_i\}_{i\in\mathcal{C}}, \overline{\delta})$ to $\mathcal{F}_{\text{Mult}}$.
   - If $1 \notin \mathcal{C}$: $\mathcal{S}$ computes $\overline{\delta} = \sum_{j\in\mathcal{C}} \lambda_j \overline{a}'_j - (-\overline{\Delta} + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i)$. Then $\mathcal{S}$ sets $\overline{z}_i = \overline{r}_i + \overline{a}'$ for $i \in \mathcal{C}$ and sends $(\{\overline{z}_i\}_{i\in\mathcal{C}}, \overline{\delta})$ to $\mathcal{F}_{\text{Mult}}$.

We argue indistinguishability.

Real world	**Ideal world**
1. The adversary sends $(\{r_j\}_{j\in\mathcal{C}}, \Delta)$ to $\mathcal{F}_{\text{D.Shar}}$.	1. The adversary sends $(\{\overline{r}_j\}_{j\in\mathcal{C}}, \overline{\Delta})$ to $\mathcal{F}_{\text{D.Shar}}$. Notice that these are equally distributed as in the real world since the adversary's view of both worlds is the same.
2. N/A (local computation)	2. N/A (local computation)

Now we divide the rest of the argument depending on whether P_1 is corrupt or not. Notice that the definitions of $\overline{\delta}$ and $\overline{a}'$ are different depending on this case. We begin with the case in which this holds, that is, $1 \in \mathcal{C}$.

Real world	**Ideal world**

3. P_1 receives $\{a_i\}_{i\in\mathcal{H}}$ from the honest parties. These are defined as $a_i = x_i y_i - r'_i$, where r'_i is uniformly random subject to $r = \Delta + \sum_{i\in\mathcal{H}} \lambda_i r'_i$, where r itself is also uniformly random. As a result, $\{a_i\}_{i\in\mathcal{H}}$ are uniformly random.	3. P_1 receives $\{\overline{a}_i\}_{i\in\mathcal{H}}$ from the virtual honest parties, which are also uniformly random.
4. Each honest party P_j receives $a^{(j)}$ from the corrupt P_1.	4. Each virtual honest party $\overline{P}_j$ receives $\overline{a}^{(j)}$ from the corrupt P_1. Notice these values follow the same distribution as in the real world since the adversary's view up to this point is indistinguishable in both worlds.
5. The honest parties output $\{z_i\}_{i\in\mathcal{H}}$, where $z_i = r_i + a^{(i)}$, with $\{r_i\}_{i\in\mathcal{H}}$, which are distributed by $\mathcal{F}_{\text{D.Shar}}$ to the honest parties, being uniformly random values subject to $\{r_i\}_{i\in\mathcal{H}} \cup \{r_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret r.	5. The real honest parties output uniformly random values $\{\overline{z}_i\}_{i\in\mathcal{H}}$ subject to $\{\overline{z}_i\}_{i\in\mathcal{H}} \cup \{\overline{z}_j\}_{j\in\mathcal{C}}$ being t-consistent and $xy + \overline{\delta} \leftarrow \text{RecSecret}_t(\{\overline{z}_i\}_{i\in\mathcal{H}})$.

To see the proof of indistinguishability in the case in which $1 \in \mathcal{C}$, we make use of the following claim.

Claim 4.2. *Assume $1 \in \mathcal{C}$. The output $\{z_i\}_{i\in\mathcal{H}}$ of the honest parties in the real world is uniformly random constrained to:*

- *$\{z_i\}_{i\in\mathcal{H}} \cup \{z_j\}_{j\in\mathcal{C}}$ is t-consistent, where, for $j \in \mathcal{C}$, $z_j = r_j + a^{(j)}$, with $(a^{(1)}, \dots, a^{(n)}) \leftarrow \text{RecShares}_t(\{a^{(i)}\}_{i\in\mathcal{H}})$.*
- *$x \cdot y + \delta \leftarrow \text{RecSecret}_t(\{z_i\}_{i\in\mathcal{H}})$, where $\delta = a' - a$, with $a = (-\Delta + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i) + \sum_{i\in\mathcal{H}} \lambda_i a_i$ and $a' \leftarrow \text{RecShares}_t(\{a^{(i)}\}_{i\in\mathcal{H}})$.*

Proof. Recall that, for $i \in \mathcal{H}$, $z_i = r_i + a^{(i)}$, with $\{r_i\}_{i\in\mathcal{H}}$ uniformly random values subject to $\{r_i\}_{i\in\mathcal{H}} \cup \{r_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret r. As a result, from their definition, it is easy to see that $\{z_i\}_{i\in\mathcal{H}}$ are uniformly random restricted to $\{z_i\}_{i\in\mathcal{H}} \cup \{z_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret $r + a'$.

Now, we have that

$$\begin{aligned} a &= (-\Delta + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i) + \sum_{i\in\mathcal{H}} \lambda_i a_i \\ &= (-\Delta + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i) + \sum_{i\in\mathcal{H}} \lambda_i (x_i y_i - r'_i) \\ &= \sum_{i=1}^{n} \lambda_i x_i y_i - (\Delta + \sum_{i\in\mathcal{H}} r'_i) = xy - r, \end{aligned}$$

so $r + a' = (r + a) + (a' - a) = xy + \delta$, as required. □

Now, it only remains to analyze the case in which P_1 is not corrupt, that is, $1 \notin \mathcal{C}$.

Real world	**Ideal world**
3. The corrupt parties send $\{a'_i\}_{i\in\mathcal{C}}$ to P_1.	3. The corrupt parties send $\{\overline{a}'_i\}_{i\in\mathcal{C}}$ to $\overline{P}_1$. Notice that these follow the same distribution as in the real world.
4. P_1 sends $a' = \sum_{i\in\mathcal{C}} \lambda_i a'_i + \sum_{j\in\mathcal{H}} \lambda_j a_j$ to the corrupt parties.	4. $\overline{P}_1$ sends $\overline{a}' = \sum_{i\in\mathcal{C}} \lambda_i \overline{a}'_i + \sum_{j\in\mathcal{H}} \lambda_j \overline{a}_j$ to the corrupt parties.
5. The honest parties output $\{z_i\}_{i\in\mathcal{H}}$, where $z_i = r_i + a'$, with $\{r_i\}_{i\in\mathcal{H}}$, which are distributed by $\mathcal{F}_{\text{D.Shar}}$ to the honest parties, being uniformly random values subject to $\{r_i\}_{i\in\mathcal{H}} \cup \{r_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret r.	5. The real honest parties output uniformly random values $\{\overline{z}_i\}_{i\in\mathcal{H}}$ subject to $\{\overline{z}_i\}_{i\in\mathcal{H}} \cup \{\overline{z}_j\}_{j\in\mathcal{C}}$ being t-consistent and $xy + \overline{\delta} \leftarrow \text{RecSecret}_t(\{\overline{z}_i\}_{i\in\mathcal{H}})$.

Claim 4.3. *The output $\{z_i\}_{i\in\mathcal{H}}$ of the honest parties in the real world is uniformly random constrained to:*

- *$\{z_i\}_{i\in\mathcal{H}} \cup \{z_j\}_{j\in\mathcal{C}}$ is t-consistent, where, for $j \in \mathcal{C}$, $z_j = r_j + a'$, with $a' = \sum_{i\in\mathcal{C}} \lambda_i a'_i + \sum_{j\in\mathcal{H}} \lambda_j a_j$.*
- *$x \cdot y + \delta \leftarrow \text{RecSecret}_t(\{z_i\}_{i\in\mathcal{H}})$, where $\delta = \sum_{j\in\mathcal{C}} \lambda_j a'_j - (-\Delta + \sum_{i\in\mathcal{C}} \lambda_i x_i y_i)$*

Proof. As before, recall that, for $i \in \mathcal{H}$, $z_i = r_i + a'$, with $\{r_i\}_{i\in\mathcal{H}}$ uniformly random values subject to $\{r_i\}_{i\in\mathcal{H}} \cup \{r_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret r. As a result, from their definition, it is easy to see that $\{z_i\}_{i\in\mathcal{H}}$ are uniformly random restricted to $\{z_i\}_{i\in\mathcal{H}} \cup \{z_j\}_{j\in\mathcal{C}}$ being t-consistent with the secret $r + a'$.

Now, we can see that

$$\begin{aligned} r + a' &= \left(\Delta + \sum_{i\in\mathcal{H}} \lambda_i r'_i\right) + \left(\sum_{i\in\mathcal{C}} \lambda_i a'_i + \sum_{j\in\mathcal{H}} \lambda_j a_j\right) \\ &= \left(\sum_{i\in\mathcal{H}} \lambda_i (x_i y_i - a_i)\right) + \left(\Delta + \sum_{i\in\mathcal{C}} \lambda_i a'_i + \sum_{j\in\mathcal{H}} \lambda_j a_j\right) \\ &= \left(\sum_{i=1} \lambda_i x_i y_i\right) + \left(\Delta - \sum_{j\in\mathcal{C}} \lambda_j x_j y_j + \sum_{i\in\mathcal{C}} \lambda_i a'_i\right) = xy + \delta, \end{aligned}$$

so $r + a' = xy + \delta$, as required.

□

□

Communication complexity of Π_{Mult}. The communication cost amounts to one call of $\Pi_{\text{D.Shar}}$, which is $\approx 4n$ elements in $\mathcal{R}$, together with each party sending one share to P_1,

and P_1 sending one value back to all the parties,[1] for a total of $2n$ ring elements more. Hence, the communication complexity of Π_{Mult} is $\approx 6n$ elements in $\mathcal{R}$.

4.3 Shares of Random Values

The goal of this section is to develop a protocol to generate a sharing $[\![s]\!]_t$, where $s \in_R \mathcal{A}$. We begin with the case in which $\mathcal{A} = \mathcal{R}$, which is relatively straightforward.

We use the same functionality as in Section 3.3.3, which we restate here for the sake of completeness.

Functionality $\mathcal{F}_{\text{Rand}}(\mathcal{A})$

- Sample $r \in_R \mathcal{A}$.
- Receive $\{r_j\}_{j\in\mathcal{C}}$ from the adversary.
- Run $(r_1, \ldots, r_n) \leftarrow \text{Share}_t(r, \{r_j\}_{j\in\mathcal{C}})$.
- For every $j \in \mathcal{H}$, send r_j to P_j.

4.3.1 $\mathcal{A} = \mathcal{R}$

Protocol $\Pi_{\text{Rand}}(\mathcal{R})$

Output: A set of sharings $\{[\![r_i]\!]\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathcal{R}$ and secret-shares it using a degree-t polynomial. The parties obtain $[\![s_i]\!]_t$.
2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_{n-t}]\!]_t \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_{n-1}]\!]_t \\ [\![s_n]\!]_t \end{pmatrix}.$$

3. The parties output the sharings $\{[\![r_i]\!]_t\}_{i=1}^{n-t}$.

The protocol is similar to Protocol $\Pi_{\text{D.Shar}}$ from Section 4.2.1 to obtain double sharings $([\![r]\!]_t, [\![r]\!]_{2t})$, but removing the degree-$2t$ part. The proof of the theorem below is similar to the one from Theorem 4.2.

[1] This step can be optimized by asking P_1 not to send a to all parties, but instead to send shares of a where t of them are defaulted to be 0.

Theorem 4.4. *Protocol $\Pi_{\text{Rand}}(\mathcal{R})$ instantiates functionality $\mathcal{F}_{\text{D.Shar}}(\mathcal{R})$ with perfect security against an active adversary corrupting $t < n/2$ parties.*

Communication complexity of $\Pi_{\text{Rand}}(\mathcal{R})$. Each party sends one share to each other party, for a total of n^2 elements communicated. Since $n - t = (n+1)/2$ sharings are produced, the cost per sharing is $2n^2/(n+1) \approx 2n$ elements in $\mathcal{R}$.

4.3.1.1 Public Random Values

A new functionality that we will require for the protocol from this chapter is $\mathcal{F}_{\text{Coin}}$, which provides the parties with fresh random elements of a given *set* $\mathcal{M}$.

Functionality $\mathcal{F}_{\text{Coin}}(\mathcal{M})$

When queried, sample $m \in_R \mathcal{M}$ and send m to all the parties.

For the case in which $\mathcal{M} = \mathcal{R}$, this functionality can be easily instantiated via the following simple protocol

Protocol $\Pi_{\text{Coin}}(\mathcal{R})$

Output: Fresh random value $r \in_R \mathcal{R}$.
Functionalities: $\mathcal{F}_{\text{Rand}}(\mathcal{R})$ and $\mathcal{F}_{\text{PublicRec}}(t)$.
Protocol: The parties execute the following

1. Parties call $\mathcal{F}_{\text{Rand}}$ to get $[\![r]\!]_t$.
2. Parties call $\mathcal{F}_{\text{PublicRec}}(2t)$ on input $[\![r]\!]_t$ to either learn r, or abort.

The following is easy to see.

Theorem 4.5. *Protocol $\Pi_{\text{Coin}}(\mathcal{R})$ instantiates functionality $\mathcal{F}_{\text{Coin}}(\mathcal{R})$ with perfect security in the $(\mathcal{F}_{\text{Rand}}(\mathcal{R}), \mathcal{F}_{\text{PublicRec}}(t))$-hybrid model against an active adversary corrupting $t < n/2$ parties.*

If the intended set $\mathcal{M}$ is not equal to $\mathcal{R}$, protocol $\Pi_{\text{Coin}}(\mathcal{R})$ can be used to instantiate $\mathcal{F}_{\text{Coin}}(\mathcal{M})$, assuming there is a surjective mapping $\mathcal{R}^\ell \to \mathcal{M}$ for some ℓ such that every element in the codomain has an equal number of preimages. This way, to instantiate $\mathcal{F}_{\text{Coin}}(\mathcal{M})$, $\Pi_{\text{Coin}}(\mathcal{R})$ is called ℓ times, and then the mapping above is applied to the resulting random value over $\mathcal{R}^\ell$.

Communication complexity of $\Pi_{\text{Coin}}(\mathcal{R})$. The cost is one call to $\Pi_{\text{Rand}}(\mathcal{R})$ ($\approx 2n$ elements) plus one call to $\Pi_{\text{PublicRec}}(t)$ ($\approx 4n$), for a total of $\approx 6n$ elements in $\mathcal{R}$.

4.3.2 $\mathcal{A} = \mathbb{Z}/2^k\mathbb{Z}$

Let $\chi \in \mathbb{Z}$ be such that $\chi \geq \kappa/\tau$. Let ϕ and ψ be the mappings from Section 3.2.5 for concatenating shares, satisfying $\phi : \mathsf{GR}(2^k,\tau)^\tau \to \mathsf{GR}(2^k,\tau^2)$ such that $\phi((\mathbb{Z}/2^k\mathbb{Z})^\tau) = \mathsf{GR}(2^k,\tau)$, and $\psi : \mathsf{GR}(2^k,\tau^2)^\chi \to \mathsf{GR}(2^k,\tau^2\chi)$ such that $\psi(\mathsf{GR}(2^k,\tau)^\chi) = \mathsf{GR}(2^k,\tau\chi)$.

The protocol in the case in which $\mathcal{A} \subsetneq \mathcal{R}$ follows the same layout as the protocol from the previous section for the case in which $\mathcal{A} = \mathcal{R}$, except that we must account for the fact that a corrupt party P_i may secret-share $[\![s_i]\!]_t$, where $s_i \notin \mathcal{A}$. This is handled by performing a check that ensures that each secret this is the case.

Protocol $\Pi_{\mathsf{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$

Output: A set of sharings $\{[\![r^{(i\ell h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$ where $r^{(i\ell h)} \in_R \mathbb{Z}/2^k\mathbb{Z}$.
Functionalities: $\mathcal{F}_{\mathsf{PublicRec}}(t)$, $\mathcal{F}_{\mathsf{Coin}}(\mathsf{GR}(2^k,\tau\chi))$.
Protocol: The parties proceed as follows

1. Each party P_i samples $s^{(ijh)} \in_R \mathbb{Z}/2^k\mathbb{Z}$ for $j \in [\tau]$ and $h \in [\chi\cdot\mu]$ and secret-shares each of these values over $\mathcal{R}$ using a degree-t polynomial. The parties obtain $\{[\![s^{(ijh)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{j\in[\tau],h\in[\chi\mu]}$.
2. The parties compute locally $[\![s^{(ih)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \leftarrow \phi([\![s^{(i,1,h)}]\!]_t,\ldots,[\![s^{(i,\tau,h)}]\!]_t)$ for $i \in [n], h \in [\chi\mu]$. Notice that $s^{(ih)} \in \phi((\mathbb{Z}/2^k\mathbb{Z})^\tau) = \mathsf{GR}(2^k,\tau)$.
3. The parties compute locally the following shares for $h \in [\chi\mu]$:
$$\begin{pmatrix} [\![r^{(1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![r^{(2,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ \vdots \\ [\![r^{(n-t,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \end{pmatrix} = \mathbf{M}^\intercal \cdot \begin{pmatrix} [\![s^{(1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![s^{(2,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ \vdots \\ [\![s^{(n-1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![s^{(n,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \end{pmatrix}.$$
4. The parties compute locally
$$[\![p_{(j-1)\mu+i}]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)} \leftarrow \psi\left([\![r^{(j,(i-1)\chi+1)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)},\ldots,[\![r^{(j,(i-1)\chi+\chi)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)}\right)$$
for $j \in [n-t], i \in [\mu]$.
5. The parties call $\mathcal{F}_{\mathsf{Coin}}(\mathsf{GR}(2^k,\tau\chi))$ to get $\omega \in_R \mathsf{GR}(2^k,\tau\chi)$.
6. The parties compute locally $[\![z]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)} \leftarrow \sum_{i=1}^{(n-t)\mu} \omega^{i-1}\,[\![p_i]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)}$.
7. The parties call $\mathcal{F}_{\mathsf{PublicRec}}^{\mathsf{GR}(2^k,\tau^2\chi)}(t)$ to reconstruct z from $[\![z]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)}$, or abort.
8. The parties compute locally $([\![r^{(i,1,h)}]\!]_t^{(\mathsf{GR}(2^k,\tau))},\ldots,[\![r^{(i,\tau,h)}]\!]_t^{(\mathsf{GR}(2^k,\tau))}) \leftarrow \phi^{-1}([\![r^{(i,h)}]\!]_t^{(\mathsf{GR}(2^k,\tau^2))})$ for $i \in [n-t], h \in [\chi\mu]$.
9. If $z \in \mathsf{GR}(2^k,\tau\chi)$, then the parties output the sharings $\{[\![r^{(i\ell h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$. Else, the parties abort.

Theorem 4.6. *Protocol $\Pi_{\mathsf{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$ instantiates functionality $\mathcal{F}_{\mathsf{Mult}}(\mathbb{Z}/2^k\mathbb{Z})$ with statistical security in the $\mathcal{F}_{\mathsf{PublicRec}}(t)$-hybrid model against an active adversary corrupting $t < n/2$ parties.*

Proof. Unlike previous proofs, we will not go down to the detail of individual sharings, given the density of sub/super-indexes present in the protocol. As usual, we begin by defining the simulator $\mathcal{S}$.

1. Each virtual party $\overline{P}_i$ and each corrupt party P_i samples $\overline{s}^{(ijh)} \in_R \mathbb{Z}/2^k\mathbb{Z}$ for $j \in [\tau]$ and $h \in [\chi \cdot \mu]$ and secret-shares each of these values over $\mathcal{R}$ using a degree-t polynomial. The parties obtain $\{[\![\overline{s}^{(ijh)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n],j\in[\tau],h\in[\chi\mu]}$, and note that $\mathcal{S}$ knows the corresponding secrets as it controls the virtual honest parties. Then $\mathcal{S}$ computes the following
   - $[\![\overline{s}^{(ih)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \leftarrow \phi([\![\overline{s}^{(i,1,h)}]\!]_t, \ldots, [\![\overline{s}^{(i,\tau,h)}]\!]_t)$ for $i \in [n], h \in [\chi\mu]$.
   - For $h \in [\chi\mu]$:

$$\begin{pmatrix} [\![\overline{r}^{(1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![\overline{r}^{(2,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ \vdots \\ [\![\overline{r}^{(n-t,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \end{pmatrix} = \mathbf{M}^{\mathsf{T}} \cdot \begin{pmatrix} [\![\overline{s}^{(1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![\overline{s}^{(2,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ \vdots \\ [\![\overline{s}^{(n-1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \\ [\![\overline{s}^{(n,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)} \end{pmatrix}.$$

   - For $j \in [n-t], i \in [\mu]$:

$$[\![\overline{p}_{(j-1)\mu+i}]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)} \leftarrow \psi\left([\![\overline{r}^{(j,(i-1)\chi+1)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)}, \ldots, [\![\overline{r}^{(j,(i-1)\chi+\chi)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)}\right)$$

2. N/A (local operations)
3. N/A (local operations)
4. N/A (local operations)
5. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}(\mathsf{GR}(2^k,\tau\chi))$ by sampling $\overline{\omega} \in_R \mathsf{GR}(2^k,\tau\chi)$ and sending this value to the corrupt parties.
6. N/A (local operations). The simulator computes locally $[\![\overline{z}]\!]_t \leftarrow \sum_{i=1}^{(n-t)\mu} \overline{\omega}^{i-1} [\![\overline{p}_i]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)}$.
7. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PublicRec}}^{\mathsf{GR}(2^k,\tau^2\chi)}(t)$ by reconstructing $\overline{z}$ from $[\![\overline{z}]\!]_t^{\mathsf{GR}(2^k,\tau^2\chi)}$, or abort.
8. N/A (local operations). The simulator computes locally $([\![\overline{r}^{(i,1,h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}, \ldots, [\![\overline{r}^{(i,\tau,h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}) \leftarrow \phi^{-1}([\![\overline{r}^{(i,h)}]\!]_t^{\mathsf{GR}(2^k,\tau^2)})$ for $i \in [n-t], h \in [\chi\mu]$.
9. If there exists $i \in \mathcal{C}, j \in [\tau], h \in [\chi\mu]$ such that $\overline{s}^{(ijh)} \notin \mathbb{Z}/2^k\mathbb{Z}$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$. Else, for $i \in [n-t] \setminus \{1\}, \ell \in [\tau], h \in [\chi\mu] \setminus [\chi]$, $\mathcal{S}$ inputs the adversary shares corresponding to $\{[\![\overline{r}^{(i\ell h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$ to the functionality $\mathcal{F}_{\mathsf{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$.

Our proof of indistinguishability in this case will contain less details, compared to the ones provided in previous parts of this chapter. First, notice that the execution in the ideal and real worlds are indistinguishable since the virtual honest parties execute the exact same protocol as the real parties do. The main potential difference can occur in the output of the two scenarios. These are the possible differences:

1. **Real world.** The honest parties abort if the opened value z is not in $\mathsf{GR}(2^k, \tau\chi)$.

   Ideal world. The honest parties abort if there exists $i \in \mathcal{C}, j \in [\tau], h \in [\chi\mu]$ such that $s^{(ijh)} \notin \mathbb{Z}/2^k\mathbb{Z}$

2. Assuming no abort happens:

   Real world. The honest parties output shares $\{[\![r^{(i\ell h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$.

   Ideal world. The honest parties output shares $\{[\![\overline{r}^{(i\ell h)}]\!]_t^{\mathsf{GR}(2^k,\tau)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$ where the secrets are uniformly random elements of $\mathbb{Z}/2^k\mathbb{Z}$, and the corrupt party's shares are the same as in the real world.

In the following we show that the differences above do not yield any distinguishing advantage to the adversary. We begin with the first item.

Claim 4.4. *If the honest parties do not abort in the ideal world, then they do not abort in the real world.*

Proof. Assume that the parties do not abort in the ideal world, so $\{\overline{s}^{(ijh)}\}_{i\in\mathcal{C},j\in[\tau],h\in[\chi\cdot\mu]}$ all belong to $\mathbb{Z}/2^k\mathbb{Z}$. Since these values are indistinguishable from their equivalent ones in the real world, we have that $\{s^{(ijh)}\}_{i\in\mathcal{C},j\in[\tau],h\in[\chi\cdot\mu]}$ all belong to $\mathbb{Z}/2^k\mathbb{Z}$, which implies that $s^{(ih)} = \phi(s^{(i,1,h)}, \ldots, s^{(i,\tau,h)})$ belong to $\mathsf{GR}(2^k, \tau)$ for $i \in \mathcal{C}, h \in [\chi\mu]$. Now, since the values $r^{(ih)}$ for $i \in [n-t], h \in [\chi\mu]$ are all $\mathsf{GR}(2^k, \tau)$-linear combinations of the values above, all these belong to $\mathsf{GR}(2^k, \tau)$ too. This in turn implies that $p_{(j-1)\mu+i} = \psi\left(r^{(j,(i-1)\chi+1)}, \ldots, r^{(j,(i-1)\chi+\chi)}\right)$ all belong to $\mathsf{GR}(2^k, \tau\chi)$ for $j \in [n-t], i \in [\mu]$.

From the above, together with the fact that $\omega \in \mathsf{GR}(2^k, \tau\chi)$, we see that $z = \sum_{i=1}^{(n-t)\mu} \omega^{i-1} p_i$ also belongs to $\mathsf{GR}(2^k, \tau\chi)$, so the honest parties do not abort. Notice that this assumes that the adversary does not cause an abort when invoking $\mathcal{F}_{\mathsf{PublicRec}}(t)$, which holds since this is the case in the ideal world and the two executions up to this point are indistinguishable. □

Claim 4.5. *If the honest parties abort in the ideal world, then, with overwhelming probability in the security parameter κ, they abort in the real world.*

Proof. We prove that the probability of the event in which the parties abort in the ideal world, yet they do not abort in the real world, is at most $((n-t)\mu - 1)2^{-\kappa}$. To see this, assume that there exists $i_0 \in \mathcal{C}, j_0 \in [\tau], h_0 \in [\chi\mu]$ such that $s^{(i_0 j_0 h_0)} \notin \mathbb{Z}/2^k\mathbb{Z}$. This implies that $s^{(i_0 h_0)} = \phi(s^{(i_0,1,h_0)}, \ldots, s^{(i_0,\tau,h_0)}) \notin \mathsf{GR}(2^k, \tau)$.

Now we show that the above implies that there exists $i_1 \in [n-t]$ such that $r^{(i_1 h_0)} \notin \mathrm{GR}(2^k,\tau)$. Indeed, we have that $(r^{(ih_0)})_{i\in[t]} = \mathbf{M}^\intercal[\,[t],\mathcal{H}\,](s^{(ih_0)})_{i\in\mathcal{H}} + \mathbf{M}^\intercal[\,[t],\mathcal{C}\,](s^{(ih_0)})_{i\in\mathcal{C}}$. $\mathbf{M}^\intercal[\,[t],\mathcal{C}\,] = \mathrm{Van}^{t\times t}((\beta_i)_{i\in\mathcal{C}})^\intercal$ is invertible from Corollary 3.1, so we can rewrite $(s^{(ih_0)})_{i\in\mathcal{C}} = \mathbf{M}^\intercal[\,[t],\mathcal{C}\,]^{-1}(r^{(ih_0)})_{i\in[t]} - \mathbf{M}^\intercal[\,[t],\mathcal{C}\,]^{-1}\mathbf{M}[\,[t],\mathcal{H}\,](s^{(ih_0)})_{i\in\mathcal{H}}$. Recall that the coefficients of $\mathbf{M}$ are in $\mathrm{GR}(2^k,\tau)$. If all the entries of $(r^{(ih_0)})_{i\in[t]}$ were in $\mathrm{GR}(2^k,\tau)$, then the equation above would imply that all the entries of $(s^{(ih_0)})_{i\in\mathcal{C}}$ are in $\mathrm{GR}(2^k,\tau)$, which is not the case since this does not hold for index $i = i_0$. Hence, there must exist $i_1 \in [t] \subseteq [n-t]$ such that $r^{(i_1 h_0)} \notin \mathrm{GR}(2^k,\tau)$.

Let $h_1 \in [\mu]$ be such that $(h_1-1)\chi+1 \le h_0 \le (h_1-1)\chi+\chi$, and let $\ell_0 = (i_1-1)\mu + h_1$. From the above we obtain that $p_{\ell_0} = \psi\left(r^{(i_1,(h_1-1)\chi+1)},\dots,r^{(i_1,(h_1-1)\chi+\chi)}\right)$ does not belong to $\mathrm{GR}(2^k,\tau\chi)$. Based on this, making use of Proposition 4.2, we see that the probability that $z = \sum_{i=1}^{(n-t)\mu}\omega^{i-1}p_i$ belongs to $\mathrm{GR}(2^k,\tau\chi)$ is at most $\frac{(n-t)\mu-1}{2^{\tau\chi}}$. Since $\tau\chi \ge \kappa$, this probability is at most $((n-t)\mu-1)2^{-\kappa}$, which is negligible in κ assuming $n-t = t+1$ is polynomial in κ. □

Finally, we need to see that the output distribution in both worlds is indistinguishable in the case that no abort is produced. This is proved in the following claim.

Claim 4.6. *The values $\{r^{(i\ell h)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$ are uniformly random over $\mathbb{Z}/2^k\mathbb{Z}$.*

Proof. Before opening $z = \sum_{i=1}^{(n-t)\mu}\omega^{i-1}p_i$, the values $\{r^{(i\ell h)}\}_{i\in[n-t],\ell\in[\tau],h\in[\chi\mu]}$ look uniformly random to the adversary since they are in a one-to-one correspondence with the values $\{r^{(ih)}\}_{i\in[n-t],h\in[\chi\mu]}$, which are also in correspondence with $\{s^{(ih)}\}_{i\in\mathcal{H},h\in[\chi\mu]}$, which are themselves equivalent to the values $\{s^{(i\ell h)}\}_{i\in\mathcal{H},\ell\in[\tau],h\in[\chi\mu]}$, which are uniformly random as these are sampled by honest parties.

In particular, from the above we see that $p_1,\dots,p_{(n-t)\mu}$ are uniformly random. However, after opening z, the adversary learns some additional information. Fortunately, it still holds that $p_2,\dots,p_{(n-t)\mu}$ are uniformly random, conditioned on the value of z. This is translated into $\{r^{(i\ell h)}\}_{i\in[n-t]\setminus\{1\},\ell\in[\tau],h\in[\chi\mu]\setminus[\chi]}$ being uniformly random conditioned on z, as desired. □

□

Communication complexity of $\Pi_{\mathbf{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$. The cost of the protocol is the aggregation of the following:

- Each party sends $\chi\mu$ shares to each other party: $\chi\mu n^2$ elements in $\mathcal{R}$.
- One call to $\Pi_{\mathrm{Coin}}(\mathrm{GR}(2^k,\tau\chi))$, which involves $6n$ elements in $\mathrm{GR}(2^k,\tau\chi)$, or $6n\chi$ elements in $\mathcal{R}$.
- One call to $\Pi^{\mathrm{GR}(2^k,\tau^2\chi)}_{\mathrm{PublicRec}}(t)$, which are $4n$ elements in $\mathrm{GR}(2^k,\tau^2\chi)$, or $4n\tau\chi$ elements in $\mathcal{R}$.

This leads to a total of $\approx \chi\mu n^2 + 6n\chi + 4n\tau\chi = \chi n(6+\mu n+4\tau)$. Since $\tau(n-t-1)(\chi\mu-\chi) = \frac{\tau\chi(n-1)(\mu-1)}{2}$ shared values are produced, the amortized cost per shared value is

$$\frac{2\chi n(6+\mu n+4\tau)}{\tau\chi(n-1)(\mu-1)} \approx \frac{12+2\mu n+8\tau}{\tau(\mu-1)} \approx \frac{2n}{\tau} + \frac{8\tau+12}{\tau(\mu-1)}.$$

We will approximate this to $2n$ elements in $\mathcal{R}$ in light that the second summand approaches zero as μ approaches infinity.

4.4 Verifying Multiplication Triples with a Galois Ring Extension

The protocol from Section 4.2 allows the parties to securely obtain $[\![x \cdot y + \delta]\!]$ from $[\![x]\!]$ and $[\![y]\!]$, where δ is an additive error chosen by the adversary. In this section we show how to ensure that this error is zero, or, in other words, that the triple $([\![x]\!], [\![y]\!], [\![x \cdot y + \delta]\!])$ is what is known as a correct multiplication triple. This is formalized by means of the following functionality.

Functionality $\mathcal{F}_{\text{MultCheck}}$

1. Receive (x_i, y_i, z_i) from each honest party P_i
2. Let $x \leftarrow \text{RecSecret}_t(\{x_i\}_{i\in\mathcal{H}})$, $y \leftarrow \text{RecSecret}_t(\{y_i\}_{i\in\mathcal{H}})$ and $z \leftarrow \text{RecSecret}_t(\{z_i\}_{i\in\mathcal{H}})$. Also, let $(x_1, \ldots, x_n) \leftarrow \text{RecShares}_t(\{x_i\}_{i\in\mathcal{H}})$, $(y_1, \ldots, y_n) \leftarrow \text{RecShares}_t(\{y_i\}_{i\in\mathcal{H}})$ and $(z_1, \ldots, z_n) \leftarrow \text{RecShares}_t(\{z_i\}_{i\in\mathcal{H}})$.
3. Send $(\{x_i\}_{i\in\mathcal{C}}, \{y_i\}_{i\in\mathcal{C}}, \{z_i\}_{i\in\mathcal{C}}, \delta)$, where $\delta = z - xy$, to the adversary,
4. Send abort to the honest parties if $\delta \neq 0$.

Below we let $\mathcal{P} = \text{GR}(2^k, \tau\chi)$, where, recall, χ is chosen so that $\tau\chi \geq \kappa$. Also recall that $\mathcal{R} = \text{GR}(2^k, \tau)$. For some functionalities we write the ring over which they operate as a superscript.

Protocol $\Pi_{\text{MultCheck}}$

Input: Secret shared values $([\![x_i]\!]_t, [\![y_i]\!]_t, [\![z_i]\!]_t)$ with $z_i = x_i \cdot y_i + \delta_i$ for some adversarially-chosen value $\delta_i \in \mathcal{R}$, for $i \in [m]$.
Output: The parties abort if $\delta_{i_0} \neq 0$ for some $i_0 \in [m]$.
Functionalities: $\mathcal{F}_{\text{Rand}}(\mathcal{R})$, $\mathcal{F}_{\text{Rand}}(\mathcal{P})$, $\mathcal{F}_{\text{PublicRec}}(t)$, $\mathcal{F}_{\text{Coin}}(\mathcal{P})$.
Protocol: The parties proceed as follows:

1. The parties call $\mathcal{F}_{\text{Rand}}(\mathcal{R})$ and $\mathcal{F}_{\text{Rand}}(\mathcal{P})$ to obtain $[\![a_i]\!]_t^{\mathcal{R}}$ and $[\![b_i]\!]_t^{\mathcal{P}}$, where $a_i \in_R \mathcal{R}$ and $b_i \in_R \mathcal{P}$, for $i \in [m]$.
2. For $i \in [m]$, the parties compute locally:
   - $[\![a_i]\!]_t^{\mathcal{P}} \leftarrow [\![a_i]\!]_t^{\mathcal{R}}$,
   - $[\![y_i]\!]_t^{\mathcal{P}} \leftarrow [\![y_i]\!]_t^{\mathcal{R}}$,
   - $[\![z_i]\!]_t^{\mathcal{P}} \leftarrow [\![z_i]\!]_t^{\mathcal{R}}$,
3. The parties call $\mathcal{F}_{\text{Mult}}^{\mathcal{P}}$ to obtain $[\![c_i]\!]_t^{\mathcal{P}}$ from $[\![a_i]\!]_t^{\mathcal{P}}$ and $[\![b_i]\!]_t^{\mathcal{P}}$, where $c_i = a_i b_i + \epsilon_i$ for some $\epsilon_i \in \mathcal{P}$ chosen by the adversary, for $i \in [m]$
4. The parties call $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ to get $\omega, \rho \in_R \mathcal{P}$.

5. For $i \in [m]$, the parties compute locally:
   - $[\![d_i]\!]_t^{\mathcal{P}} \leftarrow [\![x_i]\!]_t^{\mathcal{P}} - [\![a_i]\!]_t^{\mathcal{P}}$
   - $[\![e_i]\!]_t^{\mathcal{P}} \leftarrow \omega\,[\![y_i]\!]_t^{\mathcal{P}} - [\![b_i]\!]_t^{\mathcal{P}}$.
6. The parties call $\mathcal{F}_{\text{PublicRec}}^{\mathcal{R}}(t)$ and $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ to learn d_i and e_i for $i \in [m]$, or abort.
7. The parties compute locally the following
   - $[\![h_i]\!]_t^{\mathcal{P}} \leftarrow \omega\,[\![z_i]\!]_t^{\mathcal{P}} - (d_i\,[\![b_i]\!]_t^{\mathcal{P}} + e_i\,[\![a_i]\!]_t^{\mathcal{P}} + [\![c_i]\!]_t^{\mathcal{P}} + d_i e_i)$ for $i \in [m]$.
   - $[\![q]\!]_t^{\mathcal{P}} \leftarrow \sum_{i=1}^{m} \rho^{i-1}\,[\![h_i]\!]_t^{\mathcal{P}}$.
8. The parties call $\mathcal{F}_{\text{PublicRec}}(t)^{\mathcal{P}}$ to learn q.
9. If $q \neq 0$, the parties abort.

Theorem 4.7. *Protocol $\Pi_{\text{MultCheck}}$ instantiates functionality $\mathcal{F}_{\text{MultCheck}}$ with statistical security in the $(\mathcal{F}_{\text{Rand}}(\mathcal{R}), \mathcal{F}_{\text{Rand}}(\mathcal{P}), \mathcal{F}_{\text{PublicRec}}(t), \mathcal{F}_{\text{Coin}}(\mathcal{P}))$-hybrid model against an active adversary corrupting $t < n/2$ parties.*

Proof. The simulator is defined as follows

$\mathcal{S}$ receives $(\{x_{ij}\}_{j\in\mathcal{C}}, \{y_{ij}\}_{j\in\mathcal{C}}, \{z_{ij}\}_{j\in\mathcal{C}}, \delta_i)$ from $\mathcal{F}_{\text{MultCheck}}$, for $i \in [m]$.

1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Rand}}(\mathcal{R})$ and $\mathcal{F}_{\text{Rand}}(\mathcal{P})$ by receiving $\{\overline{a}_{ij}\}_{j\in\mathcal{C}} \subseteq \mathcal{R}$ and $\{\overline{b}_{ij}\}_{j\in\mathcal{C}} \subseteq \mathcal{P}$ from the adversary, for $i \in [m]$.
2. N/A (local computation)
3. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Mult}}^{\mathcal{P}}$ by sending $\{(\overline{a}_{ij}, \overline{b}_{ij})\}_{j\in\mathcal{C}}$ to the adversary and receiving $(\{\overline{c}_{ij}\}_{j\in\mathcal{C}}, \overline{\epsilon}_i)$ from the adversary, for $i \in [m]$.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ by sending $\overline{\omega}, \overline{\rho} \in_R \mathcal{P}$ to the adversary.
5. N/A (local computation)
6. For $i \in [m], j \in [\chi]$, $\mathcal{S}$ proceeds as follows:
   - Compute $\overline{d}_{ij} = \overline{x}_{ij} - \overline{a}_{ij}$ and $\overline{e}_{ij} = \overline{\omega}\overline{y}_{ij} - \overline{b}_{ij}$ for $j \in \mathcal{C}$.
   - Sample $\overline{d}_i \in_R \mathcal{R}$ and $\overline{e}_i \in_R \mathcal{P}$, and call $(\overline{d}_{i1}, \ldots, \overline{d}_{in}) \leftarrow \text{Share}_t^{\mathcal{R}}(d_i, \{d_{ij}\}_{j\in\mathcal{C}})$ and $(\overline{e}_{i1}, \ldots, \overline{e}_{in}) \leftarrow \text{Share}_t^{\mathcal{P}}(\overline{e}_i, \{\overline{e}_{ij}\}_{j\in\mathcal{C}})$.
   - Emulate $\mathcal{F}_{\text{PublicRec}}^{\mathcal{R}}(t)$ and $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ by sending $\{\overline{d}_{ij}\}_{j\in[n]}$ and $\{\overline{e}_{ij}\}_{j\in[n]}$ to the adversary.
7. N/A (local computation)
8. $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ as follows:
   - $\mathcal{S}$ computes $\overline{h}_{ij} = \overline{z}_{ij}\overline{\omega} - (\overline{d}_i\overline{b}_{ij} + \overline{e}_i\overline{a}_{ij} + \overline{c}_{ij} + \overline{d}_i\overline{e}_i)$ for $i \in [m], j \in \mathcal{C}$, and $\overline{q}_j = \sum_{i=1}^{m} \overline{\rho}^i \overline{h}_{ij}$ for $j \in \mathcal{C}$.
   - $\mathcal{S}$ computes $\overline{q} = \sum_{i=1}^{m} \overline{\rho}^{i-1}(\overline{\delta}_i\overline{\omega} - \overline{\epsilon}_i)$.
   - $\mathcal{S}$ calls $(\overline{q}_1, \ldots, \overline{q}_n) \leftarrow \text{Share}_t^{\mathcal{P}}(\overline{q}, \{\overline{q}_j\}_{j\in\mathcal{C}})$.
   - $\mathcal{S}$ sends $(\overline{q}_1, \ldots, \overline{q}_n)$ to the adversary as the emulation of $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$.
9. $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{MultCheck}}$ if there exists $i_0 \in [m]$ such that $\overline{\epsilon}_{i_0} \neq 0$. Also, by its definition, $\mathcal{F}_{\text{MultCheck}}$ will cause the honest parties to abort if there exists $i_0 \in [m]$ such that $\delta_{i_0} \neq 0$.

We analyze indistinguishability in the following diagram

Real world	**Ideal world**
1. The adversary sends $\{a_{ij}\}_{j\in\mathcal{C}}, \{b_{ijj}\}_{j\in\mathcal{C}}$ to $\mathcal{F}_{\text{Rand}}(\mathcal{R})$ and $\mathcal{F}_{\text{Rand}}(\mathcal{P})$, for $i \in [m]$. Let $a_i \in_R \mathcal{R}$ and $b_i \in_R \mathcal{P}$ be the random values sampled by these functionalities.	1. The adversary sends $\{\overline{a}_{ij}\}_{j\in\mathcal{C}}, \{\overline{b}_{ij}\}_{j\in\mathcal{C}}$ to the emulated $\mathcal{F}_{\text{Rand}}(\mathcal{R})$ and $\mathcal{F}_{\text{Rand}}(\mathcal{P})$, for $i \in [m]$. Notice these follow the same distribution as in the real world.
2. N/A (local computation)	2. N/A (local computation)
3. The adversary receives $\{(a_{ij}, b_{ij})\}_{j\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$, and then it sends $(\{c_{ij}\}_{j\in\mathcal{C}}, \epsilon_i)$ to $\mathcal{F}_{\text{Mult}}$, for $i \in [m]$.	3. The adversary receives $\{(\overline{a}_{ij}, \overline{b}_{ij})\}_{j\in\mathcal{C}}$ from the emulated $\mathcal{F}_{\text{Mult}}$, and then it sends $(\{\overline{c}_{ij}\}_{j\in\mathcal{C}}, \overline{\epsilon}_i)$ to the emulated $\mathcal{F}_{\text{Mult}}$, for $i \in [m]$. The distribution of these values in both worlds is indistinguishable.
4. The adversary receives $\rho, \omega \in_R \mathcal{P}$ from $\mathcal{F}_{\text{Coin}}(\mathcal{P})$.	4. The adversary receives also random values $\overline{\rho}, \overline{\omega} \in_R \mathcal{P}$ from the emulated $\mathcal{F}_{\text{Coin}}(\mathcal{P})$.
5. N/A (local computation)	5. N/A (local computation)
6. For $i \in [m]$, the adversary holds shares $d_{ij} = x_{ij} - a_{ij}$ and $e_{ij} = \omega y_{ij} - b_{ij}$ for $j \in \mathcal{C}$, and receives t-consistent vectors $(d_{i1}, \ldots, d_{in})$ and $(e_{i1}, \ldots, e_{in})$ from $\mathcal{F}^{\mathcal{R}}_{\text{PublicRec}}(t)$ and $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where the underlying secrets are $d_i = x_i - a_i$ and $e_i = y_i\omega - b_j$, which are uniformly random over $\mathcal{R}$ and $\mathcal{P}$ respectively since $a_i \in_R \mathcal{R}$ and $b_i \in_R \mathcal{P}$.	6. For $i \in [m]$, the adversary t-consistent vectors $(\overline{d}_{i1}, \ldots, \overline{d}_{in})$ and $(\overline{e}_{i1}, \ldots, \overline{e}_{in})$ from the emulated $\mathcal{F}^{\mathcal{R}}_{\text{PublicRec}}(t)$ and $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where the underlying secrets are uniformly random $\overline{d}_i \in_R \mathcal{R}$ and $\overline{e}_i \in_R \mathcal{P}$.
7. N/A (local computation)	7. N/A (local computation)
8. Let $h_{ij} = z_{ij}\omega - (d_i b_{ij} + e_i a_{ij} + c_{ij} + d_i e_i)$ for $i \in [m], j \in \mathcal{C}$. The adversary holds shares $q_j = \sum_{i=1}^m \rho^{i-1} h_{ij}$ for $j \in \mathcal{C}$, and receives t-consistent vectors $(q_1, \ldots, q_n)$ from $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where the underlying secret is $q = \sum_{i=1}^m \rho^{i-1} h_i$, with $h_i = \omega z_i - (d_i b_i + e_i a_i + c_i + d_i e_i)$. A straightforward computation shows that $q = \sum_{i=1}^m \rho^{i-1}(\delta_i\omega - \epsilon_i)$	8. The shares $\{\overline{q}_j\}_{j\in\mathcal{C}}$ computed by $\mathcal{S}$ are obtained in the same way as in the real world. The adversary receives from the emulated $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$ t-consistent vectors $(\overline{q}_1, \ldots, \overline{q}_n)$, where the underlying secret is $\overline{q} = \sum_{i=1}^m \rho^{i-1}(\delta_i\overline{\omega} - \overline{\epsilon}_i)$, which follows the same distribution as in the real world.
8. The honest parties abort in this stage if and only if $q \neq 0$.	8. The (real) honest parties abort in this stage if and only if there exists $i_0 \in [m]$ such that $\delta_{i_0} \neq 0$, or if there exists $i_0 \in [m]$ such that $\overline{\epsilon}_{i_0} \neq 0$.

From the analysis above we see that, the only place where the environment could distinguish between the two executions is the event in which the parties abort in the last stage of the protocol.[2] To see that this happens negligibly close probabilities in both worlds, we rely on the following pair of claims.

Claim 4.7. *If the honest parties abort in the real world, then they abort in the ideal world.*

[2] Note that the adversary can still cause aborts in earlier parts of the protocols, like in the calls to $\mathcal{F}_{\text{PublicRec}}(t)$, but these occur with equal probability in both executions. Hence, we only need to analyze the case of an abort in the last stage of the protocol.

Proof. We prove the counter positive statement, that is, if the honest parties do not abort in the ideal world, then they do not abort in the real world. To see this, assume that the honest parties do not abort in the ideal world, so δ_i and $\bar{\epsilon}_i$ are all zero for $i \in [m]$. As a result, ϵ_i, which follow the same distribution in the real world as the values above from the ideal world, would all be zero for $i \in [m]$. From this it can be seen that the opened value $q = \sum_{i=1}^{m} \rho^{i-1}(\delta_i \omega - \epsilon_i)$ is equal to 0, so the parties do not abort in the real world. □

Claim 4.8. *If the honest parties abort in the ideal world, then, with overwhelming probability in the security parameter κ, they abort in the real world.*

Proof. Assume that the parties abort in the ideal world, which, in terms of the real world values, means that there exists $i_0 \in [m]$ such that either (1) $\delta_{i_0} \neq 0$, or (2) $\epsilon_{i_0} \neq 0$. Let E_0 be the event in which $\delta_{i_0}\omega - \epsilon_{i_0} = 0$, where the randomness is taken over the uniformly random value $\omega \in_R \mathcal{P}$, We claim that $\Pr[E_0] \leq 2^{-\tau\chi}$. To see this, first we notice that regardless of whether case (1) or (2) above holds, $\delta_{i_0}\mathtt{X} - \epsilon_{i_0}$ is a non-zero polynomial of degree 1 over $\mathcal{P}$, so from Proposition 4.1, we have that the probability that $\delta_{i_0}\omega - \epsilon_{i_0} = 0$ is upper bounded by $1/2^{\tau\chi}$. Hence, $\Pr[E_0] \leq 1/2^{\tau\chi}$.

Recall that $q = \sum_{i=1}^{m} \rho^{i-1}(\delta_i \omega - \epsilon_i)$. Let E_1 be the event in which $h = 0$, where the randomness is taken over the uniformly random value $\rho \in_R \mathcal{P}$. We claim that $\Pr[E_1 \mid \neg E_0] \leq \frac{m-1}{2^{\tau\chi}}$. To see this, assume E_0 does not hold, so $\delta_{i_0}\omega - \epsilon_{i_0} \neq 0$. Then, $\sum_{i=1}^{m} \mathtt{X}^{i-1}(\delta_i \omega - \epsilon_i)$ would be a non-zero polynomial over $\mathcal{P}$ of degree at most $m-1$. From Proposition 4.1, the probability that $q = 0$ is upper bounded by $\frac{m-1}{2^{\tau\chi}}$, as required.

Putting together the pieces above, we see that

$$\begin{aligned}\Pr[E_1] &= \Pr[E_1|E_0]\Pr[E_0] + \Pr[E_1|\neg E_0]\Pr[\neg E_0] \\ &\leq 1 \cdot \frac{1}{2^{\tau\chi}} + \frac{m-1}{2^{\tau\chi}} \cdot 1 = \frac{m}{2^{\tau\chi}}.\end{aligned}$$

Since $\tau\chi \geq \kappa$, we have that $\Pr[E_1] \leq m2^{-\kappa}$, which is negligible in κ, assuming that m is polynomial in κ. □

□

Communication complexity of $\Pi_{\text{MultCheck}}$. The communication cost is obtained by adding the following quantities.

- m calls to $\Pi_{\text{Rand}}(\mathcal{R})$, which cost $2nm$ elements in $\mathcal{R}$.
- m calls to $\Pi_{\text{Rand}}(\mathcal{P})$, which cost $2nm$ elements in $\mathcal{P}$, or $2nm\chi$ elements in $\mathcal{R}$.
- m calls to $\Pi^{\mathcal{P}}_{\text{Mult}}$, which cost $6nm$ elements in $\mathcal{P}$, or $6nm\chi$ elements in $\mathcal{R}$.
- Two calls to $\Pi_{\text{Coin}}(\mathcal{P})$, which cost $12n$ elements in $\mathcal{P}$, or $12n\chi$ elements in $\mathcal{R}$.

- m calls to $\Pi^{\mathcal{R}}_{\text{PublicRec}}(t)$, which cost $4nm$ elements in $\mathcal{R}$
- m calls to $\Pi^{\mathcal{P}}_{\text{PublicRec}}(t)$, which cost $4nm$ elements in $\mathcal{P}$, or $4nm\chi$ elements in $\mathcal{R}$.
- One call to $\Pi^{\mathcal{P}}_{\text{PublicRec}}(t)$, which costs $4n$ elements in $\mathcal{P}$, or $4n\chi$ elements in $\mathcal{R}$.

Hence, the total is $6nm + 12nm\chi + 16n\chi$. Since m triples are verified, the amortized cost per multiplication triple is $6n + 12n\chi + \frac{16n\chi}{m}$, which we approximate to $6n + 12n\chi$ since the last term approaches 0 as m grows. Approximating $\tau\chi \approx \kappa$, we would have that the cost is $\approx 6n + \frac{12n\kappa}{\tau}$

4.5 Verifying Triples Without Security Parameter Overhead

The protocol from the previous section allows the parties to check that m triples $([\![x_i]\!], [\![y_i]\!], [\![z_i]\!])$ satisfy $z_i = x_i y_i$. However, its communication complexity per triple checked grows with $n \cdot \kappa$. The goal of this section is to design a protocol with a communication complexity per triple that is independent of the statistical security parameter κ. The ideas presented here are an adaptation of the techniques from [18], which are set in the field setting, to the Galois ring scenario. We remark that in [57] a further refinement of these techniques (over fields) is presented, where the overhead in communication complexity is made *logarithmic*, rather than linear, in the number of multiplications checked. Using a similar adaptation as the one we present below, such tools can be also made to work in the Galois ring setting without any complication.

Recall that $\mathcal{R} = \text{GR}(2^k, \tau)$. Let $\mathcal{L} = \text{GR}(2^k, \tau\mu)$ such that $2^{\tau\mu} \geq 2m + 1$, and let $\mathcal{P} = \text{GR}(2^k, \tau\mu\chi)$, where $\tau\mu\chi \geq \kappa$.

Let $\{\beta_1, \dots, \beta_{2m+1}\} \in \mathcal{P}$ be an exceptional set. Let $\mathbf{M} = \text{Van}^{(m)\times(m)}(\beta_1, \dots, \beta_m)$, which is invertible. Let $\mathbf{N} = \text{Van}^{(m-2)\times(m)}(\beta_{m+1}, \dots, \beta_{2m-1})$ Let $\mathbf{O} = \text{Van}^{(2m-1)\times(2m-1)}(\beta_1, \dots, \beta_{2m-1})$

Protocol $\Pi_{\text{MultCheck}*}$

Input: Secret shared values $([\![x_i]\!]_t, [\![y_i]\!]_t, [\![z_i]\!]_t)$ with $z_i = x_i \cdot y_i + \delta_i$ for some adversarially-chosen value $\delta_i \in \mathcal{R}$, for $i \in [m]$.
Output: The parties abort if $\delta_{i_0} \neq 0$ for some $i_0 \in [m]$.
Functionalities: $\mathcal{F}_{\text{Rand}}(\mathcal{R})$, $\mathcal{F}_{\text{PublicRec}}(t)$, $\mathcal{F}_{\text{Coin}}(\mathcal{P})$.
Protocol: The parties proceed as follows:

1. The parties compute locally $[\![x_i]\!]^{\mathcal{L}} \leftarrow [\![x_i]\!]^{\mathcal{R}}$ and $[\![y_i]\!]^{\mathcal{L}} \leftarrow [\![y_i]\!]^{\mathcal{R}}$ for $i \in [m]$,

$$\begin{pmatrix} [\![u_0]\!]^{\mathcal{L}} \\ \vdots \\ [\![u_{m-1}]\!]^{\mathcal{L}} \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} [\![x_1]\!]^{\mathcal{L}} \\ \vdots \\ [\![x_m]\!]^{\mathcal{L}} \end{pmatrix}, \qquad \begin{pmatrix} [\![v_0]\!]^{\mathcal{L}} \\ \vdots \\ [\![v_{m-1}]\!]^{\mathcal{L}} \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} [\![y_1]\!]^{\mathcal{L}} \\ \vdots \\ [\![y_m]\!]^{\mathcal{L}} \end{pmatrix},$$

and

$$\begin{pmatrix} [\![x_{m+1}]\!]^{\mathcal{L}} \\ \vdots \\ [\![x_{2m-1}]\!]^{\mathcal{L}} \end{pmatrix} = \mathbf{N} \begin{pmatrix} [\![u_0]\!]^{\mathcal{L}} \\ \vdots \\ [\![u_{m-1}]\!]^{\mathcal{L}} \end{pmatrix}, \qquad \begin{pmatrix} [\![y_{m+1}]\!]^{\mathcal{L}} \\ \vdots \\ [\![y_{2m-1}]\!]^{\mathcal{L}} \end{pmatrix} = \mathbf{N} \begin{pmatrix} [\![v_0]\!]^{\mathcal{L}} \\ \vdots \\ [\![v_{m-1}]\!]^{\mathcal{L}} \end{pmatrix}.$$

2. The parties call $\mathcal{F}_{\text{Mult}}^{\mathcal{L}}$ to obtain $[\![z_i]\!]^{\mathcal{L}}$ from $[\![x_i]\!]^{\mathcal{L}}$ and $[\![y_i]\!]^{\mathcal{L}}$, where $z_i = x_i y_i + \delta_i$, for $i \in \{m+1, \dots, 2m-1\}$.
3. The parties compute locally

$$\begin{pmatrix} [\![w_0]\!]^{\mathcal{L}} \\ \vdots \\ [\![w_{2m-2}]\!]^{\mathcal{L}} \end{pmatrix} = \mathbf{O}^{-1} \begin{pmatrix} [\![z_1]\!]^{\mathcal{L}} \\ \vdots \\ [\![z_{2m-1}]\!]^{\mathcal{L}} \end{pmatrix}.$$

   The also compute $[\![u_i]\!]^{\mathcal{P}} \leftarrow [\![u_i]\!]^{\mathcal{L}}$ and $[\![v_i]\!]^{\mathcal{P}} \leftarrow [\![v_i]\!]^{\mathcal{L}}$ for $i \in \{0\} \cup [m-1]$, and $[\![w_i]\!]^{\mathcal{P}} \leftarrow [\![w_i]\!]^{\mathcal{L}}$ for $i \in \{0\} \cup [2m-2]$.
4. The parties call $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ to get $\rho \in_R \mathcal{P}$. Repeat if necessary until $\rho \notin \{\beta_1, \dots, \beta_{2m-1}\}$.
5. The parties compute locally
   - $[\![a]\!]^{\mathcal{P}} \leftarrow \sum_{i=0}^{m-1} \rho^i [\![u_i]\!]^{\mathcal{P}}$,
   - $[\![b]\!]^{\mathcal{P}} \leftarrow \sum_{i=0}^{m-1} \rho^i [\![v_i]\!]^{\mathcal{P}}$,
   - $[\![c]\!]^{\mathcal{P}} \leftarrow \sum_{i=0}^{2m-2} \rho^i [\![w_i]\!]^{\mathcal{P}}$.
6. The parties call $\mathcal{F}_{\text{Rand}}(\mathcal{P})$ to get $[\![a']\!]^{\mathcal{P}}$, $[\![b']\!]^{\mathcal{P}}$.
7. The parties call $\mathcal{F}_{\text{Mult}}^{\mathcal{P}}$ to get $[\![c']\!]^{\mathcal{P}}$ with $c' = a'b' + \delta'$.
8. The parties call $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ to get $\omega \in_R \mathcal{P}$.
9. The parties compute locally $[\![d]\!]^{\mathcal{P}} \leftarrow [\![a]\!]^{\mathcal{P}} - [\![a']\!]^{\mathcal{P}}$ and $[\![e]\!]^{\mathcal{P}} \leftarrow \omega [\![b]\!]^{\mathcal{P}} - [\![b']\!]^{\mathcal{P}}$.
10. The parties call $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ to get d and e.
11. The parties compute locally $[\![h]\!]_t^{\mathcal{P}} \leftarrow \omega [\![c]\!]_t^{\mathcal{P}} - (d [\![b']\!]_t^{\mathcal{P}} + e [\![a']\!]_t^{\mathcal{P}} + [\![c']\!]_t^{\mathcal{P}} + de)$
12. The parties call $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ to get h.
13. If $h \neq 0$, the parties abort.

Theorem 4.8. *Protocol $\Pi_{\text{MultCheck}*}$ instantiates functionality $\mathcal{F}_{\text{MultCheck}}$ with statistical security in the $(\mathcal{F}_{\text{Rand}}(\mathcal{R}), \mathcal{F}_{\text{PublicRec}}(t), \mathcal{F}_{\text{Coin}}(\mathcal{P}))$-hybrid model against an active adversary corrupting $t < n/2$ parties.*

Proof. The simulator is defined as follows

$\mathcal{S}$ receives $(\{x_{ij}\}_{j\in\mathcal{C}}, \{y_{ij}\}_{j\in\mathcal{C}}, \{z_{ij}\}_{j\in\mathcal{C}}, \delta_i)$ from $\mathcal{F}_{\text{MultCheck}}$, for $i \in [m]$. We define $\overline{x}_{ij} = x_{ij}$, $\overline{y}_{ij} = y_{ij}$, $\overline{z}_{ij} = z_{ij}$ and $\overline{\delta}_i = \delta_i$ for $i \in [m], j \in \mathcal{C}$

1. N/A (local computation)
2. $\mathcal{S}$ computes for $j \in \mathcal{C}$:

$$\begin{pmatrix} \overline{u}_{0j} \\ \vdots \\ \overline{u}_{m-1,j} \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} \overline{x}_{1j} \\ \vdots \\ \overline{x}_{mj} \end{pmatrix}, \qquad \begin{pmatrix} \overline{v}_{0j} \\ \vdots \\ \overline{v}_{m-1,j} \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} \overline{y}_{1j} \\ \vdots \\ \overline{y}_{mj} \end{pmatrix},$$

and

$$\begin{pmatrix} \overline{x}_{m+1,j} \\ \vdots \\ \overline{x}_{2m-1,j} \end{pmatrix} = \mathbf{N} \begin{pmatrix} \overline{u}_{0j} \\ \vdots \\ \overline{u}_{m-1,j} \end{pmatrix}, \qquad \begin{pmatrix} \overline{y}_{m+1,j} \\ \vdots \\ \overline{y}_{2m-1,j} \end{pmatrix} = \mathbf{N} \begin{pmatrix} \overline{v}_{0j} \\ \vdots \\ \overline{v}_{m-1,j} \end{pmatrix}.$$

Then $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{Mult}}^{\mathcal{L}}$ by sending $\{(\overline{x}_{ij}, \overline{y}_{ij})\}_{j\in\mathcal{C}}$ to the adversary and receiving back $(\{\overline{z}_{ij}\}_{j\in\mathcal{C}}, \overline{\delta}_i)$, for $i \in \{m+1, \dots, 2m-1\}$.

3. N/A (local computation)
4. $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ by sampling $\overline{\rho} \in_R \mathcal{P}$, sending this value to the adversary, and repeating if necessary until $\rho \notin \{\beta_1, \dots, \beta_{2m-1}\}$.
5. N/A (local computation)
6. $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{Rand}}(\mathcal{P})$ by receiving $\{\overline{a}'_j\}_{j\in\mathcal{C}}$ and $\{\overline{b}'_j\}_{j\in\mathcal{C}}$ from the adversary.
7. $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{Mult}}^{\mathcal{P}}$ by sending $\{\overline{a}'_j\}_{j\in\mathcal{C}}, \{\overline{b}'_j\}_{j\in\mathcal{C}}$ to the adversary and receiving $(\{\overline{c}'_j\}_{j\in\mathcal{C}}, \overline{\delta}')$
8. $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{Coin}}(\mathcal{P})$ by sampling $\overline{\omega} \in_R \mathcal{P}$ and sending this value to the adversary
9. N/A (local computation)
10. $\mathcal{S}$ computes

$$\begin{pmatrix} \overline{w}_{0j} \\ \vdots \\ \overline{w}_{2m-2,j} \end{pmatrix} = \mathbf{O}^{-1} \begin{pmatrix} \overline{z}_{1j} \\ \vdots \\ \overline{z}_{2m-1,j} \end{pmatrix},$$

and also $\overline{a}_j = \sum_{i=0}^{m-1} \rho^i \overline{u}_{ij}$, $\overline{b}_j = \sum_{i=0}^{m-1} \rho^i \overline{v}_{ij}$ and $\overline{c}_j = \sum_{i=0}^{2m-2} \rho^i \overline{w}_{ij}$ for $j \in \mathcal{C}$. Then $\mathcal{S}$ sets $\overline{d}_j = \overline{a}_j - \overline{a}'_j$ and $\overline{e}_j = \overline{\omega} \cdot \overline{b}_j - \overline{b}'_j$ for $j \in \mathcal{C}$, samples $\overline{d}, \overline{e} \in_R \mathcal{P}$, and calls $(\overline{d}_1, \dots, \overline{d}_n) \leftarrow \text{Share}_t^{\mathcal{P}}(\overline{d}, \{\overline{d}_j\}_{j\in\mathcal{C}})$ and $(\overline{e}_1, \dots, \overline{e}_n) \leftarrow \text{Share}_t^{\mathcal{P}}(\overline{e}, \{\overline{e}_j\}_{j\in\mathcal{C}})$. Finally, $\mathcal{S}$ emulates the call to $\mathcal{F}_{\text{PublicRec}}^{\mathcal{P}}(t)$ by sending $(\overline{d}_1, \dots, \overline{d}_n)$ and $(\overline{e}_1, \dots, \overline{e}_n)$ to the adversary.

11. N/A (local computation)
12. $\mathcal{S}$ computes

$$\begin{pmatrix} \overline{\epsilon}_0 \\ \vdots \\ \overline{\epsilon}_{2m-2} \end{pmatrix} = \mathbf{O}^{-1} \begin{pmatrix} \overline{\delta}_1 \\ \vdots \\ \overline{\delta}_{2m-1} \end{pmatrix}$$

and $\overline{h} = \overline{\omega} \left(\sum_{i=0}^{2m-2} \overline{\rho}^i \overline{\epsilon}_i \right) - \overline{\delta}'$. Then $\mathcal{S}$ calls $(\overline{h}_1, \dots, \overline{h}_n) \leftarrow \text{Share}_t^{\mathcal{P}}(\overline{h}, \{\overline{h}_j\}_{j\in\mathcal{C}})$

13. $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{MultCheck}}$ if there exists $i_0 \in \{m+1, \dots, 2m-1\}$ such that $\overline{\delta}_{i_0} \neq 0$, or if $\overline{\delta}' \neq 0$. Also, by its definition, $\mathcal{F}_{\text{MultCheck}}$ will cause the honest parties to abort if there exists $i_0 \in [m]$ such that $\delta_{i_0} \neq 0$.

We analyze indistinguishability in the following diagram

Real world	**Ideal world**
1. N/A (local computation)	1. N/A (local computation)
2. The adversary receives $\{(x_{ij}, y_{ij})\}_{j\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$, and then it sends $([\![z_{ij}]\!]_{j\in\mathcal{C}}, \delta_i)$ for $i \in \{m+1, \dots, 2m-1\}$	2. The adversary receives $\{(\overline{x}_{ij}, \overline{y}_{ij})\}_{j\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$, which are equal to $\{(x_{ij}, y_{ij})\}_{j\in\mathcal{C}}$ by their definition, and then it sends $([\![\overline{z}_{ij}]\!]_{j\in\mathcal{C}}, \overline{\delta}_i)$ for $i \in \{m+1, \dots, 2m-1\}$, following the same distribution as in the real world.

3. N/A (local computation)	3. N/A (local computation)
4. The adversary learns $\rho \in_R \mathcal{P}$ from $\mathcal{F}_{\text{Rand}}(\mathcal{P})$, a process that is repeated until $\rho \in \{\beta_1, \dots, \beta_{2m-1}\}$.	4. The adversary learns $\overline{\rho} \in_R \mathcal{P}$ from the emulated $\mathcal{F}_{\text{Rand}}(\mathcal{P})$, a process that is repeated until $\overline{\rho} \notin \{\beta_1, \dots, \beta_{2m-1}\}$. The distribution of these values in both worlds is indistinguishable.
5. N/A (local computation)	5. N/A (local computation)
6. The adversary sends $[\![a'_j]\!]_{j\in\mathcal{C}}$ and $[\![b'_j]\!]_{j\in\mathcal{C}}$ to $\mathcal{F}_{\text{Rand}}(\mathcal{P})$. Let $a', b' \in_R \mathcal{P}$ be the random values sampled by $\mathcal{F}_{\text{Rand}}(\mathcal{P})$.	6. The adversary sends $[\![\overline{a}'_j]\!]_{j\in\mathcal{C}}$ and $\left[\!\left[\overline{b}'_j\right]\!\right]_{j\in\mathcal{C}}$ to the emulated $\mathcal{F}_{\text{Rand}}(\mathcal{P})$. These values follow the same distribution as in the real world.
7. The adversary receives $\{a'_j\}_{j\in\mathcal{C}}$ and $\{b'_j\}_{j\in\mathcal{C}}$ from $\mathcal{F}_{\text{Mult}}$, and it sends back $(\{c'_j\}_{j\in\mathcal{C}}, \delta')$.	7. The adversary receives $\{\overline{a}'_j\}_{j\in\mathcal{C}}$ and $\{\overline{b}'_j\}_{j\in\mathcal{C}}$ from the emulated $\mathcal{F}_{\text{Mult}}$, and it sends back $(\{\overline{c}'_j\}_{j\in\mathcal{C}}, \delta')$, which follow the same distribution as the real world given that both executions are indistinguishable up to this point.
8. The adversary learns $\omega \in_R \mathcal{P}$ from $\mathcal{F}_{\text{Rand}}(\mathcal{P})$	8. The adversary learns $\overline{\omega} \in_R \mathcal{P}$ from the emulated $\mathcal{F}_{\text{Rand}}(\mathcal{P})$ The distribution of these values in both worlds is the same.
9. N/A (local computation)	9. N/A (local computation)
10. The adversary gets $\{d_i\}_{i\in[n]}$ and $\{e_i\}_{i\in[n]}$ from $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where $d_i = a_i - a'_i$ and $e_i = \omega b_i - b'_i$ for $i \in [n]$. Since $\{a'_i\}_{i\in\mathcal{H}}$ and $\{b'_i\}_{i\in\mathcal{H}}$ are uniformly random subject to $\{a'_i\}_{i\in[n]}$ and $\{b'_i\}_{i\in[n]}$ being t-consistent with the random secrets $a', b' \in_R \mathcal{P}$, $\{d_i\}_{i\in\mathcal{C}}$ and $\{e_i\}_{i\in\mathcal{C}}$ are uniformly random restricted to $\{d_i\}_{i\in[n]}$ and $\{e_i\}_{i\in[n]}$ being t-consistent with uniformly random secrets $d, e \in_R \mathcal{P}$.	10. The adversary gets $\{\overline{d}_i\}_{i\in[n]}$ and $\{\overline{e}_i\}_{i\in[n]}$ from the emulated $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where $\{d_i\}_{i\in\mathcal{H}}$ and $\{e_i\}_{i\in\mathcal{H}}$ are uniformly random restricted to $\{d_i\}_{i\in[n]}$ and $\{e_i\}_{i\in[n]}$ being t-consistent with uniformly random secrets $\overline{d}, \overline{e} \in_R \mathcal{P}$. This is the same distribution as in the real world.
11. N/A (local computation)	11. N/A (local computation)
12. The adversary gets $(h_1, \dots, h_n)$ from $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where h_i is the share of P_i corresponding to $[\![h]\!]_t$. It can be checked with direct computation that $h = \omega\left(\sum_{i=0}^{2m-2} \rho^i \epsilon_i\right) - \delta'$, where $\begin{pmatrix} \epsilon_0 \\ \vdots \\ \epsilon_{2m-2} \end{pmatrix} = \mathbf{O}^{-1} \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_{2m-1} \end{pmatrix}$	12. The adversary gets $(\overline{h}_1, \dots, \overline{h}_n)$ from $\mathcal{F}^{\mathcal{P}}_{\text{PublicRec}}(t)$, where $\overline{h}_j$ are the corrupt parties' shares and $\{\overline{h}_i\}_{i\in\mathcal{H}}$ are uniformly random subject to $\{\overline{h}_i\}_{i\in[n]}$ being t-consistent with the secret $\overline{h} = \overline{\omega}\left(\sum_{i=0}^{2m-2} \overline{\rho}^i \overline{\epsilon}_i\right) - \overline{\delta}'$, which is the exact same distribution as in the real world.
13. The honest parties abort in this stage if and only if $h \neq 0$	13. The (real) honest parties abort in this stage if and only if there exists $i_0 \in [2m-1]$ such that $\overline{\delta}_{i_0} \neq 0$, or if $\overline{\delta}' \neq 0$.

Given the above, the only place where the environment could distinguish between the real and ideal worlds is the event in which the parties abort in the last stage of the protocol. To see that this event happens negligibly close probabilities in both worlds, we

rely on the following two claims, which are analogous to the ones found in the proof of Theorem 4.7.

Claim 4.9. *If the honest parties abort in the real world, then they abort in the ideal world.*

Proof. We prove the counter positive statement, that is, if the honest parties do not abort in the ideal world, then they do not abort in the real world. To see this, assume that the honest parties do not abort in the ideal world, so $\overline{\delta}_i$ for $i \in [2m-1]$ and $\overline{\delta}'$ are all zero. As a result, the values from the real world δ_i for $i \in [2m-1]$ and δ, which follow the same distribution as the values above from the ideal world, would all be zero. From this it can be seen that $\epsilon_i = 0$ for $i \in \{0\} \cup [2m-2]$, so the opened value $h = \omega\left(\sum_{i=0}^{2m-2} \rho^i \epsilon_i\right) - \delta'$ is equal to 0. Hence, the honest parties do not abort in the real world. □

Claim 4.10. *If the honest parties abort in the ideal world, then, with overwhelming probability in the security parameter κ, they abort in the real world.*

Proof. Assume that the parties abort in the ideal world, which means that either (1) there exists $i_0 \in [m]$ such that $\overline{\delta}_{i_0} \neq 0$, or (2) $\overline{\delta}' \neq 0$. In terms of the values from the real execution that follow the same distribution as the values above, this can be phrased as either (1) there exists $i_0 \in \cup[2m-1]$ such that $\delta_{i_0} \neq 0$, which happens if and only if there exists $j_0 \in \{0\} \cup [2m-1]$ such that $\epsilon_{j_0} \neq 0$, or (2) $\delta' \neq 0$. First, notice that if (2) holds, but (1) does not, then the final opened value by the parties in the real execution is $h = -\delta' \neq 0$, so the parties abort. Hence, it remains to analyze the case in which (1) holds.

Let E_0 be the event in which $\sum_{i=0}^{2m-2} \rho^i \epsilon_i = 0$, where the randomness is taken over the uniformly random value $\rho \in_R \mathcal{P}$, We claim that $\Pr[E_0] \leq 2^{-\tau\chi}$. Since (1) holds, $\sum_{i=0}^{2m-2} \mathtt{X}^i \epsilon_i$ is a non-zero polynomial of degree at most $m-2$ over $\mathcal{P}$, so from Proposition 4.1, we have that the probability that $\sum_{i=0}^{m-2} \rho^i \epsilon_i = 0$ is upper bounded by $(2m-2)/2^{\tau\chi}$. Hence, $\Pr[E_0] \leq (2m-2)/2^{\tau\chi}$.

Let E_1 be the event in which $h = \omega\left(\sum_{i=0}^{2m-2} \rho^i \epsilon_i\right) - \delta'$ is equal to 0, where the randomness is taken over the uniformly random value $\omega \in_R \mathcal{P}$. We claim that $\Pr[E_1 \mid \neg E_0] \leq \frac{1}{2^{\tau\chi}}$. To see this, assume E_0 does not hold, so $\sum_{i=0}^{2m-2} \rho^i \epsilon_i \neq 0$. Then, $\mathtt{X}\left(\sum_{i=0}^{2m-2} \rho^i \epsilon_i\right) - \delta'$ would be a non-zero polynomial over $\mathcal{P}$ of degree 1. From Proposition 4.1, the probability that $h = 0$ is upper bounded by $\frac{1}{2^{\tau\chi}}$, as required.

Putting together the pieces above, we see that

$$\begin{aligned}\Pr[E_1] &= \Pr[E_1|E_0]\Pr[E_0] + \Pr[E_1|\neg E_0]\Pr[\neg E_0] \\ &\leq 1 \cdot \frac{2m-2}{2^{\tau\chi}} + \frac{1}{2^{\tau\chi}} \cdot 1 = \frac{2m-1}{2^{\tau\chi}}.\end{aligned}$$

Since $\tau\chi \geq \kappa$, we have that $\Pr[E_1] \leq (2m-1)2^{-\kappa}$, which is negligible in κ, assuming that m is polynomial in κ. □

□

Communication complexity of $\Pi_{\text{MultCheck}*}$. We sum the following quantities

- $m-1$ calls to $\Pi^{\mathcal{L}}_{\text{Mult}}$, which involve $(m-1)6n$ elements in $\mathcal{L}$, or $(m-1)6n\mu$ elements in $\mathcal{R}$.
- One call to $\Pi_{\text{Coin}}(\mathcal{P})$, which costs $6n$ elements in $\mathcal{P}$, or $6n\mu\chi$ elements in $\mathcal{R}$.
- One call to $\Pi_{\text{Rand}}(\mathcal{P})$, which costs $2n$ elements in $\mathcal{P}$, or $2n\mu\chi$ elements in $\mathcal{R}$.
- One call to $\Pi^{\mathcal{P}}_{\text{Mult}}$, which costs $6n$ elements in $\mathcal{P}$, or $6n\mu\chi$ elements in $\mathcal{R}$.
- One call to $\Pi_{\text{Coin}}(\mathcal{P})$, which costs $6n$ elements in $\mathcal{P}$, or $6n\mu\chi$ elements in $\mathcal{R}$.
- Two calls to $\Pi^{\mathcal{P}}_{\text{PublicRec}}$, which costs $8n$ elements in $\mathcal{P}$, or $8n\mu\chi$ elements in $\mathcal{R}$.
- One more call to $\Pi_{\text{PublicRec}}(\mathcal{P})$, which costs $4n$ elements in $\mathcal{P}$, or $4n\mu\chi$ elements in $\mathcal{R}$.

This gives a total of $(m-1)6n\mu + 32n\mu\chi$, which, divided by the amount m of triples checked, leads to $\approx 6n\mu + \frac{32n\mu\chi}{m}$. We remove the second term since this approaches 0 as m grows. Furthermore, we approximate $\tau\mu \approx \log(2m+1)$, so the above becomes $\approx \frac{6n\log(2m+1)}{\tau}$ elements in $\mathcal{R}$. In particular, the communication complexity is independent of the security parameter κ, as desired.

4.6 Secret-Sharing Inputs

The final building block needed for our MPC protocol is a functionality that enables parties to provide inputs, while ensuring these lie in the ring $\mathbb{Z}/2^k\mathbb{Z}$. This functionality, together with the protocol instantiating it, are essentially the same as in Section 3.3.4 for the setting in which $t < n/3$.

Functionality $\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$

Let $s \in [n]$ be the index of the party providing input.

- Receive (input, x) from P_s.
  - If $x \in \mathbb{Z}/2^k\mathbb{Z}$ then store (P_s, x) in memory.
  - Else send abort to the honest parties.
- On input $\{\overline{x}_i\}_{i\in\mathcal{C}}$ from the adversary retrieve (P_s, x) from memory and do the following:
  1. Run $(x_1, \ldots, x_n) \leftarrow \text{Share}_t(x, \{\overline{x}_i\}_{i\in\mathcal{C}})$.
  2. Send x_j to each party $P_j \in \mathcal{H}$.

The functionality $\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$ can be instantiated by the following protocol, which makes use of the $\mathcal{F}_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$ functionality from Section 4.3.

Protocol $\Pi_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$

Input: Party P_s has input $x \in \mathbb{Z}/2^k\mathbb{Z}$
Output: The parties get t-consistent shares $[\![x]\!]_t$.
Functionalities: $\mathcal{F}_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$.
Protocol: The parties proceed as follows:

1. The parties call $\mathcal{F}_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$ to get $[\![r]\!]_t = (r_1, \dots, r_n)$, with $r \in \mathbb{Z}/2^k\mathbb{Z}$.
2. The parties send their shares of $[\![r]\!]_t$ to P_s.
3. P_s, upon receiving shares $(r_1, \dots, r_n)$, executes $r \leftarrow \text{RecSecret}_t(r_1, \dots, r_n)$.[a]
4. P_s broadcasts $a = x - r$ to all the parties.
5. Upon receiving a from the broadcast channel, the parties do the following:
   - If $a \notin \mathcal{A}$, then the parties abort.
   - Else, they compute $[\![x]\!]_t = a + [\![r]\!]_t$.

[a] The only difference with respect to the protocol from Section 3.3.4 is that, in that protocol, the adversary cannot cause an abort in this step, which is not the case here.

The following theorem is proven in a similar way as Theorem 3.14, by simply adapting its proof from $t < n/3$ to the case $t < n/2$.

Theorem 4.9. *Protocol $\Pi_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$ instantiates the functionality $\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$ in the $\mathcal{F}_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$-hybrid model with perfect security against an active adversary corrupting $t < n/2$ parties.*

Communication complexity of $\Pi_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$. The cost amounts to one call to $\Pi_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$, which are $\approx 2n$ elements in $\mathcal{R}$, and each party sending one element to the sender, which adds n elements. Then there is one call to the broadcast channel, for a total of $\approx 3n + \text{BC}_{\mathcal{R}}(1)$ elements in $\mathcal{R}$.

4.7 Final MPC Protocol

Finally, we put together all the pieces from the previous sections to instantiate $\mathcal{F}_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$. The protocol is identical to the one from Section 3.3.5, except that in our case we must account for the fact that multiplications may be computed incorrectly, so the functionality $\mathcal{F}_{\text{MultCheck}}$ must be called to ensure this is not the case.

As in Chapter 3, our ultimate goal is to securely compute an arithmetic circuit $F : (\mathbb{Z}/2^k\mathbb{Z})^{I_F} \to (\mathbb{Z}/2^k\mathbb{Z})^{O_F}$. For this, first we consider the same circuit as an arithmetic circuit over $\mathcal{R}$, namely $F' : \mathcal{R}^{I_F} \to \mathcal{R}^{O_F}$. By restricting inputs to be in $\mathbb{Z}/2^k\mathbb{Z}$, we obtain a secure computation protocol for F.

Protocol $\Pi_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$

Input: The parties have inputs in $\mathbb{Z}/2^k\mathbb{Z}$ for F.
Output: The parties learn the evaluation of F on these inputs.
Functionalities: $\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$, $\mathcal{F}_{\text{Mult}}$, $\mathcal{F}_{\text{MultCheck}}$, $\mathcal{F}_{\text{PublicRec}}(t)$
Protocol: The parties proceed as follows

1. For each $i \in [n]$ and each input $x \in \mathcal{A}$ held by P_i, the parties call $\mathcal{F}_{\text{Input}}(\mathcal{A})$ to get $[\![x]\!]_t$.
2. - For every addition operation with inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties locally compute $[\![x+y]\!]_t \leftarrow [\![x]\!]_t + [\![y]\!]_t$.
   - For every multiplication operation with inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties call $\mathcal{F}_{\text{Mult}}$ to get $[\![xy]\!]_t$.
3. For every multiplication gate in the circuit with inputs $([\![a]\!]_t, [\![b]\!]_t)$ and output $[\![c]\!]_t$, the parties call $\mathcal{F}_{\text{MultCheck}}$ to ensure that $c = ab$.
4. For each secret-shared output value $[\![z]\!]_t$, and if the parties did not abort in the previous verification stage, the parties call $\mathcal{F}_{\text{PublicRec}}(t)$ to learn z.

The following theorem is a direct consequence of the previous results.

Theorem 4.10. *Protocol $\Pi_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$ instantiates the functionality $\mathcal{F}_{\text{MPC}}(\mathbb{Z}/2^k\mathbb{Z})$ in the $(\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z}), \mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{MultCheck}}, \mathcal{F}_{\text{PublicRec}}(t))$-hybrid model with perfect[3] security against an active adversary corrupting $t < n/2$ parties.*

Communication complexity of $\Pi_{\text{MPC}}(\mathcal{A})$. Let I_F, O_F and M_F be the number of input, output and multiplication gates of the arithmetic circuit F, respectively. Putting together the complexity analysis from each subsection above, we obtain that the total communication complexity required to securely compute F is, in terms of elements of $\mathcal{R}$,

$$I_F \cdot (3n + \text{BC}_{\mathcal{R}}(1)) + M_F \cdot (6n + 6n\log(2M_F + 1)) + O_F \cdot (4n),$$

plus some terms that depend linearly on n and κ, and are independent of I_F, M_F or O_F. Finally, the observations regarding communication complexity at the end of Section 3.3.5 are also relevant in this section.

[3] Failure probability, or, in other words, statistical security, appears directly in the instantiations of $\mathcal{F}_{\text{CheckMult}}$ from Sections 4.4 and 4.5, and also in the instantiation of $\mathcal{F}_{\text{Rand}}(\mathbb{Z}/2^k\mathbb{Z})$ from Section 4.3, which is needed to instantiate $\mathcal{F}_{\text{Input}}(\mathbb{Z}/2^k\mathbb{Z})$.

Chapter 5

MPC over $\mathbb{Z}/2^k\mathbb{Z}$ for a Small Number of Parties

In Chapters 3 and 4 we presented actively secure protocols for securely computing any arithmetic circuit over $\mathbb{Z}/2^k\mathbb{Z}$, assuming the adversary corrupts t parties with $t < n/3$ and $t < n/2$ respectively. Furthermore, these protocols achieved strong security guarantees, namely information-theoretic security, where no assumptions on the hardness of different computational problems are made. In this chapter we explore protocols over $\mathbb{Z}/2^k\mathbb{Z}$ for the concrete cases where $t = 1, n = 4$, which falls within the bound $t < n/3$, and $t = 1, n = 3$, which falls within $t < n/2$. The motivation for this is that, even though one could use the protocols from Chapters 3 and 4 to obtain concrete protocols in these settings, respectively, it is typically the case that by tailoring the construction to a specific number of parties, much more efficient and simpler protocols can be designed. Furthermore, by introducing very mild computational assumptions, like the existence of cryptographic hash functions and pseudo-random functions, considerable performance improvements can be obtained.

This chapter has a different focus than the previous ones, and the level of formalism here will be significantly lower. This will be reflected mostly in the different security proofs, where we will not describe the simulators in full detail as done before.

5.1 Outsourced Secure Computation

Considering multiparty computation for a small number of parties is a well-motivated task. First, many relevant applications of multiparty computation only involve input from a few parties. This is the case, for example, in the setting in which one party holds, say, a sensitive medical image (perhaps an X-ray), and another party holds a machine learning model that can be used to analyze this image, which could be of sensitive nature due to the resources spent in obtaining such algorithm.[1] Another example includes internet voting within a small number of participants, like a board of directors, or games and interactions over the internet like poker and auctions.

[1] This example in particular considers two parties, which corresponds to the dishonest majority setting, whereas the protocols in this chapter consider honest majority computation for a small number of parties.

It can be argued, however, that the most relevant reason to study secret-sharing-based multiparty computation for a small number of parties lies in its potential to be leveraged to handle much more general settings. More precisely, even if a given application requires inputs from a lot of participants, the computation itself can be carried out by only a small number of parties, which can even be different from any of the participants providing input. Consider for example the setting in which a relatively large number of financial institutions want to use their joint data to train a machine learning classifier for fraud detection. It is a well-known fact that, *generally speaking*, having more training data available is likely to result in a better classifier, so, the more financial institutions provide input to the computation, the better the final outcome will be. However, for multiparty computation, the more parties involved in the execution of a protocol the less efficient the computation becomes. Hence, there is high motivation in keeping the number of parties executing the MPC protocol low, while simultaneously many scenarios like the one above demand a large number of parties providing input to the computation.

The solution to this objective mismatch is to detach the set of parties providing input to the computation from the set of parties executing the MPC protocol. Recall that secret-sharing based protocol proceed by letting the parties obtain shares of the inputs to the computation, and then proceeding in a "gate-by-gate" fashion through the computation of the circuit, until shares of the different outputs are obtained, point at which the parties can reveal the output to each other. The protocols we have designed so far contain methods for allowing the parties to secret-share their inputs, but the crucial observation is that this initial step of the protocol does not need to be executed by the same parties running the rest of the computation! More precisely, a different set of participants holding inputs, which we refer to as the *clients*, can secret-share their inputs to the set of parties who will execute the rest of the MPC protocol (referred to as the *parties*, or the *servers*), in very much the same way as if these parties themselves had provided the inputs. When shares of the outputs are obtained, the parties, instead of reconstructing these values to each other, send these shares to the clients so that they can learn the result of the computation.

This approach, typically called *client-server model* or *outsourced computation model*, is likely to be more applicable in practical settings. For instance, in terms of the example from above, the different financial institutions interested in training the fraud detection classifier can secret-share their data towards a small set of servers who will execute an MPC protocol to securely compute the desired machine learning model, obtaining shares of this output, and sending these to the involved clients so that they can learn the result. This way the financial institutions do not need to go through the complex, potentially costly and error-prone process of setting up hardware and software for executing the given multiparty computation protocol, and instead, this can be delegated to a smaller set of servers optimized and thoroughly tested specifically for this task. The servers can be thought of as some service providers, perhaps charging for their computation and hence having a natural incentive to keep a healthy and reputable service. Due to this, this model is sometimes called *MPC as-a-service*, and in the context of machine learning as in the previous example, it is also called *privacy-preserving machine learning as-a-service*.

5.2 Organization of this Chapter

Motivated on the relevance of the setting of multiparty computation with a small number of parties, we describe in Section 5.3 an actively secure protocol ensuring guaranteed output delivery for $n = 4$ parties tolerating $t = 1$ corruption. Then, in Section 5.4, we introduce an actively secure protocol with abort for $n = 3$ parties also withstanding $t = 1$ corruption.

As we have already mentioned, our main motivation to study the setting of multiparty computation with a small number of parties lies in terms of its practical value. In order to keep the exposition of the protocols as simple as possible, we will be much more lax in terms of the presentation and formalism when compared to the contents from Chapters 3 and 4. In particular, we will generally *not* introduce functionalities, and we will also not provide full simulation-based proofs. However, the statements and proofs we provide should be enough to convince the reader of the security properties of our protocols, and it is not hard, though it is cumbersome, to turn these into formal simulation-based arguments after having defined the appropriate functionalities.

Finally, for the sake of simplicity we assume that the function to be computed contains only one single output that is intended to be learned by all the clients. This is not a real restriction and our techniques can be easily adapted to accommodate other more elaborate output configurations.

About two-party computation. As mentioned above, we only consider honest majority computation in this chapter. In particular, this thesis does not address the highly relevant case of $n = 2$ and $t = 1$, which finds applications in a lot of different scenarios. The main motivation for this is twofold. First, the general case of dishonest majority computation over $\mathbb{Z}/2^k\mathbb{Z}$ is studied in Chapter 6, and, similarly to the case of multiparty computation over fields, there are no relevant optimizations nor ad-hoc constructions in the literature specifically tailored to the case of two parties, that perform noticeable better than simply instantiating the generic multiparty constructions with $n = 2$.

Secondly, as it has been explained already in several occasions, the dishonest majority setting incurs in a much larger overhead with respect to honest majority multiparty computation, given the need of computationally expensive techniques such as these arising from the domain of public-key cryptography. As a result, although contradictory at first glance, fixing $t = 1$, secure multiparty computation with $n = 3$ and $n = 4$ is considerably more efficient than the case with $n = 2$, in spite of involving more parties. This increase in efficiency comes with a detriment in security, however, since in either setting the adversary breaks the system by controlling at least two parties, and the more parties involved the "easier" it becomes fulfilling this requirement.

5.3 Four Parties and One Corruption

We begin by describing in this section a secure computation protocol for 4 parties tolerating one active corruption. Furthermore, the protocol achieves guaranteed output delivery. This protocol was introduced in the original work of [36], presented by the author of this thesis at USENIX'21. The protocol is built around the traditional secret-sharing scheme called Replicated Secret-Sharing, which enables the distribution of a secret among several participants, while being able to choose which subsets of parties are not allowed to learn anything about the secret, and which subsets should be able to completely determine the secret. These are called *general adversarial structures*, and they generalize the basic threshold structure we study in this thesis where the subsets of parties that should not be allowed to learn the secret are all these subsets with at most t parties. Here we make use of replicated secret-sharing in the context of $t = 1$ (that is, no single party should learn anything about the secret, but any two parties can reconstruct the secret together) and $n = 4$. This scheme is described in Section 5.3.3.

Our main contribution consists in designing different subprotocols for manipulating these shares, including protocols for multiplying secret-shared values, distributing shares consistently, reconstructing outputs, and other tools like distributing messages known to more than one single party correctly, which is used as a building block for some of the other protocols. These are described in Sections 5.3.4, 5.3.5 and 5.3.6. We also make use of these tools to design protocols for instantiating several primitives such as truncation and shared bit generation, among others which are described in Section 5.3.7. As an additional contribution we introduce in Section 5.3.2 a novel approach to achieve guaranteed output delivery that achieves more meaningful privacy notion in practice, since traditional techniques to obtain guaranteed output delivery ultimately rely on providing the inputs of the computation in the clear to an honest party.

The original work of [36] includes additional contributions to the ones presented in this thesis. Most notably, our protocol is fully implemented as part of the MP-SPDZ framework [61], and we perform extensive benchmarks on different applications to the domain of privacy preserving machine learning. Furthermore, we also compare the efficiency of our protocol with respect to the comparable four-party protocol from [65], showing that, although our protocol has the same theoretical communication complexity, our protocol actually performs much better than the one from [65] for meaningful tasks. We refer the reader to [36] for the complete set of experimental results and implementation-related information.

As we have mentioned above, the results of this section are based on the original work of [36], and **most of the content below is taken verbatim from that work**.

5.3.1 Preliminaries

As we already know, in the context of $t < n/3$, which is the setting in which the values $n = 4$ and $t = 1$ fall, it is possible to design perfectly and actively secure protocols, and in Chapter 3 we actually described one of such protocols. However, we can obtain great ef-

ficiency gains by removing the requirement of obtaining perfect security, and instead, we can settle for computational security by introducing certain cryptographic constructions that can help in saving in communication. These tools are pseudo-random functions and cryptographic hash functions, which are described next. We remark that, although the resulting protocol that makes use of these tools would not be secure against an arbitrarily powerful adversary anymore (since an attacker with enough computational resources could in principle break these security assumptions), this is not a practical concern since in reality these primitives would be instantiated with well-studied constructions that are widely believed to be secure against realistic adversaries, while still providing great efficiency.

5.3.1.1 Pseudo-Random Functions

We assume the existence of a pseudo-random function (or PRF for short) $\mathsf{PRF}_\mathsf{k} : \{0,1\}^{\ell_1} \mapsto \mathbb{Z}/2^k\mathbb{Z}$, which is a family of functions indexed by a key $\mathsf{k} \in \{0,1\}^{\ell_0}$. For a formal treatment on pseudo-random functions we refer the reader to [60]. For the purpose of our exposition it suffices to have the following rather informal definition.

Definition 5.1 (PRF—informal)**.** *The function* $\mathsf{PRF}. : \{0,1\}^{\ell_1} \mapsto \mathbb{Z}/2^k\mathbb{Z}$ *is a* pseudo-random function *if, for a uniformly random key* $\mathsf{k} \in \{0,1\}^{\ell_0}$*, no efficient adversary lacking knowledge of this key and having oracle access to the function* $\mathsf{PRF}_\mathsf{k}(\cdot)$ *can distinguish this mapping from a uniformly random function* $\{0,1\}^{\ell_1} \mapsto \mathbb{Z}/2^k\mathbb{Z}$ *with better-than-negligible probability.*

Remark 5.1. *In some of our protocols the output of the PRF will be assumed to lie in a different set than* $\mathbb{Z}/2^k\mathbb{Z}$*. For example, this is the case in the protocol from Section 5.3.7.4, where the PRF is assumed to output integers in* $\mathbb{Z}/2^k\mathbb{Z}$ *of bounded bit-length. In these cases, it is mentioned in the corresponding protocol description where the output is assumed to be.*

5.3.1.2 Cryptographic Hash Functions

We also assume the parties sample function $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^{\ell_2}$[2] from a collision-resistant hash function family. For a formal treatment on these functions we again refer the reader to [60]. For the purpose of our exposition it suffices to have the following informal definition.

Definition 5.2 (Cryptographic hash function—informal)**.** *A family of functions* $\{0,1\}^* \mapsto \{0,1\}^{\ell_2}$ *is a* collision-resistant family *if, for a function* H *sampled uniformly at random from this family, no efficient adversary can find* $x_1 \neq x_2$ *such that* $\mathsf{H}(x_1) = \mathsf{H}(x_2)$.

[2] Formally the domain of the function has to be bounded but we omit this technicality.

5.3.1.3 Assumed Setup

In several of our subprotocols we will need to assume that different subsets of the four parties have access to a common uniformly random key, that will be used to seed the pseudo-random function from above. This will be made clear in every relevant protocol. This is a one-time setup that is reused for any number of subsequent secure computation executions.

5.3.1.4 Multicast Channel

As in all the previous protocols we have described in this work, we assume the parties have access to a broadcast channel, which, in the case of $t = 1$ and $n = 4$, can be realized with a perfectly secure protocol given that $t < n/3$. However, for our protocol we will need a *multicast channel*, which acts in the same way as the broadcast channel except that only a subset of the parties is involved (that is, only a subset of the parties receive the *same* value distributed by the sender).

5.3.2 Achieving Guaranteed Output Delivery

As in Section 3.4, the general strategy to achieve guaranteed output delivery is to allow the parties, every time an abort takes place, to identify a pair of parties $\{P_i, P_j\}$ where at least one of them is corrupt. Such pair is called a *semi-corrupt pair*. Once such pair has been detected, even though it is not known which of these two parties is corrupted, it can be concluded that the other two parties outside the pair are honest, since the adversary is assumed to corrupt at most one party. As a result, the parties can simply provide their inputs to any of these two identified honest parties, who can carry out the computation locally and announce the outputs. This is secure according to our security definition, given that the only requirement is that the adversary does not learn anything about the honest parties' inputs.

In some of the protocols not only a semi-corrupt pair is found, but actually the actively corrupt party is identified. The same reasoning as above applies. To accommodate for these cases, we say that $\{P_i, P_j\}$ is a semi-corrupt pair if either P_i or P_j is corrupt, and we allow for $i = j$, which corresponds to the case in which the actively corrupt party has been identified.

Private Robustness: A More Meaningful Notion of Guaranteed Output Delivery

In the original work of [36], once a semi-corrupt pair has been identified, a different approach to achieve guaranteed output delivery is proposed. The main issue with the approach outlined above, that is, entrusting the computation to one of the honest parties outside the semi-corrupt pair, is that this honest party gets to see all the sensitive information in the clear. As mentioned before, formally speaking, this is not a problem since

the security definition only requires that the adversary, who is assumed to corrupt one single party, cannot learn anything about the honest parties' inputs. By delegating the computation to one party that is guaranteed to not be the corrupt party, the adversary cannot learn the private inputs of the honest parties, and security is ensured.

However, this approach is an example of a mismatch between a theoretical model and its meaning in practice. For example, consider the four servers to be large companies offering an outsourced secure computation service, such as Apple, Facebook, Google and Microsoft. Notice that the whole purpose of having such service is guaranteeing users that their data will never rest in one of these single parties, so, as long as no two of these companies are willing to share the data to each other, user data will remain private. This is reasonable to expect given that the companies, which are already diverse enough, share a strong incentive to keep a healthy service. Say the parties make use of a four-party protocol tolerating one active corruption, such as ours, so that, even if one of the four parties deviates arbitrarily from the protocol, privacy of client data is preserved. If a semi-corrupt pair is detected, a traditional protocol with guaranteed output delivery would provide all client data *in the clear* to one of the remaining parties. From the point of view of a client using the service, this is completely unacceptable: if they trusted any of these parties with their data initially then there would have been no need to execute a multiparty protocol to begin with. Tagging one of these parties as "trusted", simply because it did not cheat in a given protocol execution, is counterintuitive and insufficient in practice.

Given the above, an alternative method to obtain guaranteed output delivery once a semi-corrupt pair is identified is proposed in [36]. This consists of the following. Let $\{P_i, P_j\}$ be a semi-corrupt pair with $i < j$, and let $\{u, v\} = [4] \setminus \{i, j\}$. The parties P_i, P_u, P_v execute an actively secure three-party protocol *with abort* to securely compute the given function. If P_i is honest, the this protocol is guaranteed to terminate as it only involves honest parties, and the desired results will be obtained. On the other hand, it can be the case that P_i is the corrupt party, and in this case, it can happen that the execution of this three-party protocol results in abort. However, in this case, the parties can determine that P_i is the corrupt party, and at this point the two remaining parties P_u, P_v, which are honest, can execute a passively secure protocol to compute the given function.[3] This ensures that the sensitive information never resides in one single party, even if this party has been identified as honest.

The notion achieved above, called *private robustness* in [36], is much more meaningful in practice given that it guarantees termination of the protocol while avoiding relying on one single party computing the given function in the clear. Intuitively, this method ensures that no single party, honest or corrupt, gets to see sensitive data on its own. However, we remark that a more formal study of this model is not achieved in [36], and it is left as future work. The biggest issue with formalizing this notion is that, technically, a corrupt party may deliberately send its own state to other honest party, giving this party the possibility of holding sensitive data in the clear by joining the received state with its own internal state. Although this may seem a bit artificial, and heuristically it can be

[3] The reader may wonder why was the actively secure three-party protocol executed, given that P_u and P_v were already identified as honest parties when the semi-corrupt pair was found. The reason for this is that, although it may look counter-intuitive, actively secure three-party protocols with abort (tolerating one corruption) such as the one we will describe in Section 5.4 are much more efficient than passively secure two-party protocols.

disallowed by asking parties to reject undesired incoming information, this imposes a problem at the moment of studying the notion of private robustness formally.

A formal study of the intuitive idea of "not leaking sensitive data even to honest parties" is done in [6]. In that work, feasibility results are presented, approaching formally the issue highlighted above of honest parties receiving undesired messages that might allow them to reconstruct sensitive information.

5.3.3 Replicated Secret-Sharing for Four Parties

As we have mentioned already, although we could make use of Shamir secret-sharing as explained in Section 3.1, the main tool for the protocol from this section is a different type of secret-sharing scheme, called Replicated Secret-Sharing, which was introduced initially in [59]. This scheme, like Shamir secret-sharing, can be used for an arbitrary number of parties n, although, unlike Shamir's, the size of the share held by each party grows exponentially with n, which makes it impractical for large values of n. However, for $n = 4$ and $t = 1$, this scheme is highly practical and in fact it outperforms Shamir secret-sharing given that, as we will see, it does not require any Galois ring extension for it to work. The scheme works as follows.

Definition 5.3 (Sharing Procedure)**.** *Let $s \in \mathbb{Z}/2^k\mathbb{Z}$. We define* Share$(s)$ *as follows.*

- *Sample $s_1, s_2, s_3, s_4 \in_R \mathbb{Z}/2^k\mathbb{Z}$ uniformly at random subject to $s = s_1 + s_2 + s_3 + s_4 \bmod 2^k$.*[4]
- *Output $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$, where $\mathbf{s}_i = \{s_j : j \neq i\}$ for $i \in [4]$.*

As before, when each party P_i has $\mathbf{s}_i$ for $i \in [4]$, we denote this by $[\![s]\!]$. Furthermore, whenever we consider a secret-shared value $[\![x]\!]$, we use the subindexed values x_1, x_2, x_3, x_4 to denote the additive shares such that $x = x_1 + x_2 + x_3 + x_4$, and we use the typewriter font $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4$ to denote the shares held by each party, that is, $\mathtt{x}_i = (x_j : j \neq i)$.

Given two shared values $[\![x]\!]$ and $[\![y]\!]$, the parties can locally obtain shares of $[\![x \pm y]\!]$ by performing the respective operation on their shares. This extends to multiplication by a publicly known value. Additionally, given a value $c \in \mathbb{Z}/2^k\mathbb{Z}$ known by all parties, the parties can obtain shares $[\![c]\!]$ in a canonical way by considering $c = c_1 + c_2 + c_3 + c_4$, where $c_1 = c$ and $c_2 = c_3 = c_4 = 0$. This enables the parties to locally add/subtract the value c to a given shared value $[\![x]\!]$ to obtain $[\![x \pm c]\!]$ by first obtaining shares $[\![c]\!]$ and then proceeding as above.

Now, given only one share $\mathbf{s}_i$, the uniformly random value s_i is missing and therefore nothing is leaked about the secret s, or, in other words, $\mathbf{s}_i$ follows the uniform distribution in $(\mathbb{Z}/2^k\mathbb{Z})^3$. On the other had, any pair of shares $\mathbf{s}_i, \mathbf{s}_j$ is enough to reconstruct the secret since these together contain all the values s_1, s_2, s_3, s_4.

[4] For the rest of this chapter we will omit the $\bmod 2^k$ notation.

We now define the notion of consistent sharings, analogous to Definition 3.3 in Section 3.1.1 for the case of Shamir secret-sharing. Intuitively, the parties have consistent sharings if the correct "replication" holds among the data they possess. This is formalized in the following definition.

Definition 5.4. *Let* $\mathbf{s}_i = (s_j^{(i)} : j \in [4]\backslash\{i\}) \in (\mathbb{Z}/2^k\mathbb{Z})^3$ *for* $i \in [4]$. *We say that* $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$ *is* consistent, *and in the case that each party* P_i *holds* $\mathbf{s}_i$ *we say the parties* hold consistent shares, *if, for every* $i, j, u \in [4]$ *and* $v \in [4] \setminus \{i, j, u\}$, *it holds that* $s_v := s_v^{(i)} = s_v^{(j)} = s_v^{(u)}$. *In this case,* $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$ *constitutes sharings of* $s := s_1 + s_2 + s_3 + s_4$.

5.3.3.1 Public Reconstruction

Let $\{i, j, u, v\} = [4]$. As pointed out before, given any two shares $\mathbf{s}_i = (s_j, s_u, s_v)$ and $\mathbf{s}_j = (s_i, s_u, s_v)$, the secret s can be reconstructed by computing $s = s_i + s_j + s_u + s_v$. However, in our context each share $\mathbf{s}_i$ is held by party P_i, and in the case that, say, P_i is corrupt, the share $\mathbf{s}_i$ may be adversarially modified as $\mathbf{s}_i' = (s_j', s_u, s_v)$, with $s_j' = s_j + \delta$ for some $\delta \in \mathbb{Z}/2^k\mathbb{Z}$ chosen by the adversary. In this case, the reconstructed secret would be $s_i + s_j' + s_u + s_v = s + \delta$, which means the adversary managed to modify the secret.

Fortunately, if all of the four shares $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4$ are used for reconstruction, it is possible to detect which of the four announced vectors is incorrect, and reconstruct the correct secret from this. This must be put in contrast to the results from Section 3.1.3, where, in cases where there are enough shares, the underlying secret of Shamir secret-sharing could be *error corrected*. Recall from Section 3.1.3 that error correction is only possible if $t < (n - d)/2$, where d is the degree used for Shamir secret-sharing. In our current setting this is still true, with the degree replaced by the threshold used in the secret-sharing scheme, which is 1 in our case. We see that for $t = 1$ and $n = 4$ the bound above $1 < (4 - 1)/2 = 1.5$ holds, which is why error correction is possible. As we will see in Section 5.4 when we use Replicated secret-sharing for $t = 1$ and $n = 3$, this bound does not hold so error correction will not be possible, but error detection, which requires $t < (n - d)$, is actually possible.

We proceed to describe the method to perform error correction on some given announced shares, where one of them can be incorrect, to identify the incorrect share and therefore recover the correct secret.

Definition 5.5. *Let* $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$ *be a consistent vector, and let* $\mathbf{s}_i' \in (\mathbb{Z}/2^k\mathbb{Z})^3$ *such that* $\mathbf{s}_i = \mathbf{s}_i'$ *for all but possibly one* $i \in [4]$. *We define the method* $\mathrm{RecSecret}(\mathbf{s}_1', \mathbf{s}_2', \mathbf{s}_3', \mathbf{s}_4')$ *as follows:*

1. *Write* $\mathbf{s}_i' = (s_j^{(i)} : j \in [4] \setminus \{i\})$.
2. *For* $j \in [4]$, *let* s_j *be the majority value in* $(s_j^{(i)} : i \in [4] \setminus \{j\})$.
3. *Output* $s = s_1 + s_2 + s_3 + s_4$.

To see that the algorithm RecSecret works as intended, let i_0 be the only index for which (possibly) $\mathbf{s}'_{i_0} \neq \mathbf{s}_{i_0}$. Then, from the definition of consistency, we have that for every $j \in [4]$, $s_j^{(i)} = s_j^{(i')}$ for every $i, i' \in [4] \setminus \{i_0, j\}$. In particular, the majority value in $(s_j^{(i)} : i \in [4] \setminus \{j\})$ will be $s_j^{(i)}$ for $i \in [4] \setminus \{i_0, j\}$, which is the same value held in the unmodified $\mathbf{s}_i$. Hence, the reconstructed value is the same secret underlying the original consistent shares $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$.

5.3.3.2 Dealing Consistent Shares

Just like with Shamir secret-sharing, not every vector $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$ is consistent, as it has to satisfy certain special conditions to fulfill this definition. All of our subprotocols that produce sharings of some kind, like for example the protocol from Section 5.3.6 that produces shares of a product given shares of the two factors, return consistent shares assuming that the input shares are consistent to begin with. This invariant of holding consistent shares is then preserved throughout the circuit computation, which ensures that the sharings of the different outputs are also consistent, and allows the intended receiver to perform error correction as described in Definition 5.5

The first sharings the parties hold will be these distributed by the clients in the input phase. As a result, to preserve the invariant that all sharings are consistent, we must ensure that the inputs secret-shared by the different clients are themselves consistent. Since an actively corrupt client may send arbitrary shares to the parties, in particular inconsistent ones, the parties must execute a small subprotocol to verify the consistency of the received shares. This is described next.

Protocol $\Pi_{\text{Consistency}}$

Input: For each $i \in [4]$, party P_i receives $\mathbf{s}_i = (s_j^{(i)} : j \in [4] \setminus \{i\})$ from a given client with ID id.
Output: A flag signaling whether $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4)$ is consistent
Protocol: The parties proceed as follows.

1. For every $i \in [4]$: every P_j for $j \in [4] \setminus \{i\}$ multicasts $s_i^{(j)}$ to $\{P_u : u \in [4] \setminus \{i\}\}$.
2. For every $i \in [4]$: every P_j for $j \in [4] \setminus \{i\}$ updates $s_i^{(j)}$ to be the majority value among the multicasted values $\{s_i^{(u)} : u \in [4] \setminus \{i\}\}$. If no majority exists, then P_j broadcasts $(\text{BadClient}, \text{id}, i)$ to all the parties
3. If two messages from two different parties of the form $(\text{BadClient}, \text{id}, i)$, for the same i, are sent in the multicast channel, then the parties set all their shares to 0 (which constitute trivial consistent sharings of the value 0).

Proposition 5.1. *At the end of the execution of Protocol* $\Pi_{\text{Consistency}}$ *the parties output consistent shares. Furthermore, if the client is honest, then these shares are the ones distributed by the client initially.*

Proof. If the condition in the final step is satisfied then it is obvious that the parties output consistent shares. If, on the other hand, this does not hold, it is because for every

$i \in [4]$, there was a majority among the multicasted values $\{s_i^{(u)} : u \in [4] \setminus \{i\}\}$—indeed, if this was not the case, then the at-least-two honest parties with indexes in $[4] \setminus \{i\}$ would have broadcasted $(\mathsf{BadClient}, \mathsf{id}, i)$—so the parties in $[4] \setminus \{i\}$ all set their value $s_i^{(u)}$ to be the same. Furthermore, if the client is honest, then it cannot happen that there is no strict majority among the multicasted values $\{s_i^{(u)} : u \in [4] \setminus \{i\}\}$, since, even though a corrupt party P_u may multicast a different value than the one it received, the two remaining honest parties in $[4] \setminus \{i\}$ will multicast the values they received from the client, which are the same. □

Remark 5.2 (Optimizing the verification)**.** *One can optimize Protocol* $\Pi_{\mathsf{Consistency}}$ *to save in communication by requiring the parties to exchange digests* $\mathsf{H}(s_i^{(j)})$ *instead of the actual values* $s_i^{(j)}$*, and, in case that a party sees it needs to update its value to a different one, it can ask the other parties to run the original protocol without the hashes. This way, optimistically, the parties would only send digests to each other, which can be considerably shorter than the actual values. This is particularly the case if the same client is providing as input several values, since all the shares corresponding to these values can be concatenated together before hashing.*

5.3.4 Joint Message Passing

We begin by describing a subprotocol that enables a pair of parties, both knowing a value in common, to communicate this value to another party while ensuring that either this party receives the correct value held by both parties, or the execution ends in a semi-corrupt pair being identified. This subprotocol is used as a building block in our main secure computation protocol. More specifically, it is used in the Π_{Input2} subprotocol from Section 5.3.5.2, which in turn is used for the multiplication protocol from Section 5.3.6.

The idea behind the protocol, named Π_{JMP}, from Joint Message Passing, is quite simple: both parties knowing the common value will send this to the intended receiver, who accepts the transmission if the two values coincide. Else, the receiver announces that these two values do not match, and two senders have a chance to "defend themselves". This process eventually resolves in a semi-corrupt pair, depending on which parties try to defend themselves and how. Finally, for efficiency purposes, only one sender is required to send the message, while the other sender can simply send a digest of the intended message, which is sufficient for the purpose of verifying consistency of the two values. This is particularly useful when there a lot of messages being transmitted simultaneously, which as we will see is the case if the circuit being computed is very wide given that Π_{JMP} will be used in every multiplication gate.

Protocol $\Pi_{\mathsf{JMP}}(x, P_i, P_j, P_u)$

Input: Parties P_i and P_j share a common input x lying in some set.
Output: Party P_u learns x
Protocol: The parties proceed as follows:

1. P_i sends $m_i = x$ to P_u

2. P_j sends $m_j = \mathrm{H}(x)$ to P_u
3. P_u checks if $m_j = \mathrm{H}(m_i)$. If this holds, then P_u outputs m_i, and the protocol ends.
4. Else, P_u multicasts (accuse, P_i, P_j, m_i', m_j') to $\{P_i, P_j, P_u\}$, where $m_i' = m_j$ and $m_j' = m_j$ for an honest P_u, but a corrupt P_u may change these values.
5. If $\mathrm{H}(m_i') = m_j'$, then the parties output the set $\{P_u\}$
6. If $m_i' \neq m_i$, then P_i multicasts (accuse, P_u) to $\{P_i, P_j, P_u\}$.
7. If $m_j' \neq m_j$, then P_j multicasts (accuse, P_u) to $\{P_i, P_j, P_u\}$
8. The parties agree on a semi-corrupt pair as follows:
   - If only P_i multicasts (accuse, P_u), then the parties output the set $\{P_i, P_u\}$.
   - If only P_j multicasts (accuse, P_u), then the parties output the set $\{P_j, P_u\}$.
   - If both P_i and P_j multicasts (accuse, P_u), then the parties output the set $\{P_u\}$.
   - If none of P_i or P_j multicasts (accuse, P_u), then the parties output the set $\{P_i, P_j\}$.
9. The parties in $\{P_i, P_j, P_u\}$ send the identified semi-corrupt pair to P_v, where $\{v\} \in [4] \setminus \{i, j, u\}$. P_v outputs the set sent by at least two parties.

Proposition 5.2. *At the end of the execution of $\Pi_{\mathrm{JMP}}(x, P_i, P_j, P_u)$, either (1) P_u learns x correctly, or (2) the parties output a semi-corrupt pair. Either case, P_v does not learn anything about x.*

Proof. If the values m_i and m_j received by P_u satisfy $\mathrm{H}(m_i) = m_j$, then it must be the case that $m_i = x$. This is because either P_i or P_j is honest, which means that either $m_i = x$ or $m_j = \mathrm{H}(x)$. In the first case the claim is trivially true, and in the second, since $\mathrm{H}(x) = \mathrm{m_i}$, we cannot have $m_i \neq x$ or else a collision in the hash function H would have been found. This would be formalized by considering an adversary for the hash function challenge who runs the environment internally: if this environment can distinguish the ideal from the real world executions (having a properly defined functionality and simulator), then it can only be because such collision was found.

Now we show that if P_u multicasts (accuse, P_i, P_j, m_i', m_j') to $\{P_i, P_j, P_u\}$, then the output produced by the parties is indeed a semi-corrupt pair. Let us analyze the case in which P_u is honest first. In this case, $m_i' = m_i$ and $m_j' = m_j$, and these values satisfy $\mathrm{H}(m_i) \neq m_j$, which can only happen if either P_i or P_j is corrupt. At this point, if P_i is honest, it will not multicast (accuse, P_u) since $m_i = m_i'$. Similarly, if P_j is honest, it will not multicast (accuse, P_u) since $m_j = m_j'$. As a result, the only possibility is that either the corrupt party multicasts (accuse, P_u), in which case this party together with P_u are output as a semi-corrupt pair, or that no party among P_i and P_j multicasts (accuse, P_u), in which case the semi-corrupt pair produced is $\{P_i, P_j\}$. Either case, the pair contains the corrupt party.

In the case in which P_u is corrupt, we see that the only way in which a semi-corrupt pair not containing P_u can be produced is that none of P_i or P_j multicasts (accuse, P_u), which can only happen if $m_i' = m_i$ and $m_j' = m_j$. Since P_i and P_j are honest, these values satisfy $m_j = \mathrm{H}(m_i)$, so it holds that $m_j' = \mathrm{H}(m_i')$, which, by the protocol instructions, produces the corrupt party $\{P_u\}$. □

Communication complexity of Π_{JMP} The communication complexity of Π_{JMP} in the optimistic case in which no semi-corrupt pair is identified consists of 1 element in $\mathbb{Z}/2^k\mathbb{Z}$, plus one hash. Hence, the amortized communication complexity when many calls to this protocol are made is of only 1 element in $\mathbb{Z}/2^k\mathbb{Z}$.

5.3.5 Secret-Sharing Joint Inputs

We described in Section 5.3.3.2 a method by which the parties can verify that the inputs secret-shared by the clients are consistently distributed. Additionally, in some parts of our protocol the parties themselves will need to distribute inputs to the other parties, which can be done by using the aforementioned consistency check. However, in some of these cases, the inputs to be distributed are not only known to one single party, but sometimes they are known to two, or even three of the parties. In these cases, providing input and verifying its consistency becomes a much simpler task, which we describe next. Below we let $\{i, j, u, v\} = [4]$.

5.3.5.1 Input Known by Three Parties

This is the simplest of the settings in which the input to be secret-shared is held by three parties. In this case, as described in the protocol below, the parties can obtain shares *non-interactively*.

Protocol Π_{Input3}

Input: Parties P_i, P_j, P_u have common input x
Output: The parties obtain consistent shares $[\![x]\!]$
Protocol: Let $x_v = 0$ for $v \in \{i, j, u\}$, and $x_v = x$ for $v \in [4] \setminus \{i, j, u\}$. Each party P_ℓ for $\ell \in [4]$ outputs $\mathbf{x}_\ell = \{x_h : h \neq \ell\}$

The following is trivial to see.

Proposition 5.3. *The output of Π_{Input3} are consistent shares $[\![x]\!]$, where the view of P_v is independent of the input x*

5.3.5.2 Input Known by Two Parties

When the input to be secret-shared is known by two parties the parties can obtain consistent shares with the help of some pre-shared setup. This consists of, for each $u \in [4]$, the parties in $[4] \setminus \{u\}$ having a shared random key k_u. With this at hand, the parties can obtain shares of the given input by simply executing one call to Π_{JMP}, which is used to help one party obtain its missing share. Details are provided below.

Protocol Π_{Input2}

Input: Parties P_i, P_j have common input x
Output: The parties obtain consistent shares $[\![x]\!]$
Setup: P_i, P_j, P_u have a common random key k_v.
Protocol: The parties proceed as follows

1. Let id be a fresh common ID for the given protocol call.
   - $x_v = \mathsf{PRF}_{\mathsf{k}_v}(\mathsf{id})$ (known to P_i, P_j and P_u)
   - $x_i = x_j = 0$ (trivially known to all parties)
   - $x_u = x - x_v$ (known to P_i and P_j)
2. The parties call $\Pi_{\text{JMP}}(x_u, P_i, P_j, P_v)$ so that P_v obtains x_u, or the parties output a semi-corrupt pair.
3. If no semi-corrupt pair is produced in the previous step, each party P_ℓ for $\ell \in [4]$ outputs $\mathbf{x}_\ell = \{x_h : h \neq \ell\}$. Else, the parties output the semi-corrupt pair.

Proposition 5.4. *After the execution of $\Pi_{\text{Input2}}(x, P_i, P_j, P_u)$, either (1) the parties have consistent shares $[\![x]\!]$, or (2) the parties output a semi-corrupt pair. Either case, P_u and P_v learn nothing about x.*

Proof. The only messages transmitted are sent through the call to $\Pi_{\text{JMP}}(x_u, P_i, P_j, P_v)$, in which P_i and P_j send their common value x_u to P_v. From Proposition 5.2, this call results in either P_v learning x_u, or the parties identify a semi-corrupt pair, and in either case, P_u learns nothing about x_u, nor x.

Finally, we claim that P_v learns nothing about x, even if it learns x_u. More precisely, we claim that x_u looks indistinguishable from random to P_v. To see this, consider the uniformly random value $x'_u = x - r$, for $r \in_R \mathbb{Z}/2^k\mathbb{Z}$. P_v can only distinguish between x'_u and x_u, if it can distinguish between r and $\mathsf{PRF}_{\mathsf{k}_v}(\mathsf{id})$. However, this is not possible from the properties of the PRF, given that the key k_v is uniformly random and unknown to P_v. This would be formalized by explicitly describing an adversary for the PRF challenge that runs the environment internally. As argued above, the environment (having defined an appropriate simulator and a functionality) will only be able to distinguish the real and ideals world by distinguishing the PRF output from random, so the adversary can make use of this to solve the PRF challenge. □

Communication complexity of Π_{Input2}. This amount to one call of Π_{JMP}, which involves 1 element in $\mathbb{Z}/2^k\mathbb{Z}$.

5.3.6 Secure Multiplication

Now we make use of the protocol Π_{Input2} from Section 5.3.5.2 above to design a protocol Π_{Mult} to securely compute consistent shares $[\![xy]\!]$ from shared values $[\![x]\!]$ and $[\![y]\!]$. The protocol is described below.

Protocol Π_{Mult}

Input: Parties have consistent shares $[\![x]\!]$ and $[\![y]\!]$.
Output: The parties get consistent shares $[\![xy]\!]$, or they produce a semi-corrupt pair.
Protocol: The parties proceed as follows:

1. For every $u \in [4]$, parties call the non-interactive method $[\![x_u y_u]\!] \leftarrow \Pi_{\text{Input3}}(x_u y_u, P_i, P_j, P_v)$, where $\{i, j, v\} = [4] \setminus \{u\}$.
2. For every pair $u, v \in \{1, 2, 3, 4\}$ such that $u < v$, parties P_i and P_j with $\{i, j\} = [4] \setminus \{u, v\}$, who both know x_u, x_v, y_u and y_v, run the protocol $\Pi_{\text{Input2}}(x_u y_v + x_v y_u, P_i, P_j, P_v)$ to obtain either $[\![x_u y_v + x_v y_u]\!]$, or a semi-corrupt pair.
3. If no semi-corrupt pair was produced in the previous step, the parties locally add $[\![xy]\!] \leftarrow \sum_{u,v\in[4],u<v} [\![x_u y_v + x_v y_u]\!] + \sum_{u=1}^{4} [\![x_u y_u]\!]$ and output these sharings. Else, they output the semi-corrupt pair.

Theorem 5.1. *At the end of Protocol Π_{Mult} the parties get consistent shares $[\![xy]\!]$, and no single party learns any additional information about x or y.*

Proof. Regarding privacy, the only communication happens in the six calls to Π_{Input2}, which, from Proposition 5.4, do not leak any additional information to the parties involved. Additionally, from the same proposition, these calls either succeed or result in the identification of a semi-corrupt pair.

Finally, it remains to be seen that, in the case no semi-corrupt pair is identified, the output produced by the parties are consistent shares $[\![x \cdot y]\!]$. To see this, observe that $x = x_1 + x_2 + x_3 + x_4$ and $y = y_1 + y_2 + y_3 + y_4$, so $xy = \sum_{u,v\in[4],u<v}(x_u y_v + x_v y_u) + \sum_{u=1}^{4} x_u y_u$, which is precisely the linear combination computed by the parties in the protocol.

□

When Π_{Mult} is called on inputs $[\![x]\!]$ and $[\![y]\!]$, obtaining output $[\![z]\!]$, we denote this by $[\![z]\!] \leftarrow \Pi_{\text{Mult}}([\![x]\!], [\![y]\!])$.

Communication complexity of Π_{Mult}. This amounts to 6 calls to Π_{Input2}, which involve 6 elements in $\mathbb{Z}/2^k\mathbb{Z}$ in total. In contrast, the protocol from Chapter 3 requires the much more dramatic figure of $16n\log(2n) = 16 \cdot 4 \cdot 3 = 192$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.[5] The $\log(2n) = 3$ factor comes from the need of using a Galois ring extension of this degree to admit the construction of hyper-invertible matrices of Section 3.3.1.1. However, even without this factor, the communication complexity of the protocol based on Shamir secret-sharing is much worse than the one based on Replicated secret-sharing. This is because, in the latter setting, we exploit the fact that the parties have pre-shared keys that can help them set compute some of the shares involved in the computation without communicating to the other parties.

[5] These include the communication from the offline and online phases. The online phase only consumes $6n\log(2n)$, which is still a large number: $6 \cdot 4 \cdot 3 = 72$.

Computing dot products securely. Protocol Π_{Mult} can be easily modified to securely compute $[\![z]\!]$ from $([\![x_1]\!], \ldots, [\![x_\ell]\!])$ and $([\![y_1]\!], \ldots, [\![y_\ell]\!])$, where $z = \sum_{h=1}^{\ell} x_h y_h$, with a cost that is independent of ℓ. This is achieved by modifying Π_{Mult} as follows:

1. For every $u \in [4]$, parties call the non-interactive method $\left[\!\left[\sum_{h=1}^{\ell} x_{hu} y_{hu}\right]\!\right] \leftarrow \Pi_{\text{Input3}}(\sum_{h=1}^{\ell} x_{hu} y_{hu}, P_i, P_j, P_v)$, where $\{i, j, v\} = [4] \setminus \{u\}$.

2. For every pair $u, v \in \{1, 2, 3, 4\}$ such that $u < v$, parties P_i and P_j with $\{i, j\} = [4] \setminus \{u, v\}$, who both know x_u, x_v, y_u and y_v, run the protocol $\Pi_{\text{Input2}}(\sum_{h=1}^{\ell}(x_{hu} y_{hv} + x_{hv} y_{hu}), P_i, P_j, P_v)$ to obtain either $\left[\!\left[\sum_{h=1}^{\ell}(x_{hu} y_{hv} + x_{hv} y_{hu})\right]\!\right]$, or a semi-corrupt pair.

3. If no semi-corrupt pair was produced in the previous step, the parties locally add $\left[\!\left[\sum_{h=1}^{\ell} x_h y_h\right]\!\right] \leftarrow \sum_{u,v \in [4], u<v} \left[\!\left[\sum_{h=1}^{\ell}(x_{hu} y_{hv} + x_{hv} y_{hu})\right]\!\right] + \sum_{u=1}^{4} \left[\!\left[\sum_{h=1}^{\ell} x_{hu} y_{hu}\right]\!\right]$ and output these sharings. Else, they output the semi-corrupt pair.

Being able to securely compute dot products with a cost that is independent of the dimension is of the up-most importance for applications that make extensive use of this operation, such as these involving matrix arithmetic like in the case of deep neural networks, support vector machines, and many other machine learning models.

5.3.7 Some Primitives

Now we turn our discussion to a set of primitives that are useful when considering more advanced applications that are not expressed in a natural way as a simple combination of additions and multiplications. These are probabilistic truncation, discussed in Section 5.3.7.1 below, which is particularly useful when dealing with real-valued arithmetic, and the generation of shared random bits discussed in Section 5.3.7.2, which are a critical tool for a wide range of other primitives, and the conversion between binary and integer arithmetic, described in Section 5.3.7.3. We also present a protocol for obtaining shares of the bit-decomposition of a given shared value in Section 5.3.7.3, and finally, we discuss a protocol for generating the so-called edaBits in Section 5.3.7.4.

In this section, we augment the notation of secret-shared values $[\![s]\!]$ as $[\![s]\!]_k$, to make explicit the fact the the sharings are modulo 2^k. In particular, the notation $[\![b]\!]_1$ represent shares modulo 2 of a bit $b \in \{0, 1\}$. If the subindex is omitted then it implicitly means sharings modulo 2^k. Finally, given $s \in \mathbb{Z}/2^k\mathbb{Z}$, we denote by $(s[k-1], \ldots, s[0]) \in \{0, 1\}^k$ the bit-decomposition of s.

5.3.7.1 Probabilistic Truncation

An important primitive for a wide range of applications of secure multiparty computation consists in computing $[\![y]\!]$ from $[\![x]\!]$, where $y = \left\lfloor \frac{x}{2^m} \right\rceil$ for some $m \in [k]$. This is particularly important for applications involving real-valued arithmetic, since this type of arithmetic

is emulated by making use of fixed-point arithmetic in secure multiparty computation applications involving real numbers.

The following protocol does not compute $[\![y]\!]$ with $y = \left\lfloor \frac{x}{2^m} \right\rceil$, but rather it computes an "approximation" of this value. More precisely, the output satisfies $y = \lfloor x/2^m \rfloor + u$, where $u \in \{0, 1\}$ and u is "biased towards the right result", which means that u is more likely to be 1 (0) the closer $x/2^m$ gets to $\lceil x/2^m \rceil$ ($\lfloor x/2^m \rfloor$).

Protocol Π_{Trunc}

Input: $[\![x]\!]$ with the most significant bit of x being 0.
Setup: For $i \in \{3, 4\}$, the parties $\{P_j : j \neq i\}$ have a shared key k_i.
Output: $[\![\lfloor x/2^m \rceil]\!]$ rounded probabilistically.
Protocol:

1. Let $s_i = \mathsf{PRF}_{\mathsf{k}_i}(\mathsf{id})$ for $i \in \{3, 4\}$ and $s_i = 0$ for $i \in \{1, 2\}$. Let $r = s_1 + s_2 + s_3 + s_4$. The parties have shares $[\![r]\!]$ by defining the ℓ-th share to be $\{s_i\}_{i \neq \ell}$.
2. P_1 and P_2 compute r_{k-1} and $r' = \sum_{i=m}^{k-2} r_i \cdot 2^{i-m}$, where $r = \sum_{i=0}^{k-1} r_i \cdot 2^i$ is the bit decomposition of r. The parties call $[\![r_{k-1}]\!] \leftarrow \Pi_{\mathsf{Input2}}(r_{k-1}, P_1, P_2)$ and $[\![r']\!] \leftarrow \Pi_{\mathsf{Input2}}(r', P_1, P_2)$.
3. All parties compute $[\![c]\!] \leftarrow [\![x]\!] + [\![r]\!]$.
4. The parties call $\Pi_{\mathsf{JMP}}(c_3 + c_4, P_1, P_2, P_3)$ and $\Pi_{\mathsf{JMP}}(c_3 + c_4, P_1, P_2, P_4)$, and P_3 and P_4 reconstruct $c = \sum_{i=1}^{4} c_i$.
5. P_3 and P_4 compute $c' \leftarrow \lfloor (c \bmod 2^{k-1})/2^m \rfloor$ and $c'' = \lfloor c/2^{k-1} \rfloor$, and call $[\![c']\!] \leftarrow \Pi_{\mathsf{Input2}}(c', P_3, P_4)$ and $[\![c'']\!] \leftarrow \Pi_{\mathsf{Input2}}(c'', P_3, P_4)$.
6. All parties call $[\![r_{k-1} \cdot c'']\!] \leftarrow \Pi_{\mathsf{Mult}}([\![r_{k-1}]\!], [\![c'']\!])$ and let $[\![b]\!] \leftarrow [\![r_{k-1}]\!] \oplus [\![c'']\!] = [\![r_{k-1}]\!] + [\![c'']\!] - 2 \cdot [\![r_{k-1} \cdot c'']\!]$.
7. All parties output $[\![c']\!] - [\![r']\!] + [\![b]\!] \cdot 2^{k-m-1}$.

Cheating identification
Output the set produced by the first instance of Π_{JMP} to fail.

Theorem 5.2. *After the execution of Π_{Trunc} either the parties output a semi-corrupt pair, or the parties learn $[\![y]\!]$, where $y = \lfloor x/2^m \rfloor + u$ with $u \in \{0, 1\}$ such that $\Pr[u = 1] = (x/2^m) - \lfloor x/2^m \rfloor$. Either case, the parties do not learn anything about x.*

Proof. For the sake of the proof we make explicit the $\bmod 2^k$ notation when dealing with congruences. First, we observe that privacy is preserved throughout the computation given that the sub-primitives Π_{JMP} and Π_{Mult} are private. The only potential leakage comes from the calls to Π_{Input2}. However, this only reveals $c = x + r = x + s_3 + s_4 \bmod 2^k$ to P_3 and P_4, but since s_3 and s_4 are uniformly random and unknown to P_3 and P_4 respectively, the leakage of these calls is zero.

It remains to analyze the correctness of our construction. We begin by observing that $c = x + r - 2^k u$ as integers, where u is the potential overflow bit of adding x and r. Similarly, $(c \bmod 2^{k-1}) = (x \bmod 2^{k-1}) + (r \bmod 2^{k-1}) - 2^{k-1} v$, where v is the potential overflow bit of adding $(x \bmod 2^{k-1})$ and $(r \bmod 2^{k-1})$ modulo 2^{k-1}. Notice that, since x's most significant bit is 0, it holds that $(x \bmod 2^{k-1}) = x$ and also that $u = v \cdot r_{k-1}$. Let

$c = 2^{k-1} \cdot c'' + (c \bmod 2^{k-1})$, where $c'' = \lfloor c/2^{k-1} \rfloor$, the expressions above allow us to conclude that

$$\begin{aligned} 2^{k-1} \cdot c'' &= c - (c \bmod 2^{k-1}) \\ &= (x + r - 2^k u) - (x + (r \bmod 2^{k-1}) - 2^{k-1} v) \\ &= 2^{k-1} r_{k-1} + 2^{k-1} v - 2^k \cdot v \cdot r_{k-1} \\ &= 2^{k-1}(r_{k-1} \oplus v), \end{aligned}$$

where r_{k-1} denotes the most significant bit of r. From the above it follows that $c'' = r_{k-1} \oplus v$, or $v = c'' \oplus r_{k-1}$. This is turn shows that v is equal to b from the protocol.

Now, $(c \bmod 2^{k-1}) = x + (r \bmod 2^{k-1}) - 2^{k-1} v$, thus

$$\left\lfloor (c \bmod 2^{k-1})/2^m \right\rfloor = \left\lfloor \frac{x + (r \bmod 2^{k-1})}{2^m} \right\rfloor - 2^{k-m-1} v.$$

Furthermore, it holds that $c' = \lfloor (x + (r \bmod 2^{k-1}))/2^m \rfloor = \lfloor x/2^m \rfloor + \lfloor (r \bmod 2^{k-1})/2^m \rfloor + w$, with $w \in \{0, 1\}$. Given the above, together with the fact that the r' from the protocol equals $\lfloor (r \bmod 2^{k-1})/2^m \rfloor$, we obtain that the output produced by the protocol is

$$c' - r' + 2^{k-m-1} b = \lfloor x/2^m \rfloor + w.$$

Finally, it is easy to see that $w = 1$ with probability equal to the decimal part of $\frac{x}{2^m}$, which shows that the output is biased towards $\lfloor x/2^m \rceil$.

□

5.3.7.2 Random Bit Generation

Another important primitive is the generation of random shares $[\![b]\!]$, where $b \in_R \{0, 1\}$. These sharings have multiple uses. For example, they can be used to convert shared bits $[\![r]\!]_1$ modulo 2, where $r \in \{0, 1\}$ into $[\![r]\!]_k$, which are sharings of the same value but modulo the larger power 2^k. They can also be used to obliviously select a shared value among two options at random. More precisely, given two shared values $[\![x]\!]$ and $[\![y]\!]$, and $[\![b]\!]$ where $b \in_R \{0, 1\}$, the parties can obtain shares $[\![r]\!]$ where $r \in_R \{x, y\}$, without knowing which value between x and y was selected, by computing $[\![r]\!] \leftarrow [\![b]\!] [\![x]\!] + (1 - [\![b]\!]) [\![y]\!]$ (using Π_{Mult}). This can be generalized to sorting networks and random permutations.

Protocol Π_{RandBit}

Setup: (P_1, P_2) and (P_3, P_4) have pre-shared keys k_{12} and k_{34}, respectively.
Output: $[\![b]\!]$ for uniformly random $b \in \{0, 1\}$.
Protocol:

1. (P_1, P_2) and (P_3, P_4) use k_{12} and k_{34} to sample $b_{12} = \text{PRF}_k(\text{id})$ and $b_{23} = \text{PRF}_{k_{34}}(\text{id})$, respectively.
2. The parties call $\Pi_{\text{Input2}}(b_{12}, P_1, P_2)$ and $\Pi_{\text{Input2}}(b_{34}, P_3, P_4)$ to obtain $[\![b_{12}]\!]$ and $[\![b_{34}]\!]$.

3. The parties call $[\![b_{12} \cdot b_{34}]\!] \leftarrow \Pi_{\text{Mult}}([\![b_{12}]\!], [\![b_{34}]\!])$ and then run locally $[\![b]\!] \leftarrow [\![b_{12}]\!] + [\![b_{34}]\!] - 2 \cdot [\![b_{12} \cdot b_{34}]\!]$.

Cheating identification
Output the set produced by the first sub-protocol to fail.

Theorem 5.3. *After the execution of* Π_{RandBit} *either the parties output a semi-corrupt pair, or the parties obtain shares* $[\![b]\!]$*, where* b *is uniformly random in* $\{0,1\}$.

Proof. We can see that, in case that none of the subprotocol calls end in the identification of a semi-corrupt pair, the output produced by the parties is $[\![b]\!]$, where $b = b_{12} \oplus b_{34}$. b_{12} looks indistinguishable from random to the parties $\{P_3, P_4\}$ since it is the output of the PRF using the uniformly random key k_{12}, which is known only to $\{P_1, P_2\}$, hence, b is also indistinguishable from random. A similar argument shows that b looks indistinguishable from random to the parties $\{P_1, P_2\}$. □

Recall that we make the exponent k explicit in the secret-sharing scheme by denoting $[\![s]\!]_k$ when $s \in \mathbb{Z}/2^k\mathbb{Z}$ is secret-shared using the construction from Definition 5.3. Let $\ell \leq k$. A crucial observation is that, if each party P_i *locally* applies modulo 2^ℓ to each of its own additive shares $\{s_j : j \in [4] \setminus \{i\}\}$, then the parties obtain shares $[\![s \bmod 2^\ell]\!]_\ell$. This is denoted by $[\![s \bmod 2^\ell]\!]_\ell \leftarrow [\![s]\!]_k$.

Conversion $[\![\cdot]\!]_k \leftarrow [\![\cdot]\!]_1$. From the observation above we see that, given $[\![s]\!]_k$ where $s \in \{0,1\}$, the parties can locally obtain $[\![s \bmod 2]\!]_1 = [\![s]\!]_1$. However, given $[\![s]\!]_1$, it is not clear how the parties can obtain $[\![s]\!]_k$. This is achieved by means of the following simple protocol which makes use of a shared random bit $[\![b]\!]_k$, with $b \in_R \{0,1\}$:

1. Compute locally $[\![c]\!]_1 \leftarrow [\![s]\!]_1 \oplus [\![b]\!]_1$
2. Reconstruct $c \leftarrow [\![c]\!]_1$.
3. Compute locally $[\![s]\!]_k \leftarrow c + [\![b]\!]_k - 2c\,[\![b]\!]_k$.

Privacy follows from the fact that b is uniformly random in $\{0,1\}$ and unknown to any party, so $c = s \oplus b$ follows the same distribution, and correctness is a consequence of the fact that $c + b - 2cb = c \oplus b = (s \oplus b) \oplus b = s$. This conversion, although is *not* local, is denoted by $[\![s]\!]_k \leftarrow [\![s]\!]_1$.

5.3.7.3 Bit Decomposition

Many applications require more advanced operations than simple additions and multiplications, such as the use of truncation for real-valued arithmetic discussed in Section 5.3.7.1. Another important tool lies in accessing individual *bits* of a secret-shared

value $[\![s]\!]_k$, that is, if the bit-decomposition of $s \in \mathbb{Z}/2^k\mathbb{Z}$ is $s = \sum_{i=0}^{k-1} 2^i s[i]$, allowing the parties to obtain shares of each bit $s[i]$. For example, the computation of the secure hash algorithms (SHA), which is a family of cryptographic hash functions, requires arithmetic modulo 2^{32}, but it simultaneously makes use of operations at the bit level, like permutation of bits.

The protocol Π_{BitDec} presented below achieves the task of bit-decomposition, that is, obtaining $([\![s[k-1]]\!]_1, \dots, [\![s[0]]\!]_1)$ from a secret-shared value $[\![s]\!]_k$. Observe that these output bits are secret-shared over $\mathbb{Z}/2\mathbb{Z}$, and not over the original ring $\mathbb{Z}/2^k\mathbb{Z}$. The motivation for this is that, if the goal is to execute a binary circuit on these bits (e.g. taking as input two bit-decomposed values and returning a bit signaling which is the largest of the two), then holding smaller shares and having these be homomorphic modulo 2 leads to higher efficiency. This was observed and exploited in the original work of [44]. However, if the shares are required to be over $\mathbb{Z}/2^k\mathbb{Z}$, that is, the desired output is $([\![s[k-1]]\!]_k, \dots, [\![s[0]]\!]_k)$, which could be the case for instance if one of these bits is intended to be combined with other secret-shared values over $\mathbb{Z}/2^k\mathbb{Z}$, then either the conversion $[\![\cdot]\!]_k \leftarrow [\![\cdot]\!]_1$ from the previous section can be applied to the output, or the protocol can be easily modified to produce this output directly.

Protocol Π_{BitDec}

Input: Shared value $[\![s]\!]_k$.
Output: Binary replicated secret sharing $([\![s[k-1]]\!]_1, \dots, [\![s[0]]\!]_1)$.
Protocol:

1. For $i \in [4]$, the parties call $[\![s_i[j]]\!]_1 \leftarrow \Pi_{\text{Input3}}(s_i[j], \{P_j\}_{j\in[4]\setminus\{i\}})$ for $j \in \{0, \dots, k-1\}$.
2. Given $[\![x_i[j]]\!]_1$ for all $i \in [4]$ and $j \in \{0, \dots, k-1\}$, and the fact that $x = x_1 + x_2 + x_3 + x_4$, the parties can compute $([\![s[k-1]]\!]_1, \dots, [\![s[0]]\!]_1)$ using a binary adder.

5.3.7.4 Generating edaBits

Extended Double-Authenticated Bits, or *edaBits* for short, were introduced in the original work of [50]. These cryptographic objects can be used to greatly improve the efficiency of a wide range of primitives. An m-edaBit for some $1 \le m \le k$ consists of secret-shared values $([\![r]\!]_k, [\![r[m-1]]\!]_1, \dots, [\![r[0]]\!]_1)$, where $r = \sum_{i=0}^{m-1} 2^i r[i]$ and $r[i] \in \{0,1\}$ for $i \in \{0, \dots, m-1\}$. In Chapter 7 we discuss these objects more concretely, together with their applications, but in this section we restrict ourselves to showing how to generate edaBits in the current setting of four parties with replicated secret-sharing.

Protocol Π_{edaBits}

Setup: For $i \in [4]$, parties $\{P_j\}_{j\in[4]\setminus\{i\}}$ have a pre-shared random key k_i.
Output: $([\![r]\!]_k, [\![r[m-1]]\!]_1, \dots, [\![r[0]]\!]_1)$ for a uniformly random m-bit value $r \in \mathbb{Z}/2^k\mathbb{Z}$.
Protocol: We assume that the output of the PRF are m-bit integers.

1. For $i \in [4]$, the parties in $\{P_j\}_{j\in[4]\setminus\{i\}}$ generate a random m-bit value $r'_i = \text{PRF}_{k_i}(\text{id})$. Thus, the parties obtain $[\![r']\!]_k$, where $r' = r'_1 + r'_2 + r'_3 + r'_4$. Notice that $r' \in [0, \min(2^{m+2}, 2^k) - 1]$.

2. The parties call Π_{BitDec} to obtain $[\![r'[j]]\!]_1$ for $j \in \{m, \dots, m'\}$, where $m' = \min(\lceil \log 4 \rceil, k) - 1$.
3. The parties convert $[\![r'[j]]\!]_k \leftarrow [\![r'[j]]\!]_1$ for $j \in \{m, \dots, m'\}$.
4. The parties compute $[\![r]\!]_k = [\![r']\!]_k - \sum_{j=m}^{m'} 2^j [\![r'[j]]\!]_k$.
5. The parties output $([\![r]\!]_k, [\![r'[m-1]]\!]_1, \dots, [\![r'[0]]\!]_1)$.

Theorem 5.4. *After the execution of* Π_{edaBits} *the parties obtain* $([\![r]\!]_k, [\![r'[m-1]]\!]_1, \dots, [\![r'[0]]\!]_1)$ *where* $r = \sum_{i=0}^{m-1} 2^i r[i]$ *and* r *is uniformly random and unknown to any party.*

Proof. First, it is easy to see that the value r computed in step 4 of the protocol corresponds to $r = (r' \bmod 2^m)$, which follows from the fact that $r = \sum_{j=0}^{m-1} 2^j r'[j]$. Now, we claim that r is uniformly random in $\mathbb{Z}/2^m\mathbb{Z}$ and unknown to any party. To see this, observe that $r \equiv (r'_1 + r'_2 + r'_3 + r'_4) \bmod 2^m$, and each party P_i misses the indistinguishable-from-random value $r'_i \bmod 2^m$. □

5.4 Three Parties and One Corruption

Now we turn our attention to considering protocols for $n = 3$ and $t = 1$. These values satisfy the bound $n/3 \leq t < n/2$, which puts us in the honest majority setting. Even though in this case it is possible to obtain protocols with statistical security satisfying guaranteed output delivery, we settle for computational security by making use of a cryptographic hash function, and we aim at security with abort rather than guaranteed output delivery. This allows us to design much more concretely efficient protocols in this setting than the ones we could obtain by instantiating generic constructions for arbitrary values of t and n with the particular case of $t = 1$ and $n = 3$.

As in Section 5.3, our main tool will be Replicated secret-sharing, this time instantiated with three parties and threshold one. We describe how this secret-sharing scheme works for the particular case of $n = 3$ and $t = 1$ in Section 5.4.2. Then we present in Section 5.4.3 a series of tools that we will need for our main protocol, which is finally presented in Section 5.4.4.

This protocol is introduced in the original work of [4], which presents a generic compiler to turn *any* passively secure protocol satisfying certain extra property into an actively secure one. This extra requirement basically states that, when the corruption is active, the only thing the adversary can break is multiplication by introducing an additive error to the result of each product. This turns out to be the case, for example, with the Shamir-secret-sharing-based multiplication protocol from Section 4.2, but also with the three-party protocol based on replicated secret-sharing from Section 5.4.3.3 below. The original work of [4] considers this generic compilation process from any passively secure protocol, and instantiates the underlying protocol with a passively secure version of the protocol from Chapter 4, and also with the replicated-secret-sharing-based protocol from the current section. Furthermore, in [4] a full implementation of these two protocols is

presented, together with different benchmarks and comparisons with similar works. In this thesis we do not present the generic compilation process but rather focus on the concrete protocol with $n = 3$ and $t = 1$ using replicated secret-sharing. We also do not discuss the experimental contributions, and neither present full simulation-based proofs.

As we have mentioned, the results from this section are taken from the original work of [4], and **some paragraphs are taken verbatim from that work**. This happens mostly in the description of some of the protocols and the proofs of their security.

5.4.1 Pseudo-Random Functions and Cryptographic Hash Functions

As in Section 5.3, we make use of the tools described in sections 5.3.1.1 and 5.3.1.2. We assume the existence of a (family of) cryptographic hash function $\mathsf{H}(\cdot)$, and a pseudo-random function $\mathsf{PRF}_k(\cdot)$. The domain and codomain of these functions will be clear from context.

5.4.2 Replicated Secret-Sharing for Three Parties

In this section, given $x, y \in \mathbb{Z}$, we use $x \equiv_\ell y$ to denote the fact that x and y are congruent modulo 2^ℓ. We begin by describing replicated secret-sharing with three parties and one corruption.

Definition 5.6 (Sharing Procedure)**.** *Let $s \in \mathbb{Z}/2^\ell\mathbb{Z}$. We define* Share$(s)$ *as follows.*

- *Sample $s_1, s_2, s_3 \in_R \mathbb{Z}/2^\ell\mathbb{Z}$ uniformly at random subject to $s \equiv_\ell s_1 + s_2 + s_3$.*
- *Output $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$, where $\mathbf{s}_i = (s_i, s_{i+1})$ for $i \in [3]$.*[6]

As usual, when each party P_i has $\mathbf{s}_i$ for $i \in [3]$, we denote this by $[\![s]\!]_\ell$. Notice the subindex ℓ, making explicit the fact that the sharings are modulo 2^ℓ. Also, as before, whenever we consider a secret-shared value $[\![x]\!]_\ell$, we use the subindexed values x_1, x_2, x_3 to denote the additive shares such that $x = x_1 + x_2 + x_3$, and we use the typewriter font $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3$ to denote the shares held by each party, that is, $\mathtt{x}_i = (x_i, x_{i+1})$.

From two shared values $[\![x]\!]_\ell$ and $[\![y]\!]_\ell$, the parties can obtain shares of $[\![x \pm y]\!]_\ell$ non-interactively by performing the respective operation on their shares. This extends to multiplication by a publicly known value. Also, given a value $c \in \mathbb{Z}/2^\ell\mathbb{Z}$ known by all parties, the parties can obtain shares $[\![c]\!]_\ell$ in a canonical way by considering $c = c_1 + c_2 + c_3$, where $c_1 = c$ and $c_2 = c_3 = 0$. This enables the parties to locally add/subtract the value c to a given shared value $[\![x]\!]$ to obtain $[\![x \pm c]\!]_\ell$ by first obtaining shares $[\![c]\!]_\ell$ and then proceeding as above.

[6] Throughout this section, when clear from context we assume the subindexes that range in $[3]$ wrap around modulo 3 in that set.

As in Section 5.3.7, given $[\![s]\!]_\ell$ the parties can *locally* obtain $[\![s \bmod 2^h]\!]_h$ for any $1 \leq h \leq \ell$ by simply reducing modulo 2^h each of their additive shares. This local operation is denoted by $[\![s \bmod 2^h]\!]_h \leftarrow [\![s]\!]_\ell$. This will be important for our main protocol in Section 5.4.4 since the computation of the circuit will be done modulo 2^ℓ for some $k < \ell$, whereas the result is expected to be modulo 2^k. Using the local conversion from above the parties can obtain shares of the result in the desired set $\mathbb{Z}/2^k\mathbb{Z}$.

Now, in terms of privacy, given only one share $\mathbf{s}_i$, the uniformly random value s_i is missing and therefore nothing is leaked about the secret s, or, in other words, $\mathbf{s}_i$ follows the uniform distribution in $(\mathbb{Z}/2^\ell\mathbb{Z})^2$. On the other hand, any pair of shares $\mathbf{s}_i, \mathbf{s}_j$ is enough to reconstruct the secret since these together contain all the values s_1, s_2, s_3.

We now define the notion of consistent sharings, analogous to Definition 5.4 for the case of Replicated secret-sharing with four parties. Intuitively, the parties have consistent sharings if the correct "replication" holds among the data they possess. This is formalized in the following definition.

Definition 5.7. *Let $\mathbf{s}_i = (s_i^{(i)}, s_{i+1}^{(i)}) \in (\mathbb{Z}/2^\ell\mathbb{Z})^2$ for $i \in [3]$. We say that $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ is* consistent*, and in the case that each party P_i holds $\mathbf{s}_i$ we say the parties* hold consistent shares*, if, for every $i \in [3]$, it holds that $s_i := s_i^{(i)} = s_i^{(i-1)}$. In this case, $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ constitutes sharings of $s := s_1 + s_2 + s_3$.*

5.4.2.1 Public Reconstruction

As in Section 5.5, we need to design a method for the parties to reconstruct a given secret-shared value $[\![z]\!]_\ell$ to the clients, so that these can learn the results of the computation. However, a corrupt party may lies about its own share, which may lead to a value being reconstructed incorrectly. To this end, the clients make use of the existing replication among the parties' shares to determine whether the announced shares lead to the reconstruction of the correct secret or not. Unfortunately, unlike the method from Section 5.5 in the context of replicated secret-sharing with four parties, in our current setting the clients may not be able to reconstruct the correct secret, but it is guaranteed that they will not be fooled into reconstructing an incorrect secret. In the terminology from Section 3.1.3, this means that the clients will be able to perform error detection, which is possible in the setting $t < n/2$, in contrast to error correction, which requires the stronger bound $t < n/3$.

Definition 5.8. *Let $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ be a consistent vector, and let $\mathbf{s}'_i \in (\mathbb{Z}/2^\ell\mathbb{Z})^2$ such that $\mathbf{s}_i = \mathbf{s}'_i$ for all but possibly one $i \in [3]$. We define the method* RecSecret$(\mathbf{s}'_1, \mathbf{s}'_2, \mathbf{s}'_3)$ *as follows:*

1. *Write $\mathbf{s}'_i = (s_i^{(i)}, s_{i+1}^{(i)})$.*
2. *For $j \in [3]$, check if $s_j^{(j)} = s_j^{(j-1)}$.*
3. *If the check above passes, output $s = (s_1 + s_2 + s_3) \bmod 2^\ell$. Else, abort.*

It is clear that RecSecret works as intended, given that each share is held by two parties among which at least one is guaranteed to be honest, so, if the two reported shares coincide, they have to be equal to the share held by the honest party.

Remark 5.3. *This method can be optimized slightly by asking party P_{i-1} to send s_i, while party P_i sends $\mathsf{H}(s_i)$, for each $i \in [3]$, which is enough to check consistency of the announced shares.*

5.4.2.2 Dealing Consistent Shares

As in Section 5.3.3.2, not every vector $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ is consistent since there is a specific replication that must hold among these values. Since a malicious client may distribute an input inconsistently, the parties have to execute a protocol that checks the consistency of these sharings. To this end, we could follow a similar approach to the one from Section 5.3.3.2, in which the parties holding the supposedly same share talk to each other to determine whether they received the same value. In our case, this would mean that if each party P_i receives $\mathbf{s}_i = (s_i^{(i)}, s_{i+1}^{(i)})$ from the client, then P_i would send $s_i^{(i)}$ to P_{i-1}, who checks that $s_i^{(i)} = s_i^{(i-1)}$, raising a complaint if this is not the case. However, the problem with this approach is that a corrupt party may lie about the value it received from the client, making an honest client look as if it distributed the value inconsistently.

To alleviate this issue, we take a different approach. We assume the existence of a broadcast channel between the client C and the parties. The parties also make use of the protocol Π_{Rand}, which is presented later in Section 5.4.3.1, and generates shares of uniformly random values.

Protocol Π_{Input}

Input: Client C has input $x \in \mathbb{Z}/2^\ell\mathbb{Z}$.
Output: The parties output consistent shares $[\![x]\!]_\ell$
Protocol: The parties proceed as follows.

1. The parties call Π_{Rand} to obtain $[\![r]\!]_\ell$.
2. The parties send their shares of $[\![r]\!]_\ell$ to C.
3. C receives $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ and calls RecSecret$(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ to either learn r or abort.
4. If C does not abort, C broadcasts $e = x - r$ to the parties.
5. The parties compute locally $[\![e]\!]_\ell \leftarrow [\![r]\!]_\ell + e$.

The protocol clearly produces shares of x, where $x = r + e$. Furthermore, the parties learn nothing about x since r is uniformly random and unknown to an party. With this new approach, a corrupt party could cause an abort, for example, by sending incorrect shares to the client, who aborts as part of the execution of RecSecret. However, even if this happens, an honest client cannot be incorrectly flagged as corrupt, which is crucial for the viability of an outsourced computation service in practice.

Assumed Setup. We assume the existence of three uniformly random keys (k_1, k_2, k_3), such that, for each $i \in [3]$, P_i has (k_i, k_{i+1}).

5.4.3 Required Subprotocols

We will need several subprotocols for our main secure computation protocol These are described next.

5.4.3.1 Generating Shares of Random Values

We begin by describing a protocol for generating shares of uniformly random values unknown to any party.

Protocol Π_{Rand}

Output: The parties get consistent shares $[\![r]\!]_\ell$, where $r \in_R \mathbb{Z}/2^\ell\mathbb{Z}$ is uniformly random and unknown to any party.
Protocol: For $i \in [3]$, let $r_i = \text{PRF}_{k_i}(\text{id}) \in \mathbb{Z}/2^\ell\mathbb{Z}$, where id is a unique ID associated to the given protocol call. Each P_i outputs the share $\mathfrak{r}_i = (r_i, r_{i+1})$.

It is easy to see that the non-interactive protocol works as intended: consistency is clear, and the fact that r is uniformly random and unknown to any party holds from the fact that $r = r_1 + r_2 + r_3$, where each r_i looks indistinguishable from random to any party missing the key k_i. Since each party misses one key, the claim follows.

5.4.3.2 Checking Equality to 0

For a specific part of our protocol it will be necessary for the parties to check that a given shared value $[\![x]\!]_\ell$ satisfies $x \equiv_\ell 0$, without leaking anything about x beyond whether or not this equality holds. This is achieved by the following protocol.

Protocol $\Pi_{\text{CheckZero}}$

Input: The parties hold consistent shares $[\![x]\!]$
Output: The parties abort if x is not zero
Protocol:

1. For $i \in [3]$, P_i sends $z_i = \text{H}(-(x_i + x_{i+1}))$ to P_{i-1}.
2. For $j \in [3]$, P_j checks that $z_{j+1} = \text{H}(x_j)$, and aborts if this is not the case.

Proposition 5.5. *At the end of the execution of $\Pi_{\text{CheckZero}}$, the parties abort if x is not zero. Furthermore, nothing is leaked about x.*

Proof. Since $x = x_1 + x_2 + x_3$, if $x \neq 0$ then $x_j \neq -(x_{j+1} + x_{j+2})$ for each $j \in [3]$. Let $i - 1$ be the index of the corrupt party. The honest party P_{i+1} sends $z_{i+1} = \mathsf{H}(-(x_{i+1} + x_{i+2}))$ to the honest party P_i, who checks if $z_{i+1} = \mathsf{H}(x_i)$. Since $x_i \neq -(x_{i+1} + x_{i+2})$, this check will not pass, so the honest party P_i, and hence all the parties, abort.

Finally, to see privacy, we have to assume a stronger property of H which, informally, states that "$\mathsf{H}(w)$ leaks nothing about w". With this property we see that each P_i, upon receiving $z_{i+1} = \mathsf{H}(-(x_{i+1}+x_{i+2}))$ from P_{i+1}, learns nothing about $-(x_{i+1}+x_{i+2})$ besides whether this value equals x_i. □

Formalizing the notion that H leaks nothing. As mentioned in the proof above, we do not only need to assume that H is collision resistant, but also that $\mathsf{H}(w)$ leaks nothing about its input w. This property is formalized by making use of the *random oracle*, a standard tool in cryptography. To this end, we define H as a functionality that, upon receiving the input w from a party, samples a uniformly random string r and returns this string to the party performing the request. The functionality stores (w, r) internally, and further calls on the same input w are answered using the same previously sampled string r.

5.4.3.3 Secure Multiplication with Additive Errors

In this section we show how the parties can multiply two given shared values $[\![x]\!]$ and $[\![y]\!]$. Unfortunately, our protocol will not enable the parties to obtain $[\![x \cdot y]\!]$, but rather, the adversary will be able inject a chosen additive error δ so that the final shares are actually $[\![xy + \delta]\!]$. A corresponding functionality for this protocol would be formalized in a similar way as the one from Section 4.2.

Protocol Π_{Mult}

Input: The parties hold sharings $[\![x]\!]$ and $[\![y]\!]$
Output: The parties obtain sharings $[\![xy + \delta]\!]$ for some $\delta \in \mathbb{Z}/2^\ell\mathbb{Z}$ chosen by the adversary.
Protocol: The parties proceed as follows:

1. Each party P_i for $i \in [3]$ computes $z_i = x_i y_i + x_i y_{i+1} + x_{i+1} y_i + (r_i - r_{i+1})$, where $r_i = \mathsf{PRF}_{k_i}(\mathsf{id})$ with id being a unique ID for the given protocol call, and sends z_i to party P_{i-1}.
2. The parties output the shares $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3)$, with $\mathbf{z}_i = (z_i, z_{i+1})$.

Proposition 5.6. *After the execution of Protocol Π_{Mult} the parties get shares $[\![xy + \delta]\!]$ for some δ chosen by the adversary. Furthermore, nothing is leaked about x or y.*

Proof. Let $i \in [3]$ be the index of the corrupt party. Let $z_i' = z_i + \delta$ be the value this party sends to P_{i-1}, where $\delta = (z_i - z_i')$. The output of the parties would be consistent sharings of

$$z = z_i' + \sum_{j \in [3], j \neq i} z_j$$

$$
\begin{aligned}
&= \delta + \sum_{j=1}^{3} z_j \\
&= \delta + \sum_{j=1}^{3} (x_j y_j + x_j y_{j+1} + x_{j+1} y_j + (r_j - r_{j+1})) \\
&= \delta + \sum_{j=1}^{3} (x_j y_j + x_j y_{j+1} + x_{j+1} y_j) + \sum_{j=1}^{3} (r_j - r_{j+1}) \\
&= \delta + (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) + 0 \\
&= xy + \delta.
\end{aligned}
$$

Now, in terms of privacy, we see that the message $z_i = x_{i+1}y_{i+1} + x_{i+1}y_{i+2} + x_{i+2}y_{i+1} + (r_{i+1} - r_{i+2})$ that P_i receives from P_{i+1} indistinguishable from random to P_i given that $r_{i+2} = \mathsf{PRF}_{\mathsf{k}_{i+2}}(\mathsf{id})$ is indistinguishable from random to P_i since this party does not hold the key k_{i+2}. □

Computing dot products securely. Similar to the protocol from Section 5.3.6, the three-party protocol Π_{Mult} described above can be easily modified to securely compute $[\![z]\!]$ from $([\![x_1]\!], \ldots, [\![x_m]\!])$ and $([\![y_1]\!], \ldots, [\![y_m]\!])$, where $z = \sum_{h=1}^{m} x_h y_h$, with a cost that is independent of m. The modification consists of the following:

1. Each party P_i for $i \in [3]$ computes $z_i = \sum_{h=1}^{m} (x_{hi}y_{hi} + x_{hi}y_{h,i+1} + x_{h,i+1}y_{hi}) + (r_i - r_{i+1})$ and sends z_i to party P_{i-1}.

2. The parties output the shares $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3)$, with $\mathbf{z}_i = (z_i, z_{i+1})$.

On top of being useful for applications involving linear algebra, the ability to securely compute dot products with a communication complexity that is independent of the dimension of the vectors is important for our main computation protocol described below. This is because, in the verification stage of that protocol, the parties need to precisely compute dot products on long secret-shared vectors.

5.4.4 Secure Computation Protocol

With the different tools presented in the previous section we are now ready to describe our main secure computation protocol with abort for arithmetic circuits over $\mathbb{Z}/2^k\mathbb{Z}$. At a high level, the protocol consists by executing two copies of the same circuit, one with the actual shared values of the circuit $[\![x]\!]_k$, and another with a "randomized" version of these values $[\![r \cdot x]\!]_k$. As we will show, this will enable the parties to perform a check after the circuit has been computed to ensure that no additive error was introduced in any of the multiplication gates.

The approach sketched above is already present in the work of [30], which presents a generic compiler that turns passively secure protocols *over fields* that are secure up to

additive attacks into actively secure protocol with abort over the same domain. Over the ring $\mathbb{Z}/2^k\mathbb{Z}$ this approach does not work directly, as it ultimately relies on equations of the form $a \cdot x \equiv_k b$ not having many solutions in the variable x if $a \neq 0$, but over $\mathbb{Z}/2^k\mathbb{Z}$ this type of equations can have a lot of solutions, which is the case for example if $a = 2^{k-1}$ and $b = 0$ since every even number is a solution. The main observation of the original work of [4] is that the techniques from the original work of [32] can be used to address this major complication, enabling the adaptation of the generic compilation process from [30] to the setting of the ring $\mathbb{Z}/2^k\mathbb{Z}$.

In short, the main tool from the original work of [32] used in our protocol from this section consists of using replicated secret-sharing modulo $2^{k+\kappa}$, where κ is a statistical security parameter, instead of working plainly modulo 2^k. This way, cheating in multiplication gates can be prevented by showing that equations of the form $a \cdot x \equiv_{k+\kappa} b$ where $a \not\equiv_k 0$ have a limited number of solutions, which is indeed the case. We remark that, chronologically, this core idea was first used in the original work of [32] in the context of dishonest-majority secure computation for any number of parties, a setting that is discussed in Chapter 6 of this thesis. As pointed out above, it was in the original work of [4] that this idea was used, together with the techniques from [30], to obtain a *generic passive-to-active compiler*. The complications that arise when following this route, and the non-trivial steps required to obtain such compiler, are thoroughly documented in [4]. We recall that in this section we focus only on the concrete case of replicated secret-sharing for three parties.

For our protocol below we assume that the arithmetic circuit over $\mathbb{Z}/2^k\mathbb{Z}$ to be computed is made of M input gates and N multiplication gates. Furthermore, given that our protocol is set in the client-server model of secure computation, we assume that the parties start the execution holding shares of the inputs to the computation, which are distributed by the different clients. These sharings are set modulo $2^{k+\kappa}$, even though the underlying shared values lie in $\mathbb{Z}/2^k\mathbb{Z}$, and they are distributed by the different clients by executing together with the parties the protocol Π_{Input} from Section 5.4.2.2. Below, we let M and M be the number of input and multiplication gates, respectively, in the circuit.

Protocol Π_{MPC}

Inputs: The parties have shares $[\![v_1]\!]_{k+\kappa}, \ldots, [\![v_M]\!]_{k+\kappa}$ of the inputs to the computation.
Protocol: The parties proceed as follows

1. *Generate randomizing shares:* The parties call Π_{Rand} to receive $[\![r]\!]_{k+\kappa}$, where $r \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$.
2. *Randomization of inputs:* For each input wire sharing $[\![v_m]\!]_{k+\kappa}$ (where $m \in [M]$) the parties call Π_{Mult} on $[\![r]\!]_{k+\kappa}$ to receive $[\![r \cdot v_m]\!]_{k+\kappa}$.
3. *Circuit emulation:* The parties traverse over the circuit in topological order. For each gate the parties work as follows:
   - *Addition gates:* Given tuples $([\![x]\!]_{k+\kappa}, [\![r \cdot x]\!]_{k+\kappa})$ and $([\![y]\!]_{k+\kappa}, [\![r \cdot y]\!]_{k+\kappa})$ on the *left* and *right* input wires respectively, the parties locally compute $([\![x+y]\!]_{k+\kappa}, [\![r \cdot (x+y)]\!]_{k+\kappa})$.
   - *Multiplication gates:* Given tuples $([\![x]\!]_{k+\kappa}, [\![r \cdot x]\!]_{k+\kappa})$ and $([\![y]\!]_{k+\kappa}, [\![r \cdot y]\!]_{k+\kappa})$ on the *left* and *right* input wires respectively:
     a) The parties call Π_{Mult} on $[\![x]\!]_{k+\kappa}$ and $[\![y]\!]_{k+\kappa}$ to receive $[\![x \cdot y]\!]_{k+\kappa}$.

b) The parties call Π_{Mult} on $[\![r \cdot x]\!]_{k+\kappa}$ and $[\![y]\!]_{k+\kappa}$ to receive $[\![r \cdot x \cdot y]\!]_{k+\kappa}$. The adversary can introduce additive errors in these two calls.

4. *Verification stage:* Let $\left\{\left([\![z_i]\!]_{k+\kappa}, [\![r \cdot z_i]\!]_{k+\kappa}\right)\right\}_{i=1}^{N}$ be the tuples on the output wires of all multiplication gates and let $\left\{[\![v_m]\!]_{k+\kappa}, [\![r \cdot v_m]\!]_{k+\kappa}\right\}_{m=1}^{M}$ be the tuples on the input wires of the circuit.
   a) For $m = 1, \ldots, M$, the parties call Π_{Rand} to receive $[\![\beta_m]\!]_{k+\kappa}$.
   b) For $i = 1, \ldots, N$, the parties call Π_{Rand} to receive $[\![\alpha_i]\!]_{k+\kappa}$.
   c) *Compute linear combinations:*
      i. The parties call Π_{Mult} on $([\![\alpha_1]\!]_{k+\kappa}, \ldots, [\![\alpha_N]\!]_{k+\kappa}, [\![\beta_1]\!]_{k+\kappa}, \ldots, [\![\beta_M]\!]_{k+\kappa})$ and $([\![r \cdot z_1]\!]_{k+\kappa}, \ldots, [\![r, \cdot z_N]\!]_{k+\kappa}, [\![r \cdot v_1]\!]_{k+\kappa}, \ldots, [\![r \cdot v_M]\!]_{k+\kappa})$ to obtain $[\![u]\!]_{k+\kappa}$ with $u = \sum_{i=1}^{N} \alpha_i \cdot (r \cdot z_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m)$.
      ii. The parties call Π_{Mult} on $([\![\alpha_1]\!], \ldots, [\![\alpha_N]\!], [\![\beta_1]\!], \ldots, [\![\beta_M]\!])$ and $([\![z_1]\!]_{k+\kappa}, \ldots, [\![z_N]\!]_{k+\kappa}, [\![v_1]\!]_{k+\kappa}, \ldots, [\![v_M]\!]_{k+\kappa})$ to obtain $[\![w]\!]_{k+\kappa}$, where $w = \sum_{i=1}^{N} \alpha_i \cdot z_i + \sum_{m=1}^{M} \beta_m \cdot v_m$.
   d) The parties broadcast their shares of $[\![r]\!]_{k+\kappa}$ and call RecSecret on these to reconstruct r.
   e) The parties locally compute $[\![T]\!]_{k+\kappa} \leftarrow [\![u]\!]_{k+\kappa} - r \cdot [\![w]\!]_{k+\kappa}$.
   f) The parties call $\Pi_{\text{CheckZero}}$ on $[\![T]\!]_{k+\kappa}$.
5. *Output reconstruction:* If the parties did not abort in the call to $\Pi_{\text{CheckZero}}$, then, for each output wire of the circuit with $[\![v]\!]_{k+\kappa}$, the parties locally convert $[\![v \bmod 2^k]\!]_k \leftarrow [\![v]\!]_{k+\kappa}$. Then, they send their shares of $[\![v \bmod 2^k]\!]_k$ to the clients.

Now we analyze the security of the protocol Π_{MPC}. We recall that our focus in this chapter is more of a practical nature, focusing on practical secure multiparty computation for a small number of parties, and as such we do not provide full simulation-based proofs of security, as done with the other protocols presented so far in this chapter.

Privacy of the protocol before the output reconstruction stems from the fact that all intermediate values are secret-shared, and the only subprotocol called on sensitive data is Π_{Mult}, which, as argued in Proposition 5.6, preserves the privacy of the inputs. From this, a simulator can be defined in a straightforward manner, and this is done in the full version of the original work [4].

The main complication, however, lies in showing that, if the adversary cheats in any multiplication gate by introducing an additive error, then the parties abort. Notice that, since the computation is carried out modulo $2^{k+\kappa}$, but correctness is only required modulo 2^k since this is the original modulus of the arithmetic circuit, an additive error $\delta \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ satisfying $\delta \equiv_k 0$ does not have any harmful effect in the correctness of computation; in other words, if only additive errors that are zero modulo 2^k are introduced, then the computation is correct. On the other hand, if there is at least one error that is not zero modulo 2^k, this would potentially affect the correctness of the computation, which would ultimately make simulation impossible.

The following lemma addresses the situation in which the adversary introduces an additive error that is not zero modulo 2^k in at least one of the calls to Π_{Mult}, showing that

in this case, the check performed by the parties in the verification stage results in abort with overwhelming probability in the security parameter κ.

Lemma 5.1. *If the adversary adds an error $d \not\equiv_k 0$ in any of the calls to Π_{Mult} in the execution of Protocol Π_{MPC}, then the value T computed in the verification stage equals 0 with probability upper-bounded by $2^{-\kappa+\log(\kappa+1)}$. In other words, the parties abort with probability at least $1 - 2^{-\kappa+\log(\kappa+1)}$.*

Proof. Suppose that $(\llbracket x_i \rrbracket_{k+\kappa}, \llbracket y_i \rrbracket_{k+\kappa}, \llbracket z_i \rrbracket_{k+\kappa})$ is the multiplication triple corresponding to the i-th multiplication gate, where $\llbracket x_i \rrbracket_{k+\kappa}, \llbracket y_i \rrbracket_{k+\kappa}$ are the sharings on the input wires and $\llbracket z_i \rrbracket_{k+\kappa}$ is the sharing on the output wire. We note that the values on the input wires may not actually be the appropriate values as when the circuit is computed by honest parties. However, in the verification step, each gate is examined separately, and all that is important is whether the randomized result is $\llbracket r \cdot z_i \rrbracket_{k+\kappa}$ for whatever z_i is here (i.e., even if an error was added by the adversary in previous gates). By Proposition 5.6, a malicious adversary is able to carry out an additive attack in the calls to Π_{Mult}, meaning that it can add a value to the output of each multiplication gate.

We denote by $\delta_i \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ the value that is added by the adversary when Π_{Mult} is called with $\llbracket x_i \rrbracket_{k+\kappa}$ and $\llbracket y_i \rrbracket_{k+\kappa}$, and by $\gamma_i \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ the value added by the adversary when Π_{Mult} is called with the shares $\llbracket y_i \rrbracket_{k+\kappa}$ and $\llbracket r \cdot x_i \rrbracket_{k+\kappa}$. However, it is possible that the adversary has attacked previous gates and so $\llbracket y_i \rrbracket_{k+\kappa}$ is actually multiplied with $\llbracket r \cdot x_i + \epsilon_i \rrbracket$, where the value $\epsilon_i \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ is an accumulated error from previous gates.[7] Thus, it holds that $z_i = x_i \cdot y_i + \delta_i$, and the shared value the parties obtain as part of the randomized version of z_i is $\llbracket (r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i \rrbracket$. Similarly, for each input wire with sharing $\llbracket v_m \rrbracket$, it holds that the parties compute sharings $\llbracket r \cdot v_m + \xi_m \rrbracket$, where $\xi_m \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ is the value added by the adversary when Π_{Mult} is called with $\llbracket r \rrbracket_{k+\kappa}$ and the shared input $\llbracket v_m \rrbracket_{k+\kappa}$.

Thus, we have that the shared values $\llbracket u \rrbracket$ and $\llbracket v \rrbracket$ computed in step 4c of the protocol are equal to

$$\begin{aligned} u &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) \\ &\quad + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \Theta_1 \\ w &= \sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2 \end{aligned}$$

where $\Theta_1 \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ and $\Theta_2 \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ are the values being added by the adversary when Π_{Mult} is called in the verification step. From this, the shared value $\llbracket T \rrbracket$ computed in step 4e is equal to

$$T = u - r \cdot w =$$

[7] Although attacks in previous gates may be carried out on both multiplications, the idea is here is to fix x_i which is shared by $\llbracket x_i \rrbracket_{k+\kappa}$ at the current value on the wire, and then given the randomized sharing $\llbracket x_i' \rrbracket_{k+\kappa}$, define $\epsilon_i = x_i' - r \cdot x_i$ as the accumulated error on the input wire.

$$
\begin{aligned}
&= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \theta_1 \\
&\qquad - r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2 \right) \\
&= \sum_{i=1}^{N} \alpha_i \cdot (\epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i) \qquad (5.1) \\
&\qquad + \sum_{m=1}^{M} \beta_m \cdot \xi_m + (\Theta_1 - r \cdot \Theta_2),
\end{aligned}
$$

where the second equality holds because r is opened and so the multiplication $r \cdot [\![w]\!]_{k+\kappa}$ always yields $[\![r \cdot w]\!]_{k+\kappa}$, i.e. no additive error can be injected in this step. Let $\Delta_i = \epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i$.

Our goal is to show that T, as shown in Eq. (5.2), equals 0 with probability at most $2^{-\kappa+\log(\kappa+1)}$. We have the following cases.

- *Case 1: There exists $m \in [M]$ such that $\xi_m \not\equiv_k 0$.* Let m_0 be the smallest such m for which this holds. Then $T \equiv_{k+\kappa} 0$ if and only if

$$
\beta_{m_0} \cdot \xi_{m_0} \equiv_{k+\kappa} \left(- \sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).
$$

  Let 2^u be the largest power of 2 dividing ξ_{m_0}. Then we have that

$$
\beta_{m_0} \equiv_{k+\kappa-u} \left(\frac{- \sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2)}{2^u} \right) \cdot \left(\frac{\xi_{m_0}}{2^u} \right)^{-1}.
$$

  By the assumption that $\xi_m \not\equiv_k 0$ it follows that $u < k$ and so $k + \kappa - u > \kappa$ which means that the above holds with probability at most $2^{-\kappa}$, since β_{m_0} is uniformly distributed over $\mathbb{Z}/2^{k+\kappa}\mathbb{Z}$.

- *Case 2: All $\xi_m \equiv_k 0$.* By the assumption in the lemma, some additive value $d \not\equiv_k 0$ was sent to Π_{Mult}. Since none was sent for the input randomization, there exists some $i \in \{1, \ldots, N\}$ such that $\delta_i \not\equiv_k 0$ or $\gamma_i \not\equiv_k 0$. Let i_0 be the smallest such i for which this holds. Note that since this is the first error added which is $\not\equiv_k 0$, it holds that $\epsilon_{i_0} \equiv_k 0$. Thus, in this case, $T \equiv_{k+\kappa} 0$ if and only if $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+\kappa} Y$, where

$$
Y = \left(- \sum_{\substack{i=1 \\ i \neq i_0}}^{N} \alpha_i \cdot \Delta_i - \sum_{m=1}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).
$$

  Let q be the random variable corresponding to the largest power of 2 dividing Δ_{i_0}, where we define $q = k + \kappa$ in the case that $\Delta_{i_0} \equiv_{k+\kappa} 0$. Let E denote the event $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+\kappa} Y$. We have the following claims.

- *Claim 1: For $k < j \leq k + \kappa$, it holds that $\Pr[q = j] \leq 2^{-(j-k)}$.*

  To see this, suppose that $q = j$ and $j > k$. It holds then that $\Delta_{i_0} \equiv_j 0$, and so $\Delta_{i_0} \equiv_k 0$. We first claim that in this case it must hold that $\delta_{i_0} \not\equiv_k 0$. Assume in contradiction that $\delta_{i_0} \equiv_k 0$. In addition, by our assumption we have that $\gamma_{i_0} \not\equiv_k 0$, $\epsilon_i \equiv_k 0$ and $\Delta_{i_0} = \epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0} - r \cdot \delta_{i_0} \equiv_k 0$. However, $\epsilon_i \cdot y_{i_0} \equiv_k 0$ and $r \cdot \delta_{i_0} \equiv_k 0$ imply that $\gamma_{i_0} \equiv_k 0$, which is a contradiction.

  We thus assume that $\delta_{i_0} \not\equiv_k 0$, and in particular there exists $u < k$, such that u is the largest power of 2 dividing δ_{i_0}. It is easy to see then that $q = j$ implies that $r \equiv_{j-u} \left(\frac{\epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0}}{2^u}\right) \cdot \left(\frac{\delta_{i_0}}{2^u}\right)^{-1}$. Since $r \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ is uniformly random and $u < k$, we have that this equation holds with probability of at most $2^{-(j-u)} \leq 2^{-(j-k)}$.

- *Claim 2: For $k < j < k + \kappa$ it holds that $\Pr[E \mid q = j] \leq 2^{-(k+\kappa-j)}$.*

  To prove this let us assume that $q = j$ and that E holds. In this case we can write $\alpha_{i_0} \equiv_{k+\kappa-j} \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j}\right)^{-1}$. For $k < j < k + \kappa$ it holds that $0 < k + \kappa - j < \kappa$ and therefore this equation can be only satisfied with probability at most $2^{-(k+\kappa-j)}$, given that $\alpha_{i_0} \in \mathbb{Z}/2^\kappa\mathbb{Z}$ is uniformly random.

- *Claim 3: $\Pr[E \mid 0 \leq q \leq k] \leq 2^{-\kappa}$.*

  This is implied by the proof of the previous claim, since in the case that $q = j$ with $0 \leq j \leq k$, it holds that $k + \kappa - j \geq \kappa$, so the event E implies that $\alpha_{i_0} \equiv_\kappa \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j}\right)^{-1}$, which holds with probability at most $2^{-\kappa}$.

Putting these pieces together, we thus have the following:

$$\begin{aligned} \Pr[E] &= \Pr[E \mid 0 \leq q \leq k] \cdot \Pr[0 \leq q \leq k] + \\ &\qquad \sum_{j=k+1}^{k+\kappa} \Pr[E \mid q = j] \cdot \Pr[q = j] \\ &\leq 2^{-\kappa} + \kappa \cdot 2^{-\kappa} = (\kappa + 1) \cdot 2^{-\kappa} = 2^{-\kappa + \log(\kappa+1)}. \end{aligned} \tag{5.2}$$

To sum up the proof, in the first case we obtained that $T = 0$ with probability of at most $2^{-\kappa}$ whereas in the second case, this holds with probability of at most $2^{-\kappa+\log(\kappa+1)}$. Therefore, we conclude that the probability that $T = 0$ in the verification step is bounded by $2^{-\kappa+\log(\kappa+1)}$ as stated in the lemma. This concludes the proof. □

From this lemma it is straightforward to prove the following result, whose complete proof is presented in the full version of the original work [4].

Theorem 5.5. *Protocol Π_{MPC} securely instantiates the $\mathcal{F}_{\mathsf{MPC}}$ functionality with abort with statistical error $2^{-\kappa+\log(\kappa+1)}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Chapter 6

SPDZ2k: Dishonest Majority MPC over $\mathbb{Z}/2^k\mathbb{Z}$

Multiparty Computation in the context in which the adversary controls at least half of the parties is extremely relevant in practice. For example, in many settings assuming that the majority of the parties remain honest may not be sensible. This could be the case for instance if there is little diversification among the parties executing the protocol (for example, several parties are hosted in the same data center, or administered by the same entity), or it could also happen if the number of parties is very low. For example, if only two parties are considered, the only scenario that makes sense in this in which the adversary corrupts exactly one party, which does not constitute a minority.

As illustrated in Section 1.3.3, in spite of the advantages that MPC in the dishonest majority setting may have, such protocols come at a costly price in terms of efficiency and security guarantees. More precisely, unlike the constructions presented in Chapters 3 and 4 the previous chapter, protocols that are secure against a majority of corrupt parties must rely on computational assumptions, instead of guaranteeing security in the information-theoretic sense. On the other hand, partially implied by the above, these protocols tend to be much more inefficient than their honest majority counterparts. Unfortunately, as illustrated in the previous paragraph, several scenarios and use cases for MPC cannot afford to assume an honest majority, so improving the efficiency of dishonest majority protocols is a worthy goal with multiple implications in the applicability of MPC in practice.

In Sections 2.6 and 2.7 we presented different techniques for dishonest majority multiparty computation *over fields*, with passive and active security respectively. These protocols operated over fields, and, in fact, for the case of active security, this is a crucial aspect of their design, as we will see soon.[1] The goal of this chapter is to develop a protocol for computation modulo 2^k by extending these ideas over fields, to the case in which the algebraic structure is the ring $\mathbb{Z}/2^k\mathbb{Z}$. Our work, like the protocols presented in Sections 2.6 and 2.7, are set in the preprocessing model, which means that the execution of the protocol can be split into two phases: and offline phase, independent of the actual inputs from the participants, and an online phase, which depends on these inputs and tends to be much leaner.

This chapter is organized as follows. First, in Section 6.1 we recall the *arithmetic black-box model*, which is an alternative way of representing arithmetic circuits in a more flexible way, more compatible with reactive computation. Then, in Section 6.2, we show

[1] The passively secure protocol as presented in Section 2.6 is ported in a straightforward manner not only to the ring $\mathbb{Z}/2^k\mathbb{Z}$, but to any finite ring.

how to reduce the arithmetic black-box functionality to a simpler functionality that, in essence, does not support multiplications, but provides the ability to preprocess multiplication triples. This is simply a presentation of Beaver's results [15]. In Section 6.3 we then present the main building block and contribution of this Chapter: an authenticated secret-sharing scheme that supports homomorphic operations modulo 2^k, and we finally show in Section 6.4 how to use this construction to instantiate the partial arithmetic black-box model, assuming access to multiplication triples.

The results in this chapter are based on the original work of [32], which presents SPDZ2k (or SPD$\mathbb{Z}_{2^k}$), the first actively secure protocol over $\mathbb{Z}/2^k\mathbb{Z}$ in the dishonest majority setting. The contributions of that work can be summarized as follows. First, the authors present an authenticated secret-sharing scheme that is homomorphic modulo 2^k and enables the parties to reconstruct values correctly. This is one of the core contributions of that work, and these techniques have proven useful in other dishonest majority protocols like [69] and [26], and even other settings like honest majority as in the original work of [4], which was partially described in Section 5.4. The second contribution of [32] lies in the generation of the necessary preprocessing material to enable secure computation using the secret-sharing scheme mentioned above. This preprocessed data consists mostly of (authenticated) multiplication triples, and their generation is done with the help of Oblivious Transfer, in a way that resembles the triple generation procedure from the MASCOT protocol [62]. In this thesis we only discuss the first part, namely, the authenticated secret-sharing scheme and the method to reconstruct shares correctly. We do not include the generation of the multiplication triples using oblivious transfer, which is presented in the original work of [32].

Finally, we also point out that the SPDZ2k protocol was implemented in the subsequent work of [44], presented by the author of this thesis at S&P 2019. That work presents a series of applications of this protocol and illustrate its benefits with respect to other field-based constructions, and it also introduces different primitives for enabling these applications.

6.1 Arithmetic Black Box

We begin by recalling the arithmetic black-box model, or ABB model, for short, which was briefly discussed in Section 1.2.6.2. In this framework, which is formalized as the functionality $\mathcal{F}_{\text{ABB}}$ below, the parties have the ability of storing inputs that remain private, and they can instruct the functionality to perform basic operations on stored data like multiplications and linear affine combinations, with these computations carried out modulo 2^k. Furthermore, at any point, the parties altogether can instruct the functionality to open any stored value, making it public to all the parties. This way, the ABB model poses a generalization of the arithmetic circuit model of computation, except that the ABB model fits better the setting of *reactive computation* as considered in Section 1.1, where parties learn intermediate results of the computation and then continue with subsequent stages, perhaps inputting new data that might depend on the results learned so far by the parties.

We remark that almost all of the protocols described so far in this thesis can be phrased

in the ABB model rather than the arithmetic circuit model, and as a result, they can also be used for secure reactive computation. The only exception is the three-party protocol from Section 5.4, which, when reconstructing the output, requires the parties to reconstruct the "key" $[\![r]\!]$, which cannot be reused for further computation. This protocol is modified to enable reactive computation in the original work of [36].

We now proceed to presenting formally the functionality $\mathcal{F}_{\text{ABB}}$. This functionality keeps an internal dictionary of values stored by the parties, and performs operations on these as instructed. Also, notice that the commands are issued by the honest parties only, and the functionality, upon executing these commands, notifies the adversary about their success. This is done in order to facilitate simulation later on, since the adversary's role will be played by the simulator, who needs to be informed about the different commands being executed by the functionality $\mathcal{F}_{\text{ABB}}$ being instantiated in order to emulate the protocol execution. Details of this are given in Section 6.2.

Functionality $\mathcal{F}_{\text{ABB}}$: Arithmetic Black Box

The functionality proceeds as follows.

- On input $(\text{input}, \text{id}, i)$ from the honest parties, send $(\text{input}, \text{id}, i)$ to the adversary, wait for input $(\text{value}, \text{id}, x)$ from party P_i, where $x \in \mathbb{Z}/2^k\mathbb{Z}$, and then store (id, x) in memory.
- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$ from the honest parties, retrieve (id_i, x_i) for $i \in [\ell]$ from memory and store $(\text{id}_{\ell+1}, z)$, where $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2^k$. Then send $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$ to the adversary.
- On input $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ from the honest parties, retrieve (id_1, x) and (id_2, y) from memory and store (id_3, z), where $z = x \cdot y$. Then send $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ to the adversary.
- On input (open, id) from the honest parties, retrieve (id, x) from memory and send x to all the parties. Then send (open, id) to the adversary.

6.2 Instantiating $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{PABB}}$-Hybrid Model

Now we present an alternative functionality that we call *partial arithmetic black box*, or PABB for short, which is like $\mathcal{F}_{\text{ABB}}$, except it lacks the (input) and (mult) commands. Instead, it counts on some additional commands for storing values with certain structure, like random values and multiplication triples, plus other helper commands, as we will see below.

Functionality $\mathcal{F}_{\text{PABB}}$: Partial Arithmetic Black Box

The functionality proceeds as follows.

- On input $(\text{random}, \text{id})$ from the honest parties, sample $r \in_R \mathbb{Z}/2^k\mathbb{Z}$ and store (id, r) in memory. Then send $(\text{random}, \text{id})$ to the adversary
- On input $(\text{triple}, \text{id}_1, \text{id}_2, \text{id}_3)$ from the honest parties, sample $a, b \in_R \mathbb{Z}/2^k\mathbb{Z}$ and

store (id_1, a), (id_2, b) and $(\mathrm{id}_3, c = ab)$ in memory. Then send $(\text{triple}, \mathrm{id}_1, \mathrm{id}_2, \mathrm{id}_3)$ to the adversary

- On input $(\text{open.ind}, \mathrm{id}, i)$ from the honest parties, retrieve (id, x) from memory and send x to all the parties. Then send $(\text{open.ind}, \mathrm{id}, i)$ to the adversary

---

- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\mathrm{id}_i\}_{i\in[\ell]}, \mathrm{id}_{\ell+1})$ from the honest parties, retrieve (id_i, x_i) for $i \in [\ell]$ from memory and store $(\mathrm{id}_{\ell+1}, z)$, where $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2^k$. Then send $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\mathrm{id}_i\}_{i\in[\ell]}, \mathrm{id}_{\ell+1})$ to the adversary.
- On input $(\text{open}, \mathrm{id})$ from the honest parties, retrieve (id, x) from memory and send x to all the parties. Then send $(\text{open}, \mathrm{id})$ to the adversary.

Notice that the commands below the dashed line in the description of $\mathcal{F}_{\text{PABB}}$, namely (comb) and (open), are already present in the functionality $\mathcal{F}_{\text{ABB}}$, whereas the ones above the dashed line are new commands. The motivation behind considering the functionality $\mathcal{F}_{\text{PABB}}$ is that this functionality is enough to instantiate $\mathcal{F}_{\text{ABB}}$, and it is simpler as it does not include the (mult) command which, as usual with secure multiparty computation protocols, tends to be the bottleneck. Instead, it includes the (triple) command which stores multiplication triples, and this type of preprocessing material will be much simpler to generate.

Furthermore, and most importantly, $\mathcal{F}_{\text{PABB}}$ can be instantiated with the construction of authenticated secret-sharing we will discuss in Section 6.3. This secret-sharing scheme enables the parties to store data in a distributed fashion, take linear affine combinations modulo 2^k, and later on open the shared values in a way that the adversary cannot cheat in the reconstruction. As a result, the commands (comb) and (open) (and also (open.ind)) can be easily instantiated. The necessary preprocessing material like multiplication triples and random shared values, that is, the instantiation of the commands (triple) and (random), is done in a trivial way by assuming a functionality that distributes shares with the required distribution under the secret-sharing scheme discussed above.

The instantiation of $\mathcal{F}_{\text{PABB}}$ using our authenticated secret-sharing scheme in the preprocessing model is discussed in full detail in Section 6.4. For now, we present below the protocol Π_{ABB}, which instantiates the functionality $\mathcal{F}_{\text{ABB}}$ in the $\mathcal{F}_{\text{PABB}}$-hybrid model. As sketched above, this allows us to focus only on instantiating $\mathcal{F}_{\text{PABB}}$ in subsequent sections.

Protocol Π_{ABB}

Functionalities: $\mathcal{F}_{\text{PABB}}$.
Protocol: The parties proceed as follows.

- Upon receiving the commands (comb) and (open), the parties forward these to $\mathcal{F}_{\text{PABB}}$.
- Upon receiving the command $(\text{input}, \mathrm{id}, i)$, and P_i receiving input $(\text{value}, \mathrm{id}, x)$, the parties execute the following
  1. Call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{random}, \mathrm{id}_r)$.
  2. Call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{open.ind}, \mathrm{id}_r, i)$, so P_i obtains $r \in \mathbb{Z}/2^k\mathbb{Z}$ from $\mathcal{F}_{\text{PABB}}$.

3. P_i broadcasts $e = x - r$ to all parties
4. Call $\mathcal{F}_{\text{PABB}}$ on input $(\text{comb}, \{e, 1\}, \{\text{id}_r\}, \text{id})$.

- Upon receiving the command $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$, the parties execute the following
  1. Call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{triple}, \text{id}'_1, \text{id}'_2, \text{id}'_3)$. This results in $\mathcal{F}_{\text{PABB}}$ storing (id'_1, a), (id'_2, b) and (id'_3, c), where $c = a \cdot b$.
  2. Call $\mathcal{F}_{\text{PABB}}$ with the commands $(\text{comb}, \{0, 1, -1\}, \{\text{id}_1, \text{id}'_1\}, \text{id}''_1)$ and $(\text{comb}, \{0, 1, -1\}, \{\text{id}_1, \text{id}'_1\}, \text{id}''_1)$.
  3. Call $\mathcal{F}_{\text{PABB}}$ on inputs $(\text{open}, \text{id}''_1)$ and $(\text{open}, \text{id}''_2)$, so the parties obtain d and e, where $\mathcal{F}_{\text{PABB}}$ has stored (id''_1, d) and (id''_2, e).
  4. Call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{comb}, \{d \cdot e, e, d, 1\}, \{\text{id}'_1, \text{id}'_2, \text{id}'_3\}, \text{id}_3)$.

Proposition 6.1. *Protocol Π_{ABB} instantiates the functionality $\mathcal{F}_{\text{ABB}}$ with perfect security in the $\mathcal{F}_{\text{PABB}}$-hybrid model.*

Proof. The simulator $\mathcal{S}$ is defined as follows. $\mathcal{S}$ is in charge of emulating virtual honest parties $\overline{P}_i$ for $i \in \mathcal{H}$, and also the functionality $\mathcal{F}_{\text{PABB}}$.

- Upon receiving $(\text{comb}, \star)$ or $(\text{open}, \star)$ from $\mathcal{F}_{\text{ABB}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{PABB}}$ being called on these inputs by the virtual honest parties.
- Upon receiving $(\text{input}, \text{id}, i)$ from $\mathcal{F}_{\text{ABB}}$, $\mathcal{S}$ proceeds as follows depending on the value of i.
  - If $i \in \mathcal{H}$, then $\mathcal{S}$ emulates $\mathcal{F}_{\text{PABB}}$ as follows:
    1. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{random}, \text{id}_r)$ from the virtual honest parties.
    2. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{open.ind}, \text{id}_r, i)$ from the virtual honest parties.
    3. Sample $\overline{e} \in \mathbb{Z}/2^k\mathbb{Z}$ uniformly at random and send it in the emulated broadcast channel.
    4. Emulate $\mathcal{F}_{\text{PABB}}$ on input $(\text{comb}, \{\overline{e}, 1\}, \{\text{id}_r\}, \text{id})$ from the virtual honest parties.
  - If $i \in \mathcal{C}$, then $\mathcal{S}$ emulates $\mathcal{F}_{\text{PABB}}$ as follows:
    1. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{random}, \text{id}_r)$ from the virtual honest parties.
    2. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{open.ind}, \text{id}_r, i)$ from the virtual honest parties. Then sample $\overline{r} \in_R \mathbb{Z}/2^k\mathbb{Z}$ and send it to the corrupt party P_i.
    3. $\mathcal{S}$ receives $\overline{e}$ from P_i in the emulated broadcast channel. Then $\mathcal{S}$ sends $(\text{input}, \text{id}, x)$ to $\mathcal{F}_{\text{ABB}}$, where $x = e + r$.
    4. Emulate $\mathcal{F}_{\text{PABB}}$ on input $(\text{comb}, \{\overline{e}, 1\}, \{\text{id}_r\}, \text{id})$.
- Upon receiving $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ from $\mathcal{F}_{\text{ABB}}$, $\mathcal{S}$ proceeds as follows
  1. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{triple}, \text{id}'_1, \text{id}'_2, \text{id}'_3)$ from the virtual honest parties.
  2. Emulate $\mathcal{F}_{\text{PABB}}$ with the commands $(\text{comb}, \{0, 1, -1\}, \{\text{id}_1, \text{id}'_1\}, \text{id}''_1)$ and $(\text{comb}, \{0, 1, -1\}, \{\text{id}_1, \text{id}'_1\}, \text{id}''_1)$.
  3. Emulate $\mathcal{F}_{\text{PABB}}$ on inputs $(\text{open}, \text{id}''_1)$ and $(\text{open}, \text{id}''_2)$ by sampling uniformly random values $\overline{d}, \overline{e} \in_R \mathbb{Z}/2^k\mathbb{Z}$ and sending these to the corrupt parties.
  4. Emulate $\mathcal{F}_{\text{PABB}}$ with the command $(\text{comb}, \{\overline{de}, \overline{e}, \overline{d}, 1\}, \{\text{id}'_1, \text{id}'_2, \text{id}'_3\}, \text{id}_3)$.

We now argue indistinguishability between the real and ideal worlds. First we approach the case in which commands (comb) or (open) are issued. In this case, indistinguishability is trivially achieved given that in the ideal world these calls are handled by $\mathcal{F}_{\text{ABB}}$, and in the real world they are handled by $\mathcal{F}_{\text{PABB}}$, which behave in the exact same way.

Now we analyze the case in which the command $(\text{input}, \text{id}, i)$ is issued by the honest parties.

Real world	**Ideal world**
1. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{random}, \text{id}_r)$, so $\mathcal{F}_{\text{PABB}}$ stores (id_r, r) where $r \in_R \mathbb{Z}/2^k\mathbb{Z}$, and the adversary receives $(\text{random}, \text{id}_r)$.	1. The adversary receives $(\text{random}, \text{id}_r)$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$.
2. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{open.ind}, \text{id}_r, i)$, so the party P_i receives r and the adversary receives $(\text{open.ind}, \text{id}_r, i)$.	2. The adversary also receives $(\text{random}, \text{id}_r)$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$, and if P_i is corrupt, this party also receives a uniformly random value $\overline{r} \in_R \mathbb{Z}/2^k\mathbb{Z}$, as in the real world.
3. The party P_i broadcasts $e = x - r$.	3. If P_i is corrupt, since the execution is indistinguishable so far, the $\overline{e}$ broadcasted in the ideal world follows the same distribution as e. If P_i is honest, then e looks uniformly random to the adversary since r is uniformly random and known only to P_i. This is the same distribution as the value $\overline{e}$ received by the adversary in the ideal world.
4. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{comb}, \{e, 1\}, \{\text{id}_r\}, \text{id})$, and the adversary receives $(\text{comb}, \{e, 1\}, \{\text{id}_r\}, \text{id})$. The functionality $\mathcal{F}_{\text{PABB}}$ stores (id, z), where $z = e + 1 \cdot r$. If $i \in \mathcal{H}$, this input z is equal to x, the value received by P_i initially. If $i \in \mathcal{C}$, then z simply corresponds to $e + r$, with r being the value received and e the value broadcasted by P_i.	4. The adversary also receives $(\text{comb}, \{\overline{e}, 1\}, \{\text{id}_r\}, \text{id})$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$. Furthermore, the functionality $\mathcal{F}_{\text{ABB}}$ stored $(\text{id}, \overline{z})$. If $i \in \mathcal{H}$, $\overline{z}$ is equal to x, the value received by P_i initially, as in the real world. If $i \in \mathcal{C}$, then $\overline{z}$ corresponds to $\overline{e} + \overline{r}$, with $\overline{r}$ being the value received and $\overline{e}$ the value broadcasted by P_i. These are indistinguishable from the corresponding values in the real world.

It only remains to be seen that the two worlds remain indistinguishable when the command $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ is issued. This is shown below.

Real world	**Ideal world**
1. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{triple}, \text{id}'_1, \text{id}'_2, \text{id}'_3)$, so $\mathcal{F}_{\text{PABB}}$ samples $a, b \in_R \mathbb{Z}/2^k\mathbb{Z}$ and stores (id'_1, a), (id'_2, b) and (id'_3, c), where $c = a \cdot b$. Also, the adversary receives $(\text{triple}, \text{id}'_1, \text{id}'_2, \text{id}'_3)$ from $\mathcal{F}_{\text{PABB}}$.	1. The adversary also receives $(\text{triple}, \text{id}'_1, \text{id}'_2, \text{id}'_3)$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$.

2. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the commands $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$ and $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$. This results in $\mathcal{F}_{\text{PABB}}$ storing (id_1'', d) and (id_2'', e), where $d = x - a$ and $e = y - b$, where $\mathcal{F}_{\text{PABB}}$ has stored (id_1, x) and (id_2, y). Finally, the adversary gets $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$ and $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$ from $\mathcal{F}_{\text{PABB}}$.	2. The adversary also receives $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$ and $(\text{comb}, \{0,1,-1\}, \{\text{id}_1, \text{id}_1'\}, \text{id}_1'')$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$.
3. The honest parties call $\mathcal{F}_{\text{PABB}}$ on inputs $(\text{open}, \text{id}_1'')$ and $(\text{open}, \text{id}_2'')$, so all parties learn d and e, and the adversary gets $(\text{open}, \text{id}_1'')$ and $(\text{open}, \text{id}_2'')$. Notice $d = x - a$ and $e = y - b$ look uniformly random to the adversary since $a, b \in_R \mathbb{Z}/2^k\mathbb{Z}$ are uniformly random and never revealed to the adversary.	3. The adversary also gets $(\text{open}, \text{id}_1'')$ and $(\text{open}, \text{id}_2'')$ from the emulated $\mathcal{F}_{\text{PABB}}$. Additionally, the parties learn uniformly random values $\overline{d}, \overline{e} \in_R \mathbb{Z}/2^k\mathbb{Z}$, which follows the same distribution as the real world.
4. The honest parties call $\mathcal{F}_{\text{PABB}}$ with the command $(\text{comb}, \{d \cdot e, e, d, 1\}, \{\text{id}_1', \text{id}_2', \text{id}_3'\}, \text{id}_3)$, so $\mathcal{F}_{\text{PABB}}$ stores (id_3, z), where $z = de + ea + db + 1 \cdot c = x \cdot y$, and it sends $(\text{comb}, \{d \cdot e, e, d, 1\}, \{\text{id}_1', \text{id}_2', \text{id}_3'\}, \text{id}_3)$ to the adversary.	4. The adversary also receives $(\text{comb}, \{d \cdot e, e, d, 1\}, \{\text{id}_1', \text{id}_2', \text{id}_3'\}, \text{id}_3)$ from $\mathcal{S}$ as part of the emulation of $\mathcal{F}_{\text{PABB}}$. Furthermore, the functionality $\mathcal{F}_{\text{ABB}}$ stored (id_3, xy), which is the same entry stored by $\mathcal{F}_{\text{PABB}}$ in the real world.

□

6.3 Authenticated Secret-Sharing

In this section we present the main tool to instantiate the "storing" capabilities of $\mathcal{F}_{\text{ABB}}/\mathcal{F}_{\text{PABB}}$, and the ability to perform affine linear combinations modulo 2^k on stored data. This is achieved by means of am authenticated secret-sharing scheme that is used to distribute values in $\mathbb{Z}/2^k\mathbb{Z}$ with homomorphism over this ring. The *authenticated* part refers to the fact that the parties have some additional data that prevents the adversary to reconstruct incorrect secrets, this ensuring authenticity of the underlying data. Below we will present the construction of an authenticated secret-sharing scheme, that lies at the heart of our main protocol. We also state and prove various basic properties of the scheme that are used later on in Section 6.4 when we formally instantiate $\mathcal{F}_{\text{PABB}}$ in the preprocessing model.

As we will see, this construction closely resembles the one from Section 5.4.4, although chronologically, the core idea of working over $\mathbb{Z}/2^k\mathbb{Z}$ to obtain authenticated secret-sharing over $\mathbb{Z}/2^k\mathbb{Z}$ appeared first in the original work of [32], and was used as a building block in the original work of [4], upon which the protocol from Section 5.4.4 is based on. These techniques have also been used in other works not involving the author of the thesis, like [69] and [26]. Also, to further highlight the differences between the techniques used in this section and the ones from Section 5.4.4, we remark that in the latter the use of this construction serves the purpose of disallowing additive attacks in the multiplication gates, and there, reconstructing secret-shared values was not a problem due to the use of replicated secret-sharing, which enabled error detection. In our current setting, as common in the dishonest majority case, we need to ensure the adversary cannot cheat when reconstructing secret-shared values, which a task of a different nature.

As in Section 5.4, given two integers $x, y \in \mathbb{Z}$ we use $x \equiv_\ell y$ to denote $x \equiv y \bmod 2^\ell$. Given $s \in \mathbb{Z}/2^\ell\mathbb{Z}$, we denote by $[\![s]\!]_\ell = (s_1, \ldots, s_n)$ additive secret-sharing over $\mathbb{Z}/2^\ell\mathbb{Z}$, that is, $s_1, \ldots, s_n \in \mathbb{Z}/2^\ell\mathbb{Z}$ are uniformly random constrained to $s \equiv_\ell \sum_{i=1}^n s_i$. Notice that the parties can locally reduce their additive shares modulo 2^h for any $h \leq \ell$ to obtain $[\![s \bmod 2^h]\!]_h$, a local operation that is denoted by $[\![s \bmod 2^h]\!]_h \leftarrow [\![s]\!]_\ell$.

Definition 6.1 (Authenticated secret-sharing)**.** *Let $s \in \mathbb{Z}/2^\ell\mathbb{Z}$. We say that the parties hold authenticated shares of s if the following holds.*

Additive shares of key. *Each P_i has a global*[2] *uniformly random value $\alpha_i \in_R \mathbb{Z}/2^\ell\mathbb{Z}$. Let $\alpha = \sum_{i=1}^n \alpha_i \bmod 2^\ell$.*

Additive shares of s. *Each P_i has $s_i \in \mathbb{Z}/2^\ell\mathbb{Z}$ such that $\sum_{i=1}^n s_i \equiv_\ell s$*

Additive shares of $\alpha \cdot s$. *Each P_i has $\gamma_i \in \mathbb{Z}/2^\ell\mathbb{Z}$ such that*

$$\sum_{i=1}^n \gamma_i \equiv_\ell \left(\sum_{i=1}^n s_i\right) \cdot \left(\sum_{i=1}^n \alpha_i\right).$$

The above can be denoted as $\langle s \rangle_\ell = ([\![s]\!]_\ell, [\![\alpha \cdot s]\!]_\ell, [\![\alpha]\!]_\ell)$. When clear from context, we will omit the ℓ subindex.

Observe that this scheme is essentially the same as the one presented in Section 2.7.2 in the context of finite fields, except that, unfortunately, this scheme will not ensure that the adversary cannot cheat in the reconstruction of the shared value $\langle s \rangle$, causing the parties to reconstruct a result s' with $s' \not\equiv_\ell s$. This makes this construction insufficient to obtain MPC protocols over $\mathbb{Z}/2^\ell\mathbb{Z}$, similar to the reason why the passively secure protocol from Section 2.6 is not actively secure: the adversary can cheat in the output gates—and also in the multiplication gates—by reconstructing values incorrectly. However, as we will soon see, by setting $\ell = k + \kappa$ we will be able to show that the adversary cannot cause the reconstructed value s' to satisfy $s' \not\equiv_k s$ with probability better than (roughly) $2^{-\kappa}$. That is, even though it could be the case that $s' \not\equiv_{k+\kappa} s$, it will hold with overwhelming probability that $s' \equiv_k s$, or, in other words, the lower k bits of s will be correct upon reconstruction. This turns out to be pivotal for the construction of an MPC protocol over $\mathbb{Z}/2^k\mathbb{Z}$.

Local Operations. The parties can make use of the basic homomorphic properties of additive secret-sharing $[\![\cdot]\!]_\ell$ to perform local operations with the scheme $\langle \cdot \rangle_\ell$. More precisely, given $\langle x \rangle_\ell = ([\![x]\!]_\ell, [\![\alpha \cdot x]\!]_\ell, [\![\alpha]\!]_\ell)$ and $\langle y \rangle_\ell = ([\![y]\!]_\ell, [\![\alpha \cdot y]\!]_\ell, [\![\alpha]\!]_\ell)$ and a publicly known value $c \in \mathbb{Z}/2^\ell\mathbb{Z}$, the parties can locally compute $\langle x \pm y \rangle$, $\langle c \cdot x \rangle$ and $\langle x \pm c \rangle$ as follows:

- $\langle x \pm y \rangle = ([\![x]\!] \pm [\![y]\!],\ [\![\alpha \cdot x]\!] \pm [\![\alpha \cdot y]\!],\ [\![\alpha]\!])$
- $\langle c \cdot x \rangle = (c \cdot [\![x]\!],\ c \cdot [\![\alpha \cdot x]\!],\ [\![\alpha]\!])$

[2] The term *global* refers to the fact that this part of the sharings do not change from one shared value to another.

- $\langle x \pm c \rangle = (\llbracket x \rrbracket \pm c,\ \llbracket \alpha \cdot x \rrbracket \pm \llbracket \alpha \rrbracket \cdot c,\ \llbracket \alpha \rrbracket)$

6.3.1 Simultaneous Broadcast

Before we dive into the description of the method for the parties to reconstruct secret-shared values while ensuring the lower k bits are correct, we enhance the broadcast channel assumed by default into what we call a *simultaneous broadcast channel*. This is used in a scenario where each party P_i is supposed to broadcast a value to the other parties. To this end, the parties could make use of the underlying broadcast channel n times, one for each sender P_i. However, in the context within our protocol where all parties need to broadcast data, it is imperative that the corrupt parties do not see the values broadcasted by honest parties before they broadcast theirs, and the approach above does not prevent this since the adversary, which is ultimately modeled by the environment in the setting of UC security, can schedule the calls to the broadcast channel in any order.

A simultaneous broadcast channel is designed precisely to disallow the behavior described above. In such a channel each party broadcasts its given value "at the same time", without seeing the values sent by other parties. This is formalized by means of a functionality that operates as follows:

Simultaneous Broadcast Functionality

1. It receives a value x_i from each party P_i.
2. Once all parties have provided their input, the functionality sends $(x_1, \ldots, x_n)$ to all parties.

In order to instantiate a simultaneous broadcast channel assuming only access to the usual broadcast channel where each party individually can broadcast a value separately, we can follow a rather standard approach in the literature known as "commit-and-open", which is already briefly discussed in Section 2.7.2.1. In a nutshell, this consists of the parties first broadcasting a *commitment* to each of their to-be-broadcasted messages, which is a bit-string that "binds" the parties to the specific value they wish to send, followed by a round of the parties broadcasting the actual messages, which is only executed after all the commitments have been received. This way, the parties can verify that the broadcasted messages are consistent with the commitments sent earlier, aborting if this is not the case. As a result, the corrupt parties cannot change their message to be broadcasted after they have sent the commitment, and the only information they have possibly seen before sending their message are the commitments from the other parties, which leak nothing about the actual messages.

We refer the reader to Section 2.7.2.1 for a bit more detailed description on this method. However, for the purposes of the current section the formalization of the "commit-and-open" technique as the simultaneous broadcast functionality from above suffices, and this is the approach we will follow in our protocols and their proofs.

6.3.2 Reconstructing Shared Values

Now we show that, given a shared value $\langle s \rangle_{k+\kappa} = (\llbracket s \rrbracket_{k+\kappa}, \llbracket \alpha \cdot s \rrbracket_{k+\kappa}, \llbracket \alpha \rrbracket_{k+\kappa})$, the adversary cannot cause the reconstruction of this value to lead to $s' \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ with $s' \not\equiv_k s$, except with probability $2^{-\kappa}$. This is done via the procedure π_{Open} below. In this chapter, a *procedure*, in contrast to a *protocol*, is an algorithm executed by the parties that will be plugged in later on in an actual protocol. A useful analogy is to think of procedures as macros in programming languages such as C/C++, and protocols as `include` statements.

Procedure π_{Open}

Input: Shared value $\langle s \rangle_{k+\kappa} = (\llbracket s \rrbracket_{k+\kappa}, \llbracket \alpha \cdot s \rrbracket_{k+\kappa}, \llbracket \alpha \rrbracket_{k+\kappa})$.
Output: The parties either learn s' with $s' = s + \delta$ for some additive error δ chosen by the adversary with $\delta \equiv_k 0$, or the parties abort.
Procedure: Let $\llbracket s \rrbracket = (s_1, \ldots, s_n)$.

1. Each party P_i broadcasts s_i to the other parties. Upon receiving these values each party computes $s' = \sum_{i=1}^{n} s_i' \bmod 2^{k+\kappa}$, where s_i' is the actual value broadcasted by P_i.[a]
2. The parties compute locally $\llbracket z \rrbracket_{k+\kappa} \leftarrow \llbracket \alpha \cdot s \rrbracket - s' \llbracket \alpha \rrbracket$. Let $\llbracket z \rrbracket = (z_1, \ldots, z_n)$.
3. Each party P_i simultaneously-broadcasts z_i to the other parties.
4. The parties compute $z = \sum_{i=1}^{n} z_i' \bmod 2^{k+\kappa}$, where z_i' is the actual value broadcasted by P_i. If $z \not\equiv_{k+\kappa} 0$ then the parties abort.

[a] This can be optimized by asking each party P_i to send s_i to P_1 and then asking P_1 to broadcast $s' = \sum_{i=1}^{n} s_i' \bmod 2^{k+\kappa}$.

Proposition 6.2. *Let $s' \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ be the value computed by the parties in the first step of the procedure above. If the parties do not abort in the execution of π_{Open}, then $s' \equiv_k s$, except with probability $2^{-(\kappa+1)}$.*

Proof. For $i \in [n]$ let us write $s_i' = s_i + \delta_i$, where $\delta_i = 0$ for $i \in \mathcal{H}$. Then $s' = \sum_{i=1}^{n} s_i + \sum_{i=1}^{n} \delta_i = s + \delta \bmod 2^{k+\kappa}$. Now, if we write $z_i' = z_i + \epsilon_i$, where $\epsilon_i = 0$ for $i \in \mathcal{H}$, we have that the value reconstructed by the parties in the final step of the protocol is

$$z' = \sum_{i=1}^{n} z_i + \sum_{i=1}^{n} \epsilon_i = z + \epsilon = (\alpha s - s'\alpha) + \epsilon = -\delta \cdot \alpha + \epsilon \mod 2^{k+\kappa}.$$

Now, to see the claim we prove the counter-positive, namely, if $s' \not\equiv_k s$, which is equivalent to $\delta \not\equiv_k 0$, then the parties abort, or more precisely, $z' \not\equiv_{k+\kappa} 0$, except with probability at most $2^{-(\kappa+1)}$. To see this, observe from the above that $z' \equiv_{k+\kappa} 0$ if and only if $\delta \cdot \alpha \equiv_{k+\kappa} \epsilon$. Let v be the largest integer such that $2^v \mid \delta$, which, assuming that $\delta \not\equiv_k 0$, satisfies $v \leq k-1$. We have that $\alpha \cdot \left(\frac{\delta}{2^v}\right) \equiv_{k+\kappa-v} \frac{\epsilon}{2^v}$, and since $\delta/2^v \in (\mathbb{Z}/2^{k+\kappa-v}\mathbb{Z})^*$, we obtain $\alpha \equiv_{k+\kappa-v} \left(\frac{\delta}{2^v}\right)^{-1} \left(\frac{\epsilon}{2^v}\right)$. In particular, since $k + \kappa - v \geq \kappa + 1$, we have that $\alpha \equiv_{\kappa+1} \left(\frac{\delta}{2^v}\right)^{-1} \left(\frac{\epsilon}{2^v}\right)$.

Recall that $\alpha \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ is uniformly random and unknown to the adversary, δ is chosen by the adversary independently of α and ϵ is chosen independently of $z = -\delta \cdot \alpha$ because of the simultaneous broadcast. As a result, the probability that $\alpha \equiv_{\kappa+1} \left(\frac{\delta}{2^v}\right)^{-1} \left(\frac{\epsilon}{2^v}\right)$ holds is at most $2^{-(\kappa+1)}$. □

Finally, we also consider a similar procedure $\pi_{\text{Open.Ind}}(\langle s \rangle_{k+\kappa}, i)$ which, instead of asking each party broadcast their data, each party sends these values to party P_i, who performs the necessary checks internally.

Remark 6.1 (Privacy of the opening procedure π_{Open})**.** *When we make use of the opening procedure in our main protocol from Section 6.4, it will be crucial that, when reconstructing a shared value $\langle s \rangle_{k+\kappa}$, only $s \bmod 2^k$ is leaked, and in particular, nothing about the upper κ bits of s should be learned by the adversary. However, as described above, the procedure π_{Open} leaks all of s since the parties announce to each other their additive shares of s modulo $2^{k+\kappa}$.*

To address this issue, the procedure π_{Open} will be used in our main protocol to reconstruct not a shared value $\langle s \rangle$, but rather $\langle s \rangle + 2^k \langle r \rangle$ where r is a uniformly random value. This shared value has the same lower k bits as s, but the upper κ bits are uniformly random and reveal nothing to the adversary.

6.4 Instantiating $\mathcal{F}_{\text{PABB}}$ with Preprocessing

Now we make use of the authenticated secret-sharing scheme with homomorphism over $\mathbb{Z}/2^k\mathbb{Z}$ presented in the previous section to instantiate the functionality $\mathcal{F}_{\text{PABB}}$. Recall that this functionality handles the commands (random), (triple), (open.ind), (comb) and (open). The authenticated secret-sharing scheme $\langle \cdot \rangle$ will be used to instantiate the behavior of "storing" values, and its homomorphism modulo 2^k is used to instantiate the command (comb). Furthermore, the opening procedures from Section 6.3.2 will be useful for the (open) and (open.ind) commands.

In the original work of [32] the (random) and (triple) commands are instantiated using Oblivious Transfer, by adapting the techniques from the dishonest majority protocol MASCOT [62] over fields to the $\mathbb{Z}/2^k\mathbb{Z}$ setting. In this thesis we do not include the instantiation of these commands. Instead, we define a functionality $\mathcal{F}_{\text{Prep}}$ that instantiates these commands in the context of our authenticated secret-sharing scheme $\langle \cdot \rangle$. Observe that the data generated by the commands (random) and (triple) is actually independent of the inputs the parties provide, which means they can be generated in a *preprocessing phase*, before the inputs to the computation are known. As pointed out above, an instantiation of the $\mathcal{F}_{\text{Prep}}$ functionality is presented in the original work of [32].

Now we describe the $\mathcal{F}_{\text{Prep}}$ functionality.

Functionality $\mathcal{F}_{\text{Prep}}$

The functionality proceeds as follows.

- On input (init) from all the parties, receive $\{\alpha_i\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary, then sample $\alpha_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ and send this value to P_i for $i \in \mathcal{H}$. Let $\alpha = \left(\sum_{i=1}^n \alpha_i\right) \bmod 2^{k+\kappa}$.
- On input (random) from all the parties, proceed as follows.
  1. Receive $\{r_i\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary, then sample $r_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ and send this value to P_i for $i \in \mathcal{H}$.
  2. Let $r = \left(\sum_{i=1}^n r_i\right) \bmod 2^{k+\kappa}$. Receive $\{\gamma_i^{(r)}\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary, then sample $\gamma_i^{(r)} \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$ constrained to $\alpha \cdot r \equiv_{k+\kappa} \sum_{i=1}^n \gamma_i^{(r)}$ and, for each $i \in \mathcal{H}$, send $\gamma_i^{(r)}$ to P_i.
- On input (triple) from all the parties, proceed as follows.
  1. Receive $\{(a_i, b_i, c_i)\}_{i\in\mathcal{C}}$ from the adversary with $a_i, b_i, c_i \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$.
  2. Sample $(a_i, b_i) \in_R (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^2$ for $i \in \mathcal{H}$ and let $a = \left(\sum_{i=1}^n a_i\right) \bmod 2^{k+\kappa}$ and $b = \left(\sum_{i=1}^n b_i\right) \bmod 2^{k+\kappa}$. Sample $c_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$ uniformly at random constrained to $c \equiv_{k+\kappa} \sum_{i=1}^n c_i$, where $c = a \cdot b \bmod 2^{k+\kappa}$.
  3. Receive $\{(\gamma_i^{(a)}, \gamma_i^{(b)}, \gamma_i^{(c)})\}_{i\in\mathcal{C}} \subseteq (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^3$ from the adversary, then sample $(\gamma_i^{(a)}, \gamma_i^{(b)}, \gamma_i^{(c)}) \in_R (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^3$ for $i \in \mathcal{H}$ constrained to $\alpha \cdot a \equiv_{k+\kappa} \sum_{i=1}^n \gamma_i^{(a)}$, $\alpha \cdot b \equiv_{k+\kappa} \sum_{i=1}^n \gamma_i^{(b)}$ and $\alpha \cdot c \equiv_{k+\kappa} \sum_{i=1}^n \gamma_i^{(c)}$
  4. For each $i \in \mathcal{H}$, send $(a_i, \gamma_i^{(a)})$, $(b_i, \gamma_i^{(b)})$ and $(c_i, \gamma_i^{(c)})$ to P_i.

Observe the following:

- When the parties call the command (init), they get additive shares $[\![\alpha]\!]_{k+\kappa}$, where $\alpha \bmod 2^k$ is uniformly random in $\mathbb{Z}/2^{k+\kappa}\mathbb{Z}$.
- When the parties call the command (rand), they get authenticated shares $\langle r \rangle$, where $r \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ is uniformly random.
- When the parties call the command (mult), they get authenticated shares $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, where $a, b \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ are uniformly random and $c = a \cdot b \bmod 2^{k+\kappa}$.

With the preprocessing functionality $\mathcal{F}_{\text{Prep}}$ at hand, we can now instantiate the functionality $\mathcal{F}_{\text{PABB}}$, which, as shown in Section 6.2, is enough to instantiate the arithmetic black box functionality $\mathcal{F}_{\text{ABB}}$. The protocol Π_{PABB} below instantiates the functionality $\mathcal{F}_{\text{PABB}}$ with statistical security in the $\mathcal{F}_{\text{Prep}}$-hybrid model. As we have already mentioned, the main idea behind the protocol is to use the authenticated secret-sharing scheme $\langle\cdot\rangle$ to instantiate the storing capabilities of $\mathcal{F}_{\text{PABB}}$ together with the (comb), (open) and (open.ind) commands.

Protocol Π_{PABB}

The parties begin by calling $\mathcal{F}_{\text{Prep}}$ on input (init). Then:

- On input (random, id), the parties send (rand) to $\mathcal{F}_{\text{Prep}}$, obtaining $\langle r \rangle$. The parties store internally (id, $\langle r \rangle$).

- On input $(\text{triple}, \text{id}_1, \text{id}_2, \text{id}_3)$, the parties send (triple) to $\mathcal{F}_{\text{Prep}}$, obtaining $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$. The parties store internally $(\text{id}_1, \langle a\rangle)$, $(\text{id}_2, \langle b\rangle)$ and $(\text{id}_3, \langle c\rangle)$.
- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$, the parties retrieve $(\text{id}_i, \langle x_i\rangle)$ for $i \in [\ell]$ from memory and store $(\text{id}_{\ell+1}, \langle z\rangle)$, where $\langle z\rangle \leftarrow c_0 + \sum_{i=1}^{\ell} c_i \langle x_i\rangle$.
- On input $(\text{open.ind}, \text{id}, i)$ the parties retrieve $(\text{id}, \langle x\rangle)$ from memory and do the following:
  1. Send (random) to $\mathcal{F}_{\text{Prep}}$ to get $\langle r_x\rangle$.
  2. Execute the procedure $\pi_{\text{Open.Ind}}(\langle z\rangle, i)$, with $\langle z\rangle \leftarrow \langle x\rangle + 2^k \cdot \langle r_x\rangle$.
- On input (open, id) the parties retrieve $(\text{id}, \langle x\rangle)$ from memory and do the following:
  1. Send (rand) to $\mathcal{F}_{\text{Prep}}$ to get $\langle r_x\rangle$.
  2. Execute the procedure $\pi_{\text{Open}}(\langle z\rangle, i)$, with $\langle z\rangle \leftarrow \langle x\rangle + 2^k \cdot \langle r_x\rangle$.

Theorem 6.1. *Protocol Π_{PABB} instantiates the functionality $\mathcal{F}_{\text{PABB}}$ with statistical security in the $\mathcal{F}_{\text{Prep}}$-hybrid model.*

Proof. We define the simulator S as follows:

S begins by emulating $\mathcal{F}_{\text{Prep}}$ on input (init) by receiving $\{\overline{\alpha}_i\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary and sampling $\overline{\alpha}_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$. Let $\overline{\alpha} = \left(\sum_{i=1}^{n} \alpha_i\right) \bmod 2^{k+\kappa}$.

- Upon receiving $(\text{random}, \text{id})$ from $\mathcal{F}_{\text{PABB}}$, S proceeds as follows:
  1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Prep}}$ on input (rand) by receiving $\{\overline{r}_i\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary, then sampling $\overline{r}_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$.
  2. Let $\overline{r} = \left(\sum_{i=1}^{n} \overline{r}_i\right) \bmod 2^{k+\kappa}$, and store $(\text{id}, \langle\overline{r}\rangle)$ in memory. Receive $\{\overline{\gamma}_i^{(r)}\}_{i\in\mathcal{C}} \subseteq \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ from the adversary, then sample $\overline{\gamma}_i^{(r)} \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$ constrained to $\overline{\alpha}\cdot\overline{r} \equiv_{k+\kappa} \sum_{i=1}^{n} \overline{\gamma}_i^{(r)}$ and, for each $i \in \mathcal{H}$, send $\overline{\gamma}_i^{(r)}$ to P_i.
- On input $(\text{triple}, \text{id}_1, \text{id}_2, \text{id}_3)$ from $\mathcal{F}_{\text{PABB}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{Prep}}$ on input (triple) as follows.
  1. Receive $\{(\overline{a}_i, \overline{b}_i, \overline{c}_i)\}_{i\in\mathcal{C}}$ from the adversary with $\overline{a}_i, \overline{b}_i, \overline{c}_i \in \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$.
  2. Sample $(\overline{a}_i, \overline{b}_i) \in_R (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^2$ for $i \in \mathcal{H}$ and let $\overline{a} = \left(\sum_{i=1}^{n} \overline{a}_i\right) \bmod 2^{k+\kappa}$ and $\overline{b} = \left(\sum_{i=1}^{n} \overline{b}_i\right) \bmod 2^{k+\kappa}$. Sample $\overline{c}_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$ for $i \in \mathcal{H}$ uniformly at random constrained to $\overline{c} \equiv_{k+\kappa} \sum_{i=1}^{n} \overline{c}_i$, where $\overline{c} = \overline{a}\cdot\overline{b} \bmod 2^{k+\kappa}$.
  3. Receive $\{(\overline{\gamma}_i^{(\overline{a})}, \overline{\gamma}_i^{(\overline{b})}, \overline{\gamma}_i^{(\overline{c})})\}_{i\in\mathcal{C}} \subseteq (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^3$ from the adversary, then sample $(\overline{\gamma}_i^{(\overline{a})}, \overline{\gamma}_i^{(\overline{b})}, \overline{\gamma}_i^{(\overline{c})}) \in_R (\mathbb{Z}/2^{k+\kappa}\mathbb{Z})^3$ for $i \in \mathcal{H}$ constrained to $\overline{\alpha}\cdot\overline{a} \equiv_{k+\kappa} \sum_{i=1}^{n} \overline{\gamma}_i^{(\overline{a})}$, $\overline{\alpha}\cdot\overline{b} \equiv_{k+\kappa} \sum_{i=1}^{n} \overline{\gamma}_i^{(\overline{b})}$ and $\overline{\alpha}\cdot\overline{c} \equiv_{k+\kappa} \sum_{i=1}^{n} \overline{\gamma}_i^{(\overline{c})}$
  4. Store $(\text{id}_1, \langle\overline{a}\rangle)$, $(\text{id}_1, \langle\overline{b}\rangle)$ and $(\text{id}_1, \langle\overline{c}\rangle)$ in memory.
- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1})$ from $\mathcal{F}_{\text{PABB}}$, $\mathcal{S}$ retrieves $(\text{id}_i, \langle x_i\rangle)$ for $i \in [\ell]$ from memory and store $(\text{id}_{\ell+1}, \langle z\rangle)$, where $\langle z\rangle \leftarrow c_0 + \sum_{i=1}^{\ell} c_i \langle x_i\rangle$.
- On input $(\text{open}, \text{id}, i)$ and x from $\mathcal{F}_{\text{PABB}}$, $\mathcal{S}$ retrieves $(\text{id}, \langle x\rangle)$ from memory and then proceeds as follows:
  1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Prep}}$ on input (random) by receiving $\{(\overline{r}_i, \overline{\gamma}_i^{(\overline{r})})\}_{i\in\mathcal{C}}$ from the adversary, then sampling $r_i \in_R \mathbb{Z}/2^{k+\kappa}\mathbb{Z}$. Let $r = \left(\sum_{i=1}^{n} r_i\right) \bmod 2^{k+\kappa}$.

2. $\mathcal{S}$ samples virtual honest parties' shares of $\langle x \rangle$ so that the underlying secret is x, then computes $\langle z \rangle \leftarrow \langle x \rangle + 2^k \langle r \rangle$ and emulates the execution of the procedure $\pi_{\text{Open.Ind}}(\langle z \rangle, i)$. If the execution results in abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{PABB}}$.

- On input (open.ind, id, i)
  - If $i \in \mathcal{C}$, $\mathcal{S}$ retrieves (id, $\langle x \rangle$) from memory and proceeds as above
  - If $i \notin \mathcal{C}$, $\mathcal{C}$ emulates $\mathcal{F}_{\text{Prep}}$ on input (random) as above, and receives data from the adversary as the emulation of $\pi_{\text{Open.Ind}}$. If the execution results in abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{PABB}}$.

The indistinguishability argument for all the commands except (open) and (open.ind) is straightforward, given that the simulator simply executes the exact same steps in the emulation of these as in the real world. The most sensitive commands are (open) and (open.ind). In these two, $\mathcal{S}$ emulates the execution in the same way as in the real world, except that, in the ideal world, the parties get the correct values stored by $\mathcal{F}_{\text{PABB}}$, while in the real world they get the result of the reconstruction. However, thanks to Proposition 6.2, if the parties do not abort in the execution of the procedures π_{Open} and $\pi_{\text{Open.Ind}}$, then the parties reconstruct the same secret as the one stored by $\mathcal{F}_{\text{PABB}}$, except with probability $2^{-\kappa}$. □

Chapter 7

Primitives for Secure Computation using edaBits

The final chapter of this thesis is devoted to exploring some generic primitives for extending the range of applications of secure computation protocols. Most relevant applications of secure multiparty computation are not restricted to simple additions and multiplications over a ring, and there are many slightly more advanced operations that are common across many applications. For example, in settings involving real-valued arithmetic such as scientific computing and analysis of continuous data, either fixed or floating-point arithmetic must be used, which enable the emulation of an infinite domain such as the real numbers using bounded-machine registers, such as these in computer architectures and also the ring elements used in multiparty computation protocols. Another example includes secure integer comparison, which is useful in contexts where an action, like branching, must be taken depending on the "size" of a given value.

Given the importance and recurring use of these different primitives, it becomes a highly relevant task to design efficient subprotocols to securely compute these operations, given that these can be later used in a "plug-and-play" manner in larger application contexts. There are simply too many operations that one could consider relevant across many settings, but fortunately, many of these can be associated to *mixed computation*. In this context, different operations over integers like addition, subtraction or multiplication are required, but other operations at the level of the *bits* of the given integers are also needed, together with conversions between the arithmetic and binary "worlds". To illustrate the usefulness of mixed computation, note the operation of integer comparison can be phrased as an operation at the level of bits that compares the bit-decomposition of the two input integers and decides which of the two is the largest. In fact, since any computation can be written as an arithmetic circuit over a field, in particular every computation can be written as a binary circuit (which is an arithmetic circuit over $\mathbb{F}_2 = \{0, 1\}$), but most importantly, many relevant operations are *naturally* written as binary circuits.

The approach considered in the original work of [50] in order to enable operations beyond basic additions and multiplications consists of preprocessing some type of secret-shared data, called *edaBits*, which stands for *Extended Doubly-Authenticated Bits*, and using this correlation to securely compute more advanced primitives in the online phase of the protocol execution. An m-bit edaBit is a tuple of secret-shared values of the form $(\llbracket r \rrbracket, \llbracket r[m-1] \rrbracket_2, \ldots, \llbracket r[0] \rrbracket_2)$, where r is an m-bit uniformly random integer and $(r[m-1], \ldots, r[0])$ is its bit decomposition. Here $\llbracket \cdot \rrbracket$ is a secret-sharing scheme for modular integers, and $\llbracket \cdot \rrbracket_2$ is a secret-sharing scheme for bits, concepts that will be formalized

later on in this chapter by means of an arithmetic black box (ABB) functionality. The main remark to be made at this early stage is that edaBits are extremely powerful and flexible, not requiring any assumption about the underlying secret-sharing scheme or the different methods to instantiate the ABB, and only relying on the core basic operations like additions and multiplications, together with the ability of opening secret-shared values. From this, edaBits can be used in a wide variety of settings like $t < n/3$, $t < n/2$, $t < n$, passive/active security, or even more complex scenarios, as long as the ABB can be instantiated.

The original work of [50], in addition to introducing the concept of edaBits and some of their potential applications, also presents a generic approach to preprocessing these objects in the dishonest majority setting, without making any assumption whatsoever about the underlying secure computation protocol beyond the ability to perform basic operations such as additions and multiplications. We chose to exclude this contribution from this thesis with the goal of keeping focus on the use and applications of edabits themselves rather than their generation.[1] We also remark that, in the original work of [36], we present a full implementation of our techniques in the MP-SPDZ framework [61], together with a wide range of thorough experiments, micro-benchmarks and applications to privacy-preserving machine learning. These are also excluded from this thesis.

Organization of this chapter. First, in Section 7.1 we introduce the functionality $\mathcal{F}_{\text{MABB}}$, which extends the standard arithmetic black-box model we have previously used to support conversions between binary and arithmetic shared values; this will be crucial for the applications of *edaBits* we discuss later in the chapter, and can be easily instantiated from the standard $\mathcal{F}_{\text{ABB}}$ model by means of *daBits* [73]. Then, in Section 7.2 we formally introduce the concept of edaBits, and, even though we do not show how to fully generate them as this is very context dependent, we do show in Section 7.2.1 how to produce edaBits from what we call *private edaBits*, which are simply edaBits where the secret is known by some party. Finally, in Section 7.3 we present the different advanced primitives that can be securely computed with the help of edaBits.

We remark that, as we have mentioned already, the results of this chapter are based on the original work of [50], and as such, **parts of the text presented here are taken verbatim from that work**.

7.1 Mixed Arithmetic Black Box

In this section we define the core functionality that will support the rest of the operations on top. The mixed arithmetic black box model, or MABB for short, is like the arithmetic black box model we have used before (for example in Section 6.1), except it is expanded to admit two types of values to be stored: "large" integers and bits. The MABB model,

[1] We recall that we did discuss in Section 5.3.7.4 the generation of edaBits in the context of four-party computation. This was done because the protocol, which is set in the context of $t < n/3$ with a small number of parties, had a simple design and was simple to describe. The generation of edaBits presented in the original work of [50] for the dishonest majority setting follows a much more cumbersome design, making use of cut-and-choose techniques and an extensive and highly non-trivial probability analysis.

formalized as functionality $\mathcal{F}_{\text{MABB}}$ below, also supports basic conversion between the two types.

Functionality $\mathcal{F}_{\text{MABB}}$: Mixed Arithmetic Black Box

The functionality proceeds as follows. Below, type is a flag that equals either arithmetic or binary.

- On input $(\text{input}, \text{id}, \text{type}, i)$ from the honest parties, send $(\text{input}, \text{id}, \text{type}, i)$ to the adversary, wait for input $(\text{value}, \text{id}, \text{type}, x)$ from party P_i, where $x \in \mathbb{Z}/2^k\mathbb{Z}$ if type $=$ arithmetic and $x \in \mathbb{Z}/2\mathbb{Z}$ if type $=$ binary, and then store $(\text{id}, \text{type}, x)$ in memory.
- On input $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1}, \text{type})$ from the honest parties, retrieve $(\text{id}_i, x_i, \text{type})$ for $i \in [\ell]$ from memory and store $(\text{id}_{\ell+1}, \text{type}, z)$, where $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2^k$ if type $=$ arithmetic, and $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2$ if type $=$ binary. Then send $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i\in[\ell]}, \text{id}_{\ell+1}, \text{type})$ to the adversary.
- On input $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3, \text{type})$ from the honest parties, retrieve $(\text{id}_1, \text{type}, x)$ and $(\text{id}_2, \text{type}, y)$ from memory and store $(\text{id}_3, \text{type}, z)$, where $z = x \cdot y \bmod 2^k$ if type $=$ arithmetic, and $z = x \cdot y \bmod 2$ if type $=$ binary. Then send $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3, \text{type})$ to the adversary.
- On input $(\text{open}, \text{id}, \text{type})$ from the honest parties, retrieve $(\text{id}, \text{type}, x)$ from memory and send x to all the parties. Then send $(\text{open}, \text{id}, \text{type})$ to the adversary.
- On input $(\text{convert}, \text{id}_1, \text{id}_2, \text{type})$ from the honest parties, retrieve $(\text{id}, \text{type}, x)$ from memory and:
  - If type $=$ arithmetic, store $(\text{id}, \text{binary}, x \bmod 2)$ in memory.
  - If type $=$ binary, store $(\text{id}, \text{arithmetic}, x)$ in memory.

  Then send $(\text{convert}, \text{id}_1, \text{id}_2, \text{type})$ to the adversary.

All of our results below are set in the $\mathcal{F}_{\text{MABB}}$-hybrid model, which in practice means that we assume the existence of a secure multiparty computation protocol based on secret-sharing for the given security model (passive/active security, honest/dishonest majority, etc.). This could be instantiated for example using some of the protocols we have studied so far in this work.

Since in practice $\mathcal{F}_{\text{MABB}}$ is instantiated using a linear secret-sharing scheme, as we have seen in multiple constructions before in this thesis. Motivated by this, whenever a value $(\text{id}, \text{type}, x)$ is stored (the use of the ID is implicit), we denote this by $[\![x]\!]$ if type $=$ arithmetic and $[\![x]\!]_2$ if type $=$ binary, and we say that x is *secret-shared*. We also use the following notation:

- $[\![x]\!] \leftarrow [\![x]\!]_2$ whenever $(\text{convert}, \text{id}_1, \text{id}_2, \text{binary})$ is called (with x stored as $(\text{id}_1, \text{binary}, x)$).
- $[\![x]\!]_2 \leftarrow [\![x]\!]$ whenever $(\text{convert}, \text{id}_1, \text{id}_2, \text{arithmetic})$ is called (with x stored as $(\text{id}_1, \text{arithmetic}, x)$).
- $[\![z]\!] \leftarrow c_0 + \sum_{i=1}^{\ell} c_i [\![x_i]\!]$ whenever $(\text{comb}, \{c_i\}_{i=0}^{\ell}, \{\text{id}_i\}_{i=1}^{\ell}, \text{id}_{\ell+1}, \text{arithmetic})$ is called (with x_i stored as $(\text{id}_i, \text{arithmetic}, x_i)$ for $i \in [\ell]$); we define in a similar way $[\![z]\!]_2 \leftarrow c_0 + \sum_{i=1}^{\ell} c_i [\![x_i]\!]_2$

- $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ whenever $(\text{mult}, \text{id}_1, \text{id}_2, \text{id}_3, \text{arithmetic})$ is called, and similarly $[\![z]\!]_2 \leftarrow [\![x]\!]_2 \cdot [\![y]\!]_2$.

7.1.1 Conversions using DaBits

The only "non-standard" command in $\mathcal{F}_{\text{MABB}}$ is the conversion between arithmetic and binary secret-shared values. These can be instantiated from certain type of preprocessed data called *daBits*, which was proposed initially in [73]. As the name suggests, edaBits constitute a generalization of daBits, and using our terminology the latter can be seen a 1-bit edaBits. More precisely, a daBit is a pair $([\![r]\!], [\![r]\!]_2)$, where $r \in_R \{0,1\}$ is uniformly random and unknown to any party.

Conversions. Given $[\![b]\!]_2$, the parties can obtain $[\![b]\!]$ with the help of a daBit $([\![r]\!], [\![r]\!]_2)$ by executing the following:

1. Reconstruct $d \leftarrow [\![r]\!]_2 \oplus [\![b]\!]_2$;
2. Compute locally $[\![b]\!] \leftarrow d + [\![r]\!] - 2d\,[\![r]\!]$

Given $[\![b]\!]$, where $b \in \{0,1\}$, in the context of secret-sharing, the parties can typically obtain $[\![b]\!]_2$ *locally* by reducing their shares modulo 2. Alternatively, they can make use of a daBit in a similar way as above.

Generating a daBit. A generic method to obtain a daBit, consists in asking each party P_i for $i \in [t+1]$ to sample a uniformly random bit $r_i \in \{0,1\}$, followed by the parties executing the $(\text{input}, *, \text{arithmetic}, i)$ and $(\text{input}, *, \text{binary}, i)$ commands so that they obtain $([\![r_i]\!]_2, [\![r_i]\!])$. Once this is done, the parties can compute $[\![r]\!] \leftarrow \bigoplus_{i=1}^{t+1} [\![r_i]\!]$ and $[\![r]\!]_2 \leftarrow \sum_{i=1}^{t+1} [\![r_i]\!]_2$ (with the $\oplus$ operator, denoting XOR, computed arithmetically as $a \oplus b = a + b - 2ab$). This results in a uniformly random bit r given that there is one honest party among the parties $P_1, \ldots, P_{t+1}$ who is guaranteed to provide a uniformly random bit r_i, so the final bit $r = \bigoplus_{i=1}^{t+1} r_i$ will also be uniformly random.

The above assumes that every party's value r_i lies in $\{0,1\}$, which may not be the case for an actively corrupt party for the arithmetic sharing $[\![r_i]\!]$. In this case, the adversary can introduce some values $r_i \notin \{0,1\}$, so the value computed $[\![r]\!] \leftarrow \bigoplus_{i=1}^{t+1} [\![r_i]\!]$ may not lie in $\{0,1\}$, given that $a + b - 2ab$ is only guaranteed to be in $\{0,1\}$ if both a and b are in this set. To ensure that each party secret-shares a bit, the parties can execute the following simple protocol. For each secret-shared $[\![r_i]\!]$, the parties call $[\![z]\!] \leftarrow [\![r_i]\!] \cdot [\![r_i]\!] - [\![r_i]\!]$, and open z. If $z = 0$, then the parties determine that $r_i \in \{0,1\}$, else they abort. This works given that $x^2 - x \equiv_k 0$ if and only if $x \in \{0,1\}$.

7.2 Extended Double-Authenticated Bits

Now we present the formal definition of edaBits. These are, with the terminology we have introduced, a tuple $(\llbracket r \rrbracket, \llbracket r[m-1] \rrbracket_2, \ldots, \llbracket r[0] \rrbracket_2)$, where $r[i] \in_R \{0,1\}$ for $i \in \{0,\ldots,m-1\}$ and $r = \sum_{i=0}^{m-1} r[i]2^i$. This is formalized by means of a functionality $\mathcal{F}_{\text{edaBits}}$, presented below, that supports all the commands available to $\mathcal{F}_{\text{MABB}}$, but in addition enables a command for producing edaBits. This functionality is presented below.

Functionality $\mathcal{F}_{\text{edaBits}}$

The functionality includes the exact same commands as $\mathcal{F}_{\text{MABB}}$. In addition, it supports the following:

- On input $(\text{edabit}, \{\text{id}_\ell\}_{\ell=0}^m, m)$ from the honest parties, sample $r \in_R \mathbb{Z}/2^m\mathbb{Z}$, bit-decompose this value as $(r[m-1],\ldots,r[0]) \in \{0,1\}^m$ and store in memory $(\text{id}_0, \text{arithmetic}, r)$ and $(\text{id}_{\ell+1}, \text{binary}, r[\ell])$ for $\ell \in [m]$. Then send $(\text{edabit}, \{\text{id}_\ell\}_{\ell=0}^m, m)$ to the adversary.

7.2.1 Global EdaBits from Private EdaBits

As we have already mentioned at the beginning of this chapter, our main goal is to focus on the applications of edaBits, rather than discussing how to generate these. However, before we dive into these applications, we discuss an alternative "version" of edaBits which is potentially simpler to produce, and from which "normal" edaBits can be obtained. These simpler edabits consist of shared values $(\llbracket r_j \rrbracket, \llbracket r_j[m-1] \rrbracket_2, \ldots, \llbracket r_j[0] \rrbracket_2)$, where $r_j[i] \in \{0,1\}$ for $i \in \{0,\ldots,m-1\}$, $r_j = \sum_{i=0}^{m-1} r_j[i] \cdot 2^i$, and r_j is known by party P_j.

EdaBits of the type above, which are formalized by the functionality $\mathcal{F}_{\text{P.edaBits}}$ below, are called *Private edaBits*, which refers to the fact that the underlying secret is known by one party. In the original work of [50] it is shown how to instantiate the functionality $\mathcal{F}_{\text{P.edaBits}}$ using a novel and highly non-trivial cut-and-choose-based approach. In a nutshell, this consists of letting each party P_j sample r_j internally and distribute the sharings $(\llbracket r_j \rrbracket, \llbracket r_j[m-1] \rrbracket_2, \ldots, \llbracket r_j[0] \rrbracket_2)$. In fact, each P_j does this multiple times. In order to check that the underlying values are correct (that is, $r_j[i] \in \{0,1\}$ for $i \in \{0,\ldots,m-1\}$ and $r_j = \sum_{i=0}^{m-1} r_j[i] \cdot 2^i$), the parties execute a protocol in which some of the sharings are opened (after processing them in a specific way) to check their correctness, and via a combinatorial argument it can be shown that the unopened private edaBits are correct with high probability.

Functionality $\mathcal{F}_{\text{P.edaBits}}$

The functionality includes the exact same commands as $\mathcal{F}_{\text{MABB}}$. In addition, it supports the following:

- On input $(\text{p.edabit}, \{\text{id}_\ell\}_{\ell=0}^m, m, i)$ from the honest parties, proceed as follows:
  1. Sample $r \in_R \mathbb{Z}/2^m\mathbb{Z}$
  2. Bit-decompose this value as $(r[m-1],\ldots,r[0]) \in \{0,1\}^m$
  3. Store in memory $(\text{id}_0, \text{arithmetic}, r)$ and $(\text{id}_{\ell+1}, \text{binary}, r[\ell])$ for $\ell \in [m]$.

4. Send r to party P_i
5. Then send $(\text{edabit}, \{\text{id}_\ell\}_{\ell=0}^{m}, m, i)$ to the adversary.

Private edaBits can be used to instantiate the $\mathcal{F}_{\text{edaBits}}$ functionality in a similar way as "private daBits" can be used to obtain daBits, as discussed in Section 7.1.1: after $t+1$ private edaBits $(\llbracket r_j \rrbracket, \llbracket r_j[m-1] \rrbracket_2, \ldots, \llbracket r_j[0] \rrbracket_2)$ for $j \in \{0, \ldots, t+1\}$ are obtained, these can be "added" together to obtain uniformly random edaBits where no single party knows the underlying secrets. This is achieved in the protocol Π_{edaBits} below. We denote by BitADD a binary circuit that adds together $t+1$ m-bit values, obtaining a $m + \log(n)$ bits representing the result.

Protocol Π_{edaBits}

Functionalities: $\mathcal{F}_{\text{P.edaBits}}$.
Output: The parties get $(\llbracket r \rrbracket, \llbracket r[m-1] \rrbracket_2, \ldots, \llbracket r[0] \rrbracket_2)$ where $r = \sum_{j=0}^{m-1} r_i[j] \cdot 2^j$ and the bits are uniform to the adversary.
Protocol: The parties proceed as follows:

1. Call the functionality $\mathcal{F}_{\text{P.edaBits}}$ to get random shares $(\llbracket r_i \rrbracket, \llbracket r_i[m-1] \rrbracket_2, \ldots, \llbracket r_i[0] \rrbracket_2)$, for $i = 1, \ldots, t+1$. Party P_i additionally learns $r_i[j]$ for $j \in \{0, \ldots, m-1\}$ and $r_i = \sum_{j=0}^{m-1} r_i[j] 2^j$.
2. Call $\llbracket r' \rrbracket \leftarrow \sum_{i=1}^{t+1} \llbracket r_i \rrbracket$.
3. Compute the $m + \log n$ bits $(\llbracket b_0 \rrbracket_2, \ldots, \llbracket b_{m+\log(n)-1} \rrbracket_2) \leftarrow \text{BitADD}\left((\llbracket r_1[j] \rrbracket)_{j=0}^{m-1}, \ldots, (\llbracket r_{t+1}[j] \rrbracket)_{j=0}^{m-1} \right)$.
4. Call $\llbracket b_j \rrbracket \leftarrow \llbracket b_j \rrbracket_2$ for $j \in \{m, \ldots, m + \log(n) - 1\}$. Values b_j for $j > k$ do not need to be converted, and for the sake of notation, we denote $\llbracket b_j \rrbracket := 0$ for $j > k$.
5. Compute $\llbracket r \rrbracket \leftarrow \llbracket r' \rrbracket - 2^m \sum_{j=0}^{\log(n)-1} \llbracket b_{j+m} \rrbracket 2^j$.
6. Output $(\llbracket r \rrbracket, \llbracket b_0 \rrbracket_2, \ldots, \llbracket b_{m-1} \rrbracket_2)$.

Theorem 7.1. *Protocol Π_{edaBits} instantiates functionality $\mathcal{F}_{\text{edaBits}}$ with perfect security in the $\mathcal{F}_{\text{P.edaBits}}$-hybrid model.*

Proof. (Sketch) First, notice that the bits $(b_0, \ldots, b_{m+\log(n)-1})$ computed by the m-bit adder correspond to the bit decomposition of $r'' = \sum_{i=1}^{t+1} r_i$, where the addition is performed over the integers. On the other hand, the shared value $\llbracket r' \rrbracket$ computed by the parties in step 2 of the protocol satisfies $r' \equiv \sum_{i=1}^{t+1} r_i \bmod 2^k$, so $r' \equiv_k r''$. From this it follows that $r \equiv_k r' - 2^m \sum_{j=0}^{\log(n)-1} b_{j+m} 2^j \equiv_k \sum_{\ell=0}^{m-1} b_\ell 2^\ell$, so the output $(\llbracket r \rrbracket, \llbracket b_0 \rrbracket_2, \ldots, \llbracket b_{m-1} \rrbracket_2)$ is indeed an m-bit edaBit.

It only remains to be seen that the distribution of r is uniformly random, which can be obtained from the fact that $r = \left(\sum_{i=0}^{t+1} r_i \bmod 2^m \right)$, and there is at least one r_i for $i \in [t+1]$ that is uniformly random. □

7.3 Applications of EdaBits

Now we turn to the main contribution of this chapter, namely, an improved set of applications using edaBits as the main building block. We focus on bit decomposition and bit composition (Section 7.3.1), secure truncation (Section 7.3.2) and secure integer comparison (Section 7.3.3), although our techniques apply to a much wider set of non-linear primitives that require binary circuits for intermediate computations. For example, our techniques also allow us to compute binary-to-arithmetic and arithmetic-to-binary conversions of shared integers, by plugging in our edaBits into the conversion protocols from [27] and [44] for the field and ring cases, respectively.

7.3.1 Arithmetic/Binary Conversions

Our first primitive consists of conversions between the arithmetic and the binary world.

7.3.1.1 Arithmetic-to-Binary

In arithmetic-to-binary conversion, or bit-decomposition, we are given a shared value $[\![x]\!]$, and we would like to find shares of its bit decomposition $([\![x[k-1]]\!]_2, \ldots, [\![x[0]]\!]_2)$.

Protocol Π_{A2B}: Arithmetic to Binary Conversion

Input: Shared values $[\![x]\!]$
Functionality: $\mathcal{F}_{\text{edaBits}}$.
Output: $[\![x_1]\!]_2, [\![x_2]\!]_2, \ldots, [\![x_m]\!]_2$.
Protocol: The parties proceed as follows

1. Call $\mathcal{F}_{\text{edaBits}}$ to obtain a k-bit edaBit $([\![r]\!], [\![r[k-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$.
2. $[\![c]\!] \leftarrow [\![x]\!] - [\![r]\!]$.
3. Open $c \leftarrow [\![c]\!]$
4. Compute $([\![x[k-1]]\!]_2, [\![x[k-2]]\!]_2, \ldots, [\![x[0]]\!]_2) \leftarrow \text{BitAdd}(c, [\![r[k-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$.[a]

[a] Here BitAdd is assumed to be a binary circuit computing the addition modulo 2^k of the integer c' with the bit-decomposed value $(r[k-1], \ldots, r[0])$.

To see that the protocol works as intended, notice that $c = (x - r) \bmod 2^k$, so the result of the binary adder modulo 2^k is the bit decomposition of $(c + r) \bmod 2^k = x$. Privacy holds given that $r \in \mathbb{Z}/2^k\mathbb{Z}$ is uniformly random and unknown to the adversary.

7.3.1.2 Binary-to-Arithmetic

In binary composition, or binary-to-arithmetic conversion, we are given k secret-shared bits $([\![x[k-1]]\!]_2, \ldots, [\![x[0]]\!]_2)$, and the task is to find $[\![x]\!]$, where $x = \sum_{i=0}^{k-1} 2^i x[i]$. This is

achieved by means of the protocol Π_{B2A} below. In spirit, the protocol operates in the same way as Π_{A2B}: the edaBits is subtracted to the input, the result is opened, and the mask is undone by adding the edaBit again. This time, the procedure is executed in the "opposite direction", meaning that the initial masking is performed with the shared bits from the edaBit, and the final unmasking is done with the arithmetic edaBit. The argument of correctness and privacy remains essentially the same.

Protocol Π_{B2A}: Binary to Arithmetic Conversion

Input: Shared bits $[\![x[k-1]]\!]_2, \ldots, [\![x[0]]\!]_2$.
Functionalities: $\mathcal{F}_{\text{edaBits}}$.
Output: $[\![x]\!]$.
Protocol: The parties proceed as follows

1. Call $\mathcal{F}_{\text{edaBits}}$ to obtain a k-bit edaBit $([\![r]\!], [\![r[k-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$.
2. Compute $([\![c[k-1]]\!]_2, \ldots, [\![c[0]]\!]_2) = \text{BitSub}((([\![x[j]]\!]_2)_{j=0}^{k-1}, (([\![r[j]]\!]_2)_{j=0}^{k-1})$.[a]
3. Open $c[j] \leftarrow [\![c[j]]\!]_2$ for $j \in \{0, \ldots, k-1\}$.
4. Let $c = \sum_{j=0}^{k-1} c[j]2^j$. Output $[\![x]\!] = c + [\![r]\!]$.

[a] Here BitSub is assumed to be a binary circuit computing the subtraction modulo 2^k of the integer r with the bit-decomposed value $(x[j])_{j=0}^{k-1}$ with $(r[j])_{j=0}^{k-1}$.

7.3.2 Truncation

Let m be an integer in $\{0, \ldots, k-1\}$. The goal of a truncation protocol is to obtain $[\![y]\!]$ from $[\![x]\!]$, where $y = \lfloor \frac{x}{2^m} \rfloor$. In terms of bits, if the bit-decomposition of x is $(x[k-1], \ldots, x[0])$, this corresponds to shifting the representation to $(0, \ldots, 0, x[k-1], \ldots, x[m])$. This is a crucial operation when dealing with fixed-point arithmetic, and therefore an efficient solution for it has a substantial impact in the efficiency of MPC protocols for a wide range of applications.

An important observation is that $\lfloor \frac{x}{2^m} \rfloor = \frac{x-(x \bmod 2^m)}{2^m}$. Indeed, if $x = \sum_{j=0}^{k-1} x[j]2^j$ then $(x \bmod 2^m) = \sum_{j=0}^{m-1} x[j]2^j$ so $\frac{x-(x \bmod 2^m)}{2^m} = \frac{\sum_{j=m}^{k-1} x[j]2^j}{2^m} = \sum_{j=m}^{k-1} x[j]2^{j-m} = \sum_{\ell=0}^{k-m-1} x[\ell + m]2^\ell$.

Truncation protocols over fields typically exploit the fact that one can divide by powers of 2 modulo p. This is the case for example in the protocols presented in [27]. This is not possible when working modulo 2^k, so we take a different approach. Let $[\![x]\!]$ be the initial shared value, and let $(x[k-1], \ldots, x[0])$ be the bit-decomposition of x. For our protocol to work, we need to assume that x, which in principle lies in the range $[0, 2^k)$, actually lies in $[0, 2^\ell)$ for some $\ell < k$, which in particular means that $x[k-1]$ is guaranteed to be 0.

In order to compute $[\![\lfloor \frac{x}{2^m} \rfloor]\!]$, our protocol begins by computing shares of $x \bmod 2^m$, and subtracting them from x, which produces shares of the value with bit-decomposition $(x[k-1], \ldots, x[m], 0, \ldots, 0)$. At this point it only remains to "right-shift" this value by m positions to obtain $(0, \ldots, 0, x[k-1], \ldots, x[m])$. This is achieved by asking the parties to

open a masked version of $x-(x \bmod 2^m)$, which does not reveal the upper $k-\ell$ bits, and then shift to the right by m positions in the clear. Finally, the parties undo the truncated mask. One has to account for the overflow that may occur during this masking, but this can be calculated using a less-than binary circuit.

Protocol Π_{Trunc}: Truncation

Input: Shared value $[\![x]\!]$ with $x \in [0, 2^\ell)$ for some $\ell < k$.
Functionalities: $\mathcal{F}_{\text{edaBits}}$.
Output: $[\![y]\!]$ where $y = \left\lfloor \frac{x}{2^m} \right\rfloor$.
Protocol: The parties call $\mathcal{F}_{\text{edaBits}}$ to obtain:

- m-bit edaBit $([\![r]\!], [\![r[m-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$
- $\ell - m$-bit edaBit $([\![r']\!], [\![r'[\ell-m-1]]\!]_2, \ldots, [\![r'[0]]\!]_2)$

Then, the parties proceed as follows

1. The parties compute shares of $x \bmod 2^m$ as follows:
   a) Open $c \leftarrow 2^{k-m} \cdot ([\![a]\!] + [\![r]\!])$
   b) Compute $[\![v]\!]_2 = \text{LT}((c[i])_{i=k-m+1}^{k}, ([\![r[i]]\!]_2)_{i=0}^{m-1})$[a]
   c) Convert $[\![v]\!] \leftarrow [\![v]\!]_2$
   d) Compute $[\![x \bmod 2^m]\!] \leftarrow 2^m [\![v]\!] - [\![r]\!] + c/2^{k-m}$.
2. The parties compute the truncation:
   a) Compute $[\![b]\!] \leftarrow [\![x]\!] - ([\![x \bmod 2^m]\!])$.
   b) Open $d \leftarrow 2^{k-\ell} \cdot ([\![b]\!] + 2^m [\![r']\!])$.
   c) Compute $[\![u]\!]_2 = \text{LT}((d[i])_{i=k-\ell+m}^{k-1}, ([\![r'[i]]\!]_2)_{i=0}^{\ell-m-1})$
   d) Convert $[\![u]\!] \leftarrow [\![u]\!]_2$
   e) Output $[\![y]\!] \leftarrow 2^{\ell-m} [\![u]\!] + d/2^{k-\ell+m} - [\![r']\!]$

[a] Here LT represents a binary circuit to determine which of the two inputs is smaller.

Now we analyze the correctness of Protocol Π_{Trunc}. First, it is easy to see that $c = 2^{k-m}((x+r) \bmod 2^m)$, so $c/2^{k-m} = (x \bmod 2^m) + r - 2^m v$, where v is set if and only if $c/2^{k-m} < r$. From this we can see that the first part of the protocol $[\![x \bmod 2^m]\!]$ is correctly computed. Privacy of this first part follows from the fact that $r \bmod 2^m$ completely masks $x \bmod 2^m$ when c is opened.

For the second part, let us write $b = 2^m x'$, then $d = 2^{k-\ell+m}((x'+r') \bmod 2^{\ell-m})$, so $d/2^{k-\ell+m} = x' + r' - 2^{\ell-m}u$, where u is set if and only if $d/2^{k-\ell+m} < r'$, as calculated by the protocol. We get then that $x' = d/2^{k-\ell+m} - r' + 2^{\ell-m}u$, and since x' is precisely $\left\lfloor \frac{x}{2^m} \right\rfloor$, we conclude the correctness analysis.

Probabilistic Truncation. In some cases rounding to nearest after division is desired, that is, $[\![y]\!]$ must be computed from $[\![x]\!]$, where $y = \left\lfloor \frac{x}{2^m} \right\rceil$. This can be approached as shown directly above by relying on the protocol Π_{Trunc}. However, if certain error is allowed, then we can design a more efficient protocol for this task, which results in a *probabilistic truncation* protocol. Furthermore, this error will not be arbitrary, but instead it will be "biased towards the right result". More precisely, if we write $\frac{x}{2^m} = \left\lfloor \frac{x}{2^m} \right\rfloor + \delta$, where $\delta \in [0, 1)$, then the result of the probabilistic truncation protocol will be shares of y,

where y equals $\lceil \frac{x}{2^m} \rceil$ with probability δ, and it equals $\lfloor \frac{x}{2^m} \rfloor$ with probability $1-\delta$. In other words, if δ is too close to 1, meaning that $\frac{x}{2^m}$ is very close to $\lceil \frac{x}{2^m} \rceil$, then the protocol will result in shares of this value, but if δ is too small, meaning that $\frac{x}{2^m}$ is actually closer to $\lfloor \frac{x}{2^m} \rfloor$, then the protocol will return shares of $\lfloor \frac{x}{2^m} \rfloor$ instead; if $\delta \approx 1/2$, meaning that $\frac{x}{2^m}$ is as close to $\lceil \frac{x}{2^m} \rceil$ as it is to $\lfloor \frac{x}{2^m} \rfloor$, then the protocol makes a random choice among these two.

When the computation domain is a prime field, it turns out this operation can be carried out in a constant number of rounds without the need of any binary circuit [27], such as the circuit LT we made use of in Π_{Trunc}. Unfortunately, over the ring $\mathbb{Z}/2^k\mathbb{Z}$ this is a much more challenging task. For example, probabilistic truncation with a constant number of rounds is achieved in ABY3 [68], but it requires a 2^κ gap between the secret values and the actual modulus, where κ is a statistical security parameter, which in turn implies that only small non-negative values can be truncated.[2]

In this work we take a different approach. Intuitively, we follow the same approach as in ABY3, which consists of masking the value to be truncated with a shared random value for which its corresponding truncation is also known, opening this value, truncating it and removing the truncated mask. As we have mentioned, in ABY3 a large gap is required to ensure that the overflow that may happen by the masking process does not occur with high probability. Instead, we allow this overflow bit to be non-zero and remove it from the final expression. Doing this naively would require us to compute a less-than binary circuit, but we avoid doing this by using the fact that the overflow bit can be obtained from the opened value by making the mask value also positive. This leaks the overflow bit, which is not secure, and to avoid this we mask this single bit with another random bit. Our approach requires the input to lie in $[0, 2^\ell)$ for some $\ell < k$, but we remark that this is better than the gap from ABY3 where $\ell \leq k - \kappa$.

$\Pi_{\text{Pr.Trunc}}$: Probabilistic Truncation

Input: Shared value $[\![x]\!]$ with $x \in [0, 2^\ell)$ for some $\ell < k$.
Functionalities: $\mathcal{F}_{\text{edaBits}}$.
Output: $[\![y]\!]$ where $y = \lfloor x/2^m \rfloor + u$ with $u = 1$ with probability $(x \bmod 2^m)/2^m$.
Protocol: The parties call $\mathcal{F}_{\text{edaBits}}$ to obtain:

- m-bit edaBit $([\![r']\!], [\![r'[m-1]]\!]_2, \ldots, [\![r'[0]]\!]_2)$
- $\ell - m$-bit edaBit $([\![r]\!], [\![r[\ell-m-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$
- 1-bit edaBit $([\![b]\!], [\![b]\!]_2)$.[a]

Then, the parties proceed as follows

1. Open $c \leftarrow 2^{k-\ell-1} \cdot ([\![a]\!] + 2^\ell [\![b]\!] + 2^m [\![r]\!] + [\![r']\!])$. Write $c = 2^{k-\ell-1} c'$.
2. Compute $[\![v]\!] \leftarrow [\![b]\!] + c'_\ell - 2c'_\ell [\![b]\!]$
3. Output $[\![y]\!] \leftarrow (c' \bmod 2^\ell)/2^m - [\![r]\!] + 2^{\ell-m} [\![v]\!]$

[a] This can be optimized as the binary sharing $[\![b]\!]_2$ will not be used, only the $[\![b]\!]$ part is required.

Now we analyze the protocol. First we notice that $c = 2^{k-\ell-1} c'$ where $c' = (2^m r + r') + x + 2^\ell b - 2^{\ell+1} vb$, where v is set if and only if $(2^m r + r') + x$ overflows modulo 2^ℓ. It is

[2] Such restriction is also present in most field-based protocols.

easy to see that this implies that $c'_\ell = v \oplus b$, so we see that $v = c'_\ell \oplus b$, as calculated in the protocol.

On the other hand, we have that $(c' \bmod 2^\ell) = (2^m r + r') + x - 2^\ell v$, so $x \bmod 2^m = (c' \bmod 2^m) - r' + 2^m u$, where u is set if $(c' \bmod 2^m) < r'$. From this it can be obtained that $\lfloor (c' \bmod 2^\ell)/2^m \rfloor - r + 2^{\ell-m} = \lfloor x/2^m \rfloor + u$.

7.3.3 Integer Comparison

Another important primitive that appears in many applications is integer comparison. In this case, two secret integers $[\![x]\!]$ and $[\![y]\!]$ are provided as input, and the goal is to compute shares of $b = (x \overset{?}{<} y)$, that is, $b = 1$ if $x < y$, and it is equal to 0 otherwise.

As noticed by previous works (e.g. [27, 44]), this computation reduces to extracting the most-significant bit (MSB) from a shared integer as follows: if $x, y \in [0, 2^{k-1})$, and if $z = (x - y \bmod 2^k)$, then $z \in [0, 2^{k-1})$ if $x \geq y$, and $z \in [2^{k-1}, 2^k)$ if $x < y$. Since $z \in [0, 2^{k-1})$ if $z[k-1] = 0$ and $z \in [2^{k-1}, 2^k)$ if $z[k-1] = 1$, we see that $z[k-1]$, the MSB of z, is equal to the desired bit $(x \overset{?}{<} y)$. To extract the MSB of z, we simply note that $z[k-1] = \left\lfloor \frac{z}{2^{k-1}} \right\rfloor \bmod 2^k$, so this can be done with the protocols we have seen in the previous section.

The approach above, that is, calling Π_{Trunc} with $m = k - 1$, would require two *sequential* calls to a less-than binary circuit. This can be optimized to one single call to such circuit as follows.

Π_{MSB}: MSB Extraction

Input: Shared value $[\![z]\!]$ with $z \in [0, 2^k)$.
Functionalities: $\mathcal{F}_{\text{edaBits}}$.
Output: $[\![b]\!]$ where $b = z[k-1]$.
Protocol: The parties call $\mathcal{F}_{\text{edaBits}}$ to obtain k-bit edaBit $([\![r]\!], [\![r[k-1]]\!]_2, \ldots, [\![r[0]]\!]_2)$. Then, the parties proceed as follows

1. Open $c \leftarrow [\![a]\!] + [\![r]\!]$
2. Compute $[\![v]\!]_2 \leftarrow \text{LT}\left((c[i])_{i=0}^{k-1}, ([\![r[i]]\!]_2)_{i=0}^{k-1}\right)$
3. Compute $[\![b]\!]_2 \leftarrow [\![v]\!]_2 \oplus [\![r[k-1]]\!]_2 \oplus c[k-1]$
4. Convert $[\![b]\!] \leftarrow [\![b]\!]_2$
5. Output $[\![b]\!]$

Bibliography

[1] M. Abspoel, R. Cramer, I. Damgård, D. Escudero, M. Rambaud, C. Xing, and C. Yuan. Asymptotically good multiplicative LSSS over galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In S. Moriai and H. Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 151–180. Springer, 2020.

[2] M. Abspoel, R. Cramer, I. Damgård, D. Escudero, and C. Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In D. Hofheinz and A. Rosen, editors, *Theory of Cryptography - 17th International Conference, TCC 2019, Nuremberg, Germany, December 1-5, 2019, Proceedings, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501. Springer, 2019.

[3] M. Abspoel, R. Cramer, D. Escudero, I. Damgå rd, and C. Xing. Improved single-round secure multiplication using regenerating codes. In *Asiacrypt*, 2021.

[4] M. Abspoel, A. P. K. Dalskov, D. Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. *ACNS*, 2021.

[5] M. Abspoel, D. Escudero, and N. Volgushev. Secure training of decision trees with continuous attributes. *Proc. Priv. Enhancing Technol.*, 2021(1):167–187, 2021.

[6] B. Alon, E. Omri, and A. Paskin-Cherniavsky. Mpc with friends and foes. In *Annual International Cryptology Conference*, pages 677–706. Springer, 2020.

[7] A. Aly, K. Cong, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, O. Scherer, P. Scholl, N. Smart, T. Tanguy, et al. Scale–mamba v1. 14: Documentation, 2021.

[8] B. Applebaum, Y. Ishai, and E. Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[9] D. Aranha, D. Anders, D. Escudero, and C. Orlandi. Improved threshold signatures, proactive secret sharing, and input certification from lss isomorphisms. In *Latincrypt*, 2021.

[10] V. Arvind and T. C. Vijayaraghavan. The complexity of solving linear equations over a finite ring. In V. Diekert and B. Durand, editors, *STACS 2005*, pages 472–484, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[11] J. Baena, D. Cabarcas, D. E. Escudero, K. Khathuria, and J. Verbel. Rank analysis of cubic multivariate cryptosystems. In *International Conference on Post-Quantum Cryptography*, pages 355–374. Springer, 2018.

[12] J. B. Baena, D. Cabarcas, D. E. Escudero, J. Porras-Barrera, and J. A. Verbel. Efficient zhfe key generation. In *Post-Quantum Cryptography*, pages 213–232. Springer, 2016.

[13] M. Ball, T. Malkin, and M. Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 565–577, 2016.

[14] C. Baum, D. Escudero, A. Pedrouzo-Ulloa, P. Scholl, and J. R. Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-lwe. In *International Conference on Security and Cryptography for Networks*, pages 130–149. Springer, 2020.

[15] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[16] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008.

[17] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[18] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Annual Cryptology Conference*, pages 663–680. Springer, 2012.

[19] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 169–188. Springer, 2011.

[20] D. Bogdanov, M. Jõemets, S. Siim, and M. Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International conference on financial cryptography and data security*, pages 227–234. Springer, 2015.

[21] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[22] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 869–886, 2019.

[23] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[24] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[25] R. Canetti, A. Cohen, and Y. Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.

[26] D. Catalano, M. D. Raimondo, D. Fiore, and I. Giacomelli. Monza: Fast maliciously secure two party computation on $\mathbb{Z}/2^k\mathbb{Z}$. In *IACR International Conference on Public-Key Cryptography*, pages 357–386. Springer, 2020.

[27] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, pages 182–199. Springer, 2010.

[28] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. Ezpc: programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[29] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[30] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[31] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369, 1986.

[32] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798. Springer, 2018.

[33] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 316–334. Springer, 2000.

[34] R. Cramer, I. Damgard, and J. B. Nielsen. Secure multiparty computation and secret sharing-an information theoretic approach, 2012.

[35] A. Dalskov and D. Escudero. Honest majority mpc with abort with minimal online communication. In *Latincrypt*, 2021.

[36] A. P. K. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. *USENIX*, 2021.

[37] A. P. K. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *Proc. Priv. Enhancing Technol.*, 2020(4):355–375, 2020.

[38] I. Damgård. Commitment schemes and zero-knowledge protocols. In *School organized by the European Educational Forum*, pages 63–86. Springer, 1998.

[39] I. Damgård, D. Escudero, and D. Ravi. Information-theoretically secure mpc against mixed dynamic adversaries. In *TCC*, 2021.

[40] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010.

[41] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[42] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

[43] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[44] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pages 1325–1343, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[45] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[46] W. Diffie and M. Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[47] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[48] J. H. Ellis. The possibility of secure non-secret digital encryption. *UK Communications Electronics Security Group*, 8, 1970.

[49] D. Escudero. A crash course on MPC. https://medium.com/applied-mpc/a-crash-course-on-mpc-part-0-311fae2ce184. Accessed: Aug. 28th, 2021.

[50] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In D. Micciancio and T. Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852. Springer, 2020.

[51] D. Escudero and E. Soria-Vazquez. Efficient information-theoretic multi-party computation over non-commutative rings. *CRYPTO*, 2021.

[52] M. Franklin and M. Yung. Communication complexity of secure computation. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 699–710, 1992.

[53] M. A. Frumkin. An application of modular arithmetic to the construction of algorithms for solving systems of linear equations. In *Doklady Akademii Nauk*, volume 229, pages 1067–1070. Russian Academy of Sciences, 1976.

[54] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, 1998.

[55] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[56] S. Goldwasser and Y. Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, 2005.

[57] V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Annual International Cryptology Conference*, pages 618–646. Springer, 2020.

[58] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[59] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[60] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[61] M. Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[62] M. Keller, E. Orsini, and P. Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[63] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[64] J. Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 20–31, 1988.

[65] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[66] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–5, 2018.

[67] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, pages 277–346, 2017.

[68] P. Mohassel and P. Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–52, 2018.

[69] E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure mpc over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 254–283. Springer, 2020.

[70] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[71] A. Patra and A. Suresh. BLAZE: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042*, 2020.

[72] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.

[73] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019.

[74] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[75] R. Shostak, M. Pease, and L. Lamport. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[76] S. Singh. *The code book*, volume 7. Doubleday New York, 1999.

[77] J. von zur Gathen and M. Sieveking. Iv. weitere zum erfüllungsproblem polynomial äquivalente kombinatorische aufgaben. In *Komplexität von Entscheidungsproblemen Ein Seminar*, pages 49–71. Springer, 1976.

[78] Z.-X. Wan. *Lectures on finite fields and Galois rings*. World Scientific Publishing Company, 2003.

[79] L. R. Welch and E. R. Berlekamp. Error correction for algebraic block codes, Dec. 30 1986. US Patent 4,633,470.

[80] A. C. Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.