Author:
Rotaru, Dragos A

Title:
Optimizing Secure Multiparty Computation Protocols for Dishonest Majority

Optimizing Secure Multiparty Computation Protocols for Dishonest Majority

Dragoș Alin Rotaru

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering

Department of Computer Science
January 2020

Word count: 65000 words

Abstract

A set of parties want to compute a function $\mathcal{F}$ over their inputs without revealing them, learning only the output of $\mathcal{F}$. This is the traditional scenario introduced to show what secure Multi-Party Computation (MPC) can achieve: computing on encrypted data. Due to the initial theoretical papers appearing in the beginning of 80s describing basic protocols to achieve MPC, it has now become a hot topic in the cryptographic community where we can see dozens of startups finding good use-cases such as machine learning on encrypted data as well as high quality research constantly pushing the field's boundaries.

The goal of this thesis is to improve on dishonest majority MPC where all but one of the parties can arbitrarily deviate from the protocol and still ensure input privacy of the honest parties.

Many modern MPC protocols are realized in two stages: an input-independent but usually expensive *preprocessing phase* coupled with an input-dependent stage called *online phase*. The first contribution of this thesis is to revisit two popular protocols (SPDZ and BDOZ) based on Homomorphic Encryption (HE), and show that, with some improvements, the HE based protocols can perform better than the state-of-the-art preprocessing based on oblivious transfer.

The second contribution of the thesis is to improve upon the TinyTable protocol which evaluates lookup tables on secret data. We then evaluate more complex algorithms such as AES using the lookup table approach within SPDZ framework, and make them competitive with their Boolean counterpart based on garbled circuits for dishonest majority.

Next we build more efficient Pseudorandom Functions (PRF) protocols which have an efficient description when evaluated over an arithmetic circuit instead of binary circuits where AES shines. The resulted PRFs are then used to perform more efficient authenticated encryption using SPDZ protocol. These two applications are crucial when a set of MPC servers want to compute $\mathcal{F}$ based on inputs coming from external clients or storing outputs of $\mathcal{F}$ to an encrypted database where no party holds the decryption key but still allow them to operate on the encrypted data.

Finally, we give efficient conversion procedures between different paradigms of MPC for dishonest majority. This allows us to split $\mathcal{F}$ into chunks and evaluate each chunk using our favorite MPC protocols to then switch smoothly between each representation, realizing a more efficient evaluation of $\mathcal{F}$ overall.

Acknowledgements

Looking back at the last four years of my life it seems to me that I was extremely lucky to be constantly surrounded by many brilliant and kind people from which I could learn how to be a better researcher as well as a more thoughtful human being.

First I want to thank my Ph.D. advisor Nigel Smart for his contagious enthusiasm and incredible work ethic. I am indebted for his guidance and continuous support throughout my Ph.D. journey as well as giving me enough freedom to pursue independent research projects whenever needed. I am convinced that some of his invaluable advice will stick with me for a long time from now on.

Second I would like to thank all of my co-authors without whom this thesis would not have been possible: Martin R. Albrecht, Abdelrahaman Aly, Hao Chen, Lorenzo Grassi, Marcel Keller, Miran Kim, Reinhard Lüftenegger, Eleftheria Makri, Emmanuela Orsini, Valerio Pastro, Léo Perrin, Sebastian Ramacher, Ilya Razenshteyn, Christian Rechberger, Arnab Roy, Markus Schofnegger, Peter Scholl, Eduardo Soria-Vásquez, Nigel P. Smart, Yongsoo Song, Martijn Stam, Titouan Tanguy, Frederik Vercauteren, Srinivas Vivek, Sameer Wagh, Tim Wood. I have learned a lot from each of them and greatly enjoyed working and discussing ideas with all of them.

Third I am grateful to have two amazing friends and collaborators whom I always perceived them as excellent mentors throughout these four years: Marcel Keller and Peter Scholl. I was very lucky to have them around when my Ph.D. started and I kept asking them technical questions even after they left Bristol. Thank you for your patience guys, your help was and still is greatly appreciated.

Fourth I am grateful to Hao Chen for having me as an intern in the MSR crypto research team in the summer of 2019 and helping me understand how research is done within a large company as Microsoft. In Redmond I had the pleasure of meeting Sameer Wagh who proved to be a great friend and collaborator. I would like to thank Ilia and Irina for taking care of me while being in Redmond.

It would be a pity to not tell the story of how I got to do a Ph.D in cryptography and mention the people encountered throughout and thank them for their support. During the summer of my first year of undergraduate studies I was getting bored so I decided to take a couple of online courses, one of which was the cryptography course offered by Dan Boneh. After the summer ended it was only a matter of time that I reached out to Ruxandra Olimid, a lecturer at University of Bucharest to be my thesis supervisor and introduce me to certain aspects of cryptography research. After finishing my bachelor's degree I had the pleasure to work with Miruna Roșca and Radu Țițiu studying cryptography in the research labs of Bitdefender Romania. I am thankful to these two kind people from whom I have

learned a lot and are now close friends. A few months before the end of the year I was contacted by Bogdan Warinschi whom I met the first time at a cryptography summer school in Bucharest. Bogdan told me there is a PhD opening at Bristol supervised by Nigel. From there to being in Bristol in January 2016 was just an interview with Nigel. Here I would also want to express my gratitude to the following professors at University of Bucharest who always encouraged me to pursue research: Prof. Gheorghe Ștefănescu and Prof. Andrei Păun and to, now lecturer, Marius Dumitran.

Next I would like to thank the Bristol Crypto group for keeping me grounded for the two years spent there and the thoughtful discussions we had during the Friday pubs. Although there were few sunny days in Bristol, Marco made sure that Bristol is warm enough: he made sure I attend football every week as well inviting me to the awesome Italian dinners with him and Alessandra. Although I was far from Romania this made me feel like home. Meeting Avanthika a couple of months before I left Bristol certainly made the decision to go to Belgium even tougher but I am thankful to have her besides me a couple of weeks before submitting this manuscript.

I am grateful to have worked in the COSIC group as it was full of fun and thoughtful activities filled with football, squash and dancing salsa. I am indebted to all the people in the COSIC group, I especially enjoyed the company of Marc, Ilia, Younes, Abdel, Jose, Cyprien who were close friends to me and played dozens of football and squash matches which kept me in good physical shape to do productive research. I am profoundly grateful to Ilaria and Abdel for convincing me to start taking dance classes since I have discovered an awesome community through this which certainly contributed to my well-being during my Ph.D.

To my friends in Romania who were always eager to welcome me whenever I was back for a few days: Emil, Bogdan, Andrei, Cristina and Mihai. I am grateful for the algorithm puzzles Mihai kept throwing at me and the countless discussions we had where I would always learn something new. To some of my friends that left the country such as Andrei and Gabi but we always hanged out together when we found ourselves in the same city. Apologize to all the other friends who I am missing now.

I would also like thank my middle school math teacher, Mr. Onescu Ion who was the first math teacher I had which had patience, enthusiasm and kindness towards his students - it was a pleasure to sit through his classes and be inspired by him.

Finally, I am indebted to my family for their unconditional love and support. I am especially thankful to my mom who from the beginning of my first days in school tried to get me the best possible education.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED:_____ DATE:_____

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays when most of the people hear the word *crypto* they would probably think about cryptocurrencies. A simple Google search of the word *crypto* at the end of 2019 will yield dozens of web pages of cryptocurrencies in the top results. When I started my PhD in 2016 the popularity of the words *cryptography* and *crypto* was quite similar according to Google Trends [Goo19] whereas between December 2017 and January 2018 *crypto* would be about 50 times more popular than *cryptography*. This peak had nothing to do with the content of the thesis or the author. We only try to point that people's understanding of words can change throughout time and meaning of *cryptography* can denote different things depending on the context, whether the person comes from a practical side or with a more theoretical background. From an historical point of view cryptocurrencies started out as a small sub-topic of cryptography.

The type of cryptography this thesis deals with is about protecting data throughout computation without revealing it. This sub-field of cryptography is called secure Multi Party Computation (MPC) which was introduced by Andrew Yao in 1980. Since that time period the community polished it so much that nowadays there are various number of companies applying MPC to protect sensitive secrets.

Perhaps one of the simplest ways of explaining MPC to the general public is through the following example: in a classroom, the teacher wants to find out how many people have failed a math exam without looking at the students' individual grades. The teacher then comes up with the following protocol, write a random number r using a pencil on a sheet of paper and pass it to the first student. If the student has failed the math exam erase r and write $r + 1$ and pass the sheet onto the next student, otherwise if the student has passed the test then leaves the note as it is and send r to the next student. After the sheet of paper gets sent to every student, the last student gives a number $r + x$ to the teacher. In the end the teacher just subtracts the first message (r) from the last message ($r + x$) to get the number of students (x) who failed the math exam. In this way the students grades remain private while the teacher only learns the amount of students which failed the exam.

The idea of multiparty computation is very simple: a set of parties willing to compute a function over their inputs while keeping them private, revealing only the output of the computation. MPC takes care of what happens after parties plug in their inputs to the computation, for example things can get

complicated depending on the answers given to the following guide-through:

1. Number of parties involved in the computation. Are there two, three...one hundred, one thousand parties?
2. Type of network connection. Are parties connected on a low-latency, local area network (LAN) or through a high-latency, wide area network (WAN)?
3. Adversary behaviour: semi-honest or malicious adversary. Is there any malicious party which can arbitrarily deviate from the protocol? If yes, how many: a minority of them (less than half of the parties) or do we deal with a dishonest majority where all but one parties can act maliciously?
4. Adversary power. Do malicious parties have access to computationally unbounded machines or should we assume they only have access to probabilistic polynomial computers?
5. Corruptions over time. Do we assume parties can get corrupted throughout the computation (adaptive adversary) or is the number of corrupted parties known at the start (static adversary)?

One can see that there is a big subset of protocols depending only on the few listed questions above. If we consider the variable t as the threshold of corrupted parties and n as the number of parties then there are a few impossibility results known in the literature. One such result is that if the threshold is $t < n/3$ then there are information theoretical secure protocols which are resistant for computational unbounded adversaries. Increasing the threshold to $t < n/2$ with active security is impossible without adding some extra cryptographic assumptions [Mau06, HM97] Although successful termination in the case of full threshold MPC when $t < n$ cannot be guaranteed, if we slightly relax the assumption and allow parties to abort during the computation then it is possible to achieve MPC (with abort) for $t < n$. The latter is usually obtained through expensive public key machinery such as homomorphic encryption or oblivious transfer.

This thesis carries out work in of the most challenging models: multiparty computation with abort for dishonest majority. By default we will assume parties communicate over a synchronous network: i.e. they have access to some global clock and there exists a strict upper bound on how long a message delivery should take. These restrictions are irrelevant when benchmarking protocols as parties act as soon as they received data. However they are useful in practice to detect whether a malicious party is intentionally delaying the protocol or to formally prove guaranteed termination of certain schemes [KMTZ13].

1.1 Usecases for MPC

One can wonder if being able to compute on private data jointly owned by a set of parties is at all useful. We argue that the answer is affirmative. The idea of garbled circuits stemmed from the seminal work of Andrew Yao [Yao82] which tried to solve the millionaire's problem: two parties figuring out who is the richest without revealing their amount of money. Going from work of Lindell et al. [LPS08] implementing the first two-party active security comparison protocol (as in Yao's millionaire problem) which took around 2-3 minutes to the present where computing comparisons using SPDZ takes a few

milliseconds. Nowadays the practice advanced so much that there are cases which one can do privacy preserving analytics on millions of entries in a matter of seconds depending on the privacy algorithm and security model [MR18, SGRP19, MZ17]. We now recap some of most popular examples of MPC deployments which some of them are illustrated as well by the book of Evans, Kolesnikov and Rosulek [EKR18].

1.1.1 Sugar beet auctions

The first deployment of MPC was in Denmark 2009 when a team of researchers from Aarhus University collaborated with the Danish Government to help farmers keep renew their contracts based on which production pays off best [BCD+09]. After a survey which revealed a large concern of the farmers about their bidding details Danisco, the farmer's association DKS and the Aarhus researchers from the SIMAP project decided to run a three party protocol where clients (farmers) would submit their bids in secret shared form and let the three entities compute a double auction on their secret bids. In the double auction the farmers would specify how much they would sell for each possible price whereas buyers input the amount of sugar beets they would buy for a specific price. The goal of a double auction is to find the market clearing price which is the point where total supply equals total demand which is what the three organisations computed.

1.1.2 Estonian social studies on tax and education

Given the alarming drop-out of IT students where a total of 43% of the students quit their studies, the Estonian Association of Information and Communication Technology (ITL) wanted to study this issue in more detail trying to see whether the IT companies were hiring to aggressively causing this massive university drop out. After passing through all the legal hurdles with their Ministry of Education and Research and the Estonian Tax and Customs Board they succeeded in performing a 3PC computation where the third server belonged to Cybernetica, the company that develops Sharemind [BLW08, Bog15]. The algorithm involved sharing of 800,000 study records along with 20 million tax records, with the study concluding several interesting things: the more education one gets, the higher the salary; and it turns out that IT companies aren't hiring too much, it might be that Computer Science courses are getting increasingly difficult to pass.

1.1.3 Key Management

In real life key management is hard. If you add the fact that an attacker can get access to your server and steals your data including the precious keys, then key management becomes even more complicated. One can actually apply multiparty computation in order to harden the security: take the owned key, secret share it additively across multiple servers and then delete it. Now an attacker has to get access to all computers in order to retrieve your key. Moreover the user will get rid of the painful key management and let the servers do it for them.

The next problem arises: how to use the secret key for authentication or signing procedures? The answer is to realise the specific functionalities using MPC. This is one of the software solutions Unbound Tech offers [Tec19]. They distribute the key across different cloud providers such as Google Cloud, Microsoft's Azure or Amazon AWS at your choice and then execute authentication or threshold signatures without reconstructing the original key.

1.1.4 Boston wage gap

MPC can be used for social good as well. Lapets et al. [LJA+18] show how they aggregated salaries privately from the Boston Women Workforce Council (BWWC) to gather more informed decisions regarding the problem of gender paygap in that institution. The problem arose that no third party wanted to have the raw data of more than 30,000 salaries. The solution was to run a two party protocol between a server hosted by the Boston University and one hosted by the BWWC using JIFF framework [Uni19]. Since MPC deals with what happen after the parties input their data, Lapets at el. added some extra checks to the forms to ensure correctness along with many usability features for a less experienced tech person to use the software.

1.1.5 Password breach

Recently Google has found a new use-case to safe-guard passwords using a more specialized two party protocol denoted as Private Set Intersection (PSI) [Lak19]. PSI allows for two parties to input a list of elements and learn only the output of the intersection. Since Google has access to a massive database of leaked password from the dark web they have decided to put this to use: avoid leaking everyone's password in the plaintext to then force the users to update their credentials. Instead they have created an Chrome Extension which allows anyone to enter an username and a password before the creation of a new account and do a set intersection with Google's giant database of leaked passwords. In this way Google will not know the user's password but still be able warn users to try a different combination of username/password, one which is not compromised. Recently it was announced that they plan to roll in this feature in every login service Google has [Mar19].

1.1.6 Where is this all going?

This is a question too hard to answer but we can certainly gather some clues. In 2009 it was the first time non-researchers have applied MPC to settle sugar beet auctions in Denmark. From there it was just a few steps for Partisia company to be built.

Although a bit hard to keep track throughout the years of all startups and companies which used MPC within their software solutions there was recently announced the so-called MPC alliance calling MPC a "disruptive technology showing great promise in multiple industries". At the time of writing on their website there are 25 companies claiming to use multiparty computation, where most of them are using it for key management for block-chain wallets or applying it to privacy preserving data analytics.

Some of the major corporations such as Facebook, Google or Microsoft are not yet in the Alliance although they are applying MPC to some research projects or use it to some internal products [Fac19, O'D19, Mic19].

1.2 Outline of the thesis

In Chapter 2 we give some basic notions which will help the reader understand the content of this thesis. It starts with some notation, then reviews a basic protocol and proofs that is universally composable using Canetti's framework [Can01]. Chapter 3 offers a high level description of the dishonest majority protocols used throughout the thesis. Although most of the chapters will deal with secret shared based dishonest majority (SPDZ) the last chapter will show how to switch between SPDZ and other MPC frameworks including garbled circuits with a focus on BMR style garbled circuits [BMR90]. Since all the protocols used here work in the preprocessing model, we will describe how the online phases work for BMR and SPDZ as well as sketch the preprocessing phase.

At the core of our thesis lies Chapter 4, which describes in more details how to produce efficiently the so-called 'Beaver Triples' for SPDZ over arithmetic circuits modulo p. The end of the chapter contains some benchmarks on how the protocols described work when Beaver triples are instantiated over the field $\mathbb{F}_{2^k}$.

In the next three chapters we focus on building applications on top of the SPDZ engine. Building applications in MPC is nowadays more easy due to an existing high-level language developed by Keller et al. [KSS13a] and a continuous development ever since. Chapter 5 improves upon the state of the art evaluations of PRFs in MPC for arithmetic circuits with characteristic two ($\mathbb{F}_{2^k}$). More concretely it improves by a factor of 50 over prior work upon evaluating AES and 3-DES in SPDZ using lookup table based protocols [DNNR17]. The next two parts, Chapter 6 and 7, focus on building special protocols PRFs and Authenticated Encryption (AE) over fields of prime characteristic. This is particularly useful since computing the Beaver triples over $\mathbb{F}_p$ is faster than their variant over $\mathbb{F}_{2^k}$, and switching to evaluating AES over $\mathbb{F}_p$ would be prohibitively expensive since AES is designed for boolean circuits.

Throughout Chapter 8 we unify all the work on the dishonest majority MPC by showing an efficient way to convert between different sharing schemes and garbled circuits. We then show the benefits of switching between SPDZ and BMR based garbled circuits by improving the performance of a simple machine learning algorithm on private data by an order of magnitude when evaluated over plain SPDZ. Our method involves generating some new preprocessed material called daBits (or doubly authenticated bits) improves upon the naive conversion from SPDZ to garbled circuits and vice-versa by a factor of at least 50 in terms of communication making conversions in the realm of dishonest majority MPC practical.

The thesis concludes with Chapter 9 with a few open questions that we consider relevant upon in the next few years in order to make MPC even more practical.

1.3 Contributions of the Author

Traditionally, the order of the authors in cryptography is done via an alphabetical ordering of their last names. Since ideas are hard to quantify to check which are more valuable than others this removes the un-necessary conflict of a discussion phase to agree on some other ordering [Soc04]. The ordering was done alphabetically for the papers I have been a co-author of, and represent an equal contribution amongst all the authors, unless stated otherwise. To conclude, the thesis content is mostly based on the following publications in chronological order:

1. [GRR+16] MPC-Friendly Symmetric Key Primitives, published at CCS 2016, joint work with Lorenzo Grassi and Christian Rechberger and Peter Scholl and Nigel P. Smart.
2. [KOR+17] Faster Secure Multi-Party Computation of AES and DES Using Lookup Tables, published at ACNS 2017, joint work with Marcel Keller and Emmanuela Orsini and Peter Scholl and Eduardo Soria-Vazquez and Srinivas Vivek.
3. [RSS17] Modes of Operation Suitable for Computing on Encrypted Data, published at ToSC 2017, joint work with Nigel P. Smart and Martijn Stam.
4. [KPR18] Overdrive: Making SPDZ Great Again, published at EUROCRYPT 2018, joint work with Marcel Keller and Valerio Pastro.
5. [RW19a] MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security, published at INDOCRYPT 2019, joint work with Tim Wood.

Although any of my results would have been hard or sometimes impossible to achieve without my great co-authors, for transparency reasons I will list my contributions to the above papers:

1. Role in [GRR+16] was to implement and benchmark the protocols of Naor-Reingold PRF, MiMC, AES and LowMC (the M4R variant) in SPDZ as well as proving the security reduction for the Legendre PRF.
2. Role in [KOR+17] was to implement, optimize and benchmark all the 3-DES and AES variants using SPDZ.
3. Role in [RSS17] was to investigate existing modes of operation which are highly parallelizable and transfer the security proofs from the Boolean field case $\mathbb{F}_{2^k}$ to $\mathbb{F}_p$. I also implemented and benchmarked MiMC and Legendre inside all proposed modes (PMAC, HtMAC and OTR).
4. Role in [KPR18] was to optimize the SHE code in SPDZ to work for LowGear protocol and do a more thorough analysis of the ZK proofs involved.
5. Role in [RW19a] was to co-design the daBits protocol, come up with several optimizations and implemented the protocols using MP-SPDZ framework.

In this thesis I have added some more details where it was needed in order to be able to follow the ideas completely without skimming through different papers. These details include a more complete description of the Overdrive protocols, more details on evaluating SBoxes in [KOR+17] and a few updates on the Marbled Circuits [RW19a] paper on how to realise different share conversions. I have also been a co-author of the following publications:

1. [KRSW18] Reducing Communication Channels in MPC, published at SCN 2018, joint work with Marcel Keller and Nigel P. Smart and Tim Wood.
2. [MRSV19] EPIC: Efficient Private Image Classification (or: Learning from the Masters), published at CT-RSA 2019, joint work with Eleftheria Makri and Nigel P. Smart and Frederik Vercauteren.
3. [AGP+19] Feistel Structures for MPC, and More, published at ESORICS 2019, joint work with Martin R. Albrecht and Lorenzo Grassi and Léo Perrin and Sebastian Ramacher and Christian Rechberger and Arnab Roy and Markus Schofnegger.
4. [AOR+19] Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE, published at WAHC 2019, joint work with Abdelrahaman Aly and Emmanuela Orsini and Nigel P. Smart and Tim Wood.

There are three more unpublished manuscripts which have been submitted to Eurocrypt 2020 [RST+19, GLR+19, CKR+19]. A special mention goes to the plenty of ideas that did not work and here I am thankful to my supervisor for having huge amounts of patience and enthusiasm to convince me that sometimes it might be a better idea to pursue different problems in order to avoid being stuck.

Chapter 2

Preliminaries

In this chapter we provide some basic notation necessary to follow the content of the thesis. Most of the notation can be understood by following basic undergraduate Computer Science textbooks. Then we proceed with informally describing the UC framework of Canetti [Can01] used throughout the security proofs of our protocols. The last two sections contain proof descriptions of two essential protocols, as used to realise commitments and secure coin-tossing within the UC model.

2.1 Notation

To define various arithmetic used here, one needs to start with a finite ring R equipped with the usual addition and multiplication $(+, \cdot)$. Next, the notation $R/(nR)$ represents the quotient group generated by the ideal (nR) inside R, thus splitting R into n distinct classes. Sometimes this is also denoted as R_n. In the next chapters we have various flavours of finite fields: i) $\mathbb{F}_p \cong R/pR$ where p is a prime, ii) $\mathbb{F}_{2^k} \cong R_2[X]/f(X)$ where $R_2[X]$ is the set of polynomials with binary coefficients over R and $f(X)$ is a irreducible polynomial of degree 2^k ($\mathsf{deg}(f) = 2^k$) or iii) $\mathbb{F}_2^k$ which is the set of bit-strings of length k where addition and multiplications are defined as the component-wise XOR and AND respectively. Sets are usually written using capital letters unless stated otherwise. We define $\mathcal{R}$ as the set of real numbers and $\mathcal{R}^+$ the set of positive real numbers. As usual $\mathbb{N}$ and $\mathbb{Z}$ are the set of natural numbers and integer values respectively. The $x := y$ operator is used to define certain variables whereas $x \leftarrow y$ denotes that x is assigned value y. We also define $|x|$ the absolute value of x. On the other hand $|X|$ denotes the number of elements in the set X.

At different times we would need to sample according to a certain distribution which we denote $x \xleftarrow{\$} X$. If the distribution is not specified then assume that it is uniform across X, or each element in the set X has an equal chance of being chosen. We will often denote A as an adversary or the set of corrupted parties and H as the set of honest parties.

2.2 Some complexity theory

Here we recall few definitions used to measure the complexity of an attacker or the efficiency of a protocol. This is also called the asymptotic notation:

Definition 1. *(Polynomial function) We say that $f : \mathbb{N} \mapsto \mathcal{R}$ is a polynomial function in $\mathbb{N}$ or $f \in \mathsf{poly}(n)$ if there exists a polynomial $p : \mathbb{N} \mapsto \mathcal{R}$ such that for every $c \in \mathbb{N}$ there is a constant $n_0 \in \mathbb{N}$:*

$$\forall c \in \mathbb{N}, \exists n_0 \in \mathbb{N} \textit{ such that } \forall n > n_0, f(n) \leq p(n).$$

We can now proceed to the meaning of a negligible function:

Definition 2. *(Negligible function) We say that $f : \mathbb{N} \mapsto \mathcal{R}$ is a negligible function or $f \in \mathsf{negl}(n)$ if for every positive polynomial $p : \mathbb{N} \mapsto \mathcal{R}$, for all $c \in \mathbb{N}$ we have*

$$\forall c \in \mathbb{N}, \exists n_0 \in \mathbb{N}, \textit{ such that } \forall n > n_0, f(n) < \frac{1}{p(n)}.$$

Sometimes we also need to give worst case complexity of our protocols using the big-oh notation:

Definition 3. *(Big-oh notation) We say that $f : \mathcal{R} \mapsto \mathcal{R}$ belongs to $O(g(n))$ or $f \in O(g(n))$ if and only if there exists two positive constants $n_0 \in \mathbb{N}$ and $M \in \mathcal{R}^+$:*

$$\exists n_0 \in \mathbb{N}, \exists M \in \mathcal{R}^+, \textit{ such that } \forall n > n_0, |f(n)| \leq Mg(n).$$

The next basic notion we need is an informal definition of a Turing Machine (TM). A deterministic Turing Machine is a finite state machine with an auxiliary tape onto which it can read or write according to an internal state. A deterministic TM is able to do one step at a time, whereas a non-deterministic TM can branch out on multiple actions, i.e. going from one state to a finite number of state in a single step. A probabilistic polynomial Turing Machine (PPT) is able to branch out non-deterministically from a single state to multiple states with specific probabilities and make random decisions with the additional condition that its running time is polynomial. These TM flavours (deterministic TM, non-deterministic TM, PPT) are all equivalent sometimes with an exponential increase in the number of states.

2.3 Probabilities

Proving the security of MPC protocols often reduces to some indistinguishability arguments between two transcripts or distributions. In this thesis there are two variants of these arguments being done: computational security indexed by parameter κ and statistical security indexed by parameter sec.

We now need to quantify the distance between two distributions:

Definition 4. *(Statistical distance) We define the statistical distance between two distributions $\mathcal{D}$ and $\mathcal{E}$ over a sample space Ω:*

$$\Delta(\mathcal{D}, \mathcal{E}) := \frac{1}{2} \sum_{x \in \Omega} \Big| \Pr_{X \xleftarrow{\$} \mathcal{D}(\Omega)} [X = x] - \Pr_{Y \xleftarrow{\$} \mathcal{E}(\Omega)} [Y = x] \Big|.$$

To define statistical indistinguishability we first need to index the distributions by their statistical parameters $\mathsf{sec} \in \mathbb{N}$:

Definition 5. *(Statistical indistinguishability) Let* $\{\mathcal{D}\}_{\mathsf{sec}\in\mathbb{N}}$ *and* $\{\mathcal{E}\}_{\mathsf{sec}\in\mathbb{N}}$ *be two sets of distributions indexed by* sec*. We say that these two distributions are statistically indistinguishable (close) with statistical parameter* sec *if there exists a negligible function* $\mathsf{negl} : \mathbb{N} \mapsto \mathcal{R}$ *and there exists an integer* S *such that for all* $\mathsf{sec} > S$ *:*

$$\Delta(\mathcal{D}, \mathcal{E}) < \mathsf{negl}(\mathsf{sec}).$$

For computational indistinguishability the distributions are now indexed by $\kappa \in \mathbb{N}$ and quantified against any PPT machine $\mathsf{Adv}(1^\kappa)$ running in polynomial time $\mathsf{poly}(\kappa)$. This means that if a challenger samples a polynomial number of samples $\mathsf{poly}(\kappa)$ from any family of distribution (either $\mathcal{D}$ or $\mathcal{E}$) then any PPT $\mathsf{Adv}(1^\kappa)$ is unable to distinguish from which distribution those samples came from, given a polynomial computing power $\mathsf{poly}(\kappa)$. The definition given below is equivalent to $|\Pr[t \xleftarrow{\$} \mathcal{D}; \mathsf{Adv}(1^\kappa, \mathcal{D}) = 1] - \Pr[t \xleftarrow{\$} E; \mathsf{Adv}(1^\kappa, \mathcal{E}) = 1]| \in \mathsf{negl}(\kappa)$.

Definition 6. *(Computational indistinguishability) Let* $\{\mathcal{D}\}_{\kappa\in\mathbb{N}}$ *and* $\{\mathcal{E}\}_{\kappa\in\mathbb{N}}$ *be two sets of distributions indexed by* κ*. We say that these two distributions are computational indistinguishable (close) with computational parameter* κ *if for any PPT adversary* Adv*, there exists a negligible function* $\mathsf{negl} : \mathbb{N} \mapsto \mathcal{R}$ *and there exists an integer* K *such that for all* $\kappa > K$*:*

$$\Delta(\mathsf{Adv}(1^\kappa, \mathcal{D}), \mathsf{Adv}(1^\kappa, \mathcal{E})) < \mathsf{negl}(\kappa).$$

2.4 Universal Composability

In order to prove the security of MPC protocols throughout this thesis we will use Canetti's framework [Can01] of Universal-Composability (UC) security. Since it is hard to define what it means for a protocol to be "secure" this is modeled more formally as a protocol which securely realizes a specific functionality. The UC framework is an essential tool for cryptographers who wish to prove complex protocols by breaking them down into sub-protocols in a modular way, compose their smaller proofs and still be secure even under concurrent executions.

Stripping away the complexity of a real world MPC protocol, ideally to compute a joint function we would have parties submit their inputs to a trusted third party $\mathcal{F}$ and wait for the trusted party to send the output of the function back to the parties. Since the ideal world does not exist we would like to create a protocol Π which behaves as if it would emulate a trusted third party but keeping the parties' inputs private. Roughly speaking, proving security using Canetti's framework [Can01] reduces to construct a simulator $\mathcal{S}$ which can act on behalf of the honest parties without knowing their inputs and just interacting with the ideal functionality $\mathcal{F}$. If any external environment $\mathcal{Z}$ is unable to distinguish between a transcript created by $\mathcal{S}$ interacting with the ideal functionality computing f and the real protocol Π then we say that the protocol is secure. All the entities involved here such as the environment

$\mathcal{Z}$, simulator $\mathcal{S}$ or the parties $P_1, \ldots, P_n$ running the protocol are Interactive Turing Machines (ITMs, more details in [Can01]) where each party has two types of tapes: incoming and outgoing tapes through which they send messages between each other.

The real world (or the concrete protocol) consists of n ITMs $P_1, \ldots, P_n$ which are also known as the parties executing the protocol Π. The adversary Adv controls a subset $A \subset \{P_1, \ldots, P_n\}$. The set A is called the set of corrupted parties. The ITMs in A execute all the instructions given by the adversary Adv through the I/O tapes.

The ideal world consists of the ideal functionality $\mathcal{F}$, simulator $\mathcal{S}$ and adversary Adv as well as the n dummy parties $\hat{P}_1, \ldots, \hat{P}_n$ which only role is to forward their inputs to $\mathcal{F}$ and get the output. The simulator's task is to emulate the protocol Π using just the information from $\mathcal{F}$ - the functionality description usually involves some adversary and that is $\mathcal{S}$. To be able to create a consistent transcript $\mathcal{S}$ plays the role of the adversary as well by executing Adv commands on behalf of the corrupted parties. The Simulator also has the additional power of extracting the corrupted parties inputs since they are forwarded by the adversary Adv. Note that Adv simply acts as a proxy for the adversarial environment $\mathcal{Z}$.

The experiment flow is the following. The challenger flips a coin $b \xleftarrow{\$} \{0, 1\}$. If $b = 0$ then environment $\mathcal{Z}$ will interact with the real world, otherwise $\mathcal{Z}$ will play with the ideal world. The environment will first set the inputs of the parties $P_1, \ldots, P_n$. Note that at this point from $\mathcal{Z}$'s point of view the execution is the same for all possible b's. After setting the inputs then it is the simulator job (if $b = 1$) to make the transcript distribution be indistinguishable from the real transcript. The protocol is secure if the environment has no better strategy than flipping a random coin to reply with the correct bit b chosen by the challenger. More formally, the UC security of a protocol is defined as follows:

Definition 7. *We say that a protocol* Π *implements UC securely an ideal functionality* $\mathcal{F}$ *if for any possible PPT* Adv *there exists a PPT simulator* $\mathcal{S}$ *such that any PPT environment* $\mathcal{Z}$*, any input* z *then the following distributions are equivalent:*

$$\mathbf{REAL}_{\Pi,\mathsf{Adv},\mathcal{Z}}(z) \equiv \mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(z).$$

One key-point of UC security is that it is composable. To start composing functionalities we need to define the UC hybrid model. In particular, to prove that Π realizes a functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model the parties have access to some black-box functionality $\mathcal{G}$ for which the simulator $\mathcal{S}$ can program its output. Although the simulator $\mathcal{S}$ has this extra power it needs to satisfy the output distribution of $\mathcal{G}$ as in the real world otherwise the environment can trivially distinguish between real and ideal world.

All of our proofs are constructed in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model (Random Oracle Model, see Figure 2.1) which means that in the idela worl $\mathcal{S}$ can program the $\mathcal{F}_{\mathsf{RO}}$ outputs on parties inputs, including the adversary calls to $\mathcal{F}_{\mathsf{RO}}$. Due to the infeasibility of keeping track of all incoming inputs in polynomial time, the $\mathcal{F}_{\mathsf{RO}}$ is implemented using a hash function such as SHA-256 or SHA-3. Throughout its existence, the $\mathcal{F}_{\mathsf{RO}}$ has received some amount of criticism due to some artificial protocols that are proven

$\mathcal{F}_{\mathsf{RO}}$

Init: On input $(\mathsf{Init}, X, \mathsf{sid})$ from all parties initialize an empty dictionary $\mathtt{LT}$ with codomain X.

Random: On input (sid, x) if $(\mathsf{sid}, x) \in \mathtt{LT}$ set $o \leftarrow \mathtt{LT}[\mathsf{sid}, x]$. Otherwise sample $o \xleftarrow{\$} X$ and store it into $\mathtt{LT}[\mathsf{sid}, x]$. Send o to all parties.

Figure 2.1: $\mathcal{F}_{\mathsf{RO}}$ random oracle functionality.

in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model but cannot be securely instantiated by any hash function. Nevertheless, it is widely believed that using a hash function mitigates any practical attacks on $\mathcal{F}_{\mathsf{RO}}$.

To summarize this section, proving the UC security of a protocol requires to build a simulator which needs to extract corrupted parties' inputs. The additional power is given through a setup assumption, i.e. working in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model or parties sharing some Common Reference String (CRS) programmed by the simulator. Since simulator's knowledge is limited, then it must make use of the functionality $\mathcal{F}$ as much as possible hence in practice there is usually a race between the efficiency of a protocol and how much a functionality leaks to an adversary/simulator.

2.5 Communication channels

When an MPC protocol is executed parties have to communicate to each other to realise certain functionalities, often to multiply secrets since linear operations are for free. In some cases one player P_i has to send data which can only be known by the other player P_j and no other player - this is called a *secret channel*. Throughout this thesis parties will need access to *authenticated channels* which allows for parties to send authenticated data between each other with no attacker being able to tamper with it. Note that sending data over an authenticated channel does not imply communication secrecy.

Once the parties are able to send data individually over authenticated channels then it is straightforward to build a *broadcast channel* assuming a collision resistant hash function h. This channel gives the ability for parties to broadcast i.e. to send and receive data from all parties. The broadcast channel functionality $\mathcal{F}_{\mathsf{Broadcast}}$ is described in Figure 2.2 along with its protocol implementation $\Pi_{\mathsf{Broadcast}}$ in Figure 2.3.

2.6 Two simple UC proofs

Having the broadcast functionality described above we now proceed with two simple examples of proving UC security: building commitments and a multiparty coin tossing.

$\mathcal{F}_{\mathsf{Broadcast}}$

Init: On input $(\mathsf{Init}, \mathsf{sid})$ from all parties set $\mathsf{Abort}_{\mathsf{sid}} \leftarrow \mathsf{False}$.

Broadcast: On input $(\mathsf{Broadcast}, \mathsf{sid}, x^i)$ from P_i or from $\mathcal{S}$ if $i \in A$:

1. If $i \in H$ then send x^i to all other parties P_j where $j \neq i$ (including $\mathcal{S}$).
2. If $i \in A$ then wait for the adversary to input the values x^j for honest parties P_j where $j \in H$. If any of $x^j \neq x^i$ then set $\mathsf{Abort}_{\mathsf{sid}} \leftarrow \mathsf{True}$.

Verify: On input $(\mathsf{Verify}, \mathsf{sid})$ from all parties (including $\mathcal{S}$) await for $\mathcal{S}$ to message back: if reply is OK and $\mathsf{Abort}_{\mathsf{sid}}$ is set to False then send OK to all parties. Otherwise send Abort to all parties and halt.

Figure 2.2: $\mathcal{F}_{\mathsf{Broadcast}}$ functionality.

$\Pi_{\mathsf{Broadcast}}$

Init: On input $(\mathsf{Init}, \mathsf{sid})$ from all parties, each P_i initializes str^i with an empty string and agree to a collision resistant hash function H.

Broadcast: On input $(\mathsf{Broadcast}, \mathsf{sid}, x)$ from P_i

1. Party P_i sends x to all parties over an authenticated channel.
2. All parties P_j for $j \neq i$ update $str \leftarrow str||x$ where $a||b$ denotes concatenation of two strings a, b.

Verify: On input $(\mathsf{Verify}, \mathsf{sid})$ for all parties, each party P_i does the following:

1. Compute $h^i = H(str^i)$ and send h^i to all parties P_j where $j \neq i$ over an authenticated channel.
2. Await for h^j for all $j \neq i$. If any $h^j \neq h^i$ then send Abort to all parties and halt.

Figure 2.3: Broadcast protocol.

2.7 Commitments

Commitments are a useful cryptographic tool to ensure that parties do not change their inputs once they commit to them. Perhaps the straightforward analogy is that commitments are similar to sealed envelopes: once you send it via post the content is hidden and cannot be changed until the envelope reaches its destination. Intuitively speaking, cryptographic commitments must satisfy two properties:

1. Binding: opening two commitments: $\mathsf{Commit}(m)$ and $\mathsf{Commit}(m')$ where $m \neq m'$ should yield two different values with high probability.
2. Hiding: two commitments $\mathsf{Commit}(m)$ and $\mathsf{Commit}(m')$ are indistinguishable to an adversary. This is the equivalent to the CPA property of an encryption scheme for commitments.

They were first formalized by Goldreich in 1985 [Gol95] as unconditionally binding and hiding. Later

Π_{Commit}

Init: On input $(\mathsf{Init}, X, \mathsf{sid})$ from all parties do the following:

1. Start an instance of the Random Oracle by calling $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{Init}, \mathsf{sid}, \{0,1\}^{2\cdot\kappa})$.
2. Start an instance of the Broadcast channel by calling $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\mathsf{Init}, \mathsf{sid})$.

Commit: To commit on input x party P_i does the following:

1. Sample $r \xleftarrow{\$} \{0,1\}^{2\cdot\kappa}$ and call $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{sid}||i||x||r)$ where $a||b$ denotes the concatenation two strings a, b. Store the output of $\mathcal{F}_{\mathsf{RO}}$ as $\tau_x \in \{0,1\}^{2\cdot\kappa}$ into $\mathtt{LT}[\mathsf{sid}, i, \tau_x] := (x, r)$.
2. Calls $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\mathsf{Broadcast}, \mathsf{sid}, \tau_x)$.

Open: In order to open a commitment with identifier τ_x:

1. Party P_i retrieves $(x, r) := \mathtt{LT}[\mathsf{sid}, i, \tau_x]$ then uses $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\mathsf{Broadcast}, \mathsf{sid}||i||x||r)$. As a result each party P_j receives a message m^j.
2. Parties now check whether the broadcast was correct by calling $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\mathsf{Verify}, \mathsf{sid})$. If they receive OK then continue, otherwise abort.
3. All parties P_j call $\mathcal{F}_{\mathsf{RO}}$ with input m_j to get the output τ_x^j.
4. If $\tau_x^j = \tau_x$ then party P_j parses x from m^j and outputs x on its local tape. Otherwise call $\mathcal{F}_{\mathsf{Broadcast}}$ with $(\mathsf{Broadcast}, \mathsf{Abort}, \mathsf{sid})$, output locally $\perp$ and then halt.

Figure 2.4: $\mathcal{F}_{\mathsf{RO}}$ based commitment protocol.

it was proved by Canetti and Fischlin [CF01] that in order to universally compose commitments one needs to downgrade the security from unconditionally to computationally hard problems and have some setup assumptions such as a CRS. The commitments used throughout most MPC protocols (also used in Figure 2.4) are based on the Random Oracle assumption proposed by Hofheinz and Müller-Quade [HM04].

Theorem 8. *Protocol Π_{Commit} implements UC securely $\mathcal{F}_{\mathsf{Commit}}$ in the $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{Broadcast}}$-hybrid model against any static, malicious and computationally bounded adversary which corrupts at most $n-1$ parties.*

Proof. We need to construct a simulator $\mathcal{S}$ such that no adversary environment is able to distinguish between the real protocol Π_{Commit} where Adv is controlling $n-1$ parties and $\mathcal{S}$ interacting with $\mathcal{F}_{\mathsf{Commit}}$. It might be useful to remind that the simulator is acting as a proxy between the adversary and the corrupted parties, delivering messages back-and-forth according to the adversary wishes. All that needs to be done is to show how the simulator creates the messages for the honest parties such that the transcript is indistinguishable from the real protocol.

The trivial case is when the corrupted parties follow the protocol honestly. Since the simulator has the extra power to program the random oracle inputs to $\mathcal{F}_{\mathsf{RO}}$ then the simulation can be done in the following way: every time an honest party P_i commits a secret x in the ideal world the simulator samples

$\mathcal{F}_{\mathsf{Commit}}$

Init: On input $(\mathsf{Init}, X, \mathsf{sid})$ from all parties create an empty dictionary $\mathtt{LT}$ which indexes a tuple of the form $(\mathsf{sid}, i, \tau_x)$ where sid is the session identifier, i is the party id and τ_x is an unique handle associated to some value $x \in X$.

Commit: On input $(\mathsf{Commit}, \mathsf{sid}, i, x)$ from P_i and $(\mathsf{Commit}, \mathsf{sid}, i, \perp)$ from the other parties the functionality creates a new handle id_x and stores values x into a dictionary $\mathtt{LT}[\mathsf{sid}, i, \tau_x] := x$. Then output id_x to all parties (including $\mathcal{S}$).

Open: On input $(\mathsf{Open}, \mathsf{sid}, i, \mathsf{id}_x)$ from party P_i then retrieve $x := \mathtt{LT}[\mathsf{sid}, i, \tau_x]$ and output x to $\mathcal{S}$ and all honest parties. Otherwise wait for $\mathcal{S}$ to output OK or Abort. If message is OK then output x to all honest parties. If message from $\mathcal{S}$ is Abort then send Abort to parties and halt.

Figure 2.5: Commitment functionality.

$\tau_x \xleftarrow{\$} \mathcal{U}(\{0,1\}^{2\cdot\kappa})$ and emulates $\mathcal{F}_{\mathsf{Broadcast}}$ with the adversary Adv. To simulate the $(\mathsf{Open}, \mathsf{sid}, i, \tau_x)$ command the simulator $\mathcal{S}$ waits for the $\mathcal{F}_{\mathsf{Commit}}$ to output the secret x. Then the simulator samples $r \xleftarrow{\$} \mathcal{U}(\{0,1\}^{2\cdot\kappa})$ and programs the output of $\mathcal{F}_{\mathsf{RO}}$ to be $\mathtt{LT}[\mathsf{sid}, i, \tau_x] := (x, r)$. The remaining messages are delivered by the simulator when $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Broadcast}}$ with Adv on behalf of the honest parties by sending the string $(\mathsf{sid}||i||x||r)$.

There are two corner cases here. Since r is sampled randomly at each run by the simulator there might be situations when some $r_{\mathsf{prev}} = r$ was sampled before. In this case the environment $\mathcal{Z}$ will detect that its playing with the ideal world. The case when there is a collision on r is very low though: if Adv queries the simulator q times then, using birthday bound, there will be a chance of collision of approximatively $q^2/2^{2\cdot\kappa}$ which is negligible in κ assuming the adversary has access to $q \in \mathsf{poly}(\kappa)$ queries.

The second case is when a corrupted party executes the Open command without querying $\mathcal{F}_{\mathsf{RO}}$ by producing a fake τ_x. In this situation the simulator has no information on how to open the commitment so it will send Abort to the $\mathcal{F}_{\mathsf{Commit}}$. In the real world the parties will abort as well unless the adversary comes up with some string $(\mathsf{sid}||i||x||r)$ later for which $\mathcal{F}_{\mathsf{RO}}$ would output τ_x. Using the fact that the random oracle outputs random strings of length $2 \cdot \kappa$ then the probability of the adversary finding a pre-image to $\mathcal{F}_{\mathsf{RO}}$ is $2^{-2\cdot\kappa}$ hence negligible in κ.

In both cases the probability of $\mathcal{Z}$ distinguishing between the ideal and real world are negligible in κ which concludes our proof.

□

2.8 Coin tossing

Originally introduced by Blum [Blu82], and motivated by flipping a coin using a telephone, this shortly became one of the building blocks in many MPC protocols: the ability to sample random coins without

$\mathcal{F}_{\mathsf{Rand}}$

Init: On input $(\mathsf{Initialise}, \mathsf{sid}, \mathbb{F})$ from all parties wait for incoming messages.

Random: On input $(\mathsf{Random}, \mathsf{sid})$ from all parties sample $r \xleftarrow{\$} \mathcal{U}(\mathbb{F})$ and send it to $\mathcal{S}$. Await for $\mathcal{S}$ to message back: if reply is OK then send r to all parties. If reply is Abort then send Abort to honest parties and halt.

Figure 2.6: Functionality for generating a random number

a trusted third party. For this purpose, Blum used one way functions based on the quadratic residues problem.

In Blum's protocol there are two parties: Alice and Bob who first agree on a public modulus $n = p \cdot q$ which is a product of two distinct primes where only Bob knows p and q. The protocols goes as follows: Alice selects a random value $x \bmod n$ and sends to Bob $x^2 \bmod n$. Unlike Alice, Bob knows the prime factors p, q so he can compute all four square roots of x^2 (two modulo p and two modulo q): $(x, n-x)$ and $(x', n-x')$. Yet Bob has no idea which group Alice's root comes from. Bob makes a guess $(x^*, n-x^*)$ and sends this to Alice. If the value x^* that Alice receives is different from x then she can compute the factorization of N and output p, q to Bob and set a global random coin to 1. Otherwise if Alice receives $x^* = x$ then both parties output the coin 0.

For our case we need an efficient n party coin-toss protocol which is secure against a dishonest majority of malicious parties. In the context of a dishonest majority to realise a secure coin-toss we need stronger some stronger assumptions such as a trapdoor or random oracle [Can01, CLOS02]. One folklore protocol which is used in our constructions as well is for parties to select a random seed to then commit to it. After this all parties open their commitments and the random coin is the XOR sum of the opened seeds. The drawback of the following method is that parties need to commit per each random coin sampled incurring a large communication cost from the commitments. Instead they can use the opened value as a global seed to a random oracle. In practice the oracle will be implemented via AES in counter mode where the symmetric key is instantiated as the global seed.

Theorem 9. *Protocol Π_{Rand} implements $\mathcal{F}_{\mathsf{Rand}}$ in the $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{Commit}}$-hybrid model against any UC static, malicious and computationally bounded adversary which corrupts at most $n-1$ parties.*

Proof. We need to construct a simulator $\mathcal{S}$ such that any environment $\mathcal{Z}$ cannot distinguish between Π_{Rand} or $\mathcal{S}$ interacting with $\mathcal{F}_{\mathsf{Rand}}$. To simulate a Random command the simulator constructs dummy inputs by selecting r_i^H at random for the honest parties $i \in H$ as well as handles $\tau_{r_i^H}$ and inputs them to $\mathcal{F}_{\mathsf{Commit}}$. After the emulation of $\mathcal{F}_{\mathsf{Commit}}$ with the adversary the simulator obtains r_j^A for all $j \in A$. After emulating $\mathcal{F}_{\mathsf{Commit}}$, $\mathcal{S}$ is instructed to open the commitements via $(\mathsf{Open}, \mathsf{sid}, i, \tau_{r_i})$ for $i \in A$ it will broadcast the values $(i, r_i, \tau_{r_i})_{i \in H}$. Now the simulator computes the XOR sum $o :=$

Π_{Rand}

Init: On input $(\mathsf{Init}, \mathsf{sid}, \mathbb{F})$ from all parties do the following:

1. Call $\mathcal{F}_{\mathsf{RO}}$ with $(\mathsf{Init}, \mathsf{sid}, \mathbb{F})$.
2. Call $\mathcal{F}_{\mathsf{Commit}}$ with $(\mathsf{Init}, \mathsf{sid}, \mathbb{F})$.
3. Set the output coins to be values in $\mathbb{F}$.

Random: On input $(\mathsf{Random}, \mathsf{sid})$ from all parties do the following:

1. Each party samples $r_i \xleftarrow{\$} \{0,1\}^{2\cdot\kappa}$.
2. Each party calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{Commit}, \mathsf{sid}, i, r_i)$ to receive a handle τ_{r_i}. Then they open the commitments through the $(\mathsf{Open}, \mathsf{sid}, i, \tau_{r_i})$ command.
3. Next parties call $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{sid}, \oplus_{i=1}^{n} r_i)$ to get a string $o \in \{0,1\}^{2\cdot\kappa}$.
4. Output o as the public coin toss result.

Figure 2.7: Π_{Rand} coin-tossing protocol

$\bigoplus_{i\in H} r_i^H \bigoplus_{j\in A} r_j^A$. In the end $\mathcal{S}$ calls $\mathcal{F}_{\mathsf{Rand}}$ to get the x and programs the output of $\mathcal{F}_{\mathsf{RO}}$ to o in the last step of the protocol.

The only things we need to argue now are transcript distribution and abort probability. The transcript $\mathcal{Z}$ sees is uniform since o has at least one random r_i^H (the set H is non-empty). Next, as in the simulation of $\mathcal{F}_{\mathsf{Commit}}$ the adversary will notice that is interacting with the functionality if it receives more than once the same output from $\mathcal{F}_{\mathsf{Rand}}$ on two different inputs or predicts the output of $\oplus_{i=1}^{i} r_i$. This will only happen with non-negligible probability as the inputs to $\mathcal{F}_{\mathsf{RO}}$ are $2\cdot\kappa$ bits long, hence the chance of two inputs yielding the same output is $q^2/2^{2\cdot\kappa}$ (using the birthday bound) where $q \in \mathsf{poly}(\kappa)$ represents the number of queries an adversary can make to $\mathcal{F}_{\mathsf{RO}}$.

□

As a side note, the coins that Π_{Rand} outputs are going to be biased since there is no way honest parties can prevent malicious parties to abort. One way to decrease the bias is to repeat the protocol a number of times [BOO10] if adversary aborts. Although an adversary can abort at any point we can push corrupted parties out of the computation if identifiable abort is applied to introduce incentives for cheaters to behave correctly [BOS16] at some extra complexity cost. Nevertheless, the coins will have zero bias as long as all parties follow the protocol.

Chapter 3

Multiparty computation for dishonest majority

Most modern MPC protocols where a majority of the parties are corrupt can be split into two phases called the preprocessing phase and the online phase [DPSZ12,BDOZ11,DKL+13,WRK17a,WRK17b, DGN+17, HIMV19]. In the preprocessing phase parties work together to produce some correlated randomness using public key cryptography, then using it later in the online phase in order to compute the actual function on secret inputs. In this chapter we review two protocols for realising MPC with dishonest majority: one based on secret sharing (SPDZ [DPSZ12]), and the other based on garbled circuits (BMR [BMR90]). Description of the SPDZ protocol will focus on the online phase: while in the next chapters we show how to obtain more efficient preprocessing (Chapter 4), while also building applications on top of SPDZ (Chapters 5, 6, 7). The BMR protocol will be useful to understand the contributions in Chapter 8 to switch efficiently between SPDZ and constant round protocols such as BMR.

3.1 Secret Sharing

One can secret share x amongst a set of n parties by giving away some shares $x^{(i)}$ to all parties P_i such that there is a reconstruction algorithm to fully recover x but individual parts $x^{(i)}$ reveal no information about the original secret x. Perhaps the most popular secret sharing scheme, is the one known as Shamir secret sharing scheme [Sha79] where the secret reconstruction is done through polynomial interpolation of $t+1$ secrets since the share x_i of each party represents a point on a t-degree curve. In our protocols we will use something simpler called additive secret sharing scheme where all parties need to contribute with their share to reconstruct the secret. That is, each party will get a random $x^{(i)}$ subject to $x = x^{(1)} + \cdots + x^{(n)}$. We define a secret shared value of x as the tuple

$$[\![x]\!] := (x^{(1)}, \ldots, x^{(n)}) \text{ such that } \sum_{i=1}^{n} x^{(i)} = x.$$

3.2 Authentication

To prevent malicious parties mounting some additive attacks or changing their shares in the protocol, all the protocols will use a global MAC key α which no one knows and is secret shared amongst all parties, i.e. $[\![\alpha]\!]$. This should not be confused with the traditional sense of a MAC (message authentication code) scheme [BGR95] which ensures integrity and confidentiality when one party sends an encrypted message to another party. This extra sharing $[\![\alpha]\!]$ of an unknown MAC key mitigates additive attacks on the original secret $[\![x]\!]$ when parties cheat on their shares. To relate $[\![x]\!]$ and $[\![\alpha]\!]$ parties need to produce a sharing of $[\![\alpha \cdot x]\!]$ called $\gamma(x)$ such that $\gamma(x) = \sum_{i=1}^{n} \gamma^{(i)}(x)$. We define an authenticated share, or an additive share with the corresponding shares of the MAC on x as

$$[\![x]\!] := (x^{(1)}, \ldots, x^{(n)}, \gamma^{(1)}(x), \ldots, \gamma^{(n)}(x), \alpha^{(1)}, \ldots, \alpha^{(n)}).$$

Whenever parties broadcast their shares $x^{(i)}$ to reconstruct the secret $[\![x]\!]$ an extra equation is checked to ensure share consistency, namely whether $[\![\alpha \cdot x]\!] - [\![\alpha]\!] \cdot [\![x]\!] = 0$. This procedure is called a MAC check and will be described later in this section.

Finally, the **Share** command gives parties access to random, additive shares of a secret value stored in $\mathcal{F}_{\mathsf{ABB}}$. This essentially assumes the underlying MPC protocol uses additive secret sharing, but is only used for the Naor-Reingold PRF protocol (Section 6.4).

3.3 Arithmetic Black Box Model

To model MPC in a modular way we first need to define the ideal functionalities with their specific arithmetic. In the ideal world parties input their values to the functionality and then call various commands to compute any functions. The standard arithmetic blackbox makes use of a generic field $\mathbb{F}$ denoted by $\mathcal{F}_{\mathsf{ABB}}$ given in Figure 3.1. Each value stored in this functionality is associated with a unique identifier that is given to all parties. As specified in the prevous section, let $[\![x]\!]$ denote the identifier for an authenticated value x that is stored by the functionality and $A \subset \{1, \ldots, n\}$ be the index set of corrupted parties. Note that $\mathcal{F}_{\mathsf{ABB}}$ is built in the $\mathcal{F}_{\mathsf{Prep}}$ hybrid model described in Figure 3.2. One can think of $\mathcal{F}_{\mathsf{ABB}}$ as a better abstraction of $\mathcal{F}_{\mathsf{Prep}}$ as in the latter the functionality operates on shares rather than on secrets since for every secret generated it has to wait from the simulator's response and then sample the honest shares such that they sum up to the intial sampled secret.

3.3.1 How to evaluate circuits using $\mathcal{F}_{\mathsf{ABB}}$

The usual computation model most CS graduates are used to is the Random Access Memory (RAM) based model. In the RAM model there are two finite state machines represented by a Computer Processing Unit (CPU) and a random access memory which can store some registers. To execute instructions, the CPU fetches the instruction code from RAM along with the registers indicated by the instruction [Sav98]. For example when accessing an array A using C-style programs we need specify the index of the array i to fetch the value at that position.

Functionality $\mathcal{F}_{\mathsf{ABB}}[\mathbb{F}]$

Initialize: On input $(\mathsf{Init}, \mathbb{F})$ from all parties, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Init}, \mathbb{F})$.

Input: On $(\mathsf{Input}, \mathsf{sid}, \mathsf{id}, i, x)$ from P_i and $(\mathsf{Input}, i, \mathsf{sid}, \mathsf{id}, \perp)$ from all othe parties, if $\mathsf{id} \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ then set $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] := x$.

Add$(\llbracket x \rrbracket, \llbracket y \rrbracket)$**:** On input $(\mathsf{Add}, \mathsf{sid}, \mathsf{id}, \mathsf{id}_x, \mathsf{id}_y)$ if $\mathsf{id}_x, \mathsf{id}_y \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$, retrieve $x := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_x], y := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_y]$, compute $z = x + y$ and store z into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$.

Multiply$(\llbracket x \rrbracket, \llbracket y \rrbracket)$**:** On input $(\mathsf{Multiply}, \mathsf{sid}, \mathsf{id}, \mathsf{id}_x, \mathsf{id}_y)$ if $\mathsf{id}_x, \mathsf{id}_y \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ retrieve $x := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_x], y := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_y]$. compute $z = x \cdot y$ and store z into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$.

Output$(\llbracket x \rrbracket)$**:** On input $(\mathsf{Output}, \mathsf{sid}, \mathsf{id})$ from all parties, if $\mathsf{id} \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ retrieve $x := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$ and send x to $\mathcal{S}$. If $\mathcal{S}$ replies with Abort then halt, otherwise send x to all other parties and continue.

Figure 3.1: Ideal functionality for MPC arithmetic.

When computing on encrypted data the index might be secret as well as the data in the array A of length m. How can we retrieve the secret $A[i]$ without knowing the index i? Fortunately there are multiple answers to this question such as Oblivious RAM [SvS+13]. We will consider the simplest approach, just by using $\mathcal{F}_{\mathsf{ABB}}$ operations. Given that for every TM M there exists a circuit $\mathcal{C}$ which can compute M using just additions and multiplications (see Chapter 11 in [AB09]).

If one has access to $\mathcal{F}_{\mathsf{eq}}$ which takes as input two secret registers and returns in secret shared form $\llbracket 1 \rrbracket$ if the registers are equal or $\llbracket 0 \rrbracket$ otherwise then we can build easily a protocol which can retrieve $A[i]$ where the index is secret shared as $\llbracket i \rrbracket$. The solution is simple: iterate through all indices $j \in [1 \dots m]$ and compute $\sum_{j=1}^{m} A[j] \cdot \mathcal{F}_{\mathsf{eq}}(\llbracket i \rrbracket, j)$. Since all indices j that are compared to $\llbracket i \rrbracket$ are unique it is guaranteed that $\mathcal{F}_{\mathsf{eq}}$ will return 1 in a single position which is when $j = \llbracket i \rrbracket$, adding $A[i]$ to the final sum. This has an $O(m)$ complexity since computing the sum touches every element in A.

This simple example was selected to illustrate that there is a gap between the RAM model of computation and computing on arithmetic circuits with secret data. Another example to illustrate the gap between the two models is branching on data. Branching is a very easy task in traditional programming languages which can be done in constant time whereas branching on a secret value x is an expensive task since it requires to compute the circuit on every possible value x can have.

In the end, how should we compute programs in MPC? The short answer is to unroll the program to an arithmetic circuit and let $\mathcal{F}_{\mathsf{ABB}}$ do the job. In practice this is slightly more complicated as the programmer has to rewrite chunks of the code to eliminate branching and write specific protocols (see the lookup table approach) to realise the program more efficiently.

Functionality $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}]$

Initialize: On input $(\mathsf{Init}, \mathbb{F})$ from all parties agree and store a session identifier sid and a finite field $\mathbb{F}$.

Input: On $(\mathsf{Input}, \mathsf{sid}, \mathsf{id}, i, x)$ from P_i if $\mathsf{id} \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ store it into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] := x$. For every other party $P_j \neq P_i$ send $[\![x]\!]$.

InputTuple: On $(\mathsf{InputTuple}, \mathsf{sid}, \mathsf{id}, i)$ from P_i if $\mathsf{id} \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ sample $r \xleftarrow{\$} \mathcal{U}(\mathbb{F})$, store it into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] := r$ and send r to P_i. For every other party $P_j \neq P_i$ send $[\![r]\!]$.

Add: On input $(\mathsf{Add}, \mathsf{sid}, \mathsf{id}, \mathsf{id}_x, \mathsf{id}_y)$ if $\mathsf{id}_x, \mathsf{id}_y \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$, retrieve $x := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_x], y := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_y]$, compute $z = x + y$ and store z into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$.

Multiply: On input $(\mathsf{Multiply}, \mathsf{sid}, \mathsf{id}, \mathsf{id}_x, \mathsf{id}_y)$ if $\mathsf{id}_x, \mathsf{id}_y \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ retrieve $x := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_x], y := \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_y]$. compute $z = x \cdot y$ and store z into $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$.

Linear Combination: On input $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_l, c_1, \ldots, c_l, c)$ from all parties where $\mathsf{id}_k \in \mathtt{Reg}.\mathsf{Keys}()$ store $\mathtt{Reg}[\overline{\mathsf{id}}] = \sum_{k=1}^{l} \mathtt{Reg}[\mathsf{id}_k] \cdot c_k + c$.

RandomEntry: On input $(\mathsf{RandomEntry}, \mathsf{sid}, \mathsf{id})$ if $\mathsf{id} \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ sample $r \xleftarrow{\$} \mathbb{F}$ and store $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] := r$.

RandomBit: On input $(\mathsf{RandomBit}, \mathsf{sid}, \mathsf{id})$ if $\mathsf{id} \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ sample $b \xleftarrow{\$} \{0, 1\} \in \mathbb{F}$ and store $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] := b$.

Triple: On input $(\mathsf{Triple}, \mathsf{sid}, \mathsf{id}_a, \mathsf{id}_b, \mathsf{id}_c)$ if $id_a, \mathsf{id}_b, \mathsf{id}_c \notin \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$ sample $a, b \xleftarrow{\$} \mathcal{U}(\mathbb{F})$, set $c = a \cdot b$ and store the registers $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_a] := a, \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_b] := b, \mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}_c] := c$. Output the triple (a, b, c) to all parties.

Open: On input $(\mathsf{Open}, \mathsf{sid}, \mathsf{id}, i)$ from all parties, if $\mathsf{id} \in \mathtt{Reg}_{\mathsf{sid}}.\mathsf{Keys}$,

- if $i = 0$ then send $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary and run the procedure **Wait**. If the message was $(\mathsf{OK}, \mathsf{sid})$, await an error ε from the adversary. Send $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] + \varepsilon$ to all honest parties and if $\varepsilon \neq 0$, set the internal flag $\mathtt{Abort}_{\mathsf{sid}}$ to true.
- if $i \in A$, then send $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary and then run **Wait**.
- if $i \in [n] \setminus A$, then call the procedure **Wait**, and if not already halted then await an error ε from the adversary. Send $\mathtt{Reg}_{\mathsf{sid}}[\mathsf{id}] + \varepsilon$ to P_i and if $\varepsilon \neq 0$ then set the internal flag $\mathtt{Abort}_{\mathsf{sid}}$ to true.

Check: On input $(\mathsf{Check}, \mathsf{sid})$ from all parties send Abort to all parties if $\mathsf{Abort}_{\mathsf{sid}}$ is set to True and halt. Otherwise continue and send OK to all parties. Internal procedure:

Wait: Await a message $(\mathsf{OK}, \mathsf{sid})$ or $(\mathsf{Abort}, \mathsf{sid})$ from the adversary; if the message is $(\mathsf{OK}, \mathsf{sid})$ then continue; otherwise, send the message $(\mathsf{Abort}, \mathsf{sid})$ to all honest parties and ignore all further messages to $\mathcal{F}_{\mathsf{ABB}}$ with this sid.

Figure 3.2: Preprocessing functionality for MPC

3.4 SPDZ overview

Damgård et al. [DPSZ12] introduced a novel way of computing over secret data against a dishonest majority of parties using Somewhat Homomorphic Encryption (SHE). They avoided the computational expensive Fully Homomorphic Encryption (FHE) machinery of Gentry [Gen09] and the complex Zero Knowledge (ZK) proofs of BDOZa [BDOZ11]. In the FHE case parties could just encrypt their inputs, evaluate a circuit $\mathcal{C}$ using bootstrapping and then distribute decrypt the output to obtain a share of the final result. Due to the costly operation of bootstrapping SPDZ circumvents this by only evaluating depth-1 computation using a homomorphic encryption scheme. This comes at a cost of incurring an $O(d)$ communication increase where d is the circuit depth parties want to evaluate. Nevertheless, after the costly preprocessing phase is done the SPDZ online phase is fast as it is information theoretical requiring just basic field arithmetic.

In this section we will only focus on the online phase protocol as the next chapter deals with how to obtain better preprocessing. The online phase will be securely realised by illustrating the protocols implementing the $\mathcal{F}_{\mathsf{ABB}}$ commands such as to provide inputs and perform additions and multiplications. We illustrate the online phase of SPDZ in Figure 3.3. Note that the protocol is described in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model which assumes black box access to a functionality which outputs random Beaver triples and also checks the partially opened values throughout the computation. The protocol specifics of these procedures are described later in the next chapter.

3.5 Preprocessing for SPDZ using Oblivious Transfer (OT)

For sake of completeness we briefly describe how to realise Triple command from $\mathcal{F}_{\mathsf{Prep}}$ using the MASCOT subroutines introduced by Keller et al. in [KOS16]. We give these constructions in the $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{ROT}}$-hybrid model where we assume the OT and Random OT (ROT) functionality:

$$\mathcal{F}_{\mathsf{OT}}^{1,k} : ((s_0, s_1), b) \mapsto (\perp, s_b)$$

$$\mathcal{F}_{\mathsf{ROT}}^{1,k} : (\perp, b) \mapsto ((r_0, r_1), r_b)$$

where $r_0, r_1 \xleftarrow{\$} \{0,1\}^k$ and $b \in \{0,1\}$ is the receiver's input bit.

In $\mathcal{F}_{\mathsf{OT}}^{1,k}$ one party P_S (the sender) inputs two k-bit strings s_0, s_1 while the receiver P_R inputs a choice bit $b \in \{0,1\}$. As a result $\mathcal{F}_{\mathsf{OT}}^{1,k}$ outputs the string s_b corresponding to the choice of P_R while keeping s_{1-b} secret.

In $\mathcal{F}_{\mathsf{ROT}}^{1,k}$ the receiver inputs a choice bit b. The functionality $\mathcal{F}_{\mathsf{ROT}}^{1,k}$ then samples two random k bit-strings $r_0, r_1 \xleftarrow{\$} \{0,1\}^k$ and sends them to P_S while sending only r_b to P_R.

To multiply two random secrets, MASCOT uses a generalization of these two functionalities: $\mathcal{F}_{\mathsf{OT}}^{l,k}$ and $\mathcal{F}_{\mathsf{ROT}}^{l,k}$ where they denote l sets of k-bit string OTs. The two functionalities can be efficiently instantiated using a malicious OT extension protocol [KOS15]. These ideas can be traced back from the

$\Pi_{\mathsf{Online}}[\mathsf{SPDZ}]$

Init: On input $(\mathsf{Init}, \mathbb{F})$ from all parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Init}(\mathbb{F})$ to get an additive sharing of the MAC key $[\![\alpha]\!]$. Each party P_i receives $\alpha^{(i)}$ such that $\sum_{i=1}^{n} \alpha^{(i)} = \alpha$.

Input: To provide an input x party P_i retrieves an input tuple by calling $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Input}(P_i)$ which outputs an authenticated shared mask $[\![r]\!]$ such that only P_i knows the secret r. After P_i broadcasts $x + r$ then parties adjust the sharing of x accordingly:

1. Party P_1 computes $x^{(1)} \leftarrow (x+r) - r^{(1)}$ while all other parties do $x^{(i)} \leftarrow (x+r) - r^{(i)}, \forall i > 1$.
2. Compute MAC shares: $\gamma^{(i)}(x) \leftarrow \alpha^{(i)} \cdot (x + r) - \gamma^{(i)}(r), \forall i \in [1 \dots n]$.

and store $[\![x]\!]$ as the tuple $(x^{(1)}, \dots, x^{(n)}, \gamma^{(1)}(x), \dots, \gamma^{(n)}(x))$.

Add: To add two shared values $[\![x]\!], [\![y]\!]$ parties locally add their shares and assign $[\![x + y]\!] \leftarrow [\![x]\!] + [\![y]\!]$.

Mult: To multiply two values $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Triple}$ to retrieve a random authenticated triple $[\![a]\!], [\![b]\!], [\![c]\!]$ such that $c = a \cdot b$. Then they:

1. Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Open}$ on $[\![x]\!] - [\![a]\!]$ and $[\![y]\!] - [\![b]\!]$ to get ϵ and ρ respectively.
2. To compute the sharing of the product $[\![x \cdot y]\!]$ they set $[\![z]\!] \leftarrow [\![c]\!] + \epsilon \cdot [\![b]\!] + \rho \cdot [\![a]\!] + \epsilon \cdot \rho$.

Output: To output a secret $[\![x]\!]$ parties:

1. Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ on all previous opened values. If the result is Abort then parties abort. Otherwise continue and
2. Broadcast x_i and call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ on the opened value x using the MAC shares. Abort if $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ fails.

Figure 3.3: Online phase protocol of SPDZ.

TinyOT line of work [NNOB12, LOS14, FKOS15] designed for dishonest majority multiparty computation for Boolean circuits (fields of characteristic two).

The high-level idea can be explained with a small example for the two party case, say P_A and P_B each having some secret MAC keys $\mathbf{\Delta}_A, \mathbf{\Delta}_B \in \mathbb{F}^k$. First, P_A samples a vector of τ field elements $\mathbf{a} \xleftarrow{\$} \mathbb{F}^\tau$ while P_B samples a single field element $b \xleftarrow{\$} \mathbb{F}$. Denote with $k = \log |\mathbb{F}|$. Then parties optimistically multiply $\mathbf{a} \cdot b$ using $\mathcal{F}_{\mathsf{ROT}}^{\tau k, k}$ where P_A inputs $\mathbf{a}$ and P_B inputs b to get a sharing of their product $\mathbf{c} = \mathbf{c}_A + \mathbf{c}_B = \mathbf{a} \cdot b$. This is described as the **Multiply** step in Figure 3.6. After the **Multiply** stage parties call the $\mathcal{F}_{\mathsf{Rand}}$ functionality to get 2 random vectors $\mathbf{r}, \hat{\mathbf{r}} \in \mathbb{F}^\tau$ which are used to "collapse" the vectors into field elements $a, b, c, \hat{a}, \hat{a} \cdot b$ described in Step 3.5 from Figure 3.6.

In the **Authenticate** phase (Step 3.5) parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Input}$ to add MACs on the previous data using Figure 3.5. At the core of this method lies the protocol called Correlated Oblivious Produce Evaluation with errors (COPEe) described in Figure 3.4. In this step an adversary can introduce some additive errors which are later mitigated by checking a random linear combination of the vector entries, similar to the MAC-checking procedure in SPDZ, described in more detail later on in Chapter 4.

Protocol Π_{COPEe}

A PRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^k$ is used for OT Extension. We write $\mathbb{F}_{2^k} \cong \mathbb{F}_2[X]/(f)$ where $f \in \mathbb{F}_2[X]$ is an irreducible polynomial of degree k in $\mathbb{F}_2$.

Initialize:

1. P_A samples k pairs of seeds $((\mathbf{k}_i^0, \mathbf{k}_i^1)_{i=1}^k)$ where $\mathbf{k}_b^i \in \{0,1\}^\lambda$ for all $(i,b) \in [k] \times \{0,1\}$.
2. The parties call $\mathcal{F}_{\mathsf{OT}}^{k,\lambda}$ where P_A inputs $((\mathbf{k}_0^i, \mathbf{k}_1^i)_{i\in[k]})$ and P_B inputs $\Delta_B = (\Delta_0, \ldots, \Delta_{k-1}) \in \{0,1\}^k$.
3. $\mathcal{F}_{\mathsf{OT}}^{k,\lambda}$ outputs $(\mathbf{k}_{\Delta_i}^i)_{i\in[k]}$ to P_B.

Extend: On (local) input $(x, \mathbb{F}) \in \mathbb{F} \times \{\mathbb{F}_p, \mathbb{F}_{2^k}\}$ from P_A, the parties do the following:

1. If $\mathbb{F} = \mathbb{F}_p$, fix $\mathbf{g} \leftarrow (2^0, 2^1, \ldots, 2^{k-1}) \in \mathbb{F}_p^k$, and if $\mathbb{F} = \mathbb{F}_{2^k}$, fix $\mathbf{g} \leftarrow (X^0, X^1, \ldots, X^{k-1}) \in \mathbb{F}_{2^k}^k$.
2. For each $i = 0$ to $k-1$, the parties do the following:
   a) P_A computes
   $$t_i^0 \leftarrow \langle \mathbf{g}, F(\mathbf{k}_i^0 \| j) \rangle \quad \text{and} \quad t_i^1 \leftarrow \langle \mathbf{g}, F(\mathbf{k}_i^1 \| j) \rangle$$
   and P_B computes
   $$t_i^{\Delta_i} \leftarrow \langle \mathbf{g}, F(\mathbf{k}_{\Delta_i}^i \| j) \rangle$$
   b) Both parties compute and store $j \leftarrow j + 1$.
   c) P_A computes $u_i \leftarrow t_i^0 - t_i^1 + x$ and sends u_i to P_B.
   d) P_B computes $q_i \leftarrow \Delta_i \cdot u_i + t_i^{\Delta_i}$.
   e) P_A computes $t_i \leftarrow -t_i^0$.
3. P_A locally outputs $t \leftarrow \langle \mathbf{g}, (t_0, \ldots, t_{k-1}) \rangle$ and P_B locally outputs $q \leftarrow \langle \mathbf{g}, (q_0, \ldots, q_{k-1}) \rangle$.

Figure 3.4: Protocol Π_{COPEe} from MASCOT [KOS16].

Finally, to mitigate a possible additive error on c, i.e. all the above procedures authenticated $c = a \cdot b + \epsilon$ instead of $a \cdot b$, Keller et al. use a standard Beaver sacrifice trick with a small twist since the multiplier b is fixed here. The reader can consult their paper to see details on security proofs, in this thesis we aim to give just a brief overview on how triple generation protocols work using OT. Note that in [KOS16] the triple generation when $\mathbb{F} := \mathbb{F}_{2^k}$ is slightly faster than when $\mathbb{F} := \mathbb{F}_p$ due to the binary nature of OT though the performance for characteristic p closely matches the one for characteristic 2. As described in Chapter 4 the difference between the two fields is much larger when the preprocessing is done with Somewhat Homomorphic Encryption.

3.6 Brief overview of Garbled Circuits

The other major paradigm through which MPC can be achieved is using garbled circuits. The benefit of using garbled circuits is that the number of communication rounds required to evaluate any circuit $\mathcal{C}$ is constant as opposed to linear secret sharing schemes such as SPDZ where the communication rounds

Protocol $\Pi_{[\![\cdot]\!]}$

Initialize: Each party P_i samples a MAC key $\Delta^{(i)} \in \mathbb{F}$ and call $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Init}(\mathbb{F})$ with all other parties P_j, $j \neq i$ where each P_j inputs $\Delta^{(j)}$ (see functionality in [KOS16] as the protocol is described in Figure 3.4).

Input: On input $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l, P_j)$ from party P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, P_j)$ from all players P_i where $i \neq j$:

1. P_j samples $x_0 \xleftarrow{\$} \mathbb{F}$.
2. For $h = 0, \ldots, l$, P_j generates random sharings $\sum_i x_h^{(i)} = x_h$ and sends $x_h^{(i)}$ to party P_i.
3. For all $i \neq j$, P_i and P_j execute $\mathcal{F}_{\mathsf{COPEe}}$ where P_j inputs $(x_0, \ldots, x_l) \in \mathbb{F}^{l+1}$.
4. P_i gets $q_h^{(ij)}$ while P_j gets $t_h^{(ij)}$ such that
$$q_h^{(ij)} + t_h^{(ji)} = x_h \cdot \Delta^{(i)}, \text{ for } h = 0, \ldots, l.$$
5. All parties P_i, $i \neq j$ compute the MAC shares $m_h^{(i)} = q_h^{(ij)}$ while P_j computes the corresponding MAC share as $m_h^{(j)} = x_h \cdot \Delta^{(j)} + \sum_{i \neq j} t_h^{(ji)}$ to obtain $[\![x_h]\!]$ for all $h = 0, \ldots, l$.
6. Parties call $\mathbf{r} \xleftarrow{\$} \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^{l+1})$.
7. P_j computes and broadcasts a random linear combination of all shares $y = \sum_{h=0}^{l} r_h \cdot m_h^{(i)}$.
8. All parties P_i compute $m^{(i)} = \sum_{h=0}^{l} r_h \cdot m_h^{(i)}$.
9. Parties run $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ with y and the authentication shares $\{m^{(i)}\}_{i \in [n]}$ (this is instantiated with Π_{MACCheck} from Figure 4.9).
10. Store the shares and MAC shares under handles $\mathsf{id}_1, \ldots, \mathsf{id}_l$.

Figure 3.5: Protocol $\Pi_{[\![\cdot]\!]}$ from MASCOT [KOS16].

is proportional to the circuit depth. Garbled circuits also have the benefit that the current constructions are the most efficients to evaluate Boolean circuits, that is the circuit gates operations as additions and multiplications are represented by the traditional XOR/AND operations over $\mathbb{F}_2$ [HSS17, WRK17a, WRK17b, KY18].

The main focus of this section is to describe garbled circuits for the multiparty case while following the two-party closely to ease the explanation. In this thesis garbled circuits are needed just for for Chapter 8 as it deals with mixed-protocols, the rest of the chapters can be read without having any knowledge GC techniques. Moreover we only need a light introduction to GC since the methods in Chapter 8 use them in a black-box way, the benefit of this being able to plug in any modern GC protocol to make conversions faster.

Garbled circuits were implicitly introduced by Yao in 1986 [Yao82, Yao86]. Interestingly enough, their explicit construction was only given during Yao's talks related to those two papers [Yao19] while a full security proof was published in 2004 by Lindell and Pinkas [LP04]. Yao's garbled circuits adversary model was for two-parties semi-honest case. From a high level point of view, to evaluate a function

Protocol Π_{Triple}

An integer $\tau \geq 3$ which specifies the number of triples generated for a single output triple.

Multiply:

1. Each party P_i samples $\mathbf{a}^{(i)} \xleftarrow{\$} \mathbb{F}^\tau$ and $\mathbf{b}^{(i)} \xleftarrow{\$} \mathbb{F}$.
2. All ordered pair of parties (P_i, P_j) does the following:
   a) Call $\mathcal{F}_{\mathsf{ROT}}^{\tau k,k}$ where P_i inputs $(a_1^{(i)}, \ldots, a_{\tau k}^{(i)}) = \mathbf{g}^{-1}(\mathbf{a}^{(i)})$ (a τk-bit string).
   b) P_j gets $q_{0,h}^{(ji)}, q_{1,h}^{(ji)} \in \mathbb{F}$ and P_i gets $s_h^{(ij)} = q_{a_h^{(i)},h}^{(j)}$, for $h = 1, \ldots, \tau k$.
   c) P_j sends $d_h^{(ji)} = q_{0,h}^{(ji)} - q_{1,h}^{(ji)} + b^{(j)}, h \in [\tau k]$ to P_i.
   d) P_i sets $t_h^{(ij)} = s_h^{(ij)} + a^{(i)} \cdot d_h^{(ji)} = q_{0,h}^{(ji)} + a_h^{(i)} \cdot b^{(j)}$, for $h = 1, \ldots, \tau k$. Set $q_h^{(ji)} = q_{0,h}^{(ji)}$.
   e) Parse $(t_1^{(ij)}, \ldots, t_{\tau k}^{(ij)})$ and $(q_1^{(ji)}, \ldots, q_{\tau k}^{(ji)})$ as a concatenation of τ vectors, each of length k, i.e. $(\mathbf{t}_1, \ldots, \mathbf{t}_\tau)$ and $(\mathbf{q}_1, \ldots, \mathbf{q}_\tau)$.
   f) P_i sets $\mathbf{c}_{i,j}^{(i)} = (\langle \mathbf{g}, \mathbf{t}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{t}_\tau \rangle) \in \mathbb{F}^\tau$.
   g) P_j sets $\mathbf{c}_{i,j}^{(j)} = -(\langle \mathbf{g}, \mathbf{q}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{q}_\tau \rangle) \in \mathbb{F}^\tau$.
   h) Now the following relation holds

$$\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} = \mathbf{a}^{(i)} \cdot b^{(j)} \in \mathbb{F}^\tau.$$

3. Now each party P_i computes the sharing of the cross-product $\mathbf{a} \cdot b$:

$$\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot b^{(i)} + \sum_{j \neq i} (\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{j,i}^{(i)}) \in \mathbb{F}^\tau$$

Combine:

1. Sample $\mathbf{r}, \hat{\mathbf{r}} \xleftarrow{\$} \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^\tau)$
2. Party P_i sets

$$a^{(i)} = \langle \mathbf{a}^{(i)}, \mathbf{r} \rangle,\ c^{(i)} = \langle \mathbf{c}^{(i)}, \mathbf{r} \rangle \quad \text{and} \quad \hat{a}^{(i)} = \langle \mathbf{a}^{(i)}, \hat{\mathbf{r}} \rangle,\ \hat{c}^{(i)} = \langle \mathbf{c}^{(i)}, \hat{\mathbf{r}} \rangle$$

Authenticate: All parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Input}$ (implemented using $\Pi_{[\![\cdot]\!]}$ from Figure 3.5) on their shares to obtain $[\![a]\!], [\![b]\!], [\![\hat{a}]\!], [\![\hat{b}]\!], [\![c]\!]$.

Sacrifice:

1. Call $r \leftarrow \mathcal{F}_{\mathsf{Rand}}$.
2. Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{LinComb}$ for $r \cdot [\![b]\!] - [\![\hat{b}]\!]$ and store them as $[\![\rho]\!]$.
3. Reveal $\rho \leftarrow \mathcal{F}_{\mathsf{Prep}}.\mathsf{Open}([\![\rho]\!])$.
4. Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Open}(\cdot)$ on $\sigma \leftarrow r \cdot c - \hat{c} - \rho \cdot a$. If $\sigma \neq 0$ then abort; else continue.
5. Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ an all opened values. If any check fails then abort, otherwise continue the protocol.

Output: $[\![a]\!], [\![b]\!], [\![c]\!]$ as a valid triple.

Figure 3.6: Protocol Π_{Triple} from MASCOT [KOS16].

input wire u	input wire v	garbled gate
k_u^0	k_v^0	$E_{k_u^0}(E_{k_v^0}(k_w^0))$
k_u^0	k_v^1	$E_{k_u^0}(E_{k_v^1}(k_w^0))$
k_u^1	k_v^0	$E_{k_u^1}(E_{k_v^0}(k_w^0))$
k_u^1	k_v^1	$E_{k_u^1}(E_{k_u^1}(k_w^1))$

Table 3.1: Garbling an AND gate.

$\mathcal{C}(x, y)$ where x and y are joint inputs, one party (also denoted as the garbler) takes the circuit $\mathcal{C}$, garbles it and sends the garbled version $\widehat{\mathcal{C}}$ along with the garbled input $\hat{x}$ to the other party (called the evaluator). The evaluator can get the output of $\mathcal{C}(x, y)$ by executing several OTs (Oblivious Transfers) with the garbler to get the decryption keys of the circuit corresponding to input y without revealing it.

A couple of years later Beaver, Micali and Rogaway [BMR90] extended the Yao GC construction to multiple parties being able to jointly evaluate a circuit. This multiparty protocol for garbled circuits is known in the literature as the BMR protocol. Since in the BMR protocol parties sample jointly some random coins and perform secret shared multiplications they used the GMW protocol as a subroutine [GMW87] which works for an honest majority. To bootstrap BMR against malicious parties they had to use costly zero knowledge proofs for parties to prove that they have computed some PRGs correctly. This was mitigated for the first time in the SPDZ-BMR paper by Lindel et al. [LPSY15] which replaced the PRG calls to PRF calls, enforcing parties to correctly garble through the SPDZ MAC check.

3.6.1 Two-party GC

We now proceed with a brief description of Yao's two-party garbled circuit framework presented more formally by Lindell and Pinkas [LP04]. Consider the two parties Alice and Bob acting as the garbler and the evaluator respectively. Suppose that Alice and Bob want to compute a function $\mathcal{F}(x_A, x_B)$ where x_A is Alice's input whereas x_B is Bob's input. The first step is for Alice to create a garbled version of $\mathcal{F}$ which is represented as a boolean circuit $\mathcal{C}$.

The main task is to show the garbling and evaluation process of an AND gate and XOR. After these two procedures are shown every function $\mathcal{F}$ can be evaluated after split into ANDs and XORs.

We now proceed with the most primitive form of garbling an AND gate: two inputs u, v and one output $u \cdot v$. A garbled version of this gate is to have a set of random keys associated to each possible input wire k_u, k_v and output the key associated to the AND of the previous input wires, see Table 3.1. Garbling an XOR gate is done in an identical manner with the tiny modification that the output key is $k_w^{u \oplus v}$ instead of $k_w^{u \cdot v}$. After Alice garbles the entire circuit, she then shuffles all four entries within each garbled table $\hat{g} \in \hat{\mathcal{G}}$ and sends them to Bob along with the associated input keys of $k_{u_A}^{x_A}$. The next step is for Alice and Bob to execute OTs for each input wire which is dependent on Bob's input. In each OT, Alice places the input wire keys $k_{u_B}^{x_B}$ and Bob inputs his bit b to get the corresponding wire key to Bob's input choice.

There are a couple of improvements which can reduce drastically the cost of garbling an XOR gate as well as the number of calls to E done by the garbler. Moreover in the two-party case there is another optimization called half-gates introduced by Zahur, Rosulek and Evans [ZRE15] which make the row-reduction trick described by [NPS99, PSSW09] compatible with the free-XOR optimization reducing the cost of an AND gate to 2 ciphertexts while also maintaining the cost of a garbled XOR gate to zero ciphertexts.

Point and Permute. Note that once Bob has obtained the garbled table from Alice then he needs to perform four decryptions and see which one results in a valid plaintext. This valid plaintext can be considered a random key to which some "OK" string was appended by Alice in the garbling procedure. If Bob uses an incorrect key to decrypt then with high probability the plaintext will contain no "OK" string. In order to reduce Bob's effort to decrypt each entry of the garbled table during the evaluation phase a technique called point-and-permute was introduced in the BMR paper [BMR90]. To enable point-and-permute Alice samples for every input wire u some random mask λ_u. To garble a gate $g \in \mathcal{G}$ with input wires (u, v) Alice encrypts $E_{k_{\lambda_u \oplus u}}(E_{k_{\lambda_v \oplus v}}(k_{\lambda_w \oplus w} || \lambda_w \oplus w))$ where w is the output of $g(u, v)$. For each garbled truth table Alice sorts the entries lexicographically by the masked output wire $\lambda_w \oplus w$ and then sends all entries to Bob along with the masked inputs $\lambda_a \oplus a$ of Alice and the masks of Bob λ_b. The parties will run an OT for each of Bob's inputs where Alice inputs the keys k_{λ_b} as k_0 and $k_{\lambda_b \oplus 1}$ as k_1 and Bob inputs $b \oplus \lambda_b$. After Bob obtained $k_{\lambda_b \oplus b}$ he can proceed to decrypt the entry corresponding to $(\lambda_a \oplus a) \oplus (\lambda_b \oplus b)$. Once Bob decrypted $k_{\lambda_{a \oplus b} \oplus a \oplus b}$ along with the signal bit $\lambda_{a \oplus b} \oplus a \oplus b$ he can continue with the next output gate using the signal bit and the newly obtained key. The masked bits $\Lambda_a := \lambda_a \oplus a$ or $\Lambda_b := \lambda_b \oplus b$ are sometimes denoted in the literature as signal bits.

Free XOR. Kolesnikov and Schneider [KS08] introduced one of the most popular optimizations for garbled circuits which allows to compute XOR gates at virtually no cost, making Boolean additions just local operations as in the linear secret shared based MPC. The central idea of their construction is to correlate the input wire keys by a global difference Δ, i.e. $k_u^0 \oplus k_u^1 = \Delta$. After the global difference is sampled $\Delta \xleftarrow{\$} \{0,1\}^\kappa$ then the correlation is realised by sampling every zero key at random $k_u^0 \xleftarrow{\$} \{0,1\}^\kappa$ and setting the wire key for input 1 to be $k_u^1 \leftarrow k_u^0 \oplus \Delta$. One can combine the free-XOR optimization with the point-and-permute easily: for every input wire (u, v) to an XOR gate to obtain the signal bit of the output compute $\Lambda_w := \Lambda_u \oplus \Lambda_v$.

3.6.2 BMR Garbling

As opposed to the two-party garbling schemes, in the multiparty setting with a dishonest majority every party must contribute to the garbling procedure as well as in the process of evaluating the garbled truth tables. After introducing the main optimizations used for GC nowadays we will now describe Keller and Yanay [KY18] garbling protocol which is closely modeled after the SPDZ-BMR [LPSY15]. The main difference from the traditional SPDZ-BMR scheme is that Keller and Yanay support free-XOR

u	v	$g_{u,v}$ ciphertexts
0	0	$E_{\mathbf{k}_{u,0},\mathbf{k}_{v,0}}(\mathbf{k}_{w,0} \oplus \mathbf{\Delta} \cdot (g(0,0) \oplus \lambda_w), g_{\mathsf{id}})$
0	1	$E_{\mathbf{k}_{u,0},\mathbf{k}_{v,1}}(\mathbf{k}_{w,0} \oplus \mathbf{\Delta} \cdot (g(0,1) \oplus \lambda_w), g_{\mathsf{id}})$
1	0	$E_{\mathbf{k}_{u,1},\mathbf{k}_{v,0}}(\mathbf{k}_{w,0} \oplus \mathbf{\Delta} \cdot (g(1,0) \oplus \lambda_w), g_{\mathsf{id}})$
1	1	$E_{\mathbf{k}_{u,1},\mathbf{k}_{v,1}}(\mathbf{k}_{w,0} \oplus \mathbf{\Delta} \cdot (g(1,1) \oplus \lambda_w), g_{\mathsf{id}})$

Table 3.2: Free-XOR BMR garbled truth table. Parties will have a garbled table of $4 \cdot n$ entries since the keys $\mathbf{k}$ and global difference $\mathbf{\Delta}$ are vectors.

optimization and has better preprocessing time per AND gate since it uses MASCOT [KOS16] as an MPC subroutine. Nevertheless, the techniques explained in Chapter 8 work for any kind of garbling schemes.

In the free-XOR active BMR described in [KY18], each of the n parties P_i holds two set of keys for each wire w:

$$\mathbf{k}_{w,0} := (k^1_{w,0}, \ldots, k^n_{w,0})$$
$$\mathbf{k}_{w,1} := (k^1_{w,1}, \ldots, k^n_{w,1})$$

and such that they are shifted by an unknown global difference $\mathbf{\Delta} := (\Delta^1, \ldots, \Delta^n)$, i.e. $\mathbf{k}_{w,1} = \mathbf{k}_{w,0} \oplus \mathbf{\Delta}$. To achieve 128-bit computational security each wire key $k^i_{w,j}$ has to sampled randomly from the key space $\{0,1\}^{128}$. Hence each $k^i_{w,0}$ is going to be generated using $\mathcal{F}_{\mathsf{Prep}}$ calls to **RandomEntry** and then opened to party i. The same process is for sampling the global difference $\mathbf{\Delta}$: for each $i \in [n]$ call $[\![\Delta^i]\!] \leftarrow \mathcal{F}_{\mathsf{Prep}}.\textbf{RandomEntry}()$ command and then $\mathcal{F}_{\mathsf{Prep}}.\textbf{Output}([\![\Delta^i]\!], i)$ to party P_i. To make the garbling actively secure one has to instantiate $\mathcal{F}_{\mathsf{Prep}}$ with some active secure protocol such as SPDZ or MASCOT. To have more efficient preprocessing, in practice $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}]$ is replaced with SPDZ/MASCOT over a finite field of characteristic two ($\mathsf{SPDZ}[\mathbb{F}_{2^k}]$) where $k = 128$ to have computational security $\kappa = 128$ in the PRF used for garbling.

To garble a binary gate $g : \{0,1\} \mapsto \{0,1\}$ with an unique identifier g_{id} we need to use the input keys to mask the output wire key (see Table 3.2 using a generic encryption algorithm E).

Given the shared masking bits $[\![\lambda_a]\!], [\![\lambda_b]\!]$ of the input wires, and the keys $\mathbf{k}_u, \mathbf{k}_v$ associated to them, for each possible input value α, β parties will mask the key corresponding to the output wire $\mathbf{k}_w$, to jointly produce for each $j \in [n]$ the following table:

$$[\![g^j_{\alpha,\beta}]\!] := \bigoplus_{i=1}^{n} [\![F_{k^i_{u,\alpha},k^i_{v,\beta}}(g_{\mathsf{id}}||j)]\!] \oplus [\![k^j_w]\!] \oplus [\![\Delta^j]\!] \cdot \Big((\alpha \oplus [\![\lambda_u]\!]) \cdot (\beta \oplus [\![\lambda_w)]\!]\Big)$$

where $F_{k_l,k_r}(m)$ is a PRF which takes as input two keys and a message m. To benefit the AES-NI instructions pipeline, AES with fixed key is used on a permutation of a transformed message $(m, k_l, k_r) \mapsto M$ using $(k_l, k_r) \in \mathbb{F}^2_{2^{128}}$ [GKWY19]. First apply the transformation $M = m \oplus (k_l \cdot X) \oplus (k_r \cdot X^2) \in \mathbb{F}_{2^{128}}$ and set the PRF output as

$$F_{k_l,k_r}(m) := M \oplus \mathsf{AES}_0(\sigma(M)).$$

$\Pi_{\mathsf{Online}}[\mathsf{BMR}]$

Evaluate: On input $(\mathsf{Evaluate}, \hat{\mathsf{GC}}, \mathsf{sid})$ from all parties sort the garbled gates $g \in \hat{\mathsf{GC}}$ in topological order. Moreover, consider that g has two input wires u, v and output wire w. The case where g is an unary gate is simple do deal with by sampling an extra key but we omit it since our protocols do not use unary gates - see [WRK17b] for more details on unary gate garbling. If g is and AND gate and $g^j_{\alpha,\beta} = \mathsf{Open}([\![g^j_{\alpha,\beta}]\!]$ then all parties:

1. Set $k^j_{w,\Lambda_u \cdot \Lambda_v} := g^j_{\Lambda_u,\Lambda_v} \bigoplus_{i=1}^n F_{k^j_{\Lambda_u,\Lambda_v}}(g_{\mathsf{id}}||j)$ for all $j \in [n]$.
2. Party P_i checks that $k^i_{w,\Lambda_u \cdot \Lambda_v} \in \{k^i_{w,0}, k^i_{w,1}\}$ computed in the garbling procedures. If the check fails P_i aborts. Otherwise continue.
3. Set the signal bit $\Lambda_{w,u \cdot v} := c$ for which $k^i_{w,c} = k^i_{w,\Lambda_u \cdot \Lambda_v}$.

If g is an XOR gate then:

1. Set the signal bit $\Lambda_{w,u \oplus v} := \Lambda_u \oplus \Lambda_v$.
2. Set the output key $k^j_{w,\Lambda_u \oplus \Lambda_v} := k^j_{w,\Lambda_u} \oplus k^j_{w,\Lambda_v}$ for all $j \in [n]$.

Figure 3.7: BMR protocol for evaluating a GC.

where AES_0 represents fixed key AES with zero key and $\sigma(M_l||M_r) = (M_r \oplus M_l)||M_l$ which splits the input message $M \in \mathbb{F}_{2^{128}}$ into two halves $M_l||M_r$ which outputs the XOR of the two blocks concatenated with the first half.

What is left to show is how to compute $[\![g^j_{\alpha,\beta}]\!]$. The answer is relatively straightforward: after parties agreed on the gate number g_{id} with input wires u, v and the parties indices, then every party $j \in [n]$, for every $i \in [n]$, $\alpha, \beta \in \{0,1\}$ evaluates locally $F_{k^j_{u,\alpha},k^j_{v,\beta}}(g_{\mathsf{id}}||i)$ and calls $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Input}(F_{k^j_{u,\alpha},k^j_{v,\beta}}(g_{\mathsf{id}}||i))$ so that every other party now has sharings of $[\![F_{k^j_{u,\alpha},k^j_{v,\beta}}(g_{\mathsf{id}}||i)]\!]$. The rest of the garbled table can be obtained by performing secret shared multiplications using $4 \cdot n$ calls to $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Multiply}()$

$$[\![\lambda_u]\!] \cdot [\![\Delta^j]\!] \quad [\![\lambda_v]\!] \cdot [\![\Delta^j]\!] \quad [\![\lambda_u]\!] \cdot ([\![\lambda_v]\!] \cdot [\![\Delta^j]\!]) \quad [\![\lambda_w]\!] \cdot [\![\Delta^j]\!]$$

and a few calls to $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Add}()$ which are local computations. To summarize, computing all $4n$ ciphertexts in Table 3.2 has a total cost of n^2 calls to $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Input}$ and $4 \cdot n$ calls to $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Multiply}$.

To evaluate the circuit parties open the gates by calling $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Open}, [\![g^j_{\alpha,\beta}]\!])$ for each $\alpha, \beta \in \{0,1\}$. Then each party P_i involved in the computation which has an input wire u^i computes the signal bit corresponding to their input wires $\Lambda_{u^i} = \lambda_{u^i} \oplus u^i$ and broadcast Λ_{u^i} along with the input wire key $k^i_{u,\Lambda_{u^i}}$ to all parties. Parties are now ready to un-peel the PRF outputs using the broadcasted wire keys and signal bits in Figure 3.7.

Chapter 4

Preprocessing using SHE

This chapter is based on joint work with Marcel Keller and Valerio Pastro [KPR18] which was presented at EUROCRYPT 2018.

4.1 Contributions

In this chapter we show some overlooked methods for SHE can be better than the state-of-the-art triple generation for dishonest majority using MASCOT by Keller et al. [KOS16]. Concretely we give two improved protocols which:

1. Are up to 6 times faster over a LAN setting and up to 20x faster on a WAN for the two party case due to a reduction in communication.
2. Scale better when increasing the number of parties due to a new ZK proof, doubling the performance for 16 parties.

We also give improved descriptions of the protocols in [KPR18] and more accurate bounds on the SHE ciphertexts sizes using an updated analysis from Baum et al [BCS19]. We also give a UC simulator description instead of the limited UC simulator in the original paper.

4.2 Overview

The core idea of SPDZ is that, instead of encrypting the parties' inputs, it is easier to work with random data, conduct some checks at the end of the protocol, and abort if malicious behavior is detected. In order to evaluate a function with private inputs, the computation is separated in two phases, a preprocessing (or offline) phase and an online phase. The latter uses information-theoretic algorithms to compute the results from the inputs and the correlated randomness produced by the offline phase.

The correlated randomness consists of secret-shared random multiplication triples, that is $(a, b, a \cdot b)$ for random a and b. In SPDZ, the parties encrypt random additive shares of a and b under a global public key, use the homomorphic properties to sum up and multiply the shares, and then run a distributed decryption protocol to learn their share of $a \cdot b$. With respect to malicious parties, there are two require-

ments on the encrypted shares of a and b. First, they need to be independent of other parties' shares, otherwise the sum would not be random, and second, the ciphertexts have to be valid. In the context of lattice-based cryptography, this means that the noise must be limited. Both requirements are achieved by using zero-knowledge proofs of knowledge and bounds of the plaintext and encryption randomness. It turns out that this is the most expensive part of the protocol.

The original SPDZ protocol [DPSZ12] uses a relatively simple Schnorr-like protocol [CD09] to prove knowledge of plaintext and correctness of ciphertexts, but the later implementation [DKL+13] uses more sophisticated cut-and-choose-style protocols for both covert and active security. We have found that the simpler Schnorr-like protocol, which guarantees security against active malicious parties, is actually more efficient than the cut-and-choose proof with covert security.

Intuitively, it suffices that the encryption of the sum of all shares has to be correct because only the sum is used in the protocol. We take advantage of this by replacing the per-party proof with a global proof in Section 4.9. This significantly reduces the computation because every party only has to check one proof instead of $n - 1$. However, the communication complexity stays the same because the independence requirement means that every party still has to commit to every other party in some sense. Otherwise, a rushing adversary could make its input dependent on others, resulting in a predictable triple.

Section 4.8 contains our largest theoretical contribution. We present a replacement for the offline phase of SPDZ based solely on the additive homomorphism of BGV. This allows to reduce the communication and computation compared to SPDZ because the ciphertext modulus can be smaller. At the core of our scheme is the two-party oblivious multiplication protocol by Bendlin et al. [BDOZ11], which is based on the multiplication of ciphertexts and constants. Unlike their work, we assume that the underlying cryptosystem achieves linear targeted malleability introduced by Bitansky et al. [BCI+13], which enables us to avoid the costliest part of their protocol, the proof of correct multiplication. Instead, we replace this check by the SPDZ sacrifice, and argue that BGV with increased entropy in the secret key is a candidate for the above-mentioned assumption.

We do not consider the restriction to BGV to be a loss. Bendlin et al. suggest two flavors for the underlying cryptosystem: lattice-based and Paillier-like. For lattice-based cryptosystems, Costache and Smart [CS16] have shown that BGV is very competitive for large enough plaintext moduli such as needed by our protocol. On the other hand, Paillier only supports simple packing techniques and makes it difficult to manipulate individual slots [NWI+13]. Another advantage of BGV over Paillier is the heavy parallelization with CRT and FFT since in the lattice-based cryptosystem the ciphertext modulus can be a product of several primes (see more in Section 4.6).

To see how the two protocols combine ideas from different papers in a novel way, check Figure 4.2. In a nutshell HighGear borrows ideas from two papers but adds the global Zero Knowledge proof to achieve a better scalability with the number of parties. On the other hand LowGear avoids some ZK proofs for HE plaintext-ciphertext multiplication and replaces Paillier cryptosystem with a BGV implementation from previous work.

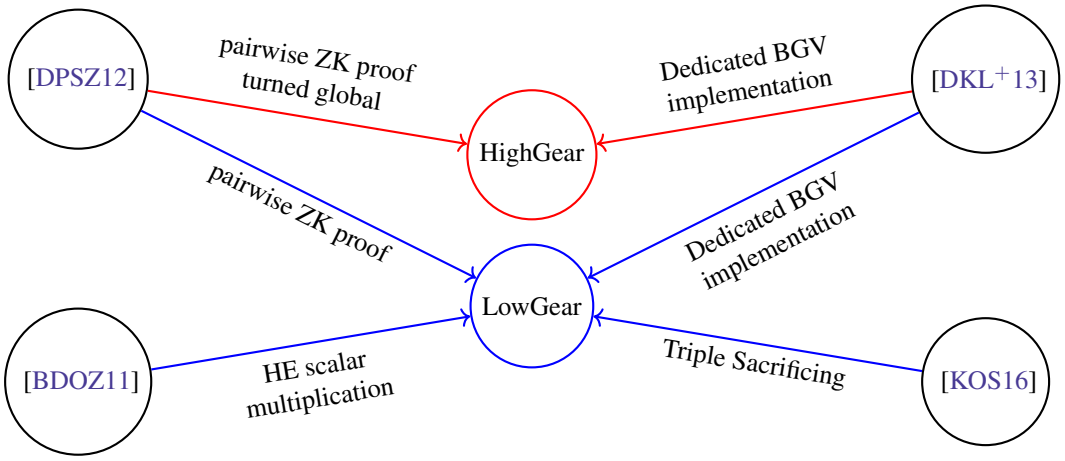

Figure 4.1: Paper dependencies for HighGear and LowGear.

4.3 Algebra

Let $R = \mathbb{Z}[X]/\langle f(x)\rangle$ be the ring of polynomials with integer coefficients modulo an irreducible monic polynomial $f(x)$. Throughout this chapter we will use $f(x) = \Phi_m(X)$ which denotes the m-th cyclotomic polynomial. Since we mostly work with cyclotomic polynomials reduced modulo an integer q recall that $\Phi_m(X) \bmod q = \prod_{i\in(\mathbb{Z}/m\mathbb{Z})^*}(X - \omega_m^i) \bmod q$ where ω_m^i is the m-th rooth of unity of $\mathbb{Z}/q\mathbb{Z}$ i.e. $(\omega_m^i)^m = 1 \bmod q$. In the context of the BGV scheme we are interested in the case when m is a power of two (due to hardness of underlying Ring-LWE problem and plaintext slot manipulation). When this happens then $\Phi_m(X) = X^{m/2} + 1$ and has degree $N = \phi(m)$ which is $|(\mathbb{Z}/m\mathbb{Z})^*|$.

4.3.1 Plaintext space

Consider the case when the plaintext modulus is $R/pR \cong R/\langle(\Phi_m(X), p)\rangle$. If p and m are carefully chosen such that $p^d \equiv 1 \mod m$ then $\Phi_m(X)$ "splits" into $\ell := N/d$ distinct irreducible polynomials, each with degree d, i.e. $\Phi_m(X) \cong F_1(X)\cdots F_\ell(X)$. Since in our MPC protocols the inputs are in a large field $\mathbb{F}_p$, the plaintext space where the triples are produced has to be isomorphic with $\mathbb{F}_p$. This is achieved by setting $d = 1$, find a $p \equiv 1 \mod m$, hence the cyclotomic polynomial splits into ℓ copies of $\mathbb{F}_p$.

As noticed by Gentry et al. [GHS12], one can see a ring element R_p as an array $\mathbf{a} = (a_1, \ldots, a_{\phi(m)})$ with $\phi(m)$ entries where each value is a coefficient modulo p of a degree $\phi(m)$ polynomial. Another way to get a representation of the polynomial is by evaluating it in every root of unity from $(\mathbb{Z}/m\mathbb{Z})^*$, i.e. $\mathbf{b} = (b_1, \ldots, b_{\phi(m)})$ where each $b_i = \mathbf{a}(\omega_m^i)$ for each $i \in (\mathbb{Z}/m\mathbb{Z})^*$. Note that $b_i = \mathbf{a} \mod (X - \omega_m^i) \in \mathbb{Z}/p\mathbb{Z}$. The second representation allows plaintexts or ciphertexts to be multiplied efficiently similar with the FFT multiplication.

4.3.2 Canonical embedding

One of the key insights Lyubashevsky, Peikert and Regev [LPR10] used for introducing the Ring-LWE problem was to switch from the traditional view of coefficient wise embedding to the canonical embedding $\kappa : R \mapsto \mathbb{C}^{\phi(m)}$ in order to bound the error distributions more tightly. One can think of the embedding κ as a map from an element $a \mapsto (\kappa_1(a), \ldots, \kappa_n(a))$ where each $\kappa_i(a)$ evaluates the polynomial a in every m-th order root of unity. We denote $\|a\|^{\mathsf{can}} = \kappa(a)$ and the coefficient embedding as $\|a\|$. These are equipped with the traditional l_p norms where $p = 1, \ldots, \infty$, i.e. $\|a\|_p^{\mathsf{can}}$ and $\|a\|_p$.

Next we highlight some inequalities which are going to be used to determine the ZK and SHE parameters:

- For any $a, b \in R$: $\|a \cdot b\|_\infty^{\mathsf{can}} \leq \|a\|_\infty^{\mathsf{can}} \cdot \|b\|_\infty^{\mathsf{can}}$ (known as the triangle inequality),
- For any $a \in R$: $\|a\|_\infty^{\mathsf{can}} \leq \|a\|_1$,
- For any $a \in R$: $\|a\|_1 \leq \phi(m) \cdot \|a\|_\infty$.

From the last two inequalities it can be deduced that $\|a\|_\infty^{\mathsf{can}} \leq \phi(m) \cdot \|a\|_\infty$.

4.3.3 Probability distributions

We now describe the sampling procedures required to encrypt a message. Note that the procedure is called for each of the $\phi(m)$ polynomial coefficients.

- $\mathcal{U}(R_q)$: draws a random element from $\mathbf{a} \xleftarrow{\$} R_q$. This is achieved by sampling a random integer $\bmod q$ for each component of $\mathbf{a}$.
- $\mathcal{DG}(\sigma^2, R_q)$: generates each coefficient from the Gaussian centered at zero and variance σ^2. In practice this is approximated using binomial distribution as in NewHope [ADPS16]: sample a few elements uniformly from $\{-1, 1\}$ and then sum them.
- $\mathcal{ZO}(0.5)$: samples each coefficient with values from $\{-1, 0, 1\}$ where $p_0 = 1/2$ and $p_{-1} = p_1 = 1/4$.
- $\mathcal{HWT}(h)$: outputs a polynomial with random coefficients from $\{-1, 0, 1\}$ where h of them are non-zero.

In order to bound the expected canonical norm of an element we need to compute the variance of $\kappa_i(a)$ for each sampled $a \in R_q$. If $a \xleftarrow{\$} \mathcal{U}(R_q)$ then its variance $V = \phi(m)q^2/12$ since each coefficient has variance $q^2/12$. If a is sampled from $\mathcal{DG}(\sigma^2, R_q)$ then the variance is $\sigma^2\phi(m)$. Next, if $a \xleftarrow{\$} \mathcal{ZO}(0.5)$ then $V = \phi(m)/2$. Lastly, when $a \xleftarrow{\$} \mathcal{HWT}(h)$ since it has h non-zero coefficients then $V = h$.

Since $\kappa_i(a)$ is the sum of $\phi(m)$ (many) independent and identical distributed variables, then by the law of large numbers it behaves similar with a Gaussian variable with standard deviation $\sqrt{V}$. In order to bound $\kappa_i(a)$, Gentry et al. [GHS12] concluded that the probability of $\kappa_i(a)$ of falling outside the range $[-6\sqrt{V}, 6\sqrt{V}]$ is $\approx 2^{-55}$. To compute this probability they used what is called in the literature as complementary error function erfc [AA92]. The function $\mathsf{erfc}(\mathsf{x})$ measures the chance of a Gaussian variable with zero mean and variance $\sigma = \sqrt{0.5}$ (or standard normal distribution) to fall outside the bounds $[-x, x]$. Notice that $\mathsf{erfc}(6) \approx 2^{-55}$.

We then denote $\mathfrak{c}_1 \cdot \sqrt{V}$ as a high probability bound of the canonical embedding of an element sampled with variance V. Next $\|a_1 \cdot a_2\|_\infty^{\mathsf{can}} \leq \mathfrak{c}_2 \cdot \sqrt{V_1 \cdot V_2}$ is the canonical bound on the product of two elements, first sampled with variance V_1 and the other with variance V_2. We set $\mathfrak{c}_i = e_i^i$ for which $\mathsf{erfc}(e_i)^i \leq 2^{-55}$.

For example, when using the NewHope parameters for the standard deviation, i.e. $\sigma = \sqrt{10}$ then the infinity norm of $\|a\|_\infty$ where $a \xleftarrow{\$} \mathcal{DG}(10, R_q)$ is equal to $6\sqrt{10}$ which is at most 20. On the other, when multiplying two ciphertexts with variance V_1 and V_2 then the expected noise bound on their product is $18 \cdot V_1 \cdot V_2$ as $\mathsf{erfc}(4.2)^2 \approx 2^{-55}$ and $4.2^2 \approx 18$. In the analysis throughout we will call the noise bound of individual randomness components as NewHopeB as in SCALE documentation [ACK+19].

4.4 Ring Learning with Errors

The problem of learning with errors was introduced by Regev in 2005 [Reg05] in which the main task is to decide whether a set of linear equation modulo p related by a secret s to which some error is added is any different than sampling a random number modulo p. More concretely, the main theorem in [Reg05] is the following:

Theorem 10. *(Informal) Let $p \leq \mathsf{poly}(n)$ be a prime integer and n samples of the form $\langle a_i, s\rangle + e_i$ where $s \in \mathbb{Z}_p^n$ and every $a_i \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_p^n)$, $b_i \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_p)$ and $e_i \xleftarrow{\$} \mathcal{D}_\chi$ is sampled independently. The decision-LWE problem states that it is computationally hard to distinguish between*

$$(a_i, \langle a_i, s\rangle + e_i) \approx^c (a, b_i).$$

Regev's main result was that under specific p and error distributions $\mathcal{D}_\chi$ then the $\mathsf{DLWE}[p, \chi]$ can be reduced to an instance of the decision SIVP (Shortest Independent Vector Problem) for which it is conjectured that this there is no quantum algorithm to solve the problem in polynomial time [AKS01, Sch87]. Recently it was shown that using classical reductions (i.e. on a classical computer) LWE is at least as hard as worst-case standard lattice problems [BLP+13].

Due to the large public key size and small encryption rate of LWE based cryptosystems, Lyubashevsky et al. [LPR10] proposed a more efficient variant of LWE called Ring-LWE independently and concurrently with Stehlé et al. [SSTX09]. In this case a Ring-LWE sample is given by sampling $a \xleftarrow{\$} \mathcal{U}(R_q), e \xleftarrow{\$} \mathcal{D}_\chi(R_q)$ and output $a \cdot s + e$.

In our HighGear protocol (Section 4.9), which we view it as a more efficient version of SPDZ, we need some specific assumptions which were used before in the original SPDZ paper namely

1. Hardness assumption of Ring-LWE with a sparse secret.
2. Key-dependent-message (KDM) assumption.

First we state the sparse-secret Ring-LWE assumption in Definition 11. Stehlé et al. [BLP+13] showed in Theorem 4.1 that the problem of solving LWE with a secret of size n and a modulus q is equivalent to solving the sparse-secret LWE problem with a Hamming weight of at least $n \log q$ and some additional constant [MP13]. The state of the art of LWE cryptanalysis with sparse secrets is presented in [CP19,

SC19]. Although the attacks are described for the LWE problem they transfer to the Ring-LWE with the same complexity.

Definition 11. *(Ring-LWE sample with sparse secret key). Let* $s \xleftarrow{\$} \mathcal{HWT}(h)$ *then*

$$(a, a \cdot s + p \cdot e) \approx^c (a, u)$$

where $e \xleftarrow{\$} \mathcal{DG}(\sigma^2, R_q)$ *and* $a, u \xleftarrow{\$} \mathcal{U}(R_q)$.

Second we state the KDM assumption in Definition 12. For a more formal definition one should check Boneh et al. paper [BHHO08] where they used linear dependence on the secret key for DDH based cryptosystem. Note that Brakerski and Vaikuntanathan [BV11] proved the KDM assumption for a more general case when f is a d-degree polynomial by expanding the ciphertext to $d+1$ components and proved that it's secure assuming hardness of Polynomial LWE (PLWE). Recently is was shown by Roşca et al. that decision/search versions of RLWE and PLWE are equivalent [RSW18].

The difference between the KDM definition used in SPDZ-2 [DKL+13] (Definition 12) and the one by Brakerski and Vaikuntanathan [BV11] is that in SPDZ-2 the key switching material contains a quadratic function over additive shares of the secret key i.e. $\mathsf{Enc}(s^2)$ where each party P_i has knowledge of s_i such that $s = \sum_{i=1}^{n} s_i$. If one uses this special key switch material under the assumption given in Definition 12 then a ciphertext $\mathbf{c}$ is a simply pair of $R_q \times R_q$ elements. Note that it is not known how to use the KDM security definition from [BV11] in our case without expanding the ciphertext to 3 components (R_q^3) since they require $d+1$ ciphertext components where d is the degree of the secret key correlation. This is why in SPDZ-1 [DPSZ12] there was little use for the extra KDM assumption since the ciphertexts were elements in R_q^3.

Definition 12. *(KDM security assumption [DKL+13]). Let* $s \xleftarrow{\$} \mathcal{HWT}(h)$ *where* $s = \sum_{i=1}^{n} s_i$ *and* f *is any two-degree polynomial then*

$$(a, a \cdot s + p \cdot e + f(s_1, \ldots, s_n)) \approx^c (a, u)$$

where $e \xleftarrow{\$} \mathcal{DG}(\sigma^2, R_q)$ *and* $a, u \xleftarrow{\$} \mathcal{U}(R_q)$.

In SPDZ-2 each party would sample $s_i \xleftarrow{\$} \mathcal{HWT}(n)$ whereas after modifying the key generation to be actively secure [RST+19] this definition can be updated so that only the final secret has specific Hamming weight and all shares s_i are random and add up to s.

4.5 Somewhat homomorphic encryption scheme

In the following section we describe the underlying cryptosystem used in our actively secure protocols (LowGear and HighGear). The SHE scheme used throughout is called BGV and was introduced by Brakerski et al. [BGV12]. This cryptosystem has been used throughout different versions of covert and active secure n-party computations such as SPDZ-1 [DPSZ12] or SPDZ-2 [DKL+13].

4.5.1 BGV procedures

$\underline{\mathsf{KeyGen}(\lambda)}$: The algorithms depend on some public parameters parameters p, h, q, σ. In our usecase we need to support at most one ciphertext-ciphertext multiplication where the plaintexts live modulo a large prime number p. The ciphertexts are elements modulo a ring R_q where q is a product of two large primes $p_0 \cdot p_1$. For the ease of notation we denote $q_0 := p_0$ and $q_1 := p_0 \cdot p_1$. Due to some internal procedures of BGV which are described later we require that $p_1 \equiv 1 \bmod p$. The key generation algorithm outputs a tuple $(\mathsf{pk}, \mathsf{sk})$. First compute

$$s \leftarrow \mathcal{HWT}(h), a \leftarrow \mathcal{U}(R_q), e \leftarrow \mathcal{DG}(\sigma^2, R_q), \text{and } b \leftarrow a \cdot s + p \cdot e.$$

The public key and secret key pair corresponds to $(\mathsf{pk}, \mathsf{sk}) \leftarrow (b, s)$.

One should note that in practice the Hamming weight of the secret key is chosen to be 64 and $\sigma = \sqrt{10}$. These parameters are related to the hardness of Ring-LWE and can be tuned using Albrecht et al. estimator [APS15].

$\underline{\mathsf{Enc}_{\mathsf{pk}}(m)}$: To encrypt a message $m \in R_p$, generate a small $v \leftarrow \mathcal{ZO}(0.5)$, the errors $e_0, e_1 \leftarrow \mathcal{DG}(\sigma^2, R_q)$ and compute $c_0 \leftarrow m + b \cdot v + p \cdot e_0 \in R_q$ followed by $c_1 \leftarrow a \cdot v + p \cdot e_1 \in R_q$. The final ciphertext is $\mathbf{c} \leftarrow (c_0, c_1) \in R_q^2$.

$\underline{\mathsf{Dec}_{\mathsf{sk}}(\mathbf{c})}$: To decrypt a ciphertext $\mathbf{c} \in R_q^2$ compute the $m \leftarrow (c_0 - s \cdot c_1 \mod q) \mod p$. This works because:

$$\begin{aligned} c_0 - s \cdot c_1 &= m + b \cdot v + p \cdot e_0 - s \cdot (a \cdot v + p \cdot e_1) \\ &= m + (a \cdot s + p \cdot e) \cdot v + p \cdot e_0 - s \cdot a \cdot v - s \cdot p \cdot e_1 \\ &= m + p \cdot (e_0 - s \cdot e_1 + e \cdot v). \end{aligned}$$

Hence, performing the operations modulo q and then reducing it mod p extracts the encrypted plaintext m.

Correctness. Decryption succeeds as long as the noise associated with the ciphertext $\mathbf{c}$ is less than $q/2$ or more formally: $\|m + p \cdot (e_0 - s \cdot e_1 + e \cdot v)\|_\infty < q/2$.

Expected noise. Next we bound (probabilistically) the expected noise of a ciphertext over a random secret key s with the canonical embedding using the decryption formula:

$$\begin{aligned} \|c_0 - s \cdot c_1\|_\infty^{\mathsf{can}} &= \|m + p \cdot (e_0 - s \cdot e_1 + e \cdot v)\|_\infty^{\mathsf{can}} \\ &\leq \|m\|_\infty^{\mathsf{can}} + p \cdot (\|e_0\|_\infty^{\mathsf{can}} + \|s \cdot e_1\|_\infty^{\mathsf{can}} + \|e \cdot v\|_\infty^{\mathsf{can}}) \\ &\leq \phi(m) \cdot p/2 + p \cdot (\mathfrak{c}_1 \cdot \sqrt{\sigma^2 \cdot \phi(m)} + \mathfrak{c}_2 \cdot \sqrt{h \cdot \sigma^2 \cdot \phi(m)} + \mathfrak{c}_2 \cdot \sqrt{\sigma^2 \cdot \phi(m) \cdot \phi(m)/2}) \\ &= \phi(m) \cdot p/2 + p \cdot \sigma \cdot (\mathfrak{c}_1 \cdot \sqrt{\phi(m)} + \mathfrak{c}_2 \cdot \sqrt{h \cdot \phi(m)} + \mathfrak{c}_2 \cdot \phi(m)/\sqrt{2}) = \\ &= B_{\mathsf{clean}}. \end{aligned}$$

4.5.2 Ciphertext multiplication

To multiply two ciphertexts $\mathbf{c} = (c_0, c_1)$ and $\mathbf{c}' = (c_0', c_1')$ at level one do the following:

$\underline{\mathsf{Mult}(\mathbf{c}, \mathbf{c}')}$:

1. $(c_0, c_1) \leftarrow \mathsf{SwitchMod}(\mathbf{c}, p)$
2. $(c'_0, c'_1) \leftarrow \mathsf{SwitchMod}(\mathbf{c}', p)$
3. $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, -c_1 \cdot c'_1)$
4. $\mathbf{c}'' \leftarrow \mathsf{SwitchKey}(d_0, d_1, d_2)$
5. Output $\mathbf{c}''$.

Note that the tuple (d_0, d_1, d_2) represents an encryption of $m \cdot m'$ w.r.t to the secret key s. One can see this easily by expanding the following expression:

$$m \cdot m' = (c_0 - s \cdot c_1)(c'_0 - s \cdot c'_1) = c_0 \cdot c'_1 - s \cdot (c_0 \cdot c'_1 + c_1 \cdot c'_0) - s^2(-c_1 \cdot c'_1)$$

Assuming the inputs to the Mult procedure have noise ν and ν' respectively then the noise of the new ciphertext $\mathbf{c}''$ becomes

$$\nu'' = (\nu/p_1 + B_{\mathsf{Scale}}) \cdot (\nu'/p_1 + B_{\mathsf{Scale}}) + (B_{\mathsf{KS}} \cdot q_0/p_1 + B_{\mathsf{Scale}})$$

since the first and second noise term come from the two calls to the SwitchMod procedure whereas the last is from one call to SwitchKey.

4.5.2.1 Key Switching

Given a ciphertext (d_0, d_1, d_2) and a key-switching matrix $W_t = (b, a)^T$ where $a \xleftarrow{\$} \mathcal{U}_{q_1}$ and $b = a \cdot s + p \cdot e - p_1 \cdot s^2 \bmod q_1$ then the procedure works as follows:

$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$:

1. $c_0 \leftarrow (p_1 \cdot d_0 + b \cdot d_2) \bmod q_1$
2. $c_1 \leftarrow (p_1 \cdot d_1 + a \cdot d_2) \bmod q_1$
3. $c'_0 \leftarrow \mathsf{Scale}(c_0, q_1, q_0)$
4. $c'_1 \leftarrow \mathsf{Scale}(c_1, q_1, q_0)$
5. $\mathbf{c}' = (c'_0, c'_1)$
6. Output $\mathbf{c}'$.

To see why this is correct we write the following expression modulo q_1:

$$\begin{aligned}(c_0 - c_1 \cdot s) &= p_1 \cdot d_0 + b \cdot d_2 - (p_1 \cdot d_1 + a \cdot d_2) \cdot s \\ &= p_1 \cdot (d_0 - d_1 \cdot s) + (a \cdot s + p \cdot e - p_1 s^2) \cdot d_2 - s \cdot d_2 \cdot a \\ &= p_1 \cdot (d_0 - d_1 \cdot s - d_2 \cdot s^2) + p \cdot e \cdot d_2\end{aligned}$$

Assuming the input to SwitchKey has noise ν then the noise bound on c' becomes $\nu' = \nu + B_{\mathsf{KS}} \cdot q_0/p_1 + B_{\mathsf{Scale}}$ where $B_{\mathsf{KS}} \cdot q_0 = \|p \cdot e \cdot d_2\|_\infty$. These bounds are derived using the same arguments as in [GHS12, DKL+13, ACK+19]. Note that the only time the condition $p_1 \equiv 1 \bmod p$ comes into play is in the key switching mechanism in order to cancel out p_1 when moving to the smaller modulus $q_0 = p_0$.

4.5.2.2 Modulus switching

This procedure starts with a ciphertext $c \bmod q_1$ having a noise ν and re-interprets it as a ciphertext $c' \bmod q_0$ encrypting the same message as c. For a complete description of the Scale procedure the reader can check Appendix D. of [GHS12].

SwitchMod(c):

1. $c'_0 = \mathsf{Scale}(c_0, q_1, q_0)$
2. $c'_1 = \mathsf{Scale}(c_1, q_1, q_0)$
3. $\mathbf{c}' = (c'_0, c'_1)$
4. Output $\mathbf{c}'$ where the fresh noise is $\nu' = \nu/p_1 + B_{\mathsf{Scale}}$.

4.6 Why BGV?

One can ask the obvious question whether BGV is a good candidate to bootstrap multiparty computation for SPDZ. And the answer is yes for a number of reasons, for example the efficiency of SIMD (Single Instruction Multiple Data) operations to batch many triples in one go and for the ciphertext size. Compared with BFV/FV and other BGV has a smaller ciphertext [CS16] although in the case of FV we could get away with a smaller ZK proof since we don't need to prove the plaintext bounds, only the randomness bounds.

For the LowGear protocol described in Section 4.8 one could also use Paillier cryptosystem [Pai99, DJ01] or Benaloh [Ben94], Kawachi et al [KTX07] and many others. For a more comprehensive study on the semi-homomorphic encryption schemes the reader can consult [AAUC18]. One interesting open question would be to survey the efficiency of recent improvements of all these semi-homomorphic encryption schemes, provide additional zero knowledge proofs of plaintext and compare their performance. For example, in Monza [CDFG20] they achieve multiparty computation over a ring $\mathbb{Z}_{2^k}$ by considering an unusual encryption scheme, namely the JL cryptosystem [BHJL17] which is based on the quadratic residues problem.

We leave this survey out of the scope of this thesis as we will focus on comparing only Paillier with BGV. One problem with Paillier scheme is that it is hard to support native computation over plaintexts modulo p. Moreover, operating many plaintexts within a single ciphertext to allow batching seems highly non-trivial as we need to multiply slots with different values for the triple generation. The only candidate good-for-all seems to be BGV.

4.7 Proofs of knowledge

The concept of proving a statement about a secret without revealing the secret was introduced in 1985 by Goldwasser, Micali and Rackoff [GMR85]. The authors formalize the idea of interactive proof systems and how to quantify the information communicated between the Interactive Turing Machines (ITMs) - for this case a prover ($\mathcal{P}$) and a verifier ($\mathcal{V}$) acting as ITMs. A language $\mathcal{L}$ is zero-knowledge

Let R a binary relation and an error function $\kappa : \{0,1\}^* \mapsto [0,1]$. Let $\mathcal{V}$ an interactive function which is computable in PPT. We say that $\mathcal{V}$ is a knowledge verifier for the relation R with a knowledge error κ if the following properties hold:

1. Non-triviality: there exists a interactive function $\mathcal{P}^*$ such that for every word in the language $x \in \mathcal{L}_R$, every possible verifier $\mathcal{V}$ then $\mathcal{P}^*$ can produce a valid transcript tr_x associated with input x such that $\Pr[\,\mathsf{accept}(\mathcal{V}, \text{tr}_{\mathcal{P}^*,\mathcal{V}}(x)) = 1\,]$. Or put it into plain words, there exists a $\mathcal{P}^*$ that can produce a transcript for $\mathcal{V}$ to accept for any given $x \in \mathcal{L}_R$.
2. Validity (with error κ). There exists a constant $c > 0$ and an oracle PPT machine $\mathcal{K}$ such that for any interactive function $\mathcal{P}$, for every $x \in \mathcal{L}_R$, the machine $\mathcal{K}$ satisfies the following: when
$$p(x) \stackrel{\text{def}}{=} \Pr[\,\mathsf{accept}(\mathcal{V}, \text{tr}_{\mathcal{P}_x,\mathcal{V}}(x)) = 1\,]) > \kappa(x)$$
then $\mathcal{K}$ will output a string from $R(x)$ with probabilty bounded by $\frac{|x|^c}{p(x)-\kappa(x)}$ when $\mathcal{K}$ has black-box query accesses to $\mathcal{P}_x$. This means that $\mathcal{K}$ can extract the secret x.

We note that here $\mathcal{K}$ is called in the literature as the **knowledge extractor** and its associated **knowledge error** κ.

Figure 4.2: Proof of knowledge definition [BG93].

if for each $x \in \mathcal{L}$ the prover reveals to the verifier that $x \in \mathcal{L}$ and nothing else. The paper breakthrough consisted in proving for the first time an NP language (quadratic residue problem) using a zero-knowledge protocol between $\mathcal{P}$ and $\mathcal{V}$.

For the triple generation we will focus on a subset of ZK proofs, namely Σ protocols. In this case the prover has a secret x, publishes $y = f(x)$ and wants to prove to $\mathcal{V}$ that y was computed correctly without revealing x. In previous SPDZ protocols [DPSZ12, BDOZ11] this was done via a classic 3-round Schnorr [Sch91] protocol, assuming a homomorphic commitment function f:

1. Prover $\mathcal{P}$ samples a random s and commits to it by sending to $\mathcal{V}$ the value $a \leftarrow f(s)$.
2. Verifier samples a random challenge $e \stackrel{\$}{\leftarrow} \mathbb{F}$ and sends it to $\mathcal{P}$.
3. $\mathcal{P}$ computes $z \leftarrow s + e \cdot x$ and sends it to $\mathcal{V}$. The verifier checks whether $f(z) = a + e \cdot y$. If the check passes then $\mathcal{V}$ accepts the proof, otherwise rejects it.

4.7.1 Definition

Proofs of Knowledge (PoKs) were informally introduced and used in 1985 by Goldwasser, Micali [GMR85]. Later they were formalized in 1992 by Bellare and Goldreich [BG93]. We restate their definition in Fig. 4.2. Note that the soundness property of a knowledge system is seen as an additional feature, i.e. for all words $x \notin \mathcal{L}$ then most of the transcripts given to the verifier will fail to accept $(\Pr[\mathsf{accept}(\mathcal{V}, \text{tr}_{\mathcal{P},\mathcal{V}}(x)) = 1)] < 1/2)$. Although this soundness property is seen as a feature to a proof of knowledge system and avoided in the definition from Fig. 4.2 it is an important tool to limit the power of a cheating prover.

What this means in layman's terms is that the only two things required to prove the security of a PoK is that for all $x \in \mathcal{L}$ the prover must convince the verifier (non-triviality) with a valid transcript and computable in PPT. The non-triviality means that non-triviality and validity (or knowledge extractor) [BG93]. In multiparty computation the soundness parameter plays a crucial role when measuring the security of protocol.

4.7.2 Proving the security of a Σ protocol

Due to the cumbersome way of constructing a knowledge extractor for a Σ-protocol the easier way to do this is using special soundness. The existence of a knowledge extractor is implied by the special soundness property: the oracle $\mathcal{K}$ queries the prover $\mathcal{P}_x$ twice using the forking lemma [PS00]. Bellare and Neven [BN06] give a more general definition of the forking lemma (3.1 in the full version of [BN06]) in which they decouple the forking lemma from the hardness reductions. Although they keep the structure similar: $\mathcal{P}_x$ has fixed randomness and the goal of $\mathcal{K}$ is to find two accepting transcripts for the same secret but different random oracle queries.

We will show step-by-step how one proves the security of a Σ protocol using the template from [Ber14]. There are three steps to follow here: i) **correctness**, ii) **security for the prover**, iii) **security for the verifier**. The last two steps need to build two different simulators, one for the prover and one for the verifier.

Correctness. Follows straightforward from the additive homomorphism of f.

Security for the prover. Sometimes called honest-verifier zero knowledge. For this case, the simulator $\mathcal{S}$ has to build a transcript that is indistinguishable between a real interaction of $\mathcal{P}$ and $\mathcal{V}$. To accomplish this, $\mathcal{S}$ acts as a prover with the additional power of rewinding the verifier. After $\mathcal{S}$ gets the challenge e from the verifier it samples a random $z \xleftarrow{\$} \mathbb{F}$. Then $\mathcal{S}$ rewinds $\mathcal{V}$ before the commitment phase and sends $a \leftarrow f(z) - e \cdot y$ and continues from there with the same challenge e. The check of $\mathcal{V}$ passes because the commitment was computed after z instead of before as in the real protocol.

Security for the verifier. Also referred to as special soundness. Here the simulator has to extract the secret s given two pairs of $(a, e, z), (a, e', z')$ accepting transcripts between $\mathcal{P}$ and $\mathcal{V}$ with different challenges ie $e \neq e'$. Since $\mathbb{F}$ is a field then the simulator can compute $(z - z')/(e - e')$, yielding:

$$((s + e \cdot x) - (s + e' \cdot x))/(e - e') = x(e - e')/(e - e') = x$$

Special soundness implies $1/|\mathbb{F}|$ soundness where $|\mathbb{F}|$ is the challenge set size. To see why this is true, consider a verifier that just received $\mathcal{P}$'s commitment and now has to select a challenge. For each challenge $e \in \mathbb{F}$ the prover has a probability $p(e)$ of providing a transcript which is going to be accepted by $\mathcal{V}$. Hence the probability of the accepted transcript between $\mathcal{P}$ and $\mathcal{V}$ is $\sum_{e \in \mathbb{F}} \frac{1}{\mathbb{F}} p(e)$. Now suppose that (a, e, z) gets accepted by the verifier although f was computed incorrectly i.e. $y \neq f(x)$ or $x \notin \mathcal{L}$. Using proof by contradiction this means that at most one term of $p(e)$ is non-zero. The contradiction comes from that if there were more than one non-zero probability terms $p(e)$ then we could extract the

$\Pi_{\mathsf{pairZKPoK}}$

Let $B_{\mathsf{plain}}, B_{\mathsf{rand}}$ defined in Figure 4.4. Let $V = 2 \cdot \mathsf{sec} - 1$ and $M_{\mathbf{e}} \in \{0,1\}^{V \times \mathsf{sec}}$ the be matrix associated with the challenge $\mathbf{e}$ such that $M_{kl} = e_{k-l+1}$ for $1 \leq k - l + 1 \leq \mathsf{sec}$ and 0 in all other entries. The randomness used for encryptions of $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ is packed into matrices $\mathbf{r}^{(i)} \leftarrow (r_1^{(i)}, \ldots, r_{\mathsf{sec}}^{(i)})$ and $\mathbf{s}^{(i)} \leftarrow (s_1^{(i)}, \ldots, s_V^{(i)})$. Hence $\mathbf{r}^{(i)}, \in \mathbb{Z}^{\mathsf{sec} \times 3}$ and $\mathbf{s}^{(i)} \in \mathbb{Z}^{V \times 3}$ (each row has 3 entries according to Enc defined in Section 4.5.1). Recall that here $\mathbf{x}^{(i)}$ is a vector with sec entries: $(\mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_{\mathsf{sec}}^{(i)})$ and $\mathbf{y}^{(i)}$ has V entries: $(\mathbf{y}_1^{(i)}, \ldots, \mathbf{y}_V^{(i)})$.
To improve readability we replace $M_{\mathbf{e}} \cdot \mathbf{x}^{\intercal}$ with $M_{\mathbf{e}} \diamond \mathbf{x}$.

1. Each party P_i broadcasts $E^{(i)} = \mathsf{Enc}_{\mathsf{pk}_i}(\mathbf{x}^{(i)}, \mathbf{r}^{(i)})$ where $\mathbf{x}^{(i)}, \mathbf{r}^{(i)} \leftarrow \mathsf{Sample}(\mathsf{Honest})$.
2. Each party P_i privately samples each entry of $\mathbf{y}^{(i)}$ and $\mathbf{s}^{(i)}$ by calling $\mathbf{y}^{(i)}, \mathbf{s}^{(i)} \leftarrow \mathsf{Sample}(\mathsf{LowGear})$. Then P_i uses the random coins $\mathbf{s}^{(i)}$ to compute $\mathbf{a}^{(i)} \leftarrow \mathsf{Enc}_{\mathsf{pk}_i}(\mathbf{y}^{(i)}, \mathbf{s}^{(i)})$ and broadcasts $\mathbf{a}^{(i)}$.
3. The parties compute $\mathbf{e} \leftarrow h(\mathbf{a}^{(i)}, E^{(i)})$ using sec bits of output from a hash function h.
4. Each party P_i computes $\mathbf{z}^{(i)} = \mathbf{y}^{(i)} + M_{\mathbf{e}} \diamond \mathbf{x}^{(i)}$ and $T^{(i)} = \mathbf{s}^{(i)} + M_{\mathbf{e}} \diamond \mathbf{r}^{(i)}$. If there is any $j \in [V]$ for which $\|\mathbf{z}_j^{(i)}\|_\infty > B_{\mathsf{LG}} \cdot p - \mathsf{sec} \cdot p$ or $\|T_j^{(i)}\|_\infty > B_{\mathsf{LG}} \cdot \rho - \mathsf{sec} \cdot \rho$ then P_i restarts the protocol as a prover.
5. If the checks have passed then P_i broadcast $(\mathbf{z}^{(i)}, T^{(i)})$.
6. Each party P_i now acts as a verifier for the proof of part P_j and computes $\mathbf{e} \leftarrow h(\mathbf{a}^{(j)}, E^{(j)})$ and $\mathbf{d}^{(ij)} = \mathsf{Enc}_{\mathsf{pk}_j}(\mathbf{z}^{(j)}, \mathbf{t}^{(j)})$ for each $j \neq i$ where $\mathbf{t}^{(j)}$ ranges through the rows of $T^{(j)}$.
7. Each party P_i compute the following checks:

$$\mathbf{d}^{(ij)} = \mathbf{a}^{(j)} + M_{\mathbf{e}} \diamond E^{(j)}, \quad \|\mathbf{z}^{(j)}\|_\infty \leq B_{\mathsf{LG}} \cdot p \quad \|T^{(j)}\|_\infty \leq B_{\mathsf{LG}} \cdot \rho$$

8. If all checks pass, parties output $E^{(j)}$ as valid ciphertexts otherwise reject.

Figure 4.3: Protocol for pairwise proof of knowledge of a ciphertext.

secret x according to the special soundness property, implying that $x \in \mathcal{L}$ - but we first assumed that $x \notin \mathcal{L}$. To conclude, since at most one of the $p(e)$ terms is non-zero and $p(e) \leq 1$ then the special soundness implies a soundness of at most $\frac{1}{\mathbb{F}}$.

4.7.3 Proofs of plaintext knowledge

In the context of multiparty computation for dishonest majority, we need to be able to prove the knowledge of plaintext inside an SHE ciphertext. In fact, what is actually proved is that the plaintext and randomness used to produce the ciphertexts are bounded. This approach by bounding data used to produce the ciphertext forces dishonest parties to produce encryptions which have a correct noise w.r.t to the SHE scheme (see Section 4.5.1). Intuitively, the slack is defined as the difference between the honest prover's language and the dishonest prover language. For example, one honest party will encrypt using a correct bound τ, whereas the final ZK proof check can only guarantee that the dishonest prover plaintext was bounded by $B \cdot \tau$. In this case the slack is B.

We sometimes need to produce ciphertexts where the plaintext and randomness used to produce

$\mathsf{Sample}(\mathsf{Type}_{\mathsf{ZK}})$

If $\mathsf{Type}_{\mathsf{ZK}} = \mathsf{Honest}$:

1. Generate $\boldsymbol{m} \xleftarrow{\$} R_p^N$ and the triple $(\boldsymbol{v}, \mathbf{e}_0, \mathbf{e}_1)$ where $\boldsymbol{v} \xleftarrow{\$} \mathcal{ZO}(0.5, N), \boldsymbol{e}_0, \boldsymbol{e}_1 \xleftarrow{\$} \mathcal{DG}(\sigma^2, N)$ and N represents the number of slots. This implies that $\|\boldsymbol{m}\|_\infty \leq p/2$, $\|\boldsymbol{v}\|_\infty \leq 1$ and $\|\mathbf{e}_i\|_\infty \leq 20$ - assuming that these are generated using NewHope algorithm for gaussian sampling [ADPS16]. We let $\rho = (1, 20, 20)$ be the tuple which bounds randomness, sometimes denoted as $\rho = (1, \mathsf{NewHopeB}, \mathsf{NewHopeB})$.
2. Output $(\mathbf{m}, \boldsymbol{v}, \boldsymbol{e}_0, \boldsymbol{e}_1)$.

If $\mathsf{Type}_{\mathsf{ZK}} = \mathsf{LowGear}$, set $B_{\mathsf{LG}} = 128 \cdot N \cdot \mathsf{sec}^2$:

1. Generate $\boldsymbol{m} \xleftarrow{\$} B_{\mathsf{LG}} \cdot p$ and the triple $(\boldsymbol{v}, \boldsymbol{e}_0, \boldsymbol{e}_1)$ where $\boldsymbol{v} \xleftarrow{\$} B_{\mathsf{LG}}$ and $\boldsymbol{e}_i \xleftarrow{\$} B_{\mathsf{LG}} \cdot 20$.
2. Output $(\boldsymbol{m}, \boldsymbol{v}, \boldsymbol{e}_0, \boldsymbol{e}_1)$.

If $\mathsf{Type}_{\mathsf{ZK}} = \mathsf{HighGear}$, set $B_{\mathsf{HG}} = 2^{\mathsf{sec}}$:

1. Generate $\boldsymbol{m} \xleftarrow{\$} B_{\mathsf{HG}} \cdot p$ and the triple $(\boldsymbol{v}, \boldsymbol{e}_0, \boldsymbol{e}_1)$ where $\boldsymbol{v} \xleftarrow{\$} B_{\mathsf{HG}}$ and $\mathbf{e}_i \xleftarrow{\$} B_{\mathsf{HG}} \cdot 20$.
2. Output $(\boldsymbol{m}, \boldsymbol{v}, \boldsymbol{e}_0, \boldsymbol{e}_1)$.

Figure 4.4: Sampling algorithms for plaintexts.

them was sampled according to some special (bounded) distributions. For the two zero knowledge proofs we give these bounds are distinct to each of the proof due to achieving different slack sizes. The sampling algorithms are given in Figure 4.4.

In Figure 4.3 we state the pairwise proof protocol from SPDZ-1 to prove ciphertext correctness. Note that in SPDZ-1 [DPSZ12] there is a single global public key whereas in Figure 4.3 we describe the protocol where each party has its own public key. Moreover the slack is much smaller than the one used for HighGear due to rejection sampling technique by Lyubashevsky [Lyu09]. The way rejection sampling works, as the name hints, is to generate masking coefficients with smaller norm then commit to the chipertexts. If the challenge reveals to have a large norm then abort the proof and start again.

The first step when designing such protocols is to first set the abort probability and afterwards compute the plaintext and randomness bounds. For example, setting an abort probability of $1/32$, in SPDZ-1, Damgård et al. end up with $B_{\mathsf{plain}} = 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$ and $B_{\mathsf{rand}} = 128 \cdot N \cdot \rho \cdot \mathsf{sec}^2$ and a soundness slack $S = (N \cdot \tau \cdot \mathsf{sec}^2 2^{\mathsf{sec}/2+8}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8})$ where the first component is the plaintext slack whereas the second is the randomness slack. We omit the security proofs of $\Pi_{\mathsf{pairZKPoK}}$ as these are identical to the ones done in SPDZ-1 [DPSZ12].

4.8 LowGear - Triples from Semi-Homomorphic Encryption

The main challenge to produce secret shared multiplication triples is to be able to perform pairwise secret multiplications. The two-party multiplication protocol can be then bootstrapped to one that produces sharings (a_i, b_i, c_i) of (a, b, c) for each party P_i such that $(\sum_{i=1}^n a_i) \cdot (\sum_{i=1}^n b_i) = \sum_{i=1}^n c_i$. We can view the product in reverse order: if each party P_i samples randomly a_i, b_i and engages in a two-

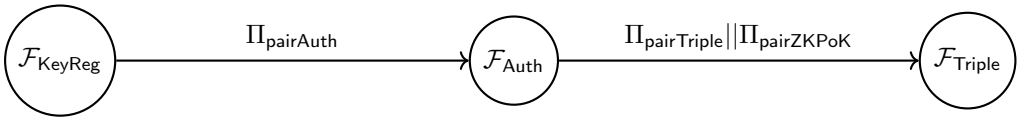

Figure 4.5: Functionality dependencies for LowGear.

party protocol with every other party P_j which computes $a_i \cdot b_j$ then the additive share of the product is $c_i = \sum_{j \neq i} a_i \cdot b_j$. This technique was applied successfully when the two-party multiplication protocol is instantiated with different building blocks: in BDOZa [BDOZ11] with semi-homomorphic encryption and Paillier cryptosystem and in MASCOT [KOS16] using oblivious transfer and OT extension.

The two-party protocol with Paillier works as follows: party P_A sends $\mathsf{Enc}(a)$ encrypted with P_A's own public key to party P_B. Next, party P_B multiplies its input and adds some extra noise encrypted with P_A's public key, i.e. it computes $C = b \cdot \mathsf{Enc}(a) - \mathsf{Enc}(c_B)$ and sends the ciphertext C to P_A. Finally, if party P_A decrypts it will hold $c_A = b \cdot a - c_B$. We can see that this is an additive sharing of the product $a \cdot b = c_A + c_B$.

Our method draws inspiration from both BDOZa and MASCOT: we replace the Paillier cryptosystem with BGV [BGV12] as the underlying encryption scheme. Moreover we use less zero-knowledge proofs by adding an extra assumption of "linear target malleability" of an encryption scheme. This extra notion assumes that given a BGV ciphertext with plaintext modulo p then an adversary is capable of performing only linear homomorphic operations on the plaintext using the ciphertext.

In Figure 4.19 we give an overview of achieving $\mathcal{F}_{\mathsf{Triple}}$ which represents the Triple command in $\mathcal{F}_{\mathsf{Prep}}$ from Figure 3.2 in the preliminaries section.

4.8.1 Key registration

Following the template of BeDOZa [BDOZ11] in the beginning of our protocol we need some setup procedure which generates the keys for the honest and corrupted parties. Since the simulator has to be able to decrypt the corrupted parties ciphertexts, the functionality $\mathcal{F}_{\mathsf{KeyReg}}$ in Figure 4.6 receives an extra input from the adversary which is the randomness source to generate the keys.

4.8.2 Input authentication

One building block in realizing MPC for dishonest majority is to be able to authenticate inputs. The authentication process ensures that if any party tries to cheat then this will be detected later in a MAC check procedure before revealing the outputs. Once the authentication is done one can proceed with creating authenticated random Beaver triples which are required to multiply secret shares.

The authentication step boils down to multiply a secret input with a global unknown MAC key Δ. There are multiple ways of achieving this:

- Somewhat homomorphic property of the BGV encryption scheme in the SPDZ line of work [DPSZ12, DKL+13],

$\mathcal{F}_{\mathsf{KeyReg}}$

The functionality does the key generation setup KeyGen in the following way:

Registration (honest): On input $(\mathsf{Register}, P_i)$ from an honest party P_i the functionality samples $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}()$ and sends $(\mathsf{Registered}, \mathsf{pk}_i, \mathsf{sk}_i)$ to party P_i and $(\mathsf{Registered}, \mathsf{pk}_i, \perp)$ to all other parties.

Registration (corrupted): On input $(\mathsf{Register}, P_i, r^*)$ from a corrupted party P_i it samples $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}()$ where the randomness seed for the key generation is r^*. Functionality then sends $(\mathsf{Registered}, \mathsf{pk}_i, \mathsf{sk}_i)$ to party P_i and $(\mathsf{Registered}, \mathsf{pk}_i, \perp)$ to all other parties.

Figure 4.6: Functionality for key registration.

- Semi-homomorphic property of Paillier cryptosystem in BDOZa [BDOZ11],
- Oblivious transfer in MASCOT [KOS16] or TinyOT [NNOB12].

We now illustrate the protocol for authenticating a secret in Figure 4.7 where most of the steps are taken verbatim from the paper [KPR18]. Note that in the original description of Keller et al. their protocol only supports a limited UC-functionality. We fix this shortcoming by having parties commit to their inputs in Step 2 using their own public key and then execute the protocol as described originally. The protocol Π_{pairAuth} implements the functionality $\mathcal{F}_{\mathsf{Auth}}$ given in Figure 4.10.

Correctness. If parties follow the protocol correctly then what remains to be verified is whether the check in Step 4.8.2 from Figure 4.7 passes:

$$\begin{aligned}\Delta^{(i)} \cdot \rho - \sigma^{(i)} - \sum_{k=1}^{m} t_k \cdot \mathbf{d}_k^{(i)} &= \Delta^{(i)} \cdot (\sum_{k=1}^{m} t_k \cdot x_k) - \sum_{k=1}^{m} t_k \cdot e_k^{(i)} - \sum_{k=1}^{m} t_k \cdot \mathbf{d}_k^{(i)} \\ &= \sum_{k=1}^{m} t_k \cdot (\Delta^{(i)} \cdot x_k - e_k^{(i)} - \mathbf{d}_k^{(i)}) = 0\end{aligned}$$

for all $i \neq j$.

Similarly we need to check whether $\sum_i m_k^{(i)} = x_k \cdot \sum_i \Delta^{(i)}$. Assuming P_j authenticates input x_k we get

$$\begin{aligned}\sum_i m_k^{(i)} = \sum_{i \neq j} m_k^{(i)} + m_k^{(j)} &= \sum_{i \neq j} \mathbf{d}_k^{(i)} + \sum_{i \neq j} e_k^{(i)} + x_k \cdot \Delta^{(j)} \\ &= \sum_{i \neq j} (x_k \Delta^{(i)} - e_k^{(i)}) + \sum_{i \neq j} e_k^{(i)} + x_k \cdot \Delta^{(j)} = x_k \cdot \Delta \quad \square\end{aligned}$$

Security: In Figure 4.11 we include the full UC description of the simulator for Π_{pairAuth} whereas in Overdrive there is only the limited UC version.

Π_{pairAuth}

Initialize: Each party P_i does the following:

1. Sample a MAC key $\Delta^{(i)} \xleftarrow{\$} \mathbb{F}$.
2. Parties register their keys using $\mathcal{F}_{\mathsf{KeyReg}}$ each receiving receiving $(\mathsf{pk}_i, \mathsf{sk}_i)$
3. Using $\Pi_{\mathsf{pairZKPoK}}$ party P_i sends an encryption $\mathsf{Enc}_{\mathsf{pk}_i}(\boldsymbol{\Delta}^{(i)})$ to every other party where $\boldsymbol{\Delta}^{(i)}$ denotes a plaintext with all slots set to $\Delta^{(i)}$.

Input: On input $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l, P_j)$ from P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, P_j)$ from all P_i where $i \neq j$:

1. We assume that $l < m$ where m is the number of ciphertext slots in the encryption scheme. Let $\mathbf{x}$ denote the vector containing x_k in the first l entries and a random number in the m-th one.
2. Party P_j commits to its inputs by broadcasting $\mathsf{Enc}_{\mathsf{pk}_j}(\mathbf{x})$.
3. For each input x_k where $k \in [1 \ldots l]$ P_j samples randomly $x_k^{(i)} \xleftarrow{\$} \mathbb{F}$ and sends them to the designated party i. Then P_j sets its corresponding share $x_k^{(j)}$ accordingly such that $\sum_{l=1}^{n} x_k^{(l)} = x_k$.
4. For every party P_i:
   a) P_j computes $C^{(i)} = \mathbf{x} \cdot \mathsf{Enc}_{\mathsf{pk}_i}(\boldsymbol{\Delta}^{(i)}) - \mathsf{Enc}'_{\mathsf{pk}_i}(\mathbf{e}^{(i)})$ for random $\mathbf{e}^{(i)}$ and sends $C^{(i)}$ to P_i. Enc' denotes encryption with noise $p \cdot 2^{\mathsf{sec}}$ larger than in normal encryption.
   b) P_i decrypts $\mathbf{d}^{(i)} = \mathsf{Dec}_{\mathsf{sk}_i}(C^{(i)})$.
5. P_j sets its MAC share associated to x_k as $m_k^{(j)} \leftarrow \sum_{i \neq j} e_k^{(i)} + x_k \cdot \Delta^{(j)}$ and each party P_i does so for $m_k^{(i)} \leftarrow d_k^{(i)}$.
6. The parties use $\mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^m)$ to generate random t_k for $k = 1, \ldots, m$.
7. P_j broadcasts $\rho = \sum_{k=1}^{m} t_k \cdot x_k$ and all parties set $\sigma^{(i)} = \sum_{k=1}^{m} t_k \cdot m_k^{(i)}$.
8. Parties now call Π_{MACCheck} with ρ and $\sigma^{(i)}$. If check fails abort otherwise continue.
9. All parties store their authenticated shares $x_k^{(i)}, m_k^{(i)}$ as $[\![x]\!]$ under the identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_l$.

Linear Combination: On input $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_l, c_1, \ldots, c_l, c)$ from all parties, every P_i retrieves the share-MAC pairs $x_k^{(i)}, m(x_k)_{k \in [1 \ldots l]}^{(i)}$ and computes:

$$y^{(i)} = \sum_{k=1}^{l} c_k \cdot x_k^{(i)} + c \cdot s_1^{(i)}$$

$$m(y)^{(i)} = \sum_{k=1}^{l} c_k \cdot m(x_k)^{(i)} + c \cdot \Delta^{(i)}$$

where $s_1^{(i)}$ denotes a fixed sharing of 1, for example, $(1, 0, \ldots, 0)$.

Continued in Figure 4.8

Figure 4.7: Protocol for n-party input authentication, part 1.

Π_{pairAuth}

Open: On input $(\mathsf{Open}, \mathsf{id})$ from all parties, each P_i looks up the share $x^{(i)}$ with identifier id and broadcasts it. Then each party reconstructs $x = \sum_{i=1}^{n} x^{(i)}$.

Check: On input $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l)$ from all parties:

1. Sample public vector $r \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^l)$.
2. Compute $y = \sum_{k=1}^{l} r_k x_k^{(i)}$ and $m(y)^{(i)} = \sum_{k=1}^{m} r_k m_k^{(i)}$.
3. Run Π_{MACCheck} with $y, m(y)^{(i)}$.

Figure 4.8: Protocol for n-party input authentication, part 2. (continued from Figure 4.7)

Π_{MACCheck}

Each party P_i uses $y, m(y)^{(i)}, \Delta^{(i)}$ in the following way:

1. Compute $\sigma^{(i)} \leftarrow m(y)^{(i)} - \Delta^{(i)} y$.
2. Call $\mathcal{F}_{\mathsf{Commit}}$ with $(\mathsf{Commit}, \sigma^{(i)})$ to receive handle τ_i.
3. Broadcast $\sigma^{(i)}$ to all parties by calling $\mathcal{F}_{\mathsf{Commit}}$ with (Open, τ_i).
4. If $\sigma^{(1)} + \cdots + \sigma^{(n)} \neq 0$ then abort and output $\perp$; otherwise continue.

Figure 4.9: Protocol for MAC checking

$\mathcal{F}_{\mathsf{Auth}}$

Input: On input $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l, P_j)$ from party P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, P_j)$ from all players P_i where $i \neq j$ wait for adversary's input. If receives OK then store $\mathtt{Reg}_{[}\mathsf{id}_k] \leftarrow x_k$ for all $k \in [1 \ldots l]$ otherwise abort.

Linear Combination: On input $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_l, c_1, \ldots, c_l, c)$ from all parties where $\mathsf{id}_k \in \mathtt{Reg.Keys}()$ store $\mathtt{Reg}[\overline{\mathsf{id}}] = \sum_{k=1}^{l} \mathtt{Reg}[\mathsf{id}_k] \cdot c_k + c$.

Open: On input $(\mathsf{Open}, \mathsf{id})$ from all parties, send $\mathtt{Reg}[\mathsf{id}]$ to the adversary; wait for input x from the adversary and then send x to all parties.

Check: On input $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l)$ from all parties, wait for the adversary's input. If the input is OK and $\mathtt{Reg}[\mathsf{id}_k] = x_k$ for all $k \in [1 \ldots l]$ then send OK to every party, otherwise send $\perp$ and terminate.

Abort: On input Abort from the adversary send $\perp$ to all parties and terminate.

Figure 4.10: Functionality $\mathcal{F}_{\mathsf{Auth}}$

$\mathcal{S}_{[\![\cdot]\!]}$

Let H denote the set of honest parties and A the complement thereof.
Init:

1. Emulating $\mathcal{F}_{\mathsf{KeyReg}}$, generate $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}()$ for all parties, using the corrupted randomness seed r^* received from the adversary when calling KeyGen for $i \in A$. $\mathcal{S}$ then sends the keys $(\mathsf{pk}_j, \mathsf{sk}_j)$ for $j \in [n]$ to all parties (including A).
2. Emulating $\Pi_{\mathsf{pairZKPoK}}$, send $\mathsf{Enc}_{\mathsf{pk}_i}(\Delta^{(i)})$ for random $\Delta^{(i)}$ to the adversary for all $i \in H$.

Input: We assume that $j \in A, i \in H$. Simulator can decrypt the input vector $\mathsf{Enc}_{\mathsf{pk}_j}(\mathbf{x})$ as it knows secret key of corrupted parties. Simulator $\mathcal{S}$ plays on behalf of the honest parties.

1. $\mathcal{S}$ receives $C^{(i)}$ and decrypts $\mathbf{d}^{(i)} = \mathsf{Dec}_{\mathsf{sk}_i}(C^{(i)})$ for all honest P_i.
2. Emulating $\mathcal{F}_{\mathsf{Rand}}$, sample random t_i for $i = 1, \ldots, m$.
3. Receive $(\rho, \sigma^{(i)})$ from the adversary for all $i \in H$.
4. Check whether $\sigma^{(i)} + \sum_{k=1}^{m} t_k \cdot d_k^{(i)} = 0$ for all $i \in H$ and abort if not.
5. Compute $m_k^{(j)}$ for every $j \in A$ by first setting $\mathbf{e}^{(i)} \leftarrow \mathbf{d}^{(i)} - \mathbf{x}^{(j)} \cdot \boldsymbol{\Delta}^{(i)}$ and then proceed as setting $m_k^{(j)} \leftarrow \sum_{i \neq j} e_k^{(i)} + x_k \cdot \Delta_k^{(j)}$.
6. Input $(x_1, \ldots, x_l)$ to $\mathcal{F}_{\mathsf{Auth}}$.

Linear Combination: For every $i \in A$, compute shares and MACs as an honest party would.

Open:

1. Receive the value x from the functionality $\mathcal{F}_{\mathsf{Auth}}$.
2. If x is a linear combination of previously computed shares then the simulator adjusts the honest parties' shares accordingly. Otherwise, sample new random shares $\{x^i\}_{i \in H}$.
3. Emulate the broadcast with the adversary using the simulated honest parties' and receive the corrupted parties' shares.
4. Simulator computes the sum of the shares $x^* = \sum_{i \in H} x^{(i)} + \sum_{i \in A} x^{(i)}$ and then forward x^* back to $\mathcal{F}_{\mathsf{Auth}}$ while also updating the dictionaries of HS and CS.

Check:

1. Emulate $\mathcal{F}_{\mathsf{Rand}}$, send r to corrupted parties.
2. Emulate $\mathcal{F}_{\mathsf{Commit}}$ receive $\sigma^{(i)}$ for all $i \in A$, and adjust honest parties' shares $\sigma^{(i)} = m(y)^{(i)} - \Delta^{(i)} y$ using HS and complete the emulation of $\mathcal{F}_{\mathsf{Commit}}$.
3. If $\sum_{i \in A} \sigma^{(i)}$ does not match the result computed from stored shares, abort $\mathcal{F}_{\mathsf{Auth}}$.

Figure 4.11: Simulator for Π_{Auth}.

$\Pi_{\mathsf{pairTriple}}$

Multiply:

1. Each party P_i samples $\mathbf{a}^{(i)}, \mathbf{b}^{(i)}, \hat{\mathbf{b}}^{(i)} \xleftarrow{\$} \mathbb{F}$ (such that the length of every vector matches the number of slots in the encryption scheme).
2. Every unordered pair (P_i, P_j) executes the following:
   a) P_i uses $\Pi_{\mathsf{pairZKPoK}}$ to send P_j the encryption $\mathsf{Enc}_{\mathsf{pk}_i}(\mathbf{a}^{(i)})$.
   b) P_j computes $C^{(ij)} = \mathbf{b}^{(j)} \cdot \mathsf{Enc}_{\mathsf{pk}_i}(\mathbf{a}^{(i)}) - \mathsf{Enc}'_0(\mathbf{e}^{(ij)})$ for random $\mathbf{e}^{(ij)} \xleftarrow{\$} \mathbb{F}$ and sends it to P_i. Enc'_0 denotes encryption with noise $p \cdot 2^{\mathsf{sec}}$ larger than normal encryption times the slack in the zero-knowledge proof. More concretely, $\mathsf{Enc}'_0(\mathbf{e}_k^{(ij)}) = (\mathbf{e}_k^{(ij)} + p \cdot e_0, p \cdot e_1)$ where for each slot $k \in [N]$ sample $e_0, e_1 \xleftarrow{\$} \mathcal{U}(p \cdot 2^{\mathsf{sec}} \cdot B_{\mathsf{clean-lg}}^{\mathsf{dishonest}})$
   c) P_i decrypts $\mathbf{d}^{(ij)} = \mathsf{Dec}_{\mathsf{sk}_i}(C^{(ij)})$.
   d) Repeat the last two steps with $\hat{\mathbf{b}}^{(i)}$ to get $\hat{\mathbf{e}}^{(ij)}$ and $\hat{\mathbf{d}}^{(ij)}$.
3. Each party P_i computes $\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot \mathbf{b}^{(i)} + \sum_{j \neq i}(\mathbf{e}^{(ij)} + \mathbf{d}^{(ij)})$ and $\hat{\mathbf{c}}^{(i)}$ similarly.

Authenticate: Party P_i calls $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Input}$ with $(\mathbf{a}^{(i)}, \mathbf{b}^{(i)}, \hat{\mathbf{b}}^{(i)}, \mathbf{c}^{(i)}, \hat{\mathbf{c}}^{(i)})$ and then $\mathcal{F}_{\mathsf{Auth}}.\mathsf{LinComb}$ to get vectors of handles of the sum of shares. E.g., we denote by $[\![\mathbf{a}]\!]$ the vector of handles for the respective sums of elements $\{\mathbf{a}^{(i)}\}_{i=1\ldots n}$.

Sacrifice: The parties do the following:

1. Call $r \leftarrow \mathcal{F}_{\mathsf{Rand}}$.
2. Call $\mathcal{F}_{\mathsf{Auth}}.\mathsf{LinComb}$ for $r \cdot [\![\mathbf{b}]\!] - [\![\hat{\mathbf{b}}]\!]$ and store them as $[\![\boldsymbol{\rho}]\!]$.
3. Reveal $\boldsymbol{\rho} \leftarrow \mathcal{F}_{\mathsf{Auth}}.\mathsf{Open}([\![\boldsymbol{\rho}]\!])$.
4. Call $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Open}(\cdot)$ on $\boldsymbol{\tau} \leftarrow r \cdot \mathbf{c} - \hat{\mathbf{c}} - \boldsymbol{\rho} \cdot \mathbf{a}$. If $\boldsymbol{\tau} \neq 0$ then abort; else continue.
5. Call $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Check}$ an all opened values. If any check fails then abort, otherwise continue the protocol.

Output: $([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ as a vector of valid triples.

Figure 4.12: Protocol for random triple generation.

4.8.3 Triple generation protocol

Recall that the goal is to produce random authenticated triples $([\![a]\!], [\![b]\!], [\![ab]\!])$ such that a, b are randomly sampled from $\mathbb{F}$ as described in Figure 4.13. Our protocol in Figure 4.12 is modeled closely after MASCOT [KOS16], replacing oblivious transfer with semi-homomorphic encryption. The construction of a "global" multiplication from a two-party protocol works exactly the same way in both cases. The **Sacrifice** step is exactly the same as in SPDZ and MASCOT and essentially guarantees that corrupted parties have used the same inputs in the **Multiplication** and **Authentication** steps. This is the only freedom the adversary has because all other arithmetic is handled by $\mathcal{F}_{\mathsf{Auth}}$ at this stage.

Theorem 13. $\Pi_{\mathsf{pairTriple}}$ *implements* $\mathcal{F}_{\mathsf{Triple}}$ *in the* $(\mathcal{F}_{\mathsf{Auth}}, \mathcal{F}_{\mathsf{Rand}})$*-hybrid model with a dishonest ma-*

$\mathcal{F}_{\mathsf{Triple}}$

$\mathcal{F}_{\mathsf{Triple}}$ offers the same interface as $\mathcal{F}_{\mathsf{Auth}}$ and the following function:

Triple: On input $(\mathsf{Triple}, \mathsf{id}_a, \mathsf{id}_b, \mathsf{id}_c)$ from all parties sample $a, b \xleftarrow{\$} \mathbb{F}$ and store $(\mathsf{Val}[\mathsf{id}_a], \mathsf{Val}[\mathsf{id}_b], \mathsf{Val}[\mathsf{id}_c]) = (a, b, c)$ where $c = a \cdot b$.

Figure 4.13: Functionality for random triple generation.

jority of parties.

Sketch. For the proof we use $\mathcal{S}_{\mathsf{pairTriple}}$ in Figure 4.14. The simulator is based on two important facts. First, it can decrypt $C^{(ji)}$ for a corrupted party P_j because it generates the keys emulating $\mathcal{F}_{\mathsf{KeyReg}}$. Second, the adversary is committed to all shares of corrupted parties by the input to $\mathcal{F}_{\mathsf{Auth}}$ in the **Authenticate** step. This allows the simulator to determine exactly whether the **Sacrifice** step in Π_{Auth} will fail. Furthermore, the adversary only learns encryptions of honest parties' shares, corrupted parties' shares, $\boldsymbol{\rho}$, and the result of the check. If the check fails, the protocol aborts, $\boldsymbol{\rho}$ is independent of any output information because $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$ are discarded at the end, and finally, an environment deducing information from the encryptions can be used to break the enhanced-CPA security of the underlying cryptosystem. In addition, the environment only learns handles to triples in the **Output** steps, from which no information can be deduced. □

4.8.4 Enhanced CPA Security

We want to reduce the security of our protocol to an enhanced version of the CPA game for the encryption scheme. In other words, if the encryption scheme in use is enhanced-CPA secure, then even a selective failure caused by the adversary does not reveal private information.

We say that an encryption scheme is *enhanced-CPA secure* if, for all PPT adversaries in the game from Figure 4.15, $\Pr[b = b'] - 1/2$ is negligible in κ.

Achieving enhanced-CPA security. The game without zero-checks in step 3 clearly can be reduced to the standard CPA game. Furthermore, we have to make sure that the oracle queries cannot be used to reveal information about m. The cryptosystem is only designed to allow affine linear operations limiting the adversary to succeed only with negligible probability due to the high entropy of m. However, if the cryptosystem would allow to generate an encryption of a bit of m from $\mathsf{Enc}_{\mathsf{pk}}(m)$, the adversary could test this bit for zero with success probability $1/2$. Therefore, we have to assume that non-linear operations on ciphertexts are not possible. To this end, Bitansky et al. [BCI+13] have introduced the notion of linear targeted malleability. A stronger notion thereof, linear-only encryption, has been conjectured by Boneh et al. [BISW17] to apply to the cryptosystem by Peikert et al. [PVW08], which is based on the ring learning with errors problem. The definition by Bitansky et al. is as follows:

$\mathcal{S}_{\mathsf{pairTriple}}$

Let H denote the set of honest parties and A the complement thereof.

Initialize: Emulating $\mathcal{F}_{\mathsf{KeyReg}}$, for every $i \in A$ and $j \in H$, generate all key pairs $(\mathsf{pk}_i, \mathsf{sk}_i)$ and send the relevant parts to the corresponding party.

Multiply:

1. For every $i \in A$ and $j \in H$, emulate two instances of $\Pi_{\mathsf{pairZKPoK}}$:
   a) When P_j is the prover: send $\mathsf{Enc}_{\mathsf{pk}_j}(0)$ to the adversary and receive $C^{(ji)}$.
   b) When P_i is the prover: receive $\mathsf{Enc}_{\mathsf{pk}_i}(\mathbf{a}^{(i)})$. $\mathcal{S}$ decrypts and obtains $\mathbf{a}^{(i)}$. Next the simulator replies to the adversary with $-\mathsf{Enc}'_{\mathsf{pk}_j}(\mathbf{e}^{(ij)})$ for random $\mathbf{e}^{(ji)}$. This way they will obtain a sharing of zero.

Authenticate: Emulating $\mathcal{F}_{\mathsf{Auth}}$, receive $\mathbf{a}^{(i)}, \mathbf{b}^{(i)}, \hat{\mathbf{b}}^{(i)}, \mathbf{c}^{(i)}, \hat{\mathbf{c}}^{(i)}$ for all $i \in A$ from the adversary and return the desired handles.

Sacrifice:

1. Emulating $\mathcal{F}_{\mathsf{Rand}}$, sample $r \xleftarrow{\$} \mathbb{F}_p$ and send it to the adversary.
2. Sample $\boldsymbol{\rho} \xleftarrow{\$} \mathbb{F}_p^m$ and send it to the adversary emulating $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Open}$. Set Fail if the adversary inputs a different value in response.
3. Given the adversary's inputs in **Authenticate** and $\mathsf{Dec}_{\mathsf{sk}_j}(C^{(ji)})$, we can compute $\boldsymbol{\tau}$. Send it to the adversary emulating $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Open}$. If the response is different, or $\boldsymbol{\tau} \neq 0$, set Fail.
4. Emulating $\mathcal{F}_{\mathsf{Auth}}.\mathsf{Check}$, abort if Fail is set.

Figure 4.14: Simulator for $\mathcal{F}_{\mathsf{Triple}}$ (LowGear).

$\mathcal{G}_{\mathsf{cpa}+}$

1. The challenger samples $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\kappa)$, sends pk to the adversary.
2. The challenger sends $c = \mathsf{Enc}_{\mathsf{pk}}(m)$ for a random message m.
3. For $j \in \mathsf{poly}(\kappa)$:
   a) The adversary sends c_j to the challenger.
   b) The challenger checks if $\mathsf{Dec}_{\mathsf{sk}}(c_j) = 0$; if this is the case the challenger sends OK to the adversary; else, the challenger sends FAIL to the adversary and aborts.
4. The challenger samples $b \xleftarrow{\$} \{0, 1\}$ and sends m to the adversary if $b = 0$ and a random m' otherwise.
5. The adversary sends $b' \in \{0, 1\}$ to the challenger and wins the game if $b = b'$.

Figure 4.15: Enhanced CPA game.

Definition 14. *An encryption scheme has the* linear targeted malleability *property if for any polynomial-size adversary A and plaintext generator $\mathcal{M}$ there is a polynomial-size simulator S such that, for any sufficiently large $\lambda \in \mathbb{N}$, and any auxiliary input $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$, the distributions*

$$\left\{ \begin{array}{l|r} & (\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ \mathsf{pk}, & (s, a_1, \ldots, a_m) \leftarrow \mathcal{M}(\mathsf{pk}) \\ a_1, \ldots, a_m, & (c_1, \ldots, c_m) \leftarrow (\mathsf{Enc}_{\mathsf{pk}}(a_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(a_m)) \\ s, & (c'_1, \ldots, c'_k) \leftarrow A(\mathsf{pk}, c_1, \ldots, c_m; z) \\ \mathsf{Dec}_{\mathsf{sk}}(c'_1), \ldots, \mathsf{Dec}_{\mathsf{sk}}(c'_k) & \textit{where} \\ & \mathsf{ImVer}_{\mathsf{sk}}(c'_1) = 1, \ldots, \mathsf{ImVer}_{\mathsf{sk}}(c'_k) = 1 \end{array} \right\}$$

and

$$\left\{ \begin{array}{l|r} \mathsf{pk}, & (\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ a_1, \ldots, a_m, & (s, a_1, \ldots, a_m) \leftarrow \mathcal{M}(\mathsf{pk}) \\ s, & (\Pi, \mathbf{b}) \leftarrow S(\mathsf{pk}; z) \\ a'_1, \ldots, a'_k & (a'_1, \ldots, a'_k)^\top \leftarrow \Pi \cdot (a_1, \ldots, a_m)^\top + \mathbf{b} \end{array} \right\}$$

are computationally indistinguishable where $\Pi \in \mathbb{F}^{k \times m}$, $\mathbf{b} \in \mathbb{F}^k$, and s is some arbitrary string (possibly correlated with the plaintexts).

In the context of BGV, the definition can easily be extended to vectors of field elements. Furthermore, verifying whether a ciphertext is the image of the encryption (ImVer) can be trivially done by checking membership in $R_q \times R_q$, which is possible without the secret key.

It is straightforward to see that linear targeted malleability allows to reduce the enhanced-CPA game to a game without a zero-test oracle. We simply replace the decryption of the adversary's queries by $a'_1, \ldots, a'_k$ computed using S according to the definition, which can be tested for zero without knowing the secret key. The two games are computationally indistinguishable by definition, and the modified one can be reduced to the normal CPA game as argued above.

We now argue that BGV as used by us is a valid candidate for linear targeted malleability. First, the definition excludes computation on ciphertexts other than affine linear maps. Most notably, this excludes multiplication. Since we do not generate the key-switching material used by Damgård et al. [DKL+13], there is no obvious way of computing multiplications or operations of any higher order.

Second, the definition requires the handling of ciphertexts that were generated by the adversary without following the encryption algorithm. For example, $\mathsf{Dec}_{\mathsf{sk}}(0,1) = s \bmod p$. The decryption of such ciphertexts can be simulated by sampling a secret key and computing the decryption accordingly. However, to avoid a security degradation due to independent consideration of standard CPA security and linear targeted malleability, we add sec bits of entropy to the secret key.

4.8.5 A different conjecture

We now describe a different conjecture from Appendix A. of BDOZa paper [BDOZ11] which looks similar to the enhanced CPA security but given for Paillier's scheme. Suppose we have an adversary A

MultSec

1. Challenger generates $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\kappa)$, chooses $y, s \xleftarrow{\$} \mathbb{F}_p$ and samples $\mathbf{r}$. Additionally it samples b and set $z_b = y$ if $b = 0$ or $z_b = s$ otherwise. Next, B computes $Y = \mathsf{Enc}_{\mathsf{pk}}(y, \mathbf{r})$ and sends it to A.
2. Adversary outputs x and a ciphertext C, where x is small enough such that $x \cdot y$ fits in the plaintext space.
3. B checks whether $\mathsf{Dec}_{\mathsf{sk}}(C) = x \cdot y$. If the check passes B sends z_b to A.
4. A outputs a bit b' as the guess whether it thinks that $z_b = y$ or s. A wins if $b = b'$.

Figure 4.16: Multiplication security property.

and a challenger B. They call an encryption scheme "multiplication secure" if for all PPT adversaries A the probability of winning the game in Figure 4.16 is $1/2 + \mathsf{negl}(\kappa)$. Note that traditional Paillier encryption scheme is insecure w.r.t to the definition. The proposed fix that would potentially (and conjectured) make Paillier multiplication secure is by setting $\hat{\mathsf{Enc}}_{\mathsf{pk}}(y) := \mathsf{Enc}_{\mathsf{pk}}(y + vp)$ where v is a random element and p is the prime field characteristic.

4.8.6 Parameter analysis

Recall the soundness slack bound guarantees on a valid ciphertext after a ZK proof was verified successfully:

$$\|m\|_\infty \leq N{\cdot}p/2{\cdot}\mathsf{sec}^2{\cdot}2^{\mathsf{sec}/2+8} \quad \|v\|_\infty \leq N{\cdot}\mathsf{sec}^2{\cdot}2^{\mathsf{sec}/2+8} \quad \|e_0, e_1\|_\infty \leq N{\cdot}\mathsf{NewHopeB}{\cdot}\mathsf{sec}^2{\cdot}2^{\mathsf{sec}/2+8}$$

To be able to bound the noise on a freshly generate ciphertext after the ZK proof, as in Section 4.9.3 we compute the worst case noise as

$$\begin{aligned}
\|c_0 - s \cdot c_1\|_\infty^{\mathsf{can}} &= \|m + p \cdot (e_0 - s \cdot e_1 + e \cdot v)\|_\infty^{\mathsf{can}} \\
&\leq \|m\|_\infty^{\mathsf{can}} + p \cdot (\|e_0\|_\infty^{\mathsf{can}} + \|s \cdot e_1\|_\infty^{\mathsf{can}} + \|e \cdot v\|_\infty^{\mathsf{can}}) \\
&\leq \phi(m)^2 \cdot p \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8} + p \cdot \phi(m) \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8} \cdot \\
&\qquad \cdot (\phi(m) \cdot \mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{h} \cdot \mathsf{NewHopeB} \cdot \phi(m) + \\
&\qquad + \mathfrak{c}_1 \cdot \sqrt{\sigma \cdot \phi(m)} \cdot \phi(m)) \\
&= \phi(m)^2 \cdot p \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8} \cdot (\mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{h} \cdot \mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{\sigma \cdot \phi(m)}) \\
&= B_{\mathsf{clean-lg}}^{\mathsf{dishonest}}.
\end{aligned}$$

Note that in Protocol 4.12, step 2b every party will multiply $b \cdot \mathsf{Enc}(a)$ and add an encryption of $e \xleftarrow{\$} \mathbb{F}$ with $2^{\mathsf{sec}} \cdot p$ larger noise than $B_{\mathsf{clean-lg}}^{\mathsf{dishonest}}$. The p factor comes from multiplying the noise of $\mathsf{Enc}(a)$ by $b \in \mathbb{F}$ whereas 2^{sec} is from statistically hiding the noise of $b \cdot \mathsf{Enc}(a)$. This means that our SHE modulus q_0 has to fulfill:

$$2^{\mathsf{sec}} \cdot p \cdot B_{\mathsf{clean-lg}}^{\mathsf{dishonest}} < q_0/2.$$

Π_{HighGear}

Initialize: Parties call $\mathcal{F}_{\mathsf{KeyGen}}.\mathsf{Init}$ from Figure 4.18 where each obtains privately a secret key sk_i such that $\mathsf{sk} = \sum_{i=1}^{n} \mathsf{sk}_i$ and a global public key pk corresponding to the SHE cryptosystem.

Multiply: Parties multiply and authenticate random additives shares.

1. Each party P_i samples $a_i, b_i \leftarrow \mathsf{Sample}(\mathsf{Honest})$.
2. Each P_i publishes the encryptions $\mathfrak{ct}_{a_i}, \mathfrak{ct}_{b_i}$ using the randomness from their samplers along with the $\Pi_{\mathsf{globalZKPoK}}$ proof that the ciphertexts are well formed.
3. Parties now multiply the sum of public ciphertexts $\mathfrak{ct}_c \leftarrow \sum_{i=1}^{n} \mathfrak{ct}_{a_i} \cdot \sum_{i=1}^{n} \mathfrak{ct}_{b_i}$.
4. They perform a distributed decryption $\Pi_{\mathsf{DDec}}(\mathfrak{ct}_c, \mathsf{NewCiphertext})$ where each party gets an additive share of the plaintext c_i such that $\sum_{i=1}^{n} c_i = c$. Given the additional argument $\mathsf{NewCiphertext}$, Π_{DDec} also outputs a fresh ciphertext $\mathfrak{ct}'_c$ that decrypts to c.
5. Now parties need to authenticate shares so they publicly multiply the ciphertexts $\mathfrak{ct}_a, \mathfrak{ct}_b, \mathfrak{ct}'_c$ with $\mathfrak{ct}_\alpha$ to get $\mathfrak{ct}_{a\cdot\alpha}, \mathfrak{ct}_{b\cdot\alpha}, \mathfrak{ct}'_{c\cdot\alpha}$.
6. In order to get additive sharings of the authentications $\gamma_i(a), \gamma_i(b), \gamma_i(c)$ they call Π_{DDec} on each ciphertext $\mathfrak{ct}_{a\cdot\alpha}, \mathfrak{ct}_{b\cdot\alpha}, \mathfrak{ct}'_{c\cdot\alpha}$ with the additional argument $\mathsf{NoNewCiphertext}$.

Sacrifice: Parties now check if the adversary inserted any errors on the triples. They do the following using a batch of $2 \cdot n_T$ triples:

1. Sample $t \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.
2. Take every consecutive tuple of triples $(\llbracket a_{2i} \rrbracket, \llbracket b_{2i} \rrbracket, \llbracket c_{2i} \rrbracket), (\llbracket a_{2i+1} \rrbracket, \llbracket b_{2i+1} \rrbracket, \llbracket c_{2i+1} \rrbracket)$ for $i \in n_T$ and parse them as $\llbracket a_1 \rrbracket, \llbracket b_1 \rrbracket, \llbracket c_1 \rrbracket$ and $\llbracket a_2 \rrbracket, \llbracket b_2 \rrbracket, \llbracket c_2 \rrbracket$.
3. Open $t \cdot \llbracket a_1 \rrbracket - \llbracket a_2 \rrbracket$ and $\llbracket b_1 \rrbracket - \llbracket b_2 \rrbracket$ to get σ and ρ.
4. Compute $\tau \leftarrow \mathsf{Open}(t \cdot \llbracket c_1 \rrbracket - \llbracket c_2 \rrbracket - \sigma \cdot \llbracket a_2 \rrbracket - \rho \cdot \llbracket b_2 \rrbracket - \sigma \cdot \rho)$.
5. If $\tau \neq 0$ parties abort, otherwise output $\llbracket a_1 \rrbracket, \llbracket b_1 \rrbracket, \llbracket c_1 \rrbracket$ as a valid triple.

Figure 4.17: SPDZ triple generation protocol with global ZKPok (HighGear).

4.9 HighGear: SPDZ With a Global ZKPoK

The HighGear protocol is essentially SPDZ-1 with a better zero knowledge proof. Our protocol reduces the proving cost by a factor of n where n is the number of parties. To produce a triple in SPDZ [DPSZ12] parties encrypt their randomness $\mathfrak{ct}_{a_i}, \mathfrak{ct}_{b_i}$ along with a zero knowledge proof that they are well formed. In the next phase, they sum the ciphertexts $\mathfrak{ct}_a \leftarrow \sum_{i=1}^{n} \mathfrak{ct}_{a_i}$ and $\mathfrak{ct}_b \leftarrow \sum_{i=1}^{n} \mathfrak{ct}_{b_i}$ and publicly multiply $\mathfrak{ct}_c \leftarrow \mathfrak{ct}_a \cdot \mathfrak{ct}_b$. In the final step they perform a distributed decryption the ciphertext $\mathfrak{ct}_c$ to obtain an additive sharing of the share product. One also has to ensure the MAC authentication on these additive shares but for a more detailed description, check the protocol in Figure 4.17. We skip the security proofs as they follow through from the original SPDZ-1 [DPSZ12] protocol once security of the global ZK proof is provided in the next subsection.

$\mathcal{F}_{\mathsf{KeyGen}}$

Init:
1. On input $(\mathsf{Init}, \mathsf{KeyGen})$ from all parties run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}()$. Send pk to the adversary.
2. Receive corrupted shares $\mathsf{sk}_{i \in A}$. Sample honest parties' shares $\mathsf{sk}_{i \in H}$ such that $\sum_{i=1}^{n} \mathsf{sk}_i = \mathsf{sk}$. Note this is always possible as there is at least one honest party.
3. Send secret key shares sk_i to party P_i.

Figure 4.18: Key Generation functionality for HighGear.

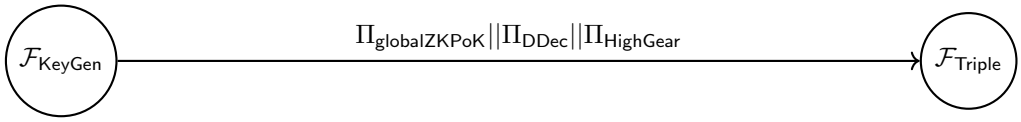

Figure 4.19: Functionality dependencies for HighGear.

4.9.1 Global proof of plaintext knowledge

One of the key insights in the HighGear protocol is that parties can do a single ZK proof verification for all the ciphertexts at once. In the SPDZ protocol [DPSZ12] each party has to verify every other party's proof doing $n-1$ checks. We notice that since every proof is done using the same public key pk then parties can first sum all $n-1$ proofs together and then do a final check. In this way we reduce the computational complexity because summing the proofs is faster then checking each of it individually whereas the communication complexity stays the same - one proof broadcasted per party. The protocol is described in Figure 4.20, and shares many similarities with the pairwise proof while the difference being in the final check. Next, we argue the correctness and security of the global ZK proof from Figure 4.20.

Correctness. The proof goes similar to Damgård et al [DPSZ12]. If the prover is honest then for each $j = 1, \ldots, V$:

$$
\begin{aligned}
\mathbf{d}_j &= \mathbf{a}_j + (M_{e_j} \diamond E) \\
&= \mathsf{Enc}_{\mathsf{pk}}\Big(\sum_{i=1}^{n} \mathbf{y}_j^{(i)}, \sum_{i=1}^{n} \mathbf{s}_j^{(i)}\Big) + \sum_{k=1}^{\mathsf{sec}} (M_{e_{jk}} \cdot \mathsf{Enc}_{\mathsf{pk}}\Big(\sum_{i=1}^{n} \mathbf{x}_k^{(i)}, \sum_{i=1}^{n} \mathbf{r}_k^{(i)}\Big)) \\
&= \mathsf{Enc}_{\mathsf{pk}}\Big(\sum_{i=1}^{n} \mathbf{y}_j^{(i)} + \sum_{k=1}^{\mathsf{sec}} (M_{e_{jk}} \cdot \sum_{i=1}^{n} \mathbf{x}_k^{(i)}), \mathbf{s}_j^{(i)} + \sum_{k=1}^{\mathsf{sec}} M_{e_{jk}} \sum_{i=1}^{n} \mathbf{r}_k^{(i)}\Big) \\
&= \mathsf{Enc}_{\mathsf{pk}}\Big(\sum_{i=1}^{n} (\mathbf{y}_j^{(i)} + \sum_{k=1}^{\mathsf{sec}} M_{e_{jk}} \mathbf{x}_k^{(i)}), \sum_{i=1}^{n} (\mathbf{s}_j^{(i)} + \sum_{k=1}^{\mathsf{sec}} M_{e_{jk}} \mathbf{r}_k^{(i)})\Big) = \mathsf{Enc}_{\mathsf{pk}}\Big(\sum_{i=1}^{n} \mathbf{z}_j^{(i)}, \sum_{i=1}^{n} T_j^{(i)}\Big) \\
&= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_j, T_j).
\end{aligned}
$$

The equality in step 6 follows trivially from the linearity of the encryption. It remains to check the probability that an honest prover will fail the bounds check on $\|\mathbf{z}\|_\infty$ and $\|\mathbf{t}\|_\infty$ where the infinity norm

$\|\cdot\|_\infty$ denotes the maximum of the absolute values of the components.

Remember that the honestly generated $E^{(i)}$ are (τ, ρ) ciphertexts. The bound check will succeed if the infinity norm of $\sum_{i=1}^{n}(\mathbf{y}^{(i)} + \sum_{k=1}^{\mathsf{sec}}(M_{e_{jk}} \cdot \mathbf{x}^{(i)}))$ is at most $2 \cdot n \cdot B_{\mathsf{plain}}$. This is always true because $\mathbf{y}^{(i)}$ is sampled such that $\|\mathbf{y}^{(i)}\|_\infty \leq B_{\mathsf{plain}}$ and $\|M_{\mathbf{e}} \cdot \mathbf{x}^{(i)}\|_\infty \leq \mathsf{sec} \cdot \tau \leq 2^{\mathsf{sec}} \cdot \tau = B_{\mathsf{plain}}$. A similar argument holds regarding ρ and B_{rand}.

Special soundness. To prove this property one must be able to extract the witness given responses from two different challenges. In this case consider the transcripts $(\mathbf{x}, \mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and $(\mathbf{x}, \mathbf{a}, \mathbf{e}', (\mathbf{z}', T'))$ where $\mathbf{e} \neq \mathbf{e}'$. Recall that each party has a different secret $\mathbf{x}^{(i)}$. Because both challenges have passed the bound checks during the protocol, we get that:

$$(M_{\mathbf{e}} - M_{\mathbf{e}'}) \cdot E^{\intercal} = (\mathbf{d} - \mathbf{d}')^{\intercal}$$

To solve the equation for E notice that $M_{\mathbf{e}} - M_{\mathbf{e}'}$ is a matrix with entries in $\{-1, 0, 1\}$ so we must solve a linear system where $E = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)$ for $k = 1, \ldots, \mathsf{sec}$. This can be done in two steps: solve the linear system for the first half: $\mathbf{c}_1, \ldots, \mathbf{c}_{\mathsf{sec}/2}$ and then for the second half: $\mathbf{c}_{\mathsf{sec}/2+1}, \ldots, \mathbf{c}_{\mathsf{sec}}$. For the first step identify a square submatrix of $\mathsf{sec} \times \mathsf{sec}$ entries in $M_{\mathbf{e}} - M_{\mathbf{e}'}$ which has a diagonal full of 1's or -1's and it is lower triangular. This can be done since there is at least one component j such that $e_j \neq e'_j$. Recall that the plaintexts $\mathbf{z}_k, \mathbf{z}'_k$ have norms less than B_{plain} and the randomness used for encrypting them, $\mathbf{t}_k, \mathbf{t}'_k$, have norms less than B_{rand} where k ranges through $1, \ldots, \mathsf{sec}$.

Solving the linear system from the top row to the middle row via substitution we obtain in the worst case: $\|\mathbf{x}_k\|_\infty \leq 2^k \cdot n \cdot B_{\mathsf{plain}}$ and $\|\mathbf{y}_k\|_\infty \leq 2^k \cdot n \cdot B_{\mathsf{rand}}$ where k ranges through $1, \ldots, \mathsf{sec}/2$. The second step is similar to the first with the exception that now we have to look for an upper triangular matrix of $\mathsf{sec} \times \mathsf{sec}$. Then solve the linear system from the last row to the middle row. In this way we extract $\mathbf{x}_k, \mathbf{r}_k$ which form $(2^{\mathsf{sec}/2+1} \cdot n \cdot B_{\mathsf{plain}}, 2^{\mathsf{sec}/2+1} \cdot n \cdot B_{\mathsf{rand}})$ or $(2^{3\mathsf{sec}/2+1} \cdot n \cdot \tau, 2^{3\mathsf{sec}/2+1} \cdot n \cdot \rho)$ ciphertexts. This means that the slack is $2^{3\mathsf{sec}/2+1}$.

Honest verifier zero-knowledge. Here we give a simulator $\mathcal{S}$ in Figure 4.21 for an honest verifier (each party P_i acts as one at one point during the protocol). The simulator's purpose is to create a transcript with the verifier which is indistinguishable from the real interaction between the prover and the verifier. To achieve this, $\mathcal{S}$ samples uniformly $\mathbf{e} \xleftarrow{\$} \{0,1\}^{\mathsf{sec}}$ and then creates the transcript accordingly: sample $\mathbf{z}^{(i)}$ such that $\|\mathbf{z}^{(i)}\|_\infty \leq B_{\mathsf{plain}}$ and $T^{(i)}$ such that $\|T^{(i)}\|_\infty \leq B_{\mathsf{rand}}$ and then fix $\mathbf{a}^{(i)} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}^{(i)}, T^{(i)}) - (M_{\mathbf{e}} \cdot E^{(i)})$, where the encryption is applied component-wise. Clearly the produced transcript $(\mathbf{a}^{(i)}, \mathbf{e}^{(i)}, \mathbf{z}^{(i)}, T^{(i)})$ passes the final checks and the statistical distance to the real one is $2^{-\mathsf{sec}}$, which is negligible with respect to sec.

Recently a successor of HighGear, TopGear [BCS19], obtained a smaller soundness slack which reduces the parameters even further from $2^{3\mathsf{sec}/2+1}$ to $2^{\mathsf{sec}+1}$. Their contribution is taken into account inside SCALE engine and well documented [ACK+19].

$\Pi_{\mathsf{globalZKPoK}}$

Follows the same notations as in $\Pi_{\mathsf{pairZKPoK}}$ but with different bounds: B_{HG} instead of B_{LG} from Figure 4.4. As in the pairwise proof, we prove the bounds for sec ciphertext at once, $\mathbf{x}$ is actually a vector of sec ciphertexts each containing N items due to BGV batching.

1. Each party P_i broadcasts $E^{(i)} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}^{(i)}, \mathbf{r}^{(i)})$ where $\mathbf{x}^{(i)}, \mathbf{r}^{(i)} \leftarrow \mathsf{Sample}(\mathsf{Honest})$. These ciphertexts come from $\mathfrak{ct}_{a_i}$ or $\mathfrak{ct}_{b_i}$ or from the output of Π_{DDec} protocol with NewCiphertext in the SPDZ protocol from Figure 4.17.
2. Each party P_i samples each entry of $\mathbf{y}^{(i)}$ and $\mathbf{s}^{(i)}$ using $\mathsf{Sample}(\mathsf{HighGear})$ w.r.t to the bounds $B_{\mathsf{plain}}, B_{\mathsf{rand}}$. Then P_i uses the random coins $\mathbf{s}^{(i)}$ to compute $\mathbf{a}^{(i)} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}^{(i)}, \mathbf{s}^{(i)})$ and broadcasts $\mathbf{a}^{(i)}$. Note that $\mathbf{y}^{(i)}$ is a plaintext vector of length $V = 2 \cdot \mathsf{sec} - 1$ whereas $\mathbf{s}^{(i)}$ has the same length containing the randomness associated to produce the encryption $\mathbf{a}^{(i)}$ of $\mathbf{y}^{(i)}$.
3. The parties use $\mathcal{F}_{\mathsf{Rand}}$ to sample $\mathbf{e} \in \{0,1\}^{\mathrm{sec}}$.
4. Each party P_i computes $\mathbf{z}^{(i)} = \mathbf{y} + M_{\mathbf{e}} \diamond \mathbf{x}^{(i)}$ and $T^{(i)} = \mathbf{s}^{(i)} + M_{\mathbf{e}} \diamond \mathbf{r}^{(i)}$ and broadcasts $(\mathbf{z}^{(i)}, T^{(i)})$.
5. Each party P_i computes $\mathbf{d}^{(i)} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}^{(i)}, \mathbf{t})$ where $\mathbf{t}$ ranges through all rows of $T^{(i)}$, then stores the sum $\mathbf{d} = \sum_{i=1}^{n} \mathbf{d}^{(i)}$.
6. The parties compute $E = \sum_i E^{(i)}$, $\mathbf{a} = \sum_i \mathbf{a}^{(i)}$, $\mathbf{z} = \sum_i \mathbf{z}^{(i)}$ and $T = \sum_i T^{(i)}$ and conduct the checks (allowing the norms to be $2n$ times bigger to accommodate the summations):

$$\mathbf{d} = \mathbf{a} + M_{\mathbf{e}} \diamond E, \quad \|\mathbf{z}\|_\infty \leq 2 \cdot n \cdot B_{\mathsf{HG}} \cdot p, \quad \|T\|_\infty \leq 2 \cdot n \cdot B_{\mathsf{HG}} \cdot \rho.$$

7. If the check passes, the parties output the global sum E as a ciphertext with valid encryption bounds.

Figure 4.20: Protocol for global proof of knowledge of a ciphertext.

$\mathcal{S}^S_{\mathsf{globalZKPoK}}$

Let A denote the set of corrupted parties, and H the set of honest ones.

1. Receive $E^{(i)}$ for all $i \in H$.
2. Sample $\mathbf{e} \xleftarrow{\$} \{0,1\}^{\mathsf{sec}}$.
3. Use the honest-verifier zero-knowledge simulator above to generate transcripts $(\mathbf{a}^{(i)}, \mathbf{e}, (\mathbf{z}^{(i)}, T^{(i)}))$ for $i \in H$.
4. Send $\{\mathbf{a}^{(i)}\}_{i \in H}$ to the adversary.
5. Receive $(E^{(i)}, \mathbf{y}^{(i)}, \mathbf{a}^{(i)})$ for every corrupted party P_i from the adversary.
6. Emulating $\mathcal{F}_{\mathsf{Rand}}$, send $\mathbf{e}$ to the adversary.
7. Receive $(\mathbf{z}^{(i)}, T^{(i)})$ for every corrupted party P_i from the adversary.
8. Check whether $\sum_{i \in A} \mathbf{z}^{(i)}$ and $\sum_{i \in A} T^{(i)}$ meets the bounds. Abort if not.
9. Rewinding the adversary, sample $\tilde{\mathbf{e}} \neq \mathbf{e}$ and conduct the same check for the adversary's responses $\{\tilde{\mathbf{z}}^{(i)}, \tilde{T}^{(i)}\}_{i \in A}$ until the check passes.
10. Use the Σ-protocol extractor on $\{(E^{(i)}, \mathbf{y}^{(i)}, \mathbf{a}^{(i)}, \mathbf{e}, \mathbf{z}^{(i)}, T^{(i)}, \tilde{\mathbf{e}}, \tilde{\mathbf{z}}^{(i)}, \tilde{T}^{(i)})\}_{i \in A}$ to compute $\{\mathbf{x}^{(i)}\}_{i \in A}$ and input $\sum_{i \in A} \mathbf{x}^{(i)}$ to $\mathcal{F}^S_{\mathsf{gZKPoK}}$.

Figure 4.21: Simulator for global proof of knowledge of ciphertext.

Π_{DDec}

If the second argument is NewCiphertext then parties also get a "fresh" (i.e. level 1) ciphertext $\mathfrak{ct}'_m$ which decrypts to the same plaintext as $\mathfrak{ct}_m$.
Input is a ciphertext $\mathsf{Enc}_{\mathsf{pk}}(m)$ or $\mathfrak{ct}_m$ with some additional arguments such as NewCiphertext or NoNewCiphertext. Players need to parse the ciphertext as a tuple $(c_0, c_1) \in R_q^2$. All sub-routines output an additive sharing m_i such that $\sum_{i=1}^n m_i = m$.
MAC DDec: On input $(\mathsf{Enc}_{\mathsf{pk}}(m), \mathsf{NoNewCiphertext})$ from all parties:

1. Each P_i samples $f_i \xleftarrow{\$} [0 \dots B_{\mathsf{Dec}} \cdot 2^{\mathsf{DD}_{\mathsf{sec}}}]$.
2. P_1 sets $v_1 = (c_0 - s_1 \cdot c_1) - f_1$, whereas every other P_i sets $v_i = -s_i \cdot c_1 - f_i$.
3. All parties broadcast $v_i \bmod q_0$.
4. P_1 set its share $m_1 \leftarrow \sum_{i=1}^n v_i + f_1 \bmod p$. All other parties set their share $m_i \leftarrow f_i \bmod p$.

Fresh DDec: On input $(\mathsf{Enc}_{\mathsf{pk}}(m), \mathsf{NewCiphertext})$ from all parties

1. Each P_i samples $f_i \xleftarrow{\$} \mathbb{F}$.
2. Parties broadcast encryptions of their partial shares $\mathfrak{ct}_{\mathbf{f}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(f_i)$ along with their proofs of correctness using $\Pi_{\mathsf{globalZKPoK}}(\mathfrak{ct}_{\mathbf{f}_i})$.
3. Using the output of $\Pi_{\mathsf{globalZKPoK}}$ they can now compute $\mathfrak{ct}_{\mathbf{m}+\mathbf{f}} \leftarrow \mathfrak{ct}_{\mathbf{m}} + \mathfrak{ct}_{\mathbf{f}}$ using publicly available information. Parse $\mathfrak{ct}_{\mathbf{m}+\mathbf{f}}$ as (c_0, c_1).
4. P_1 sets $v_1 = (c_0 - s_1 \cdot c_1) \bmod q_0$ whereas all other parties P_i set $v_i = -s_i \cdot c_1 \bmod q_0$.
5. All parties broadcast their $v_i + p \cdot r_i$ where $r_i \xleftarrow{\$} [0 \dots B_{\mathsf{Dec}} \cdot 2^{\mathsf{DD}_{\mathsf{sec}}}/p]$.
6. Parties sum up the broadcasts and set $\mathbf{m} + \mathbf{f} \leftarrow (\sum_{i=1}^n (v_i + p \cdot r_i) \bmod q_0) \bmod p$.
7. Adjust the shares the following: P_1 does $\mathbf{m}^1 \leftarrow \mathbf{m} + \mathbf{f} - f_1$ whereas all the other parties $i \neq 1$ set their share as $\mathbf{m}^{(i)} \leftarrow -f_i$.
8. Finally, they adjust the encryption of $\mathbf{m}$ as $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}) \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f}) - \mathfrak{ct}_{\mathbf{f}}$ where the randomness used in producing $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f})$ was sampled using $\mathcal{F}_{\mathsf{Rand}}$.

Figure 4.22: Distributed decryption for SPDZ.

4.9.2 Distributed decryption

In Figure 4.22 we describe two methods of producing additive sharings of a given ciphertext. The first method is used for generating additive MAC shares whereas the second one (**Fresh DDec** command in Figure 4.22) is used to produce a fresh ciphertext after multiplying $\mathfrak{ct}_a \cdot \mathfrak{ct}_b$ to be able to multiply further with $\mathfrak{ct}_\alpha$ and place a MAC on top of the product.

One may wonder what is the reason behind sampling r_i from an interval which is p times smaller than $B_{\mathsf{dec}} \cdot 2^{\mathsf{DD}_{\mathsf{sec}}}$. In the case of **Fresh DDec** command in Figure 4.22 the decryption output is public so there is no need to mask it! We only need to mask the secret noise which is stretched up by a factor of p in the decryption.

4.9.3 Parameter analysis

We now follow similar steps as SCALE documentation [ACK+19]. Following the introduction of BGV noise analysis in Section 4.5.1 this becomes an easy task. We recap what are the bound guarantess on a valid ciphertext after the ZK proofs:

$$\|m\|_\infty \leq n \cdot p \cdot 2^{3\mathsf{sec}/2+1}, \|v\|_\infty \leq n \cdot 2^{3\mathsf{sec}/2+1}, \|e_0, e_1\|_\infty \leq n \cdot \mathsf{NewHopeB} \cdot 2^{3\mathsf{sec}/2+1}.$$

First we need to bound the noise of a ciphertext generated dishonestly which passed the ZK proof checks:

$$\begin{aligned}
\|c_0 - s \cdot c_1\|_\infty^{\mathsf{can}} &= \|m + p \cdot (e_0 - s \cdot e_1 + e \cdot v)\|_\infty^{\mathsf{can}} \\
&\leq \|m\|_\infty^{\mathsf{can}} + p \cdot (\|e_0\|_\infty^{\mathsf{can}} + \|s \cdot e_1\|_\infty^{\mathsf{can}} + \|e \cdot v\|_\infty^{\mathsf{can}}) \\
&\leq \phi(m) \cdot p \cdot 2^{3\mathsf{sec}/2+1} \cdot n + p \cdot 2^{3\mathsf{sec}/2+1}. \\
&\qquad \cdot (\phi(m) \cdot n \cdot \mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{h} \cdot n \cdot \mathsf{NewHopeB} \cdot \phi(m) + \\
&\qquad + \mathfrak{c}_1 \cdot \sqrt{\sigma \cdot \phi(m)} \cdot n \cdot \phi(m)) \\
&= \phi(m) \cdot p \cdot n \cdot 2^{3\mathsf{sec}/2+1} (\mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{h} \cdot \mathsf{NewHopeB} + \mathfrak{c}_1 \cdot \sqrt{\sigma \cdot \phi(m)}) \\
&= B_{\mathsf{clean-hg}}^{\mathsf{dishonest}}.
\end{aligned}$$

where we used the following facts:

1. $\|e\|_\infty = \mathfrak{c}_1 \sqrt{\sigma \cdot \phi(m)}$ as this is honestly generated public key,
2. $\|s\|_\infty = \mathfrak{c}_1 \sqrt{h}$ (noted as $\mathfrak{c}_1 \cdot V_s$ in SCALE), honest generated secret key,
3. $\sum_{i=1}^n \|e_0^i\|_\infty$ and $\sum_{i=1}^n \|e_1\|_\infty \leq n \cdot \phi(m) \cdot 2^{3\mathsf{sec}/2+1} \cdot \mathsf{NewHopeB}$,
4. $\sum_{i=1}^n \|v\|_\infty \leq n \cdot 2^{3\mathsf{sec}/2+1}$.

4.9.4 The impact of modulus switching on the slack

Now that we obtained an exact form of $B_{\mathsf{clean-hg}}^{\mathsf{dishonest}}$ it is time to see what happens to the noise of a ciphertext in the triple generation protocol. By inspection of the protocol Π_{HighGear} there is one ciphertext-ciphertext multiplication and then a re-sharing for computing the MAC shares. To compute a sharing of the product $c = a \cdot b$ there is one ciphertext-ciphertext multiplication to which one fresh ciphertext is added (see **Fresh DDec** command) in Step 3). The noise bound after n additions of the ciphertext and one multiplication is

$$U_1 = (B_{\mathsf{clean-hg}}^{\mathsf{dishonest}}/p_1 + B_{\mathsf{Scale}}) \cdot (B_{\mathsf{clean-hg}}^{\mathsf{dishonest}}/p_1 + B_{\mathsf{Scale}}) + B_{\mathsf{KS}} \cdot p_0/p_1 + B_{\mathsf{Scale}}.$$

After this we add a fresh proven ciphertext via $\Pi_{\mathsf{globalZKPoK}}$ to get the final noise

$$U_2 = U_1 + (B_{\mathsf{clean-hg}}^{\mathsf{dishonest}}/p_1 + B_{\mathsf{Scale}}).$$

The reason why the original noise is scaled down is that before performing the addition between the two ciphertexts we need to scale down from q_1 to q_0 the newly generated ciphertext in **Fresh DDec**.

Now the only condition that needs to be fulfilled comes from the fact that we are masking the $c_0 - s \cdot c_1$ with a value larger by a factor of 2^{sec} to statistically hide the noise:

$$U_2 + U_2 \cdot n \cdot 2^{\mathsf{sec}} < q_0/2 \tag{4.1}$$

which means that the modulus q_0 has to be at least of size $2 \cdot U_2 \cdot (1 + n \cdot 2^{\mathsf{sec}})$.

In practice, if one needs to produce Beaver triples over a prime field $\mathbb{F}_p$ where $p \approx 2^{128}$ with a batching parameter $\phi(m) = 32768$ the prime p_1 turns out to be around 224 bits long using the Albrecht et al. estimator, which means that $B^{\mathsf{dishonest}}_{\mathsf{clean-hg}}/p_1$ disappears completely after the modulus switching operation. It turns out that even for larger values of sec the slack will have no impact over the size of q_0 because B_{Scale} is much bigger than the bound on the dishonest parties' ciphertext noise. Even though TopGear has a zero knowledge slack smaller by a factor of $\sqrt{2^{\mathsf{sec}}}$ than HighGear it gives no improvements over the ciphertext size. Where TopGear shines is in memory usage as it gains more soundness using the SHE slots. Nevertheless we give the ciphertext sizes for both LowGear and HighGear computed at the time of writing Overdrive in Table 4.1 The primes p_0 and p_1 are selected heuristically in the following way: brute force through each possible $q' = p_0 \cdot p_1$ less than the one given by the Albrecht et al. estimator, then find minimal $p_0 = q_0$ such that Equation 4.1 is fulfilled. In the last step p_1 is chosen as the minimum number for which $p_1 \equiv 0 \bmod (2 \cdot N)$ and $p_1 \equiv 1 \bmod p$.

4.9.5 Concrete parameters

In the Overdrive paper the slack estimations were taken verbatim from SPDZ-1 [DPSZ12] which used B_{clean} instead of $B^{\mathsf{dishonest}}_{\mathsf{clean-hg}}$ and gave a noise bound of

$$B^{\mathsf{dishonest}}_{\mathsf{clean-hg}}* = (B_{\mathsf{clean}} - \phi(m) \cdot p/2) \cdot 2^{3\mathsf{sec}/2+1} + \phi(m) \cdot p/2.$$

According to the calculations done in this thesis, as well as TopGear paper or SCALE documentation the slack in Overdrive is slightly mistaken due to some incorrect bounds. Nevertheless, the ciphertext size only decreases slightly due to the modulus switching operation which removes the slack by a large amount.

Damgård et al. [CDXY17] presented an improved version of the cut-and-choose proof used in a previous implementation of SPDZ [DKL+13], but the reduced slack does not justify the increased complexity caused by several additional ciphertexts being computed and sent in the proof. Consider that, even for $\mathsf{sec} = 128$ and $N = 2^{15}$ (the latter being typical for our parameters), $\log S$ is about 100, increasing the ciphertext modulus length by less than 25 percent.

In Table 4.1 we have calculated the ciphertext modulus q's bit length for various parameters and for our protocol with semi-homomorphic encryption (supporting only plaintext-ciphertext multiplication) and SPDZ (using somewhat homomorphic encryption which supports ciphertext-ciphertext multiplication as well). Then we instantiated both protocols with several ZK proofs like the Schnorr-like protocol [CD09, DPSZ12] and the recent cut-and-choose proof [CDXY17]. Table 4.1 shows the results of our calculation as well as the results given by Damgård et al. [DKL+13]. One can see that using

LowGear		SPDZ			[BCS19]			
							sec	$\log \lvert\mathbb{F}\rvert$
[CD09]	[CDXY17]	1 [DPSZ12]	2 [CDXY17]	2 [DKL+13]	HighG	TopG		
238	199	330	330	332	291	291	40	64
367	327	526	526	526	490	490	40	128
276	224	378	378	N/A	340	340	64	64
406	352	572	572	N/A	540	540	64	128
504	418	700	700	N/A	660	660	128	128

Table 4.1: Ciphertext modulus bit length ($\log(q)$) for two parties.

cut-and-choose instead of the Schnorr-like protocol does not make any difference for SPDZ. This is because the scaling (also called modulus switching) involves the division by a number larger than the largest possible slack of the Schnorr-like protocol (roughly 2^{200}), hence the slack will be eliminated. For our LowGear protocol, the slack has a slight impact, increasing the size of a ciphertext by up to 25 percent. However, this does not justify the use of a cut-and-choose proof because it involves sending seven instead of two extra ciphertexts per proof.

Table 4.1 also shows LowGear ciphertexts are about 30 percent shorter than SPDZ ciphertexts. Consider that Table 4.3 in Section 4.10 shows a reduction in the communication from SPDZ to LowGear of up to 50 percent. The main reason for the additional reduction is the fact that for one guaranteed triple, SPDZ involves producing two triples $(a, b, c), (d, e, f)$, of which (a, b, d, e) require a zero-knowledge proof. In LowGear on the other hand, we produce $(a, b, c, \hat{b}, \hat{c})$, of which only a requires a zero-knowledge proof. We have also updated the table containing the correct slack analysis of HighGear from Section 4.9.4 along with the TopGear parameters [BCS19] where HighGear is denoted as HighG and TopGear as TopG to then compare with the old analysis done in Overdrive. It turns out that using a more rigorous methodology to analyze the SHE bounds give rise to smaller ciphertexts, a little more than 5% of improvement.

4.10 Implementation

We have implemented all three approaches to triple generation in this paper and measured the throughputs achieved by them in comparison to previous results with SPDZ [DKL+12, DKL+13] and MASCOT [KOS16]. We have used the optimized distributed decryption in for SPDZ-1, SPDZ-2, and HighGear. Our code is written in C++ and uses MPIR [MPI19] for arithmetic with large integers.[1] We use Montgomery modular multiplication and the Chinese reminder theorem representation of polynomials wherever beneficial. See Gentry et al. [GHS12] for more details.

[1] We extensively use the function `mpn_addmul_1`, which we found to be 10–20 percent faster in MPIR compared to GMP. Both libraries have implemented this function in Assembly but MPIR has a more specialized version, including a specific one for Sandybridge/Ivybridge and one for Broadwell/Haswell while GMP features one just for the latter.

	Triples/s	Security	BGV impl.	$\log_2(\lvert\mathbb{F}_p\rvert)$
SPDZ-1 [DKL+12]	79	40-bit active	NTL	64
SPDZ-2 [DKL+13]	158	20-covert	specific	64
SPDZ-2 [DKL+13]	36	40-bit active	specific	64
MASCOT [KOS16]	5,100	64-bit active	$\perp$	128
SPDZ-1 (ours)	12,000	40-bit active	specific	64
SPDZ-1 (ours)	6,400	64-bit active	specific	128
SPDZ-1 (ours)	4,200	128-bit active	specific	128
SPDZ-2 (ours)	3,900	20-covert	specific	64
SPDZ-2 (ours)	1,100	40-bit active	specific	64
LowGear (Section 4.8)	59,000	40-bit active	specific	64
LowGear (Section 4.8)	30,000	64-bit active	specific	128
LowGear (Section 4.8)	15,000	128-bit active	specific	128
HighGear (Section 4.9)	11,000	40-bit active	specific	64
HighGear (Section 4.9)	5,600	64-bit active	specific	128
HighGear (Section 4.9)	2,300	128-bit active	specific	128

Table 4.2: Triple generation for 64 and 128 bit prime fields with two parties on a 1 Gbit/s LAN.

Note that the parameters chosen by Damgård et al. [DKL+13][Appendix A] for the non-interactive zero-knowledge proof imply that the prover has to re-compute the proof with probability $1/32$ as part of a technique called rejection sampling. We have increased the parameters to reduce this probability by up to 2^{20} as long as it would not impact on the performance, i.e., the number of 64-bit words needed to represent p_0 and p_1 would not change.

All previous implementations have benchmarks for two parties on a local network with 1 Gbit/s throughput on commodity hardware. We have have used i7-4790 and i7-3770S CPUs with 16 to 32 GB of RAM, and we have re-run and optimized the code by Damgård et al. [DKL+13] for a fairer comparison. Table 4.2 shows our results in this setting. SDPZ-1 and SPDZ-2 refer to the two different proofs for ciphertexts, the Schnorr-like protocol presented in the original paper [DPSZ12] and the cut-and-choose protocol in the follow-up work [DKL+13], the latter with either covert or active security. The c-covert security is defined as a cheating adversary being caught with probability $1/c$, and by sec-bit security we mean a statistical security parameter of sec. Throughout this section, we will round figures to the two most significant digits for a more legible presentation.

To allow direct comparisons with previous work, we have benchmarked our protocols for several choices of security parameters and field size. Note that the computational security parameter is set everywhere to $k = 128$ and we highlight how the statistical parameter impacts the performance. The main difference between our implementation of SPDZ with the Schnorr-like protocol to the previous one [DKL+12], is the underlying BGV implementation because the protocol is the same.

In Table 4.3 we also analyze the communication per triple of some protocols with active security and compared the actual throughput to the maximum possible on a 1 Gbit/s link (network through-

	Communication	Security	$\log_2(\mathbb{F}_p\|)$	Triples/s	Maximum
SPDZ-2	350 kbit	40	64	1,100	2,900
MASCOT [KOS16]	180 kbit	64	128	5,100	5,600
SPDZ-1	23 kbit	40	64	12,000	44,000
SPDZ-1	32 kbit	64	128	6,400	31,000
SPDZ-1	37 kbit	128	128	4,200	27,000
LowGear (Section 4.8)	9 kbit	40	64	59,000	110,000
LowGear (Section 4.8)	15 kbit	64	128	30,000	68,000
LowGear (Section 4.8)	17 kbit	128	128	15,000	60,000
HighGear (Section 4.9)	24 kbit	40	64	11,000	42,000
HighGear (Section 4.9)	34 kbit	64	128	5,600	30,000
HighGear (Section 4.9)	42 kbit	128	128	2,300	24,000

Table 4.3: Communication per prime field triple (one way) and actual vs. maximum throughput with two parties on a 1 Gbit/s link.

put divided by the communication per triple). The higher the difference between actual and maximum possible, the more time is spent on computation. The figures show that MASCOT has very low computation; the actual throughput is more than 90% of the maximum possible. On the other hand, all BGV-based implementations have a significant gap, which is to be expected. Experiments have shown that the relative gap increases in LowGear with a growing statistical parameter. This is mostly because the ciphertexts become larger and 32 GB of memory is not enough for one triple generator thread per core, hence there is some computation capacity left unused.

4.10.1 WAN setting

For a more complete picture, we have also benchmarked our protocols in the same WAN setting as Keller et al. [KOS16], restricting the bandwidth to 50 Mbit/s and imposing a delay of 50 ms to all communication. Table 4.4 shows our results in similar manner to Table 4.3. As one would expect, the gap between actual throughput and maximum possible is more narrow because the communication becomes more of a bottleneck, and the performance is closely related to the required communication.

4.10.2 More than two parties.

Increasing the number of parties, we have benchmarked our protocols and our implementation of SPDZ with up to 64 r4.16xlarge instances on Amazon Web Services. Figure 4.23 shows that both Low and High Gear improve over SPDZ-1, with HighGear taking the lead from about ten parties. Missing figures do not indicate failed experiments but rather omitted experiments due to financial constraints.

At the time of writing, one hour on an r4.16xlarge instance in US East costs $4.256. Therefore, the number of triples per dollar and party varies between 190 million (two parties with LowGear) and 13 million (64 parties with HighGear).

	Communication	Security	$\log_2(\mathbb{F}_p\|)$	Triples/s	Maximum
MASCOT [KOS16]	180 kbit	64	128	214	275
SPDZ-1	23 kbit	40	64	1,800	2,200
SPDZ-1	32 kbit	64	128	1,400	1,600
SPDZ-1	37 kbit	128	128	1,100	1,400
LowGear (Section 4.8)	9 kbit	40	64	4,500	5,600
LowGear (Section 4.8)	15 kbit	64	128	3,200	3,400
LowGear (Section 4.8)	17 kbit	128	128	2,600	3,000
HighGear (Section 4.9)	24 kbit	40	64	1,600	2,100
HighGear (Section 4.9)	34 kbit	64	128	1,300	1,500
HighGear (Section 4.9)	42 kbit	128	128	700	1,200

Table 4.4: Communication per prime field triple (one way) and actual vs. maximum throughput with two parties on a 50 Mbit/s link.

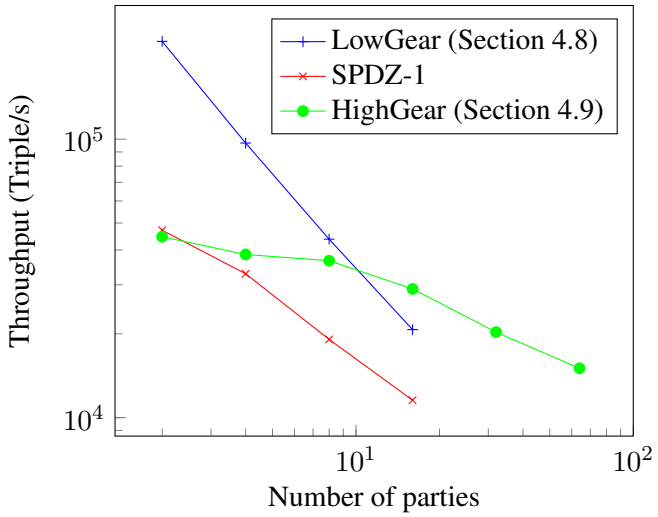

Figure 4.23: Triple generation for a 128 bit prime field with 64 bit statistical security on AWS r4.16xlarge instances.

4.10.3 Vickrey Auction for 100 Parties

As a motivation for computation with a high number of parties, we have implemented a secure Vickrey second price auction [Vic61], where 100 parties input one bid each. Table 4.5 shows our online phase timings for two different Amazon Web Services instances. The Vickrey auction requires 44,571 triples. In Table 4.6, we compare the offline cost of MASCOT and our High Gear protocol on AWS m3.2xlarge instances.

AWS instance	Time	Cost per party
t2.nano	9.0 seconds	$0.000017
c4.8xlarge	1.4 seconds	$0.000741

Table 4.5: Online phase of Vickrey auction with 100 parties, each inputting one bid.

	Time	Cost per party
MASCOT [KOS16]	1,300 seconds	$0.190
HighGear (Section 4.9)	98 seconds	$0.014

Table 4.6: Offline phase of Vickrey auction with 100 parties, each inputting one bid.

	Triples/s	Security	BGV impl.	$\mathbb{F}_{2^n}$
SPDZ-1 [DKL+12]	16	40-bit active	NTL	40
MASCOT [KOS16]	5,100	64-bit active	$\perp$	128
SPDZ-1 (ours)	67	40-bit active	specific	40
SPDZ-2 (ours)	24	20-covert	specific	40
SPDZ-2 (ours)	8	40-bit active	specific	40
LowGear (Section 4.8)	117	40-bit active	specific	40
HighGear (Section 4.9)	67	40-bit active	specific	40

Table 4.7: Triple generation for characteristic two with two parties on a 1 Gbit/s LAN.

4.11 Alternatives for fields of characteristics two

For a more thorough comparison with MASCOT, we have also implemented our protocols for the field of size 2^{40} using the same approach as Damgård et al. [DKL+12]. Table 4.7 shows the low performance of homomorphic encryption-based protocols with fields of characteristic two. This has been observed before: in the above work, the performance for $\mathbb{F}_{2^{40}}$ is an order of magnitude worse than for $\mathbb{F}_p$ with a 64-bit bit prime. The main reason is that BGV lends itself naturally to plaintexts modulo some integer p. The construction for $\mathbb{F}_{2^{40}}$ sets $p = 2$ and uses 40 slots to represent an element whereas an element of $\mathbb{F}_p$ for a prime p only requires one ciphertext slot.

Chapter 5

PRFs for fields of characteristics two

This chapter is based on joint work with Marcel Keller and Emmanuela Orsini and Peter Scholl and Eduardo Soria-Vazquez and Srinivas Vivek [KOR+17] which was presented at ACNS 2017.

5.1 Contributions

In this chapter we focus on evaluating traditional blockciphers (eg., AES and DES) in MPC using the SPDZ protocol. We investigate several approaches which involve representing the specific S-boxes for AES or DES as either a polynomial or a look-up table. We then improve on the TinyTable protocol by Damgård et al. [DNNR17] (CRYPTO 2017) by at least a factor of 50 while also extending their protocols to the multiparty setting.

5.2 Overview

The TinyTable protocol provides a very efficient online phase evaluation for S-boxes or lookup tables by creating special correlated randomness in the preprocessing phase which is input independent. Their online phase requires each party to send $\log_2 N$ bits to evaluate a look-up table of size N with a secret index. The downside is that in the preprocessing phase they have to obtain a scrambled version of the table for each individual entry. Hence for evaluating an AES table (256 entries) the price one needs to pay for a fast online phase is increasing the overall runtime of the protocol by a factor of 256.

Our work extends the TinyTable by porting their protocol from the two-party semi-honest setting to work with any number of partes and resistant against a dishonest majority. Moreover, we reduce drastically the number of field multiplications in the preprocessing phase. When instantiating AES with our improved TinyTable protocol for arithmetic circuits this approach turns out to be competitive in the *overall* running time against the fastest known evaluations of AES while also preserving the highest throughput performance.

We also look at some of the techniques from the side-channel countermeasures literature to take the most efficient polynomial representation of an S-box. There is a rich work regarding the evaluation

of blockciphers using masking (a popular technique against side-channel attacks) [CGP+12, RV13, CRV14, PV16] which translates very well to the multiparty computation domain. In fact we are going to use some of the techniques used by Pulkus and Vivek [PV16] to optimize DES S-box evaluation. Since an DES S-box maps a 6 bit input to 4 bit output via a 62 degree polynomial it would take 62 field multiplications to evaluate an S-box naïvely in MPC. Pulkus and Vivek showed that instead of looking at the DES S-boxes as polynomials over $\mathbb{F}_{2^6}$, one can interpret it as a string of bits in $\mathbb{F}_{2^8}$ where it can be evaluated using only 3 non-linear multiplications. This brings a 25% improvement on the number of multiplications (vs. evaluating the S-box traditional $\mathbb{F}_{2^6}$) which can be done in 4 multiplications (vs. 62 multiplications naïvely). This means that with Pulkus and Vivek method of embedding the S-box calculation into $\mathbb{F}_{2^8}$ we can evaluate an entire DES round using 24 multiplications.

In the end we show how that our implementation using lookup tables achieves the highest online throughput and lowest online latency whereas using the side-channel inspired techinques speeds up the costly preprocessing phase achieving one of the fastest overall protocol execution for the discussed blockciphers.

5.3 Preliminaries

5.3.1 Advanced Encryption Standard

The algorithm supports three security parameters: 128, 192 and 256 which represents the key-length as the block size has always 128 bit length. In our work we will focus on the standard AES specification which guarantees 128 bit security, i.e. has a key k size of 128 bits long. AES starts with a key-schedule which expands the key into 10 sub-keys called round keys. During the encryption algorithm for each round a state is produced by taking the previous state and applying the following steps:

1. SubBytes: applies the non-linear layer (S-box) for each consecutive byte in the state.
2. ShiftRows: simply rotates the state cyclically to the left.
3. MixColumns: computes a linear transformation of the state with a fixed matrix.
4. AddRoundKey: XORs the round key to the state which gives the state for the next round: s_{r+1}.

Each round is described in more details in Figure 5.1 and it is iterated 10 times to get security level of 128 bits. We do not describe how to obtain the round keys but the process is very similar to the encryption: copy the key into an initial state and then apply AES encryption rounds using the key k over the state a slightly different S-Box in the SubBytes step. The expanded key represents each intermediary state at the end of each of the 10 AES iterations.

5.3.2 Data Encryption Standard and Triple-DES

The Data Encryption Standard (DES) was introduced by the National Institute of Standards and technology (NIST) in 1977. The DES blockcipher has a key size is of 56 bits with a block size equal to 64 bits long. Unlike AES, DES uses a Feistel substitution network where for each round the input is split

Consider a round r with a 128 bit key k_r and 128 bit state s_{r-1} output from round $r-1$. If $r = 1$ then $s_0 := m$ where m is the initial message to be encrypted.

1: //SubBytes
2: **for** $i = 0, \mathsf{len}(s)$ **do** $s_r[i] \leftarrow \mathsf{S\text{-}box}(s_{r-1}[i])$
3: //ShiftRows
4: **for** $i = 0, \mathsf{len}(s)$ **do** $s_r[i ::] \leftarrow \mathsf{Rotate}(s_r[i ::], i)$
5: // Doing the MixColumns step
6: $s'_r[0] \leftarrow 2 \cdot s_r[0] \oplus 3 \cdot s_r[1] \oplus 1 \cdot s_r[2] \oplus 1 \cdot s_r[3]$
7: $s'_r[1] \leftarrow 1 \cdot s_r[0] \oplus 2 \cdot s_r[1] \oplus 3 \cdot s_r[2] \oplus 1 \cdot s_r[3]$
8: $s'_r[2] \leftarrow 1 \cdot s_r[0] \oplus 1 \cdot s_r[1] \oplus 2 \cdot s_r[2] \oplus 3 \cdot s_r[3]$
9: $s'_r[2] \leftarrow 3 \cdot s_r[0] \oplus 1 \cdot s_r[1] \oplus 1 \cdot s_r[2] \oplus 2 \cdot s_r[3]$
10: // AddRoundKey
11: $s_r \leftarrow s'_r \oplus k_r$

Figure 5.1: 1 AES encryption round.

in two equal halves: left (L) and right (R). At round i the left half $L_i \leftarrow R_{i-1}$ and the right half takes a non-linear function dependent on the round key and the previous half: $R_i \leftarrow L_{i-1} \oplus f_i(k_i, R_{i-1})$.

This Feistel network process of switching input halves is iterated 16 times where the function $f_i(k_i, R_{i-1})$ consists in:

1. Expansion: Compute a 48 bit vector R'_{i-1} which just duplicates some of the bits in the 32 bit string R_i.
2. KeyMixing: Extracts from the original 56-bit key a 48-bit round key according to a key schedule mechanism. This round key is then viewed as 8 consecutive pieces of 6 bit-string.
3. Substitution: A non-linear layer is applied to each 6-bit chunk and maps it to a 4-bit output. This non-linear layer is slightly weaker than the one used in AES as it can be represented by a lower degree polynomial.
4. Permutation. Takes the 32-bit output (8 chunks times 4-bit S-box output) from the previous Substitution step and permutes the bits using a fixed permutation.

The advantage of a Feistel network is that during the decryption process f does not need to be invertible as to compute DES^{-1} one needs to set (L_{i-1}, R_{i-1}) as $R_{i-1} \leftarrow L_i$ and $L_{i-1} \leftarrow L_i \oplus f_i(k_i, R_i)$. Triple DES (known as 3-DES) is computed using three keys k_1, k_2, k_3, each 56-bit long, by defining $\text{3-}\mathsf{DES}_{k_1,k_2,k_3}(x) := \mathsf{DES}_{k_1}(\mathsf{DES}^{-1}_{k_2}(\mathsf{DES}_{k_3}(x)))$.

Although 3-DES has three independent keys, it is vulnerable to attacks based on the state length. Due to its short 64-bit state, one of the newest attacks called Sweet32 exploits the vulnerability that there is a high chance of state collisions, successfully mounting an attack using only $2^{36.8}$ although researchers were lucky to find a collision only after 2^{20} queries [BL16]. Even though 3DES is entirely broken, it is used by at least 3000 vendors as of end of 2019 [NISb] and more than 5000 supporting AES [NISa].

Consider an AES input block $x \in \mathbb{F}_{2^8}$ where $\mathbb{F}_{2^8} = \mathbb{F}_2/(x^8 + x^4 + x^3 + x + 1)$.

1: $(x_0, \ldots, x_7) \leftarrow \mathsf{BitDec}(x)$
2: $y_0 \leftarrow \oplus_{i=0}^{7} x_i$
3: $y_1 \leftarrow x_1 \oplus x_3 \oplus x_5 \oplus x_7$
4: $y_2 \leftarrow x_2 \oplus x_3 \oplus x_6 \oplus x_7$
5: $y_3 \leftarrow x_3 \oplus x_7$
6: $y_4 \leftarrow x_4 \oplus x_5 \oplus x_6 \oplus x_7$
7: $y_5 \leftarrow x_5 \oplus x_7$
8: $y_6 \leftarrow x_6 \oplus x_7$
9: $y_7 \leftarrow x_7$
10: **return** $\sum_{i=0}^{7} y_i \cdot X^{5i}$

Figure 5.2: $\mathbb{F}_{2^8} \hookrightarrow K_{40}$ embedding.

5.4 MPC Evaluation of AES using polynomials

We now recap some previous work which focused on minimizing the number of non-linear multiplications to evaluate in AES S-box. To be able to catch a cheating adversary with probability $1 - \frac{1}{2^{40}}$ in SPDZ we need to perform operations over $\mathbb{F}_{2^{40}}$. If we choose to work over $\mathbb{F}_{2^8}$ (as AES natively does) then the computation has to be repeated at least five times. As a consequence the online phase will be considerably slower due to repeating the protocol several times hence we stick to evaluating the circuit over $\mathbb{F}_{2^{40}}$.

5.4.1 Embedding AES blocks into $\mathbb{F}_{2^{40}}$

AES algorithm consists in manipulating 16 blocks of 8-bit strings. The approach taken by Damgård et al. [DKL+12] is to embed each block into

$$K_{40} = \mathbb{F}_2/(y^{40} + y^{20} + y^{15} + y^{10} + 1).$$

This embedding works by mapping each monomial $X^i \in \mathbb{F}_{2^8}$ to $X^{5i} + 1 \in K_{40}$. Concretely, after having the bit-decomposition of an element $x \in \mathbb{F}_{2^8}$, the algorithm is described in Figure 5.2.

To get the final output one needs to be able to invert the embedding. As its forward embedding counterpart, first we do a bit decomposition and perform the operations from Figure 5.2 in reverse order. The precise algorithm is described in Figure 5.3. One small optimization to reduce the communication overhead is that we only need to 8 random bits to mask the embedded output $y \in K_{40}$ since we know that the remaining entries of y are useless.

After inputs are embedded into K_{40} the issue that needs to be solved is computing the AES S-box in MPC. The AES S-box consists in mapping an element $x \in \mathbb{F}_{2^8}$ to its inverse x^{-1} or x^{254} in $\mathbb{F}_{2^8}$, where zero element is mapped to zero, and then apply an affine transformation on x^{-1}. There are other ways of computing an S-Box using Keller and Damgård [DK10] method but we choose to avoid it since that can make a malicious adversary detect zero inputs to S-box with a high probability of $1/2^8$. When

Consider an AES input block $y \in K_{40}$ where $K_{40} = \mathbb{F}_2/(x^{40} + x^{20} + x^{15} + x^{10} + 1)$.

1: $(y_0, \ldots, y_7) \leftarrow \mathsf{BitDec}(y, \mathsf{step} = 5)$ // skip 5 bits at a time
2: $x_7 \leftarrow y_7$
3: $x_6 \leftarrow y_6 \oplus y_7$
4: $x_5 \leftarrow y_5 \oplus y_7$
5: $x_4 \leftarrow y_4 \oplus x_5 \oplus x_6 \oplus x_7$
6: $x_3 \leftarrow y_3 \oplus y_7$
7: $x_2 \leftarrow y_2 \oplus x_3 \oplus x_6 \oplus x_7$
8: $x_1 \leftarrow y_1 \oplus x_3 \oplus x_5 \oplus x_7$
9: $x_0 \leftarrow y_0 \oplus_{i=1}^{7} x_i$
10: **return** $\sum_{i=0}^{7} x_i \cdot X^i$

Figure 5.3: $K_{40} \hookrightarrow \mathbb{F}_{2^8}$ un-embedding.

computing x^{254} using the following two methods it can be seen that the only power an adversary has is to find out the SPDZ MAC key which is can be guessed with probability 2^{-40}.

5.4.2 Rivain-Prouff method [RP10]

The method below is a variant, also used by Gentry et al. [GHS12], of the method of Rivain–Prouff [RP10] to evaluate the AES S-box polynomial using only 4 non-linear multiplications in $\mathbb{F}_{2^8}[X]$. They compute a sequence of monomials as below:

$$\{X, X^2\} \xrightarrow{\times} \{X^3, X^{12}\} \xrightarrow{\times} \{X^{14}\} \xrightarrow{\times} \{X^{15}, X^{240}\} \xrightarrow{\times} X^{254}.$$

Although in Gentry et al. [GHS12] the squaring operation comes for free by doing in local operations on the SHE slots, in MPC this requires some communication between the parties with additional pre-processing. To square one authenticated element $[\![x]\!]$, parties extract the bits of $[\![x]\!]$ by opening $[\![x]\!] \oplus [\![r]\!]$ and then manipulate locally the bits of $[\![r]\!]$ and $x + r$ to get any squaring of $[\![x]\!]$. Fortunately generating a single random shared bit $[\![r_i]\!]$ in a characteristic two field is extremely cheap, around $\log |\mathbb{F}|$ times faster than generating a multiplication triple in $\mathbb{F}$. In our experiments we denote this method by `AES-RP`.

MPC complexity. The multiplication chain can be achieved using four multiplication triples and 7 communication rounds due to several bit-decompositions between multiplications. The four triples come from computing $X^3, X^{14}, X^{15}, X^{254}$. Although we need to do a BitDec at each step, some of them can be done in parallel resulting in a circuit with a multiplicative depth equal to six. To compute MixColumns an extra call to BitDec will be used.

5.4.3 Bit-Decomposition method of Damgård et al. [DKL+12]

As described in previous section, squaring in MPC is for free as long as one has the shared bits of a secret. The approach taken by Damgård et al. [DKL+12] tries to minimize the number of non-linear multiplications by computing a higher number of squarings.

The evaluation proceeds as follow: first X is bit-decomposed so that all the squarings can be locally evaluated, and then X^{254} is obtained as described in [DKL+12] via

$$X^{254} = ((X^2 \cdot X^4) \cdot (X^8 \cdot X^{16})) \cdot ((X^{32} \cdot X^{64}) \cdot X^{128}).$$

This requires 4 rounds, out of which one is a call to BitDec. We also need an extra round for computing the inverse of the field embedding $\mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{40}}$ to evaluate the S-box linear layer. In our experiments we denote this method by `AES-BD`.

5.5 MPC Evaluation of DES using polynomials

Recall that in each Substitution round DES applies eight 6-to-4 bit S-boxes. The naïve method of evaluating a DES S-box is by constructing a polynomial over $\mathbb{F}_{2^6}$ where the input and output bit-strings are elements in $\mathbb{F}_{2^6}$. To get the DES S-box 4-bit output from such a polynomial evaluation one has to remove the two most significant bits due to zero padding. Roy and Vivek [RV13] show that this polynomial has a degree of at most 62.

To be able to evaluate DES S-boxes as a polynomials, we first recall the class of cyclotomic polynomials. Over $\mathbb{F}_{2^m}[X]$, define

$$C_i^m := \left\{ X^{i \cdot 2^j} \ : \ j = 0, 1, \ldots, m-1 \right\} \text{ for } 0 < i < 2^m. \tag{5.1}$$

Note that $X^{2^m} = X$ in $\mathbb{F}_{2^m}[X]/(X^{2^m} + X)$.

In order to minimize the number of nonlinear multiplications, we can do a Breadth-First-Search (BFS) over the cyclotomic classes. Each node will be represented by one class whereas edges are drawn from one class C_a to another class C_b whether C_b can be derived from C_a using one non-linear multiplication. In order to cover all monomials, a BFS search starting from C_0 will reveal the minimum number of non-linear operations. As previously mentioned, once a monomial is computed, it is repeatedly squared to generate more monomials without costing additional non-linear multiplications. Once the BFS computes the monomials graph, the S-box evaluation will consist in performing a linear combination of the resulted monomials.

For example, if we compute 13 distinct classes of monomials then all the monomials can be used to cover every linear combination of up to degree 62 in $\mathbb{F}_{2^6}[X]$:

$$C_0^6, C_1^6, C_3^6, C_5^6, C_7^6, C_9^6, C_{11}^6, C_{13}^6, C_{15}^6, C_{21}^6, C_{23}^6, C_{27}^6, C_{31}^6.$$

In this case C_0^6 and C_1^6 can be computed directly from the input X, meaning the DES S-boxes can be evaluated with at most 11 non-linear multiplications by combining different classes C_i^m.

5.5.1 Embedding $\mathbb{F}_{2^6}$ multiplications in $\mathbb{F}_{2^{42}}$

The naïve method to compute a DES S-box is to first do a Lagrange interpolation on the S-box lookup tables. Afterwards the 6-bit input block is embedded into $\mathbb{F}_{2^{42}}$ to have at least the amount of statistical

Consider an DES input block $x \in \mathbb{F}_{2^8}$ where $\mathbb{F}_{2^6} = \mathbb{F}_2/(X^6 + X^4 + X^3 + X + 1)$.

1: $(x_0, \ldots, x_5) \leftarrow \mathsf{BitDec}(x)$
2: $y_0 \leftarrow \oplus_{i=0}^{5} x_i$
3: $y_1 \leftarrow x_1 \oplus x_3 \oplus x_5$
4: $y_2 \leftarrow x_2 \oplus x_3$
5: $y_3 \leftarrow x_3$
6: $y_4 \leftarrow x_4 \oplus x_5$
7: $y_5 \leftarrow x_5$
8: **return** $\sum_{i=0}^{6} y_i \cdot X^{7i}$

Figure 5.4: $\mathbb{F}_{2^6} \hookrightarrow \mathbb{F}_{2^{42}}$ embedding.

security against a cheating party (as described for AES in the previous section). Since the Lagrange polynomial has degree 62, in the end we can naïvely perform 62 the non-linear multiplications. This will serve as the baseline for the blockcipher computation and is denoted in the experiments as `3DES-Raw`. It is straightforward to see that this can be done using 62 rounds with 62 triples in $\mathbb{F}_{2^{42}}$.

In Figure 5.4 we describe how to embed:

$$\mathbb{F}_2[X]/(X^6 + X^4 + X^3 + X + 1) \hookrightarrow \mathbb{F}_2[X]/(X^{42} + X^{21} + 1)$$

via $X \mapsto X^7 + 1$.

5.5.2 Pulkus–Vivek Method [PV16]

Pulkus and Vivek [PV16] propose an improvement over the method Coron–Roy–Vivek line of work [RV13, CRV14] to evaluate arbitrary polynomials over finite fields of characteristic two. They view a DES S-box input as a field element over $\mathbb{F}_{2^8}$ instead of $\mathbb{F}_{2^6}$ where the most significant digits are two zeros. To get the S-box output an $\mathbb{F}_{2^8}$ polynomial is evaluated and then discard the top-most four coefficients to get a 4 bit output.

To understand their idea, a set of monomials $L = C_1^8 \cup C_3^8 \cup C_7^8$ in $\mathbb{F}_{2^8}[X]$ is computed. We know from Equation 5.1 that

$$\begin{aligned} C_1^8 &= \{X, X^2, X^4, X^8, X^{16}, X^{32}, X^{64}\}, \\ C_3^8 &= \{X^3, X^6, X^{12}, X^{24}, X^{48}, X^{96}, X^{65}\}, \\ C_7^8 &= \{X^7, X^{14}, X^{28}, X^{56}, X^{112}, X^{97}, X^{67}\}. \end{aligned}$$

Consider that a polynomial $P(X)$ represents the S-box computation. Then $P(X)$ can be written down in the following form

$$P(X) = p_1(X) \cdot q_1(X) + p_2(X)$$

where $p_1(X)$, $q_1(X)$, and $p_2(X)$ have monomials only from the set L. These three polynomials are computed by first assigning to $q_1(X)$ a random set of monomials from L. After $q_1(X)$ is fixed they

Functionality $\mathcal{F}_{\mathsf{ABB-LUT}}$

This functionality has all the features of $\mathcal{F}_{\mathsf{ABB}}$, operating over $\mathbb{F}_{2^k}$, plus the following command.
Table Look-up: On command $(\mathsf{T}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties, where $\mathsf{T} : \{0,1\}^\ell \to \{0,1\}^m$, for $\ell, m \leq k$, and id_1 is present in memory, retrieve (id_1, x) and store $(\mathsf{id}_2, \mathsf{T}(x))$.

Figure 5.5: The ideal functionality for MPC using lookup tables.

set up a linear system of equations for the coefficients of $p_1(X)$ and $p_2(X)$ obtained from evaluating a DES S-box in each entry. Pulkus and Vivek noticed that using this process, with high probabilty we obtain a valid $P(X)$ equivalent to computing a DES S-box.

Pulkus and Vivek noticed that one can choose the same $q_1(X)$ for all S-boxes and end up with a valid $P(X)$. Alas, this has no impact on the performance of the MPC evaluation. Summing up, the PV method requires 3 non-linear multiplications in $\mathbb{F}_{2^8}[X]$ to compute each DES S-box: two multiplications for computing C_3^8 and C_7^8 and one to multiply $p_1(X) \cdot q_1(X)$, the remaining operations are simply linear computations and can be obtain by doing local computations in MPC.

MPC Complexity. Note that, although in side-channel world computing the squares is for free, since it is an $\mathbb{F}_2$-linear operation, in a secret-shared based MPC with MACs this is no longer true and we need to bit-decompose. The squares from C_1^8, C_3^8, C_7^8, are obtained locally after X, X^3, X^7 are bit-decomposed. Here we need two multiplications, since $X^3 = X \cdot X^2$ and $X^7 = X^3 \cdot X^4$. The third multiplication occurs when computing the product $p_1(X) \cdot q_1(X)$, resulting in an S-box cost of only 3 triples, 24 bits and 5 communication rounds.

The number of rounds is given by the 3 calls to BitDec (on X^3, X^7 and $p_1(X) \cdot q_1(X) + p_2(X)$) and 3 non-linear multiplications. Although at a first glance there seems to be six rounds, we have that $\mathsf{BitDec}(X^7)$ is independent of the $\mathsf{BitDec}(X^3)$, as we can compute X^7 without the call $\mathsf{BitDec}(X^3)$, resulting in only five rounds.

5.6 MPC Evaluation of Boolean Circuits using Look-up Tables

We now proceed to describe how to evaluate Look-up Tables in MPC over arithmetic fields of characteristic two. The protocols are given in the preprocessing model and has the same online phase as [DZ16]. Furthermore, our protocols significantly improve over the preprocessing while also maintaining a competitive online phase to [DNNR17]. For a table of size N parties need to communicate $\log_2 N$ bits in the online phase.

The functionality that we implement is $\mathcal{F}_{\mathsf{ABB-LUT}}$ (Figure 5.5), which augments the standard $\mathcal{F}_{\mathsf{ABB}}$ functionality with a table look-up command. The concrete online cost of each table look-up is just $\log_2 N$ bits of communication per party, where N is the size of the table. Note that the functionality $\mathcal{F}_{\mathsf{ABB-LUT}}$ works over a finite field $\mathbb{F}_{2^k}$, and has been simplified by assuming that the size of the range and domain of the look-up table T is not more than 2^k. However, our protocol actually works for general

Functionality $\mathcal{F}_{\mathsf{Prep-LUT}}$

This functionality has all of the same features as $\mathcal{F}_{\mathsf{ABB}}$, with the following additional command.

Masked Table: On Input $(\mathsf{MaskedTable}, \mathsf{T}, \mathsf{id})$ from all parties, where $\mathsf{T} : \{0,1\}^\ell \to \{0,1\}^m$ for $\ell, m \leq k$, sample a random value s, set $(\mathsf{Val}[\mathsf{id}_s], \mathsf{Val}[\mathsf{id}_{\mathsf{T}(s)}], \ldots, \mathsf{Val}[\mathsf{id}_{\mathsf{T}(s\oplus(2^\ell-1))}]) \leftarrow (s, \mathsf{T}(s), \ldots, \mathsf{T}(s \oplus (2^\ell - 1))$, and return $(\mathsf{id}_s, (\mathsf{id}_{\mathsf{T}(s)}, \ldots, \mathsf{id}_{\mathsf{T}(s\oplus(2^\ell-1))}))$.

Masked Function: On Input $(\mathsf{MaskedFunction}, f, \mathsf{id})$ from all parties denote with q the number of non-linear operations in f. Next decompose the non-linear operations of f into q successive evaluations of look-up tables $T^1, \ldots, T^q$ where $T^i : \{0,1\}^\ell \to \{0,1\}^m$, sample a set of random values $o_1, \ldots, o_q$ and set $(\mathsf{Val}[\mathsf{id}_{o_1}], \mathsf{Val}[\mathsf{id}_{\mathsf{T}^1(o_1)\oplus o_2}], \ldots, \mathsf{Val}[\mathsf{id}_{o_q}], \mathsf{Val}[\mathsf{id}_{\mathsf{T}^q(o_q)}], \ldots, \mathsf{Val}[\mathsf{id}_{\mathsf{T}^q(o_q\oplus(2^l-1)}]$.

Figure 5.6: Ideal functionality for the preprocessing of masked look-up tables.

Protocol Π_{LT}

Table Look-up: On Input $[\![x]\!]$ compute $[\![\mathsf{T}(x)]\!]$ as follows.

1. Call $\mathcal{F}_{\mathsf{Prep-LUT}}$ on Input $(\mathsf{MaskedTable}, \mathsf{T})$, and obtain a precomputed masked table $([\![s]\!], [\![\mathsf{Table}(s)]\!])$.
2. The parties open the value $h = x \oplus s$.
3. Locally compute $[\![\mathsf{T}(x)]\!] = [\![\mathsf{Table}(s)]\!][h]$, where $[\![\mathsf{Table}(s)]\!][h]$ is the hth component of $[\![\mathsf{Table}(s)]\!]$.

Figure 5.7: Secure online evaluation of SBox using look-up tables.

table sizes, and $\mathcal{F}_{\mathsf{ABB-LUT}}$ can easily be extended to model this by representing a table look-up result with several field elements instead of one.

We now show how Protocol 5.7 implements the **Table Look-up** command of $\mathcal{F}_{\mathsf{ABB-LUT}}$, given the right preprocessing material. For any non-linear function T, with ℓ Input and m output bits, it is well known that it can be implemented as a look-up table of 2^ℓ components of m bits each. To evaluate $\mathsf{T}(\cdot)$ on a secret authenticated value $[\![x]\!], x \in \mathbb{F}_{2^\ell}$, the parties use a random authenticated T evaluation from $\mathcal{F}_{\mathsf{Prep-LUT}}$ (Figure 5.6). More precisely, we would like the preprocessing to output values $([\![s]\!], [\![\mathsf{Table}(s)]\!])$, where $[\![s]\!]$ is a random authenticated value unknown to the parties and $[\![\mathsf{Table}(s)]\!])$ is the table

$$[\![\mathsf{Table}(s)]\!] = \big([\![\mathsf{T}(s)]\!], [\![\mathsf{T}(s \oplus 1)]\!], \ldots, [\![\mathsf{T}(s \oplus (2^\ell - 1))]\!] \big),$$

so that $[\![\mathsf{Table}(s)]\!][j], 0 \leq j \leq 2^\ell - 1$, denotes the element $[\![\mathsf{T}(s \oplus j)]\!]$. Given such a table, evaluating $[\![\mathsf{T}(x)]\!]$ is straightforward: first the parties open the value $h = x \oplus s$ and then they locally retrieve the value $[\![\mathsf{Table}(s)]\!][h] = [\![\mathsf{T}(s \oplus h)]\!] = [\![\mathsf{T}(s \oplus s \oplus x)]\!] = [\![\mathsf{T}(x)]\!]$.

Correctness easily follows from the linearity of the $[\![\cdot]\!]$-representation and the discussion above.

Privacy follows from the fact that the value s used in **Table Look-up** is randomly chosen and is used only once, thus it perfectly blinds the secret value x.

5.6.1 More Efficient Variant with TinyTable

The method just described is similar to, but not quite as efficient as, the approach in the two-party TinyTable protocol [DNNR16]. We can modify this to match the efficiency of TinyTable, generalized to the multi-party setting. The online cost of a secure look-up to a table $\mathsf{T} : \{0,1\}^\ell \to \{0,1\}^m$ becomes that of opening an m-bit value, whereas the above method requires opening ℓ bits. This reduces the cost of an AND gate from opening 2 bits to just 1 bit, and the cost of a DES S-box from 6 bits to 4. Additionally, when implemented using SPDZ, the cost of linear operations becomes the same as computing the same operation on clear data; there is no need to also compute on (much larger) MACs, which gives a significant saving. The downside of this approach is that the preprocessing phase depends on the precise function being computed, so is less general.

We assume the gates in the circuit to be evaluated are all either $\mathbb{F}_2$-linear operations on bit vectors, or table look-up gates from $\{0,1\}^\ell \to \{0,1\}^m$. During the online phase, for each wire of the circuit, all parties will obtain a public value $x \oplus s$, where $x \in \{0,1\}^\ell$ is the *actual* value being computed on, and s is a random mask for that wire. The previous table look-up preprocessing is modified so that the i-th entry of the masked table contains a secret-sharing of $\mathsf{T}(s \oplus i) \oplus o$, where s and o are the random masks for the Input and output wires (resp.) of that table look-up gate.

This means that $x \oplus s$ does not need to be opened in the online phase. Instead, the parties open the $(x \oplus s)$-th table entry, which is $[\![\mathsf{T}(x) \oplus o]\!]$). This gives them the public value for the output wire, which can be used in the next gate. Linear gates are computed in the clear on the public values. To obtain outputs at the end of the computation, the parties open the shared mask $[\![o]\!]$ (from the preprocessing) for every output wire. The online cost of each table look-up in this variant is that of opening an m bit value, instead of ℓ bits for the previous method. The preprocessing for the m-bit variant can be done with essentially the same cost as the previous section, but requires knowing the structure of the circuit in advance. The full online protocol to evaluate a sequence of look-up tables is given in Protocol 5.8.

5.6.2 The Preprocessing Phase: Securely Generating Masked Look-up Tables

In this section we describe how to securely implement the **Table Look-up** command in $\mathcal{F}_{\mathsf{Prep-LUT}}$ (see Figure 5.6), and in particular how to generate masked look-up tables which can be used for the online phase evaluation. We omit the **MaskedFunction** command as that is a trivial extension of **Table Look-up**. Recall that the goal is to obtain the shared values

$$[\![\mathsf{Table}(s)]\!] = ([\![\mathsf{T}(s)]\!], [\![\mathsf{T}(s \oplus 1)]\!], \ldots, [\![\mathsf{T}(s \oplus (2^\ell - 1))]\!]).$$

Protocol 5.9 begins by taking a secret, random ℓ-bit mask $[\![s]\!] = ([\![s_0]\!], \ldots, [\![s_{\ell-1}]\!])$. Then, the parties expand s into a secret-shared bit vector $(s'_0, \ldots, s'_{2^\ell-1})$ which has a 1 in the s-th entry and is 0 else-

Protocol $\Pi_{\mathsf{F-LT}}$

Function Look-up: On input $[\![x]\!]$ compute $[\![\mathsf{T}(\mathsf{T}(\ldots\mathsf{T}(x)))]\!]$ (considering the table is evaluated n times during f) as follows:

1. Call $\mathcal{F}_{\mathsf{Prep-LUT}}$ on Input $(\mathsf{MaskedFunction}, f)$, and obtain a set of precomputed masked tables $([\![o_1]\!], [\![\mathsf{Table}(x)]\!] \oplus o_1)$ and $([\![o_2]\!], [\![\mathsf{Table}(x) \oplus o_1 \oplus i]\!] \oplus o_2), \ldots, [\![o_q]\!], [\![\mathsf{Table}(x \oplus o_{q-1} \oplus i)]\!] \oplus o_q)$.
2. The parties open the value $y_1 \leftarrow [\![\mathsf{Table}(x) \oplus o_1]\!]$. Afterwards they open $y_2 \leftarrow [\![\mathsf{Table}(y_1)]\!] \oplus o_2$ and so on, until the last opening of $y_q \leftarrow [\![\mathsf{Table}(y_{q-1})]\!] \oplus o_q$.
3. Assign $f([\![x]\!]) \leftarrow y_q \oplus [\![o_q]\!]$ as the final function output.

Figure 5.8: More efficient online phase using look-up tables.

Protocol $\Pi_{\mathsf{MaskedTable}}$

MaskedTable: On Input $(\mathsf{MaskedTable}, \mathsf{T}, P_i)$ from all the parties, do the following:

1. Take ℓ random authenticated bits $[\![s_0]\!], \ldots, [\![s_{\ell-1}]\!]$, where each s_i is unknown to all the parties.
2. Compute $([\![s'_0]\!], \ldots, [\![s'_{2^\ell-1}]\!]) \leftarrow \mathsf{Demux}([\![s_0]\!], \ldots, [\![s_{\ell-1}]\!])$
3. $\forall i = 0, \ldots, 2^\ell - 1$, locally compute

$$[\![\mathsf{T}(i \oplus s)]\!] = \mathsf{T}(i) \cdot [\![s'_0]\!] + \mathsf{T}(i \oplus 1) \cdot [\![s'_1]\!] + \cdots + \mathsf{T}((2^\ell - 1) \oplus i) \cdot [\![s'_{2^\ell-1}]\!]$$

Figure 5.9: Protocol to generate secret shared table look-up.

where. We denote this procedure — the most expensive part of the protocol — by Demux, and describe how to perform it in the next section.

Once this is done, the parties can obtain the i-th entry of the masked look-up table by computing

$$\mathsf{T}(i) \cdot [\![s'_0]\!] + \mathsf{T}(i \oplus 1) \cdot [\![s'_1]\!] + \cdots + \mathsf{T}(i \oplus (2^\ell - 1)) \cdot [\![s'_{2^\ell-1}]\!],$$

which is clearly $[\![\mathsf{T}(i \oplus s)]\!]$ as required. Note that since the S-box is public, this is a local computation for the parties. In the following we give an efficient protocol for computing Demux.

5.6.3 Computing Demux with Finite Field Multiplications

We now present a general method for computing Demux using fewer than $N/k + \log N$ multiplications over $\mathbb{F}_{2^k}$, when k is any power of 2 and $N = 2^\ell$ is the table size. Launchbury et al. [LDDA12] previously described a protocol with $O(N)$ multiplications in $\mathbb{F}_2$, but our protocol has fewer multiplications than theirs for all choices of k.

As said before, Demux maps a binary representation $(s_0, \ldots, s_{\ell-1})$ of an integer $s = \sum_{i=0}^{\ell-1} s_i \cdot 2^i$ into a unary representation of fixed length 2^ℓ that contains a one in the position s and zeros elsewhere.

Protocol 1 $(\llbracket s'_0 \rrbracket, \ldots, \llbracket s'_{N-1} \rrbracket) \leftarrow \mathsf{Demux}(k, \llbracket s_0 \rrbracket, \ldots, \llbracket s_{\ell-1} \rrbracket)$

Require: k a power of two, $u = N/k$, $\ell = \log_2 N$
Input: Bit decomposition of $s \in \{0, \ldots, N-1\}$, with LSB first
Output: Satisfies $s'_s = 1$ and $s'_i = 0$ for all $i \neq s$

1: $\llbracket p \rrbracket = (1 - \llbracket s_0 \rrbracket, \llbracket s_0 \rrbracket)$ // p starts in $\mathbb{F}_2^2$
2: **for** $j = 1$ to $\ell - 1$ **do**
3: $\quad \llbracket t \rrbracket = \llbracket s_j \rrbracket \cdot \llbracket p \rrbracket$ // $\mathbb{F}_2 \times \mathbb{F}_2^{2^j}$ multiplication, 1 round
4: $\quad \llbracket p \rrbracket = (0^{2^j} \| \llbracket t \rrbracket) + (\llbracket p \rrbracket - \llbracket t \rrbracket) \| 0^{2^j})$ // p now in $\mathbb{F}_2^{2^{j+1}}$
5: Write $\llbracket p \rrbracket = (\llbracket b_0 \rrbracket, \ldots, \llbracket b_{u-1} \rrbracket)$ // $b_i \in \mathbb{F}_2^k$
6: **for** $i = 0$ to $u - 1$ **do**
7: $\quad (\llbracket s'_{ki} \rrbracket, \ldots, \llbracket s'_{ki+k-1} \rrbracket) = \mathsf{BitDec}(\llbracket b_i \rrbracket)$ // 1 round
8: **return** $(\llbracket s'_0 \rrbracket, \ldots, \llbracket s'_{N-1} \rrbracket)$

A straightforward way to compute Demux is by computing

$$\llbracket s' \rrbracket = \prod_{i=0}^{\ell-1} (\llbracket s_i \rrbracket \cdot X^{2^i} + (1 - \llbracket s_i \rrbracket)).$$

over $\mathbb{F}_{2^N}$[1]. Notice that if $s_i = 1$ then the i-th term of the product equals X^{2^i}, whereas the term equals 1 if $s_i = 0$. This means the entire product evaluates to $s' = X^s$, where s is the integer representation of the bits $(s_0, \ldots, s_{\ell-1})$. Bit decomposing s' obtains the demuxed output as required. Unfortunately, this approach does not scale well with N, the table size, as we must exponentially increase the size of the field.

We now show how to compute this more generally, using operations over $\mathbb{F}_{2^k}$, where k is a power of two. We will only ever perform multiplications between elements of $\mathbb{F}_2$ and $\mathbb{F}_{2^k}$, and will consider elements of $\mathbb{F}_{2^k}$ as vectors over $\mathbb{F}_2$. Define the partial products, for $j = 1, \ldots, \ell$:

$$p_j(X) = \prod_{i=0}^{j-1} (s_i \cdot X^{2^i} + (1 - s_i)) \in \mathbb{F}_{2^N}$$

and note that $p_{j+1}(X) = p_j(X) \cdot (s_j \cdot X^{2^j} + (1 - s_j))$, for $j < \ell$.

Note also that the polynomial $p_j(X)$ has degree $< 2^j$, so $p_j(X)$ can be represented as a vector in $\mathbb{F}_2^{2^j}$ containing its coefficients, and more generally, a vector p_j containing $\lceil 2^j/k \rceil$ elements of $\mathbb{F}_2^k$. This is the main observation that allows us to emulate the computation of s' using only $\mathbb{F}_{2^k}$ arithmetic.

Given a sharing of p_j represented in this way, a sharing of $p_j(X) \cdot X^{2^j}$ can be seen as the vector (increasing the powers of X from left to right):

$$(0^{2^j} \| p_j) \in \mathbb{F}_2^{2^{j+1}}$$

and a vector representation of $p_{j+1}(X)$ is:

[1] A similar trick was used by Aliasgari et al. [ABZS13] for binary to unary conversion over prime fields.

N	$k=1$	8	40	64	128
64	62	9	5	5	5
128	126	17	7	6	6
256	254	33	11	8	7
512	510	65	18	12	9
1024	1022	129	31	20	13

Table 5.1: Number of $\mathbb{F}_2 \times \mathbb{F}_{2^k}$ multiplications for creating a masked look-up table of size N, for varying k.

$$\left((0^{2^j} \| s_j \cdot p_j) + ((1-s_j) \cdot p_j \| 0^{2^j})\right) \in \mathbb{F}_2^{2^{j+1}}.$$

Thus, given $[\![p_j]\!]$ represented as $\lceil 2^j/k \rceil$ shared elements of $\mathbb{F}_{2^k}$, we can compute $[\![p_{j+1}]\!]$ in MPC with $\lceil 2^j/k \rceil$ multiplications between $[\![s_j]\!]$ and a shared $\mathbb{F}_{2^k}$ element, plus some local additions.

Starting with $p_1(X) = s_0 \cdot X + (1 - s_0)$ we can iteratively apply the above method to compute $p_\ell = s'$, as shown in Protocol 1. The overall complexity of this protocol is given by

$$\sum_{j=1}^{\ell-1} \lceil 2^j/k \rceil < N/k + \ell$$

multiplications between bits and $\mathbb{F}_{2^k}$ elements.

Table 5.1 illustrates this trade-off between the field size and number of multiplications for some example parameters. We note that the main factor affecting the best choice of k is the cost of performing a multiplication in $\mathbb{F}_{2^k}$ in the underlying MPC protocol, and this may change as new protocols are developed. However, we compare costs of some current protocols in Section 5.7.

We now show how to use the look-up table protocol from the previous section to evaluate AES and DES in MPC. We use the more general method from Protocol 5.7, and leave an implementation of the faster variant with circuit-dependent preprocessing to future work.

5.6.4 MPC Evaluation of AES

We require an MPC protocol which performs operations in $\mathbb{F}_{2^8}$. In practice, we actually embed $\mathbb{F}_{2^8}$ in $\mathbb{F}_{2^{40}}$, since we use the SPDZ protocol which requires a field size of at least 2^κ, for statistical security parameter κ. We implement the AES S-box using the table look-up method from Figure 5.9 combined with Demux (Protocol 1) over $\mathbb{F}_{2^{40}}$, since this yields a lower communication cost (see Table 5.5). Notice that the data sent is highly dependent on the number of bits, triples and the field size.

In a naive implementation of this approach, we would have call BitDec on $[\![\mathsf{Table}(s)]\!]$, in order to perform the embedding $\mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{40}}$. This is required since the table output is not embedded, but the MixColumns step from Figure 5.1 needs the bit decomposition of the input to perform multiplication by $X \in \mathbb{F}_{2^8}$ on each state.

With a more careful analysis we can avoid the BitDec calls by locally embedding the bit shares inside Figure 5.9. We store the masked S-box table in bit decomposed form and then its bits are multiplied (in the clear) with Demux's output (secret-shared). This trick reduces the online communication by a factor of 8, halves the number of rounds required to evaluate AES and gives a very efficient online phase with only 10 rounds and 160 openings in $\mathbb{F}_{2^{40}}$.

5.6.5 MPC Evaluation of DES.

Using the fact that DES S-boxes have size 64, we chose to use the Demux given in Protocol 1 with multiplications in $\mathbb{F}_{2^{40}}$, based on the costs in Table 5.5. Like AES, we try to isolate the Input-dependent phase as much as possible with no extra cost.

Every DES round performs only bitwise addition and no embedding is necessary here. The masked table can be bit-decomposed without interaction, exactly as described above for AES, by multiplying clear bits with secret shared values. This yields a low number of openings, one per S-box look-up, so the total online cost for 3-DES is 46 rounds with 384 openings.

5.7 Performance Evaluation

This section presents timings for 3-DES and AES using the methods presented in previous sections. We also discuss trade-offs and different optimizations which turn out to be crucial for our running-times. The setup we have considered is that both the key and message used in the cipher are secret shared across two parties. We consider the input format for each block cipher as already embedded into $\mathbb{F}_{2^{40}}$ for AES, or as a list of shared bits for DES. We implemented the protocols using the SPDZ software,[2] and estimated times for computing the multiplication triples and random bits needed based on the costs of MASCOT [KOS16].

The results, shown in Tables 7.2, 5.3 and 5.4, give measurements in terms of *latency* and *throughput*. Latency indicates the online phase time required to evaluate one block cipher, whereas throughput (which we consider for both online and offline phases) shows the maximum number of blocks per second which can be evaluated in parallel during one execution. We also measure the number of rounds of interaction of the protocols, and the number of *openings*, which is the total number of secret-shared field elements opened during the online evaluation.

Benchmarking Environment. The experiments were ran across two machines each with Intel i7-4790 CPUs running at 3.60GHz, 16GB of RAM connected over a 1Gbps LAN with an average ping of 0.3ms (round-trip). The WAN experiments were simulated using the Linux `tc` tool with an average ping latency of 100ms (round-trip) and a bandwidth of 50Mbps.

For experiments with 3–5 parties, we used three additional machines with i7-3770 CPUs at 3.1GHz. In order to get accurate timings each experiment was averaged over 5 executions, each with at least 1000 cipher calls.

[2] https://github.com/bristolcrypto/SPDZ-2

Security Parameters and Field Sizes. Secret-sharing based MPC can be usually split into 2 phases — preprocessing and online. In SPDZ-like systems, the preprocessing phase depends on a computational security parameter, and the online phase a statistical security parameter which depends on the field size. In our experiments the computational security parameter is $\lambda = 128$. The statistical security κ is 40 for every cipher except for `3DES-Raw` which requires an embedding into a 42 bit field.

Results. The theoretical costs and practical results are shown in Table 7.2 and Table 5.3, respectively. Timings are taken only for the encryption calls, excluding the key schedule mechanism. `AES-BD` is AES implemented by embedding each block into $\mathbb{F}_{2^{40}}$, and then squaring the shares locally after the inputs are bit-decomposed. In this manner, each S-box computation costs 5 communication rounds and 6 multiplications. This method was described in [DKL+12].

Surprisingly, `AES-RP` (the polynomial-based method from Section 5.4.2) has a better throughput than `AES-BD` although it requires 20 more rounds and 2 times more shared bits to evaluate. The explanation for this is that in `AES-RP` there are fewer openings, thus less data sent between parties. However, for the WAN experiments in `AES-RP` the latency increases dramatically because of the extra rounds and the round-trip time.

`3DES-Raw` represents the 3-DES cipher with the S-box evaluated as a polynomial of degree 62 over the field $\mathbb{F}_{2^6} = \mathbb{F}_2[x]/(x^6+x^4+x^3+x+1)$. To make the comparisons relevant with other ciphers in terms of active security we chose to embed the S-box Input in $\mathbb{F}_{2^{42}}$, via the embedding $\mathbb{F}_{2^6} \hookrightarrow \mathbb{F}_{2^{42}}$, where $\mathbb{F}_{2^{42}} = \mathbb{F}_2[y]/(y^{42} + y^{21} + 1)$ and $y = x^7 + 1$. The S-boxes used for interpolating are taken from the PyCrypto library [Lit19]. `3DES-Raw` is implemented only for benchmarking purposes and it has no added optimizations. One S-box has a cost of 62 multiplications and 62 rounds.

`3DES-PV` is 3-DES implemented with the Pulkus-Vivek method from Section 5.5.2. Since it has only a few multiplications in $\mathbb{F}_{2^{40}}$, the amount of preprocessing data required is very small, close to `AES-BD`. It suffers in terms of both latency and throughput due to the high number of communication rounds (needed for bit decomposition to perform the squarings).

`AES-LT` and `3DES-LT` are the ciphers obtained with the lookup table protocol from Section 5.6. `AES-LT` achieves the lowest latency and the highest throughput in the online phase for both LAN and WAN settings. The communication in the preprocessing phase is roughly twice the cost of the previous method, `AES-BD`.

Packing optimization. We notice that in the online phase of `AES-LT` each opening requires to send 8 bit values embedded in $\mathbb{F}_{2^{40}}$. Instead of sending 40 bits across the network we select only the relevant bits, which for `AES-LT` are 8 bits. This reduces the communication by a factor of 5 and gives a throughput of 236k AES/second over LAN and a multi-threaded MPC engine.

The same packing technique is applied for `3DES-LT` since during the protocol we only open 6 bit values from Protocol 5.7. These bits are packed into a byte and sent to the other party. Here the multi-threaded version of `3DES-LT` improves the throughput only by a factor of 4.2x (vs. `AES-LT` 4.5x) due to the higher number of rounds and openings along with the loss of 2 bits from packing.

Computation vs. Communication. Notice that for AES the throughput in the WAN setting compared

Cipher	Online cost			Preprocessing cost			
	Rounds	Openings	Field	Triples	Bits	Field	Comm.(MB)
AES-BD	50	2240	$\mathbb{F}_{2^{40}}$	960	2560	$\mathbb{F}_{2^{40}}$	4.3
AES-RP	70	1920	$\mathbb{F}_{2^{40}}$	640	5120	$\mathbb{F}_{2^{40}}$	2.9
AES-LT	10	160	$\mathbb{F}_{2^{40}}$	1760	42240	$\mathbb{F}_{2^{40}}$	8.4
3DES-Raw	2979	48024	$\mathbb{F}_{2^{42}}$	23808	2688	$\mathbb{F}_{2^{42}}$	112
3DES-PV	230	3456	$\mathbb{F}_{2^{40}}$	1152	9216	$\mathbb{F}_{2^{40}}$	5.2
3DES-LT	46	384	$\mathbb{F}_{2^{40}}$	1920	26880	$\mathbb{F}_{2^{40}}$	8.8

Table 5.2: Communication cost for AES and 3-DES in MPC.

Cipher	Online (single-thread)			Online (multi-thread)			Preprocessing[a]
	Latency (ms)	Batch size	ops/s	Batch size	ops/s	Threads	ops/s
AES-BD	5.20	64	758	1024	3164	16	30.7
AES-RP	7.19	1024	940	64	3872	16	**46.1**
AES-LT	**0.928**	1024	51654	512	**236191**	32	16.79
3DES-Raw	270	512	130	-	-	-	1.24
3DES-PV	36.98	512	86	512	366	32	**25.6**
3DES-LT	**4.254**	1024	10883	512	**45869**	16	15.3

Table 5.3: 1 Gbps LAN timings for evaluating AES and 3-DES in MPC.

to LAN decreases by at least 8 times. Surprisingly, the throughput of 3DES decreases by at most four - single threaded 3DES-LT can perform around 10000 ops/s over LAN whereas over WAN it has 300 ops/s (with the same ratio for the multi-threaded variant). Profiling suggests that AES has a lower computation cost than 3DES. This means that increasing the round-trip time between machines has a slightly worse effect for AES than 3DES since the CPU can do more work between subsequent rounds.

General costs of the table look-up protocol. In Table 5.5, we estimate the communication cost for creating preprocessed, masked tables for a range of table sizes, using our protocol from Section 5.6.2. This requires multiplication triples over $\mathbb{F}_{2^k}$, where k is a parameter of the protocol. When $k = 1$, we give figures using a recent optimized variant [WRK17b] of the two-party TinyOT protocol [NNOB12]. For larger choices of k, the costs are based on the MASCOT protocol [KOS16]. We note that even though MASCOT has a communication complexity in $O(k^2)$, it still gives the lowest costs (with $k = 40$) for all the table sizes we considered.

Cipher	Online (single-thread)			Online (multi-thread)			Preprocessing
	Latency (ms)	Batch size	ops/s	Batch size	ops/s	Threads	ops/s
AES-BD	2550	4096	79	2048	325	32	1.52
AES-RP	3569	4096	83	4096	346	32	**2.28**
AES-LT	**510**	4096	928	4096	**29055**	32	0.83
3DES-PV	11727	2048	35	512	185	32	**1.27**
3DES-LT	**2344**	4096	383	4096	**12165**	32	0.76

Table 5.4: 50 Mbps WAN timings for evaluating AES and 3-DES in MPC.

N	$k = 1$	40	64	128
64	35.01	21.8	43.52	112.64
128	71.16	30.52	52.22	135.17
256	143.45	47.96	69.63	157.7
512	288.02	78.48	104.45	202.75
1024	577.17	135.16	174.08	292.86

Table 5.5: Total communication cost (kBytes) of the $\mathbb{F}_2 \times \mathbb{F}_{2^k}$ multiplications needed in creating a masked lookup table of size N, with two parties. The $k = 1$ estimates are based on TinyOT [WRK17b], the others on MASCOT [KOS16].

5.7.1 Multiparty Setting

We also ran the AES-LT protocol with different numbers of parties and measured the throughput of the preprocessing and online phases. Figure 5.10 indicates that the preprocessing gets more expensive as the number of parties increases, whereas the online phase throughput does not decrease by much. This is likely to be because the bottleneck for the preprocessing is in terms of communication (which is $O(n^2)$ in total), whereas the online phase is more limited by the local computation done by each party.

5.7.2 Comparison with Other Works

We now compare the performance of our protocols with other implementations in similar settings. Table 8.3 gives an overview of the most relevant previous works. We see that our AES-LT protocol comes very close to the best online throughput of TinyTable, whilst having a far more competitive offline cost. [3] Our AES-RP variant has a slower online phase, but is comparable to the best garbled circuit protocols overall.

TinyTable Protocol. The original, 2-party TinyTable protocol [DNNR17] presented implementations

[3]The reason for the very large preprocessing cost of TinyTable is due to the need to evaluate the S-box 256 times per table lookup.

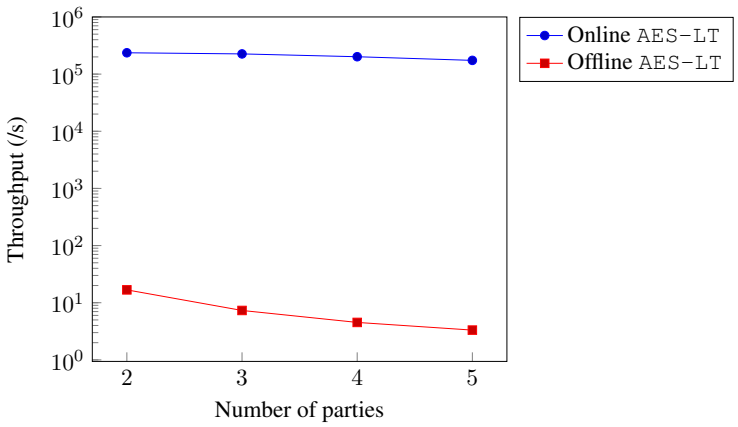

Figure 5.10: Table lookup-based AES throughput for multiple parties.

Protocol	Online		Comms.	Notes
	Latency (ms)	Throughput (/s)	(total)	
TinyTable (binary) [DNNR17]	4.18	24500	3.07 MB	
TinyTable (optim.) [DNNR17]	1.02	339000	786.4 MB	
Wang et al. [WRK17b]	0.93	1075	2.57 MB	10 Gbps
Rindal-Rosulek [RR16]	1.0	1000	1.6 MB	10 Gbps
OP-LUT [DKS+17]	5	41670	0.103 MB	passive
SP-LUT [DKS+17]	6	2208	0.044 MB	passive
AES-LT	0.93	236200	8.4 MB	
AES-RP	7.19	940	2.9 MB	

Table 5.6: Performance comparison with other 2-PC protocols for evaluating AES in a LAN setting.

of the online phase only, with two different variants. The fastest variant is based on table lookup and obtains a throughput of around 340 thousand AES blocks per second over a 1Gbps LAN, which is 1.51x faster than our online throughput. The latency (for sequential operations) is around 1ms, the same as ours. We attribute the difference in throughput to the additional local computation in our implementation, since we need to compute on MACs for every linear operation (this could be avoided if we used the protocol from Section 5.6.1.

TinyTable does not report figures for the preprocessing phase. However, we estimate that using TinyOT and the naive method suggested in the paper would need would need over 1.3 million TinyOT triples for AES (34 ANDs for each S-box, repeated 256 times to create one masked table, for 16 S-boxes in 10 rounds). In contrast, our table lookup method uses around 160 thousand TinyOT triples, or just 2080 triples over $\mathbb{F}_{2^{40}}$ (cf. Table 5.1), per AES block.

Garbled Circuits. There are many implementations of AES for actively secure 2-PC using garbled circuits [LR15, RR16, NST17, WRK17b]. When measuring online throughput in a LAN setting, using

garbled circuits gives much worse performance than methods based on table lookup, because evaluating a garbled circuit is much more expensive computationally. For example, out of all these works the lowest reported online time (even over a 10Gbps LAN) is 0.93ms [WRK17b], and this does not improve in the amortized setting.

Some recent garbled circuit implementations, however, improve upon our performance in the preprocessing phase, where communication is typically the bottleneck. Wang et al. [WRK17b] require 2.57MB of communication when 1024 circuits are being garbled at once, while Rindal and Rosulek need only 1.6MB [RR16]. The runtime for both of these preprocessing phases is around 5ms over a 10Gbps LAN; this would likely increase to at least 15–20ms in a 1Gbps network, whereas our table lookup preprocessing takes around 60ms using MASCOT. If a very fast online time is not required, our implementation of the Rivain–Prouff method would be more competitive, since this has a total amortized time of only 23ms per AES block.

Secret-Sharing Based MPC. Other actively implementations of AES/DES using secret-sharing and dishonest majority based on secret sharing include those using SPDZ [DKL+12, KSS13a] and MiniMAC [DZ13, DLT14]. Our `AES-BD` method is the same as [DKL+12] and obtains faster performance than both SPDZ implementations. For DES, our TinyTable approach improves upon the times of the binary circuit implementation from [KSS13a] (which are for single-DES, so must be multiplied by 3) by over 100 times. Regarding MiniMAC, the implementation of [DLT14] obtains slower online phase times than our work and TinyTable, and it is not known how to do the preprocessing with concrete efficiency.

OP-LUT and SP-LUT. The proposed 2-party protocols by Dessouky et al. [DKS+17] only offer security in the semi-honest setting. The preprocessing phase for both the protocols are based on 1-out-of-N oblivious transfer. In particular, the cost of the OP-LUT setup is essentially that of 1-out-of-N OT, while the cost of SP-LUT is the cost of 1-out-of-N *random* OT, which is much more efficient in terms of communication.

The online communication cost of OP-LUT is essentially the same as our online phase, since both protocols require each party to send $\log_2 N$ bits for a table of size N. However, we incur some additional local computation costs and a MAC check (at the end of the function evaluation) to achieve active security. The online phase of SP-LUT is less efficient, but the overall communication of this protocol is very low, only 0.055MB for a single AES evaluation over a LAN setting with 1GB network.

The work [DKS+17] reports figures for both preprocessing and online phase: using OP-LUT gives a latency of around 5ms for 1 AES block in the LAN setting, and a throughput of 42000 blocks/s. These are both slower than our online phase figures using `AES-LT`. The preprocessing runtimes of both OP-LUT and SP-LUT are much better than ours, however, achieving over 1000 blocks per second (roughly 80 times faster than `AES-LT`). This shows that we require a large overhead to obtain active security in the preprocessing, but the online phase cost is the same, or better.

Protocol 2 $[\![x']\!] \leftarrow \mathsf{Mod2m}([\![x]\!], k, \ell)$, as in [sec]

Input: k is the bit-length of the input $[\![x]\!]$
Output: Satisfies $x' = x \bmod 2^\ell$

1: $[\![b]\!] \leftarrow 2^{k-1} + [\![x]\!]$
2: **for** $i = 0$ to $\ell - 1$ **do**
3: $\quad [\![r_i]\!] = \mathcal{F}_{\mathsf{Prep}}.\mathsf{RandomBit}()$
4: $[\![r']\!]_B \leftarrow (r_{\ell-1}, \ldots, r_0)$
5: $[\![r']\!] = \sum_{i=0}^{\ell-1} 2^i \cdot [\![r_i]\!]$
6: **for** $i = 0$ to $k + \mathsf{sec} - \ell$ **do**
7: $\quad [\![r''_i]\!] = \mathcal{F}_{\mathsf{Prep}}.\mathsf{RandomBit}()$
8: $[\![r'']\!] \leftarrow \sum_{i=1}^{k+\mathsf{sec}-\ell} 2^i \cdot [\![r''_i]\!]$
9: $[\![r]\!] \leftarrow 2^\ell \cdot [\![r'']\!] + [\![r']\!]$
10: $c \leftarrow \mathsf{Open}([\![b]\!] + [\![r]\!])$
11: $c' \leftarrow c \bmod 2^\ell$
12: $[\![u]\!] \leftarrow \mathsf{BitLT}(c', [\![r']\!]_B)$ // BitLT takes $\log \ell$ rounds and $2l - 2$ openings
13: $[\![x']\!] \leftarrow c' - [\![r']\!] + [\![u]\!] \cdot 2^\ell$
14: **return** $[\![x']\!]$

Protocol 3 $x' \leftarrow \mathsf{Mod2m}([\![x + s]\!], \ell)$

Input: x, s are l bit integers from the look-up table protocol
Output: Satisfies $x' = (x + s) \bmod 2^\ell$

1: **for** $i = 0$ to $\mathsf{sec} + \ell$ **do**
2: $\quad [\![r_i]\!] \leftarrow \mathcal{F}_{\mathsf{Prep}}.\mathsf{RandomBit}()$
3: $c \leftarrow \mathsf{Open}([\![x + s]\!]) + 2^\ell \cdot \sum_{i=0}^{\mathsf{sec}+\ell} [\![r_i]\!]$
4: **return** $c \bmod 2^\ell$

5.8 Extension to $\mathbb{F}_p$

We now give a new constant round protocol (independent of the table size) to evaluate the online phase of a look-up table $[\![\mathsf{T}]\!]$ of size 2^ℓ where each entry of $\mathsf{T}(i) \in \mathbb{F}_p, \forall i \in [2^\ell]$. Recall again that our goal is to obtain secret shares of

$$[\![\mathsf{Table}(s)]\!] = ([\![\mathsf{T}(s)]\!], [\![\mathsf{T}((s+1) \bmod 2^\ell]\!], \ldots, [\![\mathsf{T}((s + (2^\ell - 1) \bmod 2^\ell)]\!]).$$

To evaluate such a look-up table on input $[\![x]\!]$ in the online phase we need to retrieve $[\![s]\!]$ and call $\mathsf{Open}([\![x]\!] + [\![s]\!])$. In the Boolean case, the reduction modulo 2 happens automatically and the revealed sum looks random to an adversary as s was sampled at random. When the arithmetic shares are in $\mathbb{F}_p$ we need to work harder since opening $x + s \in \mathbb{F}_p$ can reveal some information about the input x. The straightforward solution is to do the reduction $\bmod 2^\ell$ in MPC, as long as $s \xleftarrow{\$} \{0,1\}^\ell$ then $(x + s) \bmod 2^\ell$ is also random. Computing $[\![x + s]\!] \bmod 2^\ell$ is relatively costly though. This can be done using techniques introduced by Catrina and de Hoogh [Cd10a] and [sec] illustrated in Figure 2.

Our improvement comes from noticing that for the look-up table protocol we need the public output of $[\![x + s]\!] \bmod 2^\ell$. Hence we can devise something simpler described in Figure 3 and get the output

in the public by first masking $x + s$ with enough random bits, open the result and then do the operation modulo 2^ℓ in clear. This simple protocol reduces the cost from $\ell + \mathsf{sec}$ random bits, ℓ triples, $\ell + 2$ communication rounds, $2 \cdot \ell$ openings to just one opening and sampling $\ell + \mathsf{sec} + 1$ random bits.

Chapter 6

PRFs for fields of characteristics p

This chapter is based on joint work with Lorenzo Grassi and Christian Rechberger and Peter Scholl and Nigel P. Smart. [GRR+16] which was presented at CCS 2016.

6.1 Contributions

In this chapter we focus on designing and evaluating efficient PseudoRandom Functions (PRFs) for arithmetic circuits modulo a prime p using the SPDZ protocol. The use of PRFs in MPC has broad implications when dealing with encrypted databases where the keys are unknown and is detailed below in the next Section. The main contributions of this work is to investigate several low-complexity blockciphers such as LowMC, MiMC and to design efficient protocols for PRFs such as Naor-Reingold and a less-known one based on the Legendre symbol. In the case of Legendre PRF we give the first constant-round protocol which can be evaluated in any secret shared based MPC system, including SPDZ.

6.2 Overview

Before proceeding with the preliminaries, we first outline some applications we have in mind. Our focus is on secret sharing based MPC systems such as that typified by BDOZ [BDOZ11], SPDZ [DPSZ12, DKL+13], and VIFF [DGKN09]; or indeed any classical protocol based on Shamir secret sharing. In such situations data is often shared as elements of a finite field $\mathbb{F}_p$, of large prime characteristic. Using such a representation one then has efficient protocols to compute relatively complex functions such as integer comparison [DFK+06], fixed point arithmetic [CS10], and linear programming [Cd10b]. Indeed the most famous of such efficient high level protocols is that needed to compute the output of an auction [BCD+09].

Given such applications, evaluated by an MPC "engine", the question arises as to how to get data securely in and out of the engine. In traditional presentations the data is entered by the computing parties, and the output is delivered to the computing parties. However, this in practice will be a sim-

plification. Input and output may need to be securely delivered/received by third parties, in addition in a long term reactive functionality the intermediate secure data may need to be stored in a database, or other storage device.

If we examine the case of long term storage of data, which is stored by the MPC engine only to be used again at a later date, the trivial way to store such shared data is for each party to encrypt their share with a symmetric key, and then store each encrypted share. However, this incurs an N-fold increase in storage at the database end (for N MPC servers), which may be prohibitive. A similar trivial solution also applies for data input and output, except data input is now performed using N public keys (one for each MPC server) and output is performed by each server producing a public key encryption of its share to the recipient's public key.

A more efficient solution would be to use a direct evaluation of a symmetric key primitive within the MPC engine. Such a symmetric key primitive should be able to be efficiently evaluated by the MPC engine[1]. We call such a symmetric key primitive "MPC-Friendly". Given almost all symmetric key primitives can be constructed easily from Pseudo-Random Functions (PRFs), the goal is therefore to produce an MPC-Friendly PRF.

The main problem of using "traditional" PRFs such as AES is that these are built for computational engines which work over data types that do not easily match the operations possible in the MPC engine. For example AES is very much a byte/word oriented cipher, which is hard to represent using arithmetic in $\mathbb{F}_p$. Thus we are led to a whole new area of PRF design, with very different efficiency metrics compared to traditional PRF design.

6.2.1 Related Work

At the time of writing the paper [GRR+16] there was little direct work on this problem, despite the recent plethora of proposed MPC applications; indeed the only paper we knew of which explicitly designs PRFs for use in MPC, was [ARS+15], which we shall discuss below. The three lines of work most related to the work in this thesis, apart from re-purposing designs from elsewhere, are

- Low complexity, "lightweight" ciphers for use in IoT and other constrained environments.
- Block and stream ciphers suited to evaluation by a Fully Homomorphic or Somewhat Homomorphic Encryption scheme, i.e., SHE-Friendly ciphers.
- Designs for use in SNARKs.

We now elaborate on the prior work in these areas.

Low Complexity Lightweight Ciphers: Block ciphers often iterate a relatively simple round function a number of times to achive security goals. Most early designs in this domain focused on small area when implemented as a circuit in hardware. There, large depth (via a large number of rounds) is

[1]Note that public key encryption applications as mentioned above can be built from the symmetric key key primitives in the standard KEM-DEM manner. The KEM component being relatively easy to implement, in most cases, in an MPC friendly manner. Thus we focus on symmetric key primitives in this thesis.

of no concern, since it simply means repeating a circuit that implements a single round more times. Notable exceptions are mCrypton [LK06] and Noekeon [DPVAR00] which also feature a relatively low depth. The more recent trend to emphasize low latency (with designs like PRINCE [BCG+12]) fits much better with our requirement of having low-depth. A property of all these designs is that they lend themselves well to implementations where binary NAND gates, XOR gates, or multiplexers are the basic building blocks in the used libraries. As explained above the majority of secret sharing based MPC applications require description via $\mathbb{F}_p$. Whilst bit operations are possible over $\mathbb{F}_p$ using standard tricks (which alas turn XOR into a non-linear operation), applying such ciphers would require the $\mathbb{F}_p$ data types to be split into a shared bit representation over $\mathbb{F}_p$ to apply the cipher. Such a conversion is expensive.

SHE-Friendly Ciphers: Perhaps due to the recent theoretical interest in SHE/FHE schemes, this area has had more attention than the more practical issues addressed in this thesis. The motivating scenario for a SHE-Friendly cipher is to enable data to be securely passed to a cloud environment, using a standard encryption scheme, which the cloud server then homomorphically decrypts to obtain a homomorphic encryption of the original data.

This line of work has resulted in a handful of designs. A block cipher called LowMC [ARS+15], a stream cipher called Kreyvium [CCF+16] (based on the Trivium stream cipher) and FLIP [MJSC16] (based on a filter permutation, although recently cryptanalysed in [DLR16]). The block cipher LowMC is designed for both MPC and FHE implementation, but actually does not meet the MPC design goals we have set. It does indeed have low depth, but it is a cipher based on operations in characteristic two. The two SHE friendly stream cipher designs of Kreyvium and FLIP also suffer from the same problem as the lightweight designs describe above, since they are also bit-oriented.

SNARK-friendly Constructions: Being SNARK-friendly means that the number of constraints is low. This generally favours larger data types like $\mathbb{F}_p$ or $\mathbb{F}_{2^n}$, and the depth of the circuit is of no concern. MiMC [AGR+16] was originally designed for this use case and seems to be the only one in this area. As the depth is not too high either, we choose it for detailed evaluation.

6.2.2 Recent related work

At the time of writing this thesis, there have been many improvements to evaluate lightweight PRFs in MPC. Two years after, the work of Agrawal et al. [AMMR18] at CCS'18 constructed a more efficient evaluation for the Naor-Reingold PRF in the distributed setting, i.e., they avoid generic secret sharing techniques to execute a two-round protocol with applications to distributed key management and enterprise network authentication. Next, Albrecht et al. [AGP+19] showed how to use Feistel constructions using MiMC to reduce the preprocessing cost when evaluating a blockcipher using generic MPC.

6.3 Preliminaries

The goal of this work is to investigate the efficient evaluation of PRFs in a secret-sharing based MPC setting. We leave the construction of the various higher level primitives (SSE, ORE, AE etc.) to future work, although many of these can easily be constructed directly from a PRF.

To fix notation we will consider a PRF of the following form

$$F : \begin{cases} (\mathbb{F}_p)^\ell \times (\mathbb{F}_q)^n & \longrightarrow & (\mathbb{F}_r)^m \\ (k_1, \ldots, k_\ell, x_1, \ldots, x_n) & \longmapsto & F_k(x_1, \ldots, x_n). \end{cases}$$

The various finite fields $\mathbb{F}_p$, $\mathbb{F}_q$ and $\mathbb{F}_r$ may be distinct. Our MPC engine is assumed to work over the finite field $\mathbb{F}_p$, as we always assume the key to the PRF will be a secret shared value. As a benchmark, we compare all of our candidates to the baseline AES example used in prior work, and to implementations of the given PRFs on clear (public) data.

Depending on the precise application, there are several distinct design criteria which we may want to consider. Thus, there will not be a one size fits all PRF which works in all applications. We then have various potential cases:

- In some applications the input is public and we need to embed the public elements $x_1, \ldots, x_n \in \mathbb{F}_q$ into $\mathbb{F}_p$. However, the more general case is when the input is secret shared itself, and we have $\mathbb{F}_q = \mathbb{F}_p$.
- In some applications the output of the PRF will be public, and thus $\mathbb{F}_r$ can be any field. In other applications we also want the output to be secret shared, so we can use it in some other processing such as a mode of operation. In this latter case we will have $\mathbb{F}_r = F_p$. In addition, some applications, such as when using the (leaky) ORE scheme presented in [CLWW16] require PRF outputs in $\{0, 1, 2\}$, and we may (or may not) require these to be secret shared (and hence embedded in $\mathbb{F}_p$).
- In some applications we would like a PRF which is just efficient in the MPC engine, and we do not care whether the equivalent standard PRF is efficient or not. In other applications we also require that the standard PRF is also efficient. For example when an external third party is encrypting data for the MPC engine to decrypt.

In this thesis we consider four candidate PRFs for use in MPC systems, as well as the comparison case of AES. Two of these are number theoretic in nature (the Naor-Reingold PRF, based on DDH, and a PRF based on the Legendre symbol), whilst MiMC [AGR+16] and LowMC [ARS+15] are more akin to traditional symmetric block cipher constructions.

AES: Since AES does not lend itself well to secure computation over prime fields, we use this purely as a benchmark. We assume an MPC system which is defined over the finite field $\mathbb{F}_{2^8}$, allowing for efficient evaluation of the S-box [DK10, DKL+12]. We have

$$F_{\text{AES}} : (\mathbb{F}_{2^8})^{16} \times (\mathbb{F}_{2^8})^{16} \to (\mathbb{F}_{2^8})^{16}.$$

LowMC: This is a block cipher candidate [ARS+15] designed to be suitable for FHE and MPC style applications; thus it has a low multiplicative depth and a low number of multiplications. It operates over $\mathbb{F}_2$, so like AES, is not well-suited to the MPC applications for which we envisage our block ciphers being used for. Thus we only consider LowMC as an additional base line comparison (along with AES) for our ciphers. LowMC has block size n bits and key size k bits (we use $n = 256$ and $k = 128$ since it has a lower rate of ANDs per output bit), giving

$$F_{\mathsf{LowMC}} : (\mathbb{F}_2)^k \times (\mathbb{F}_2)^n \to (\mathbb{F}_2)^n.$$

Naor-Reingold: Let $\mathbb{G} = \langle g \rangle$ be an elliptic curve group of prime order p in which DDH is hard, and $\mathsf{encode}(\cdot)$ be a hash function that maps elements of $\mathbb{G}$ into elements of $\mathbb{F}_p$. The Naor-Reingold PRF takes a uniform secret-shared key in $\mathbb{F}_p^{n+1}$, a message in $\mathbb{F}_2^n$ (secret-shared over $\mathbb{F}_p$), and outputs a *public* $\mathbb{F}_p$ element as follows:

$$\begin{aligned} F_{\mathsf{NR}(n)} : (\mathbb{F}_p)^{n+1} \times (\mathbb{F}_2)^n &\to \mathbb{F}_p \\ (\mathbf{k}, \mathbf{x}) &\mapsto \mathsf{encode}(g^{k_0 \cdot \prod_{i=1}^{n} k_i^{x_i}}). \end{aligned}$$

To evaluate F_{NR} in MPC naively would require computing exponentiations (or Elliptic-Curve scalar multiplications) on secret exponents, which is very expensive. However, if the PRF output is public, we show how the exponentiation (and hence PRF evaluation) can be done very efficiently, with active security, using any MPC protocol based on secret sharing.

Legendre Symbol: We also consider an unusual PRF based on the pseudorandomness of the Legendre symbol. This is a relatively old idea, going back to a paper of Damgård in 1988 [Dam90], but has not been studied much by the cryptographic community. The basic version of the PRF is defined as,

$$\begin{aligned} F_{\mathsf{Leg(bit)}} : \mathbb{F}_p \times \mathbb{F}_p &\to \mathbb{F}_2 \\ (k, x) &\mapsto L_p(x + k) \end{aligned}$$

where $L_p(a)$ computes the usual Legendre symbol $\left(\frac{a}{p}\right) \in \{-1, 0, 1\}$ and maps this into $\{0, 1, (p + 1)/2\}$, by computing

$$L_p(a) = \frac{1}{2}\left(\left(\frac{a}{p}\right) + 1\right) \pmod{p}.$$

The output is embedded into $\mathbb{F}_p$, giving a secret-shared output in $\mathbb{F}_p$. If needed, the range can easily be extended to the whole of $\mathbb{F}_p$ by using a key with multiple field elements and performing several evaluations in parallel. This gives a PRF

$$F_{\mathsf{Leg}(n)} : (\mathbb{F}_p)^{((n+1)\cdot \ell)} \times (\mathbb{F}_p)^n \to \mathbb{F}_p,$$

for some value $\ell = O(\log_2 p)$ chosen large enough to ensure a sufficient statistical distance from uniform of the output. This PRF takes n finite field elements as input and produces an element in $\mathbb{F}_p$ as output, where n is some fixed (and relatively small) number, say one or two.

PRF	$\log_2 p$	Output (type)	Online cost		Assumption
			Mult.	Rounds	
$F_{\mathtt{AES}}$	8	shared	960	50	–
F_{LowMC}	2	shared	1911	13	–
$F_{\mathsf{NR}}(n)$	256	public	$2 \cdot n$	$3+\log(n+1)$	EC-DDH
$F_{\mathsf{Leg}}(\mathsf{bit})$	128	shared	2	3	DSLS
$F_{\mathsf{Leg}}(n)$	128	shared	$256 \cdot n$	3	DSLS
F_{MiMC}	128	shared	146	73	–

Table 6.1: Overview of the cost of evaluating the PRFs in MPC.

Perhaps surprisingly, we show that the Legendre PRF can be evaluated *very efficiently* in MPC, at the cost of just two multiplications in three rounds of interaction for $F_{\mathsf{Leg}(\mathsf{bit})}$. To the best of our knowledge, this is the only PRF that can be evaluated in a constant number of rounds on secret-shared data, using any arithmetic MPC protocol. Since the underlying hard problem is less well-studied than, say, DDH or factoring, we also provide a brief survey of some known attacks, which are essentially no better than brute force of the key.

MiMC: This is a very recent class of designs whose primary application domain are SNARKs [AGR+16]. In addition to a cryptographic hash function, the design also includes a block cipher which is also usable as a PRF, with up to birthday bound security. The input, output and keys are all defined over $\mathbb{F}_p$, so we get

$$F_{\mathsf{MiMC}} : \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p.$$

The core of the round function is the simple map $x \mapsto x^3$ over $\mathbb{F}_p$. The number of rounds is quite high (for a 128-bit prime p 82 for full security, 73 for PRF security), but in terms of $\mathbb{F}_p$ multiplications the performance turns out to be competitive.

The reason for selecting MiMC as a "standard" block cipher is that firstly it works over a finite prime field of large characteristic, which is a common requirement for applications of secret-sharing based MPC that perform arithmetic on integers or fixed-point data types. Secondly, the depth of the computation is not too large, being 146. Thirdly, the number of non-linear operations is also 146, this means that the offline preprocessing needed (to produce multiplication triples) will be very small compared to other constructions.

In Table 6.1 we present an overview of the MPC-friendly PRFs we consider. The table shows the number of secure multiplication needed to execute the online evaluation of the function on shared inputs (since in secret-sharing based MPC, additions are free) as well as the number of rounds of communication.

Length Extension for $F_{\mathsf{Leg}(1)}$

1. $c_0 \leftarrow n$.
2. For $i = 1, \ldots, n$ do
   a) $c_i \leftarrow x_i + F_{\mathsf{Leg}(1)}(k, c_{i-1})$.
3. $a \leftarrow F_{\mathsf{Leg}(1)}(k, c_n)$.
4. Return a.

Figure 6.1: Using CBC Mode With $F_{\mathsf{Leg}(1)}$.

Length Extension for $F_{\mathsf{Leg}(2)}$

1. $c_0 \leftarrow n$.
2. For $i = 1, \ldots, n$ do
   a) $c_i \leftarrow F_{\mathsf{Leg}(2)}(k, c_{i-1}, x_i)$.
3. Return c_n.

Figure 6.2: Using Merkle-Damgård With $F_{\mathsf{Leg}(2)}$.

6.3.1 Length Extension

We end this overview by noting that F_{MiMC} and $F_{\mathsf{Leg}(n)}$ can be extended to cope with arbitrary length inputs in the standard way; either by using a CBC-MAC style construction or a Merkle–Damgård style construction. For example, to extend $F_{\mathsf{Leg}(1)}$and F_{MiMC}, so that they can be applied to an input $x_1, \ldots, x_n \in \mathbb{F}_p$ we can use CBC mode as in Figure 6.1. Whereas, to extend $F_{\mathsf{Leg}(2)}$ we can apply Merkle–Damgård as in Figure 6.2. These two extension techniques are often more efficient than using an arbitrary length PRF as a base building block. The next chapter will deal with adding modes of operation on top of the PRFs to manipulate arbitrary length data in encrypted form using MPC.

6.3.2 Multi-Party Computation Model

We use the same functionality as in Figure 3.1 initialized over a prime field $\mathbb{F}_p$. We recall that additions (and linear operations) are local operations so essentially for free. A multiplication uses a preprocessed multiplication triple and requires sending two field elements in the online phase, with one round of interaction. Squaring can be done using a square pair and sending just one field element, again in one round.

The preprocessing can be implemented using Somewhat Homomorphic Encryption (SHE) (as in the original SPDZ protocols, or protocol of Keller et al. [KPR18]) or Oblivious Transfer (OT), using MASCOT protocol. [KOS16]. We present runtimes using the OT-based offline phase only, as at the time of writing the paper this was the faster than their SHE counterpart. After Keller et al. [KPR18]

improvements of SHE based preprocessing, one can easily take the preprocessing runtimes given in this chapter and divide them by a factor of 6.

6.3.3 MPC Evaluation of AES and LowMC

As a means of comparison for the other PRFs we use as a base line a two party implementation of AES using a SPDZ engine over the finite field $\mathbb{F}_{2^8}$, embedded into $\mathbb{F}_{2^{40}}$, as in [DKL+12]. Note that recently, much lower latencies have been obtained by evaluating AES using secure table look-up [DNNR17, KOR+17]. As the benchmark setup is the same with the previous chapter we will use the `AES-BD` numbers and avoid the lookup table protocols due to their slightly more expensive preprocessing phase. One should also bear in mind that this is only the time needed to evaluate the PRF. In a given application, which is likely to be over a different finite field, the MPC engine will also need to convert data between the two fields $\mathbb{F}_p$ and $\mathbb{F}_{2^{40}}$. This is likely to incur a more significant cost than the evaluation of the PRF itself

In addition to AES, we also present comparison executions for the low complexity block cipher LowMC. This is to enable a comparison with our $\mathbb{F}_p$ based block ciphers against not only a standard in-use block cipher (AES), but also a block cipher designed for use in MPC/FHE environments.

6.3.3.1 F_{LowMC} Definition

LowMC [ARS+15] is a flexible family of block ciphers with operations over $\mathbb{F}_2$, designed to have a low number of multiplications and a low multiplicative depth when implemented in MPC. Similar to AES, it is based on an SPN structure where the block size n, the key size k, the number of S-boxes m in the substitution layer and the allowed data complexity d of attacks can independently be chosen. The number of rounds r needed to reach the security claims is then derived from these parameters. The two most relevant parts of the round transformation are the SBOXLAYER and the LINEARLAYER. SBOXLAYER is an m-fold parallel application of the same 3-bit S-box (of multiplicative depth 1) on the first $3m$ bits of the state. If $n > 3m$ then for the remaining $n - 3m$ bits, the SBOXLAYER is the identity. LINEARLAYER is the multiplication in $\mathbb{F}_2$ of the state with a predetermined dense randomly chosen invertible binary $n \times n$ matrix that is different for every round.

Using the most recent v2 [ARS+15] formula for r, we need at least 13 rounds to achieve a security comparable to AES as a PRF, i.e. $k = 128$ and $d = 64$. Using $n = 256$, the minimal number S-boxes m for which this is true turns out to be 49.

6.3.3.2 Computing F_{LowMC} in MPC

To evaluate LowMC in MPC, we consider two approaches. In the first method, denoted $F_{\mathsf{LowMC}}(\mathsf{vector})$, we work over $\mathbb{F}_{2^{128}}$ and compute the matrix multiplications and XOR operations by parallelizing over 128-bit vectors. Specifically, each column M_i of the $n \times n$ matrix M is packed into $\mathbb{F}_{2^{128}}$ elements; to compute the product $M[\mathbf{x}]$ we take the inner product of all columns with $[\mathbf{x}]$. For $n = 256$, this

requires 512 XORs and 512 local finite field multiplications. However, we then need to switch back to $\mathbb{F}_2$ to evaluate the S-box (with three $\mathbb{F}_2$ multiplications), which requires bit decomposition, adding one round of interaction for every round of the cipher.

In the second approach, denoted $F_{\mathsf{LowMC}}(\mathsf{M4R})$, we use the "Method of Four Russians" [ABH10] to perform each matrix multiplication in $O(n^2/\log n)$ bit operations. We do not parallelize the computation by packing bits into vectors, so this actually results in a higher computation cost than the vector method, but avoids the need for bit decomposition in each round.

In both methods, the total number of multiplications over $\mathbb{F}_2$ is $3 \cdot m \cdot r$. The vector approach requires $256 \cdot r$ additional random bits, and also $2r$ rounds of communication, instead of r rounds for M4R.

6.3.3.3 Performance

With parameters $n = 256, m = 49, r = 13$, we obtained a latency of 4ms and a throughput of almost 600 blocks per second.

As for AES, the need to convert from a $\mathbb{F}_p$ representation to a bit-oriented representation for application of LowMC is likely to dominate the run-time for the actual PRF evaluation, making LowMC unsuitable for the applications we discussed at the beginning.

6.4 Naor–Reingold PRF

In this section we describe the Naor-Reingold PRF, originally presented in [NR97]. We then go on to describe how it can be efficiently implemented in a secret sharing based MPC system.

6.4.1 F_{NR} Definition

Let $\mathbb{G} = \langle g \rangle$ be a multiplicatively written group of prime order p in which DDH is hard, and $\mathsf{encode}(\cdot)$ be a hash function that maps group elements into elements of $\mathbb{F}_p$. For a message $\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$, the Naor-Reingold PRF [NR97] is defined by

$$F_{\mathsf{NR}(n)}(\mathbf{k}, \mathbf{x}) = \mathsf{encode}(g^{k_0 \cdot \prod_{i=1}^{n} k_i^{x_i}})$$

where $\mathbf{k} = (k_0, \ldots, k_n) \in \mathbb{F}_p^{n+1}$ is the key.

In practice, we choose $\mathbb{G}$ to be a 256-bit elliptic curve group over the NIST curve P-256, so require an MPC protocol for $\mathbb{F}_p$ with a 256-bit prime p.

6.4.2 Public Output Exponentiation Protocol

The main ingredient of our method to evaluate F_{NR} in MPC, when the key and message are secret-shared over $\mathbb{F}_p$, is an efficient protocol for publicly computing g^s, for some secret value $s \in \mathbb{F}_p$. The protocol, shown in Figure 6.3, uses any arithmetic MPC protocol based on linear secret sharing over $\mathbb{F}_p$.

Protocol $\Pi_{\mathsf{Exp}}([s])$

1. The parties call $\mathcal{F}_{\mathsf{ABB}}$ with command (**Share**, $[s]$), so that each party P_i obtains an additive share $s_i \in \mathbb{F}_p$
2. Each party P_i broadcasts $y_i = g^{s_i}$
3. Compute $y = \prod_i y_i$
4. Take a random shared $[r]$, and compute $[t] = [r] \cdot [s]$
5. Call $\mathcal{F}_{\mathsf{ABB}}$ with (**Share**, $[t]$) so that each P_i obtains t_i. Broadcast $z_i = g^{t_i}$
6. Open $[r]$ and check that $\prod_i z_i = y^r$
7. Output y

Figure 6.3: Securely computing a public exponentiation.

Functionality $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$

Let $G = \langle g \rangle$ be a group of prime order p. This functionality has all of the features of $\mathcal{F}_{\mathsf{ABB}}$ (running in $\mathbb{F}_p$), plus the following command:

PubExp: On receiving $(\mathtt{exp}, [s])$ from all parties, where s is stored in memory, retrieve s, then send $y = g^s$ to the adversary and wait for a response. If the adversary responds with **Deliver** then send y to all parties. Otherwise output $\perp$ to all parties.

Figure 6.4: Ideal functionality for public exponentiation.

This is modeled for the case of additive secret sharing by the **Share** command of the $\mathcal{F}_{\mathsf{ABB}}$ functionality, which produces random shares of secret values.

Given additive shares $s_i \in \mathbb{F}_p$, each party P_i first broadcasts g^{s_i}, so the result $y = \prod g^{s_i}$ can be computed. To obtain active security, we must ensure that each party used the correct value of s_i. We do this by computing an additional public exponentiation of g^t, where $t = r \cdot s$ for some random, secret value r. This serves as a one-time MAC on s, which can then be verified by opening r and checking that $g^t = y^r$. If an adversary cheats then passing the check essentially requires guessing the value of r, so occurs only probability $1/p$.

Note that the functionality $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$ (Figure 6.4) models an unfair computation, whereby the adversary first learns the output, and can then decide whether to give this to the honest parties or not. This is because in the protocol, they can always simply stop sending messages and abort after learning y.

Theorem 15. *The protocol* Π_{Exp} *securely computes the functionality* $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$ *in the* $\mathcal{F}_{\mathsf{ABB}}$*-hybrid model.*

Proof. We construct a simulator $\mathcal{S}$, which interacts with any adversary Adv (who controls the corrupt parties $\{P_i : i \in A\}$) and the ideal functionality $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$, such that no environment can distinguish between an interaction with $\mathcal{S}$ and a real execution of the protocol Π_{Exp}.

- In the first round $\mathcal{S}$ receives s_i for $i \in A$, as the corrupt parties' inputs to the $\mathcal{F}_{\mathsf{ABB\text{-}Share}}$ command. $\mathcal{S}$ calls $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$ with $(\mathtt{exp}, [s])$ and receives $y = g^s$. Then $\mathcal{S}$ samples $s_i \xleftarrow{\$} \mathbb{F}_p$ and sets $y_i = g^{s_i}$ for all $i \notin A$. $\mathcal{S}$ modifies one honest party's share y_i to $g^s \prod_{j \neq i} y_j^{-1}$, then sends y_i for all $i \notin A$ to the adversary and gets back the corrupted parties' response y_i^*, for $i \in A$.
- Proceed similarly to the previous step: $\mathcal{S}$ samples $r_i \xleftarrow{\$} \mathbb{F}_p$, sets $z_i = y_i^{r_i}$ such that $\prod_i z_i = y^r$, and sends z_i to Adv on behalf of the honest parties. Receives back courrupted parties z_i^*.
- Sends $r \leftarrow \sum_i r_i$ to the adversary. $\mathcal{S}$ performs the checking phase with z_i^* from Adv and the honest z_i. If the check passes send **Deliver** to $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$.

The indistinguishability argument follows from the fact that all broadcasted values g^{x_i} by $\mathcal{S}$ and the real protocol Π_{Exp} have uniform distribution over $\mathbb{F}_p$ with output in $\mathbb{G}$ with respect to $\prod_i g^{x_i} = g^x$.

Correctness is straightforward if all parties follow the protocol. An adversary Adv wins if it changes the distribution of the functionality to output **Deliver**. Alas, this happens with negligible probability: suppose a corrupt party P_j sends y_j^* instead of $y_j = g^{s_j}$. We can write $y_j^* = g^{s_j} \cdot e$, for some error $e \neq 1 \in \mathbb{G}$, and so $y = g^s \cdot e$. Then the check passes if Adv can come up with z_j^* such that $\prod_i z_i = g^{rs} \cdot e^r$. Writing $z_j^* = z_j \cdot f$, this is equivalent to coming up with $f \in \mathbb{G}$ such that $f = e^r$. Since r is uniformly random and unknown to the adversary at the time of choosing e and f, passing this check can only happen with probability $1/|\mathbb{G}|$. Note that this requires $\mathbb{G}$ to be of prime order, so that e (which is adversarially chosen) is always a generator of $\mathbb{G}$. □

More Efficient Protocol based on SPDZ. When using the SPDZ MPC protocol with the secret-shared MAC representation from [DKL+13], we can save performing the multiplication $[t] = [r] \cdot [s]$. Instead, we can take the shared MAC value $[m]$ (on the shared s), which satisfies $m = s \cdot \alpha$ for a shared MAC key α, and use $[m]$ and $[\alpha]$ in place of $[t]$ and $[r]$. However, in this case α cannot be made public, otherwise all future MACs could be forged. Instead, steps 4–6 are replaced with:

- Each party commits to $z_i = y^{\alpha_i} \cdot g^{-m_i}$.
- All parties open their commitments and check that $\prod_i z_i = 1$.

If the parties are honest, we have $z_i = g^{s \cdot \alpha_i - m_i}$, so the check will pass. Since in SPDZ, the honest parties' MAC shares m_i are uniformly random, the shares of α_i are perfectly masked by the g^{-m_i} factor in z_i, so no information on α is leaked. The main difference here is that the parties must commit to the z_i shares before opening, to prevent a rushing adversary from waiting and forcing the product to always be 1. The number of rounds and exponentiations is the same, but one multiplication is saved compared with the previous protocol.

6.4.3 Secure Computation of Naor-Reingold

Given the protocol for public exponentiation, it is straightforward to evaluate the Naor-Reingold PRF with public output when given a bit-decomposed, secret-shared input $[x_1], \ldots, [x_n]$ and key $[k_0], \ldots, [k_n]$. First compute

$$[s] = [k_0] \cdot \prod_{i=1}^{n} ([x_i] \cdot [k_i] + (1 - [x_i]))$$

Protocol Π_{NR}

KeyGen: Call $\mathcal{F}_{\mathsf{ABB}}$.Random to generate $n+1$ random keys $[\![k_0]\!] \dots [\![k_n]\!]$.

Eval: To evaluate $F_{\mathsf{NR}(n)}(k, x)$ on input $[\![x]\!]$ with key $[\![k]\!]$:

1. Bit decompose $[\![x]\!]$ into $[\![x_1]\!] \dots [\![x_n]\!]$.
2. Compute $[\![s]\!] = [\![k_0]\!] \cdot \prod_{i=1}^{n} ([\![k_i]\!][\![x_i]\!] + (1 - [\![x_i]\!])$ (see text for details).
3. Call $\mathcal{F}_{\mathsf{ABB\text{-}Exp}}$ on input $[\![s]\!]$.

Figure 6.5: Computing $F_{\mathsf{NR}(n)}(\mathbf{k}, \mathbf{x})$.

using $\mathcal{F}_{\mathsf{ABB}}$, and then use Π_{Exp} to obtain g^s.

The product can be computed in $\lceil \log_2 n + 1 \rceil$ rounds using a standard binary tree evaluation. Alternatively, we can obtain a constant, 4-rounds protocol using the prefix multiplication protocol of Catrina and de Hoogh [Cd10a], (which is an optimized variant of the trick of Bar-Ilan and Beaver [BIB89]) at the expense of $2(n+1)$ additional multiplications.

Security of the Π_{NR} protocol is straightforward, since there is no interaction outside of the arithmetic black box functionality.

Handling Input in $\mathbb{F}_p$. If the input is given as a field element rather than in bit-decomposed form, then we must first run a bit decomposition protocol, such as that of Catrina and de Hoogh [Cd10a] or Damgård et al. [DFK+06]. The latter works for arbitrary values of x, whilst the former is more efficient, but requires x is ℓ bits long, where $p > 2^{\ell+\kappa}$ for statistical security κ.

Complexity. For the logarithmic rounds variant based on SPDZ, with n-bit input that is already bit decomposed, the protocol requires $2n$ multiplications of secret values and three exponentations, in a total of $\lceil \log_2 n + 1 \rceil + 3$ rounds. The constant rounds variant takes $4n + 2$ multiplications in 7 rounds. Note that there is a higher cost for the secure multiplications, as we require an MPC protocol operating over $\mathbb{F}_p$ for a 256-bit prime p (for 128-bit security), whereas our other PRF protocols only require MPC operations in 128-bit fields.

6.4.4 Performance

The main advantage of this PRF is the small number of rounds required, which leads to a low latency in our benchmarks (4.4ms over LAN). However, the high computation cost (for EC operations) slows down performance and results in a low throughput. We found that with a 256-bit prime p and $n = 128$, the logarithmic rounds variant outperformed the constant rounds protocol in all measures in a LAN environment. In a WAN setting, the constant round protocol achieves a lower latency, but is worse for throughput and preprocessing time.

6.5 PRF from the Legendre Symbol

In this section we consider a PRF based on the Legendre symbol, which to the best of our knowledge was first described in [vHI03]. Whilst this PRF is very inefficient when applied to cleartext data, we show that with secret-shared data in the MPC setting it allows for a very simple protocol.

6.5.1 F_{Leg} Definition

In 1988, Damgård proposed using the sequence of Legendre symbols with respect to a large prime p as a pseudorandom generator [Dam90]. He conjectured that the sequence

$$\left(\frac{k}{p}\right), \left(\frac{k+1}{p}\right), \left(\frac{k+2}{p}\right), \ldots$$

is pseudorandom, when starting at a random seed k. Although there have been several works studying the *statistical* uniformity of this sequence, perhaps surprisingly, there has been very little research on cryptographic applications since Damgård's paper. Damgård also considered variants with the Jacobi symbol, or where p is secret, but these seem less suitable for our application to MPC.

We first normalize the Legendre symbol to be in $\{0, 1, (p+1)/2\}$, by defining

$$L_p(a) = \frac{1}{2}\left(\left(\frac{a}{p}\right) + 1\right) \pmod{p}.$$

We now define the corresponding pseudorandom function (as in [vHI03]) as

$$F_{\mathsf{Leg(bit)}}(k, x) = L_p(k + x)$$

for $k, x \in \mathbb{F}_p$, where $p \approx 2^\kappa$ is a public prime. The security of this PRF is based on the following two problems

Definition 16 (Shifted Legendre Symbol Problem)**.** *Let k be uniformly sampled from $\mathbb{F}_p$, and define $\mathcal{O}_{\mathsf{Leg}}$ to be an oracle that takes $x \in \mathbb{F}_p$ and outputs $\left(\frac{k+x}{p}\right)$. Then the* Shifted Legendre Symbol *(SLS) problem is to find k, with non-negligible probability.*

Definition 17 (Decisional Shifted Legendre Symbol Problem)**.** *Let $\mathcal{O}_{\mathsf{Leg}}$ be defined as above, and let $\mathcal{O}_{\mathsf{R}}$ be a random oracle that takes values in $\mathbb{F}_p$ and produces outputs in $\{-1, 1\}$. The* Decisional Shifted Legendre Symbol *(DSLS) problem is to distinguish between $\mathcal{O}_{\mathsf{Leg}}$ and $\mathcal{O}_{\mathsf{R}}$ with non-negligible advantage.*

The following proposition is then immediate.

Proposition 18. *The function $F_{\mathsf{Leg(bit)}}$ is a pseudorandom function if there is no probabilistic polynomial time algorithm for the DSLS problem.*

6.5.2 Hardness of the Shifted Legendre Symbol Problem

The SLS problem has received some attention from the mathematical community, particularly in the quantum setting. We briefly survey some known results below.

A naive algorithm for *deterministically* solving the SLS problem is to compute $\left(\frac{k+x}{p}\right)$ for all $(k, x) \in \mathbb{F}_p^2$ and compare these with $\mathcal{O}_{\mathsf{Leg}}(x)$ for all $x \in \mathbb{F}_p$, which requires $\tilde{O}(p^2)$ binary operations. Russell and Shparlinski [RS04] described a more sophisticated algorithm using Weil's bound on exponential sums, which reduces this to $\tilde{O}(p)$.

Van Dam, Hallgren and Ip [vHI03] described a quantum polynomial time algorithm for the SLS problem that recovers the secret k if the oracle can be queried on a quantum state. They conjectured that classically, there is no polynomial time algorithm for this problem. Russell and Shparlinski [RS04] also extended this quantum algorithm to a generalization of the problem where the secret is a polynomial, rather than just a linear shift.

One can also consider another generalization called the *hidden shifted power problem*, where the oracle returns $(k + x)^e$ for some (public) exponent $e|(p - 1)$. The SLS problem is a special case where $e = (p - 1)/2$. Vercauteren [Ver08] called this the *hidden root problem* and described efficient attacks over small characteristic extension fields, with applications to fault attacks on pairings-based cryptography. Bourgain et al. [BGKS12] showed that if $e = p^{1-\delta}$ for some $\delta > 0$ then this problem has classical query complexity $O(1)$. Note that neither of these attacks apply to the SLS problem, which cannot be solved with fewer than $\Omega(\log p)$ queries [VD02].

At the time of writing the paper we were not aware of any classical algorithms for the SLS problem in better than $\tilde{O}(p)$ time, nor of any method for solving the DSLS problem without first recovering the secret. There has been some progress within the last year concerning the security of the Legendre PRF by Khovratovich [Kho19] or by Beulens et al [BBUV19]. They are able to solve the SLS problem on a classical computer within time complexity $O(\sqrt{p} \log p)$ whereas for the "high-degree" version of SLS problem, where $\left(\frac{k_0+k_1x+x^2}{p}\right)$, the complexity increases to $O(p \log p)$. In conclusion, the cryptanalysis of the linear version $\left(\frac{k+x}{p}\right)$ is still an open problem for primes of at least 256-bits long whereas for high degree SLS this seems to be a hard problem for the primes we consider in our paper ($p \approx 2^{128}$).

6.5.3 Secure Computation of $F_{\mathsf{Leg(bit)}}$

It turns out that $F_{\mathsf{Leg(bit)}}$ can be evaluated in MPC very efficiently, at roughly the cost of just 2 multiplications in 3 rounds of communication. Although this only produces a single bit of output, composing together multiple instances in parallel with independent keys allows larger outputs to be obtained (see later).

We first describe how to evaluate $F_{\mathsf{Leg(bit)}}$ when the output is public, and then show how to extend this to secret-shared output, with only a small cost increase.

Public output. Suppose we have a shared, non-zero $[a]$ and want to compute the public output, $L_p(a)$. Since the output is public, we can simply take a random preprocessed non-zero square $[s^2]$, compute

Protocol Π_{Legendre}

Let α be a fixed, quadratic non-residue modulo p.

KeyGen: Call $\mathcal{F}_{\mathsf{ABB}}$.Random to generate a random key $[\![k]\!]$.

Eval: To evaluate $F_{\mathsf{Leg(bit)}}$ on input $[\![x]\!]$ with key $[\![k]\!]$:

1. Take a random square $[\![s^2]\!]$ and a random bit $[\![b]\!]$
2. $[\![t]\!] \leftarrow [\![s^2]\!] \cdot ([\![b]\!] + \alpha \cdot (1 - [\![b]\!]))$
3. $u \leftarrow \mathsf{Open}([\![t]\!] \cdot ([\![k]\!] + [\![x]\!]))$
4. Output $[\![y]\!] \leftarrow (\left(\frac{u}{p}\right) \cdot (2[\![b]\!] - 1) + 1)/2$

Figure 6.6: Securely computing the $F_{\mathsf{Leg(bit)}}$ PRF with secret-shared output.

$[c] = [s^2] \cdot [a]$ and open c. By the multiplicativity of the Legendre symbol, $L_p(c) = L_p(a)$.

By composing the PRF n times in parallel, this gives an n-bit output PRF that we can evaluate in MPC with just n multiplications and n openings in 2 rounds. The preprocessing requires n random squares and multiplication triples.

Shared output. Now suppose we instead want shared output, $[\![L_p(a)]\!]$. If we have a random non-zero value $[\![t]\!]$, and also the shared value $[\![L_p(t)]\!]$, then this is easy. Just open $[\![a]\!] \cdot [\![t]\!]$, and compute the Legendre symbol of this to get $c = L_p(a \cdot t)$. The shared value $[\![L_p(a)]\!]$ can then be computed locally using c and $[\![L_p(t)]\!]$, as c is public.

Generating a random value with a share of its Legendre symbol can be done very cheaply. Our key observation is that we can do this without having to compute *any* Legendre symbols in MPC. Let $\alpha \in \mathbb{Z}_p$ be a (public) quadratic non-residue, and perform the following:

- Take a random square $[\![s^2]\!]$ and a random bit $[\![b]\!]$.
- Output $(2[\![b]\!] - 1, [\![b]\!] \cdot [\![s^2]\!] + (1 - [\![b]\!]) \cdot \alpha \cdot [\![s^2]\!])$

Note that since α is a non-square, the second output value is clearly either a square or non-square based on the value of the random bit b (which is mapped into $\{-1, 1\}$ by computing $2 \cdot b - 1$). Finally, note that since s^2 provides fresh randomness each time, α can be reused for every PRF evaluation. This gives us the protocol in Figure 6.6, which realizes the functionality $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$ shown in Figure 6.7. Notice that all bar the computation of u can be performed in a preprocessing phase if needed.

Security. At first glance, the security of the protocol appears straightforward: since t and k are uniformly random, the opened value u should be simulatable by a random value, and this will be correct except with probability $1/p$ (if $s^2 = 0$). However, proving this turns out to be more tricky. We need to take into account that if $x = -k$ then the protocol causes $u = 0$ to be opened, but in the ideal world the simulator does not know k so cannot simulate this. This reflects the fact that an adversary who solves the SLS problem can find k and run the protocol with $x = -k$. Therefore, we need to assume hardness of the SLS problem and show that any environment that distinguishes the two worlds (by causing $x = -k$ to be queried) can be used to recover the key k. The reduction must use the SLS oracle, $\mathcal{O}_{\mathsf{Leg}}$,

Functionality $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$

This functionality has all of the same commands as $\mathcal{F}_{\mathsf{ABB}}$, plus the following:

KeyGen: On receiving $(\texttt{keygen})$ from all parties, sample $k \xleftarrow{\$} \mathbb{F}_p$ and store k.

PRF: On receiving $(\texttt{legendre}, [x])$ from all parties, where x is stored in memory, compute $y = L_p(x + k)$ and store y in memory.

Figure 6.7: Ideal functionality for the Legendre symbol PRF, $F_{\mathsf{Leg(bit)}}$.

to detect whether $x = -k$, in order to simulate the u value to the environment. To do this, they simply obtain the value $y = \left(\frac{x+k}{p}\right)$ from $\mathcal{O}_{\mathsf{Leg}}$ and check whether $y = 0$, for each **Eval** query made by the adversary.

Theorem 19. *The protocol Π_{Legendre} securely computes the functionality $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$ in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model, if the SLS problem is hard.*

Proof. We construct a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ corrupting up to $n - 1$ parties can distinguish between the real protocol Π_{Legendre}, and $\mathcal{S}$ interacting with the ideal functionality $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$.

In the **KeyGen** stage, $\mathcal{S}$ simply calls $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$ with the `keygen` command. In the **Eval** stage, the main task of $\mathcal{S}$ is to simulate the opened value u, which is done by sampling $u \xleftarrow{\$} \mathbb{F}_p$, and then call $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$ with $(\texttt{legendre}, [\![x]\!])$.

We now argue indistinguishability of the two executions. In the real world, since t is computed as $s^2 \cdot (b + (1 - b) \cdot \alpha)$ for a uniform quadratic residue s^2 and random bit b, then t is uniform in $\mathbb{F}_p$. This is because the map defined by multiplication by α is a bijection between the sets of squares and non-squares modulo p. Therefore, if s^2 is a uniformly random square, then $\alpha \cdot s^2$ is a uniformly random non-square.

Now, since t is a fresh uniformly random value on each evaluation, the real world value u and output y, as seen by $\mathcal{Z}$, will be identically distributed to the simulated values as long as $k + x \neq 0$ and $s \neq 0$. Whenever the former happens in the real world $u = 0$ is opened, whereas the ideal world still simulates a random value, so the environment can distinguish. In the latter case, $s = 0$, the output y will be incorrectly computed in the real world, but this can only happen with probability $1/p$.

However, any environment $\mathcal{Z}$ that causes $k + x = 0$ to happen with non-negligible probability can be used to construct an algorithm $\mathcal{A}^*$ that breaks the SLS problem, as follows.

$\mathcal{A}^*$ runs $\mathcal{Z}$, emulating a valid execution of Π_{Legendre} by replacing $L_p(x+k)$ computation with calls to $\mathcal{O}_{\mathsf{Leg}_k}$. These modified transcripts have the same distribution since the SLS oracle and $(\texttt{keygen})$ both generate a random key. When $\mathcal{A}^*$ runs $\mathcal{Z}$ internally, it knows the inputs provided by $\mathcal{Z}$ to all parties, so knows the x value on each invocation of Π_{Legendre}. Once $\mathcal{Z}$ constructs a query for which $\mathcal{O}_{\mathsf{Leg}_k}$ returns 0 then $\mathcal{A}^*$ responds to the SLS challenge with $k = -x$. Finally, the algorithm looks like this:

1. Interact with $\mathcal{Z}$ as the simulator $\mathcal{S}$ would do.

2. Instead of computing the Legendre symbol $L_p(x+k)$ as in $\mathcal{F}_{\mathsf{ABB\text{-}Leg}}$, make a call to $\mathcal{O}_{\mathsf{Leg}_k}$.
3. If $\mathcal{O}_{\mathsf{Leg}_k}(x) = 0$, return $-x$ as the SLS secret.

The only way $\mathcal{Z}$ can distinguish between $\mathcal{S}$ and Π_{Legendre} — except with probability $1/p$ — is by producing a query x for which $\mathcal{O}_{\mathsf{Leg}_k}(x) = 0$, since the two worlds are statistically close up until this point. If $\mathcal{Z}$ can do this with probability ϵ then the probability that $\mathcal{A}^*$ solves the SLS problem is the same.

Overall, $\mathcal{S}$ correctly simulates the protocol Π_{Legendre} as long as $u \neq 0$, which happens with probability $\leq 1/p + \epsilon$ ($s = 0$ or solving SLS with probability ϵ). □

Perfect Correctness. The basic protocol above is only statistically correct, as $s^2 = 0$ with probability $1/p$, and if this occurs the output will always be zero. Although this suffices for most applications, we note that perfect correctness can be obtained, at the expense of a protocol that runs in *expected* constant rounds. We can guarantee that the square s^2 is non-zero by computing it as follows:

- Take a random square $[\![s^2]\!]$ and a random value $[\![y]\!]$.
- Compute $[\![v]\!] = [\![y \cdot s^2]\!]$ and open v. If $v = 0$ then return to the first step.

Note, that the iteration of the first step only happens if $y = 0$ or $s = 0$, which occurs with probability $2/p$, so the expected number of rounds for this stage of the protocol is one.

6.5.4 Domain and Codomain Extension

Some applications may require a PRF which takes multiple finite field elements as input, and outputs a finite field element. We now present how to extend the basic PRF $F_{\mathsf{Leg(bit)}}$ to a function which takes messages consisting of n finite field elements and outputs a single uniformly random finite field element. Indeed our input could consist of up to t elements in the finite field where $t \leq n$. In practice we will take $n = 1$ or 2, and can then extend to larger lengths using CBC-mode or Merkle-Damgård (as in Section 6.3.1).

We first define a statistical security parameter $2^{-\mathsf{stat}}$, which bounds the statistical distance from uniform of the output of our PRF. We let define p' to be the nearest power of two to the prime p and set $\alpha = |p - p'|$. Then if $\alpha/p < 2^{-\mathsf{stat}}$ we set $\ell = \lceil \log_2 p \rceil$, otherwise we set $\ell = \lceil \log_2 p \rceil + \mathsf{stat}$. A standard argument will then imply that the following PRF outputs values with the correct distribution.

The key for the PRF is going to be an $\ell \times (n+1)$ matrix K of random elements in $\mathbb{F}_p$, except (for convenience) that we fix the first column to be equal to one. To apply the PRF to a vector of elements $\mathbf{x} = (x_1, \ldots, x_t)$ we "pad" $\mathbf{x}$ to a vector of $n+1$ elements as follows $\mathbf{x}' = (x_1, \ldots, x_t, 0, \ldots, 0, t)$ and then product the matrix-vector product $\mathbf{y} = K \cdot \mathbf{x}' \in (\mathbb{F}_p)^\ell$. The output of $F_{\mathsf{Leg}(n)}$ is then given by

$$F_{\mathsf{Leg}(n)}(K, \mathbf{x}) = \left(\sum_{i=0}^{\ell-1} 2^i \cdot L_p(y_i) \right) \pmod{p}.$$

This extended PRF requires one extra round of $\ell \cdot (n-1)$ secure multiplications compared to $F_{\mathsf{Leg(bit)}}$.

Since the matrix K is compressing, the distribution of $\mathbf{y}$ will act, by the leftover hash lemma, as a random vector in $\mathbb{F}_p^\ell$. With probability ℓ/p we have $y_i \neq 0$ for all i, which implies that the values of $L_p(y_i)$ behave as uniform random bits, assuming our previous conjectures on the Legendre symbol. Thus the output value of $F_{\mathsf{Leg}(n)}(K, \mathbf{x})$ will, by choice of ℓ, have statistical distance from uniform in $\mathbb{F}_p$ bounded by $2^{-\mathsf{stat}}$.

Our choice of padding method, and the choice of the first matrix column to be equal to one, is to ensure that in the case of $n = 1$ we have

$$F_{\mathsf{Leg}(n)}(K, \mathbf{x}) = \left(\sum_{i=0}^{\ell-1} 2^i \cdot F_{\mathsf{Leg(bit)}}(k_i, y_i) \right) \pmod{p}.$$

In addition, the padding method ensures protection against length extension attacks.

6.5.5 Performance

We measured performance using the prime $p = 2^{127} + 45$, which implied for $F_{\mathsf{Leg}(n)}$ we could take $\ell = 128$. Both $F_{\mathsf{Leg(bit)}}$ and $F_{\mathsf{Leg}(1)}$ obtain very low latencies (0.35ms and 1.2ms over LAN, respectively) due to the low number of rounds. For a PRF with small outputs, $F_{\mathsf{Leg(bit)}}$ achieves by far the highest throughput, with over 200000 operations per second. For full field element outputs, $F_{\mathsf{Leg}(1)}$ is around 128 times slower, but still outperforms AES in all metrics except for cleartext computation.

6.6 MiMC

6.6.1 F_{MiMC} Definition

MiMC is a comparatively simple block cipher design, where the plaintexts, the ciphertexts and the secret key are elements of $\mathbb{F}_p$ and can be seen as a simplification of the KN-cipher [NK95]. Its design is aimed at achieving an efficient implementation over a field $\mathbb{F}_p$ by minimizing computationally expensive field operations (e.g. multiplications or exponentiations).

Let p a prime that satisfies the condition $\gcd(3, p-1) = 1$. For a message $x \in \mathbb{F}_p$ and a secret key $k \in \mathbb{F}_p$, the encryption process of MiMC is constructed by iterating a round function r times. At round i (where $0 \leq i < r$), the round function $F_i : \mathbb{F}_p \to \mathbb{F}_p$ is defined as:

$$F_i(x) = (x + k + c_i)^3,$$

where c_i are random constants in $\mathbb{F}_p$ (for simplicity $c_0 = c_r = 0$). The output of the final round is added with the key k to produce the ciphertext. Hence, the output of $F_{\mathsf{MiMC}}(x, k)$ is then given by

$$F_{\mathsf{MiMC}}(x, k) = (F_{r-1} \circ F_{r-2} \circ \ldots \circ F_0)(x) + k.$$

The condition on p ensures that the cubing function creates a permutation.

The number of rounds for constructing the keyed permutation is given by $r = \lceil \log_3 p \rceil$ - for prime fields of size 128 bits the number of rounds is equal to $r = 82$. This number of round r provides security against a variety of cryptanalytic techniques. In particular, due to the algebraic design principle of MiMC, the most powerful key recovery methods are the algebraic cryptanalytic attacks, as the Interpolation Attack and the GCD Attack. In the first one introduced by Jakobsen and Knudsen in [JK97], the attacker constructs a polynomial corresponding to the encryption function without any knowledge of the secret key. In particular, the attacker guesses the key of the final round, constructs the polynomial at round $r-1$ and checks it with one extra plaintext/ciphertext pair. In the second one, given two plaintext/ciphertext pairs (p^j, c^j) for $j = 1, 2$, the attacker constructs the polynomials $F_{\mathsf{MiMC}}(p^1, K) - c^1$ and $F_{\mathsf{MiMC}}(p^2, K) - c^2$ in the fixed but unknown key K. Since these two polynomials share $(K - k)$ as a factor (where k is the secret key), the attacker can find the value of k by computing the GCD of them.

If the attacker has access to a limited number of plaintext/ciphertext pairs only (at most $n < p$), then the number of round r can be reduced. In this case, the number of rounds is given by $r = \max\{\lceil \log_3 n \rceil, \lceil \log_3 p - 2\log_3(\log_3 p) \rceil\}$ - for prime field of size 128 bits, the number of rounds is equal to $r = 73$ if $n \leq 2^{115}$, while $r = \lceil \log_3 n \rceil$ otherwise.

6.6.2 Computing F_{MiMC} in MPC

We consider two different approaches for computing F_{MiMC} in MPC, with a secret shared key and message. The basic approach is simplest, whilst the second variant has half the number of rounds of communication, with slightly more computation.

$\mathsf{MiMC}^{\mathsf{basic}}$: The naive way to evaluate F_{MiMC} requires one squaring and one multiplication for each of the r rounds. Using SPDZ, the squaring costs one opening in one round of communication, and the multiplication costs two openings in one round, giving a total of $3r$ openings in $2r$ rounds of communication.

$\mathsf{MiMC}^{\mathsf{cube}}$: If for each round we first compute a tuple $([r], [r^2], [r^3])$, where $r \xleftarrow{\$} \mathbb{F}_p$, then given a secret-shared value $[x]$, we can open $y = x - r$ and obtain a sharing of x^3 by the computation

$$[x^3] = 3y[r^2] + 3y^2[r] + y^3 + [r^3]$$

which is linear in the secret-shared values so does not require interaction.

For a single MiMC encryption, we first compute all of the cube triples for each round, which takes just one round of communication by taking a preprocessed random square pair $([r], [r^2])$ and performing one multiplication to obtain $[r^3]$. Each round of the cipher then requires just one opening and a small amount of interaction. The total communication complexity is still $3r$ openings, but in only r rounds.

6.6.3 Performance

Using $r = 73$, we measured a latency of 12ms per evaluation for the simple protocol $\mathsf{MiMC}^{\mathsf{basic}}$, which halves to 6ms for the lower round variant, $\mathsf{MiMC}^{\mathsf{cube}}$. $\mathsf{MiMC}^{\mathsf{basic}}$ gives a very high throughput of over 8500 blocks per second (around 20% higher than $\mathsf{MiMC}^{\mathsf{cube}}$), and the offline cost is fairly low, at 34 blocks per second. In fact, apart from in latency, MiMC outperforms all the other PRFs we studied.

6.7 Performance Evaluation

In this section, we evaluate the performance of the PRFs using the SPDZ multi-party computation protocol [DPSZ12, DKL+13], which provides active security against any number of corrupted parties. We focus here on the two-party setting, although the protocol easily scales to any number of parties with roughly a linear cost.

The two main metrics we use to evaluate performance are *latency* and *throughput*, both of which relate to the online phase of the SPDZ protocol. Latency measures the waiting time for a *single* PRF evaluation; the best possible latency is recorded by simply timing a large number of sequential executions of the PRF, and taking the average for one operation. In contrast, throughput is maximized by running many operations in parallel to reduce the number of rounds of communication. Of course, this comes at the expense of a higher latency, so a tradeoff must always be made depending on the precise application. In addition to latency and throughput, we present the cost of running the preprocessing phase and computing the PRF on cleartext data, for comparison.

Implementation Details: We implemented the protocols using the architecture of Keller et al. [KSS13a], which runs the online phase of SPDZ. This system automatically uses the minimum number of rounds of communication for a given program description, by merging together all independent openings. We extended the software to use the Miracl library for elliptic curve operations over the NIST P-256 curve, as required for the Naor-Reingold protocol. Note that although the SPDZ implementation supports multi-threading, all of our online phase experiments are single-threaded to simplify the comparison.

	Data type	$\mathbb{F}_p$ (ms)		$\mathbb{F}_{2^{128}}$ (ms)
		128-bit	256-bit	
LAN	Triple/Sq.	0.204	0.816	0.204
	Bit	0.204	0.816	0.00014
WAN	Triple/Sq.	4.150	16.560	4.150
	Bit	4.150	16.560	0.00285

Table 6.2: Time estimates for generating preprocessing data in various fields using oblivious transfer.

PRF	Best latency	Best throughput		Prep. (ops/s)	Cleartext (ops/s)
	(ms/op)	Batch size	ops/s		
AES	7.713	2048	530	5.097	106268670
$F_{\mathsf{LowMC}}(\mathsf{vector})$	4.302	256	591	2.562	7000
$F_{\mathsf{LowMC}}(\mathsf{M4R})$	4.148	64	475	2.565	1420
$F_{\mathsf{NR}(128)}(\mathsf{log})$	4.375	1024	370	4.787	1359
$F_{\mathsf{NR}(128)}(\mathsf{const})$	4.549	256	281	2.384	1359
$F_{\mathsf{Leg}(\mathsf{bit})}$	0.349	2048	202969	1225	17824464
$F_{\mathsf{Leg}(1)}$	1.218	128	1535	9.574	115591
$F_{\mathsf{MiMC}}(\mathsf{basic})$	12.007	2048	8788	33.575	189525
$F_{\mathsf{MiMC}}(\mathsf{cube})$	5.889	1024	6388	33.575	189525

Table 6.3: Performance of the PRFs in a LAN setting.

To estimate the cost of producing the preprocessing data (multiplication triples, random bits etc.), we used figures from the recent MASCOT protocol [KOS16], which uses OT extensions to obtain what are currently the best reported triple generation times with active security. Although in [KOS16], figures are only given for triple generation in a 128-bit field, we can also use these times for random square and random bit generation, since each of these can be easily obtained from one secret multiplication [DFK+06]. For the Naor-Reingold PRF, we multiplied these times by a factor of 4 to obtain estimates for a 256-bit field (instead of 128), reflecting the quadratic communication cost of the protocol. [2] The costs for all of these preprocessing data types are summarized in Table 6.2.

Note that LowMC only requires multiplication triples in $\mathbb{F}_2$, for which the protocol of [FKOS15] could be much faster than using $\mathbb{F}_{2^{128}}$ triples. However, we are not currently aware of an implementation of this protocol, so use the $\mathbb{F}_{2^{128}}$ times for now.

Benchmarking Environment: In any application of MPC, one of the most important factors affecting performance is the capability of the network. We ran benchmarks in a standard 1Gbps LAN setting, and also a simulated WAN setting, which restricts bandwidth to 50Mbps and latency to 100ms, using the Linux `tc` tool. This models a real-world environment where the parties may be in different countries or continents. In both cases, the test machines used have Intel i7-3770 CPUs running at 3.1GHz, with 32GB of RAM.

Results: The results of our experiments in the LAN and WAN environments are shown in Tables 6.3 and 6.4, respectively. All figures are the result of taking an average of 5 experiments, each of which ran at least 1000 PRF operations. We present timings for AES and LowMC purely as a comparison

[2] The experiments in [KOS16] showed that communication is the main bottleneck of the protocol, so this should give an accurate estimate.

PRF	Best latency (ms/op)	Best throughput		Prep. (ops/s)
		Batch size	ops/s	
AES	2640	1024	31.947	0.256
$F_{\mathsf{LowMC}}(\mathsf{vector})$	1315	2048	365	0.1259
$F_{\mathsf{LowMC}}(\mathsf{M4R})$	659	2048	334	0.1261
$F_{\mathsf{NR}(128)}(\mathsf{log})$	713	1024	59.703	0.2359
$F_{\mathsf{NR}(128)}(\mathsf{const})$	478	1024	30.384	0.1175
$F_{\mathsf{Leg}(\mathsf{bit})}$	202	1024	2053	60.241
$F_{\mathsf{Leg}(1)}$	210	512	68.413	0.4706
$F_{\mathsf{MiMC}}(\mathsf{basic})$	7379	512	59.04	1.650
$F_{\mathsf{MiMC}}(\mathsf{cube})$	3691	512	79.66	1.650

Table 6.4: Performance of the PRFs in a simulated WAN setting.

metric; as explained in the introduction, these are not suitable for many MPC applications as they do not operate over a large characteristic finite field.

LowMC obtains slightly better throughput and latency than AES over a LAN, with both the vector and M4R methods achieving similar performance here. In the WAN setting, LowMC gets a very high throughput of over 300 blocks per second. This is due to the low online communication cost for multiplications in $\mathbb{F}_2$ instead of $\mathbb{F}_{2^n}$ or $\mathbb{F}_p$, and the fact that local computation is less significant in a WAN. The M4R method gets half the latency of the vector method in this scenario, since the number of rounds is halved. As discussed earlier, the preprocessing for LowMC would likely be much better than AES if implemented with the protocol of [FKOS15].

In both scenarios, the Legendre PRF gives the lowest latency, even when outputing 128-bit field elements rather than bits, due to its low round complexity. The single-bit output variant achieves by far the highest throughput of all the PRFs, so would be ideally suited to an application based on a short-output PRF, such as secure computation of the (leaky) order-revealing encryption scheme in [CLWW16]. The Legendre PRF with large outputs is useful in scenarios where low latency is very important, although the preprocessing costs are expensive compared to MiMC below. However, the high cost of the Legendre PRF "in the clear" may not make it suitable for applications in which one entity is encrypting data to/from the MPC engine.

The Naor-Reingold PRF also achieves a low latency — though not as good as the Legendre PRF — but it suffers greatly when it comes to throughput. Notice that in the LAN setting, the constant rounds protocol actually performs worse than the logarithmic rounds variant in all measures, showing that here the amount of computatation and communication is more of a limiting factor than the number of rounds. Profiling suggested that over 70% of the time was spent performing EC scalar multiplications, so it seems that computation rather than communication is the bottleneck in these timings. The requirement for a 256-bit field (for 128-bit security) will be a limiting factor in many applications, as will the need to bit decompose the input, if it was previously a single field element.

The MiMC cipher seems to provide a good compromise amongst all the prime field candidates, especially as it also performs well when performed "in the clear". The cube variant, which halves the number of rounds, effectively halves the latency compared to the naive protocol. This results in a slightly worse throughput in the LAN setting due to the higher computation costs, whereas in the WAN setting round complexity is more important. Although the latency is much higher than F_{Leg}, due to the large number of rounds, MiMC achieves the best throughput for $\mathbb{F}_p$-bit outputs, with over 6000 operations per second. In addition, the preprocessing costs of MiMC are better than that of both Legendre and the Naor-Reingold PRFs.

So in conclusion there is no single PRF which meets all the criteria we outlined at the beginning. But one would likely prefer the Legendre PRF for applications which require low latency, and which do not involve any party external to the MPC engine, and MiMC for all other applications.

Chapter 7

Modes of operation over $\mathbb{F}_p$

This chapter is based on joint work with Nigel P. Smart and Martijn Stam [RSS17] which was published in Transactions on Symmetric Cryptology 2017 and presented at FSE 2018.

7.1 Contributions

In this chapter we being our search for building more advanced protocol on top of the PRFs described earlier, looking at the concrete case of Authenticated Encryption (AE) in MPC. After finding several candidates suitable for doing AE in MPC such as OTR, PMAC, Hash-then-MAC we then formally prove that they are secure with concrete query bounds. We then showed experimentally their performance by instantiating them with the most efficient PRFs found in the previous chapter. Contrary to our belief, when multiple blocks get authenticated MiMC turned out to be more efficient than the Legendre PRF due to fewer openings done in parallel and a lower computational complexity.

7.2 Overview

In the previous chapter we have seen that there are many applications built on top of PRFs evaluated on encrypted data which deals with sending data to and from an MPC engine $\mathcal{E}$. Consider the example when a client wants to send data securely to $\mathcal{E}$: the client encrypts her data using a PRF in counter mode to then let the system jointly decrypt using a PRF in MPC. The advantage of this method is that it abstracts away the underlying structure of $\mathcal{E}$, such as the number of parties or the type of MPC protocol, and allows an easy key management due to the client storing just a single symmetric key.

But what happens if the data needs to be authenticated during transit? In this chapter we try to answer this question by ensuring data integrity plus confidentiality to and from $\mathcal{E}$, as well as proving the client's identity. In order to achieve this we survey the literature for highly parallel modes of operation for AE and adapt them to work over a prime field. The restriction to work over a prime field comes from the fact that most of the secret-sharing based MPC engines designed to work with a dishonest majority are more efficient over a prime field.

We examine how the currently best PRFs for secret shared MPC over $\mathbb{F}_p$, namely MiMC and Leg, can be used to enable nonce-based authenticated encryption, where we benchmark a number of orthogonal options. As alluded to before, we are assuming that the key and the plaintext message are held in secret shared form, but that the nonce and the resulting ciphertext are in the clear. This assumption crucially informs our study.

Our first step is to select potential modes of operation for secret-sharing based MPC-driven, nonce-based AE. In making such a selection there are a number of design desiderata to take into account the somewhat unusual computational model. Firstly, the underlying PRF is only ever evaluated in the forward direction, both during encryption and decryption (even though MiMC as a blockcipher does have an inverse, it is rather inefficient). Secondly, the mode should allow a high degree of parallelism of the PRF calls to take full advantage of the ability of secret-sharing based MPC engines to evaluate many operations in parallel. Finally, the further computational overhead (beyond PRF calls) may be complicated, provided it can be performed locally. To enable local computation, it can be worth opening secret shared elements, provided this opening does not negatively affect security.

When examining various possible modes for authenticated encryption, we found two candidates that best met our overall design criteria: on the one hand, a single combined mode based on OTR [Min14], and on the other an Encrypt-then-MAC methodology using either CTR-Mode plus PMAC [BR02], or CTR-Mode plus Hash-then-Encrypt (where in both cases the CTR-Mode is nonce-based by exploiting a tweakable PRF). We converted the original PMAC and OTR algorithms (which use finite fields of characteristic two) into variants that process blocks consisting of finite field elements in $\mathbb{F}_p$, where p is a large prime (say $p > 2^{128}$). The resulting algorithms we dub pPMAC and pOTR. Here we took care to ensure that the modifications made do not invalidate any of the original security proofs.

Modern modes of operation, including PMAC and OTR, are usually cleanest described based on a tweakable primitive, and we follow suit. This obviously does necessitate the investigation of tweakable PRFs in our MPC context. Luckily, creating tweakable PRFs turns out much easier than in the traditional, binary field setting. In that latter setting, Rogaway's XE transform [Rog04] takes a PRF $\mathsf{E}_\mathsf{k}(m)$ and turns it into a tweakable PRF $\widetilde{\mathsf{E}}_\mathsf{k}^{i,N}(m)$ with a tweak (i, N) using a sequence of constants M_i in the following manner:

$$\widetilde{\mathsf{E}}_\mathsf{k}^{i,N}(m) = \mathsf{E}_\mathsf{k}(m \oplus (M_i \cdot \mathsf{E}_\mathsf{k}(N))) \; .$$

It is important that the constants M_i do not repeat, and be easy to compute. This led many authors to select $M_i = 2^{T_1} \cdot 3^{T_2}$ for two functions T_1, T_2 depending on i. This choice is prompted by the characteristic two field, with the exact tweak applied depending on the field order. In our setting of large prime characteristic, we obtain a trivial schedule by using a standard integer representation of the field:

$$\widetilde{\mathsf{E}}_\mathsf{k}^{i,N}(m) = \mathsf{E}_\mathsf{k}(m + (i \cdot \mathsf{E}_\mathsf{k}(N))) \; .$$

With CTR-then-pPMAC as an encryption methodology on a message consisting of ℓ finite field elements (i.e. ℓ blocks in this context), we apply one round of ℓ tweakable-PRF evaluations to encrypt the ℓ message blocks, then another round of $\ell - 1$ tweakable-PRF evaluations to produce a final MAC

block, to which a final tweakable-PRF evaluation is performed. Ignoring non-message dependent PRF evaluations this means we need to evaluate $2 \cdot \ell$ PRF evaluations in a total of three parallel rounds. For the CTR+Hash-then-MAC mode we apply one round of ℓ tweakable-PRF evaluations to encrypt the ℓ message blocks, then a hash function in the clear to produce an intermediate open value, to which a final tweakable-PRF evaluation is performed. This means we need to evaluate $\ell + 1$ PRF evaluations in a total of two parallel rounds. For the OTR mode we evaluate first a PRF on a nonce block, then apply ℓ PRF calls in two rounds (essentially performing a two round Feistel network). A final PRF evaluation produces the tag. Overall, we require $\ell + 2$ PRF evaluations over four parallel rounds to evaluate OTR mode. Not surprisingly, we find that CTR+Hash-then-MAC turns out to be the most efficient of these modes of operation.

In a second step we implemented the modes using the above two PRFs to see which performed better in practice. Our experiments are carried out using the publicly available SPDZ engine [DPSZ12, DKL+13], though any classical protocol based on Shamir secret sharing could also be used. Previously, Grassi et al. [GRR+16] conducted experiments under the assumptions that the input *and* the output to the PRF need to be kept in secret shared form. However, when used within one of the above modes of operation this *may* no longer be true, enabling further optimizations to be made into precisely how the PRFs are evaluated within the MPC system, a topic which we explore in this thesis.

Grassi et al. furthermore imply that the Leg PRF is to be preferred over the MiMC PRF, as the Leg PRF (based on the Legendre symbol) had a lower online round cost and slightly higher preprocessing costs. Their experiments seemed to confirm the preference for the Leg PRF. Interestingly, when used within a mode of operation supporting parallel processing of the blocks, we find that the MiMC PRF online phase performs much better. Though the Leg PRF has low round complexity and low computational cost (when computational cost is measured in an MPC environment), its per-round communication cost is high. Thus for each round of communication the number of bits sent between the MPC servers is much larger than that for MiMC. When many PRF applications are done in parallel this high per round communication cost causes network bottlenecks, resulting in a linear scaling in the runtime as the number of blocks processed increases. For MiMC, reaching network saturation takes longer and so, as the number of blocks processed is increased, the runtime only degrades sub-linearly. Hence, MiMC will often significantly outperform Leg.

7.3 Preliminaries

In this section we recall the basic notions of Authenticated Encryption and its constituent building block PseudoRandom Function, as well as generic design considerations in the context of MPC, including details on the two existing PRFs designed for MPC that we will build upon. Throughout we will write $\mathsf{Adv}^{O_1,\ldots,O_c}$ for an algorithm Adv with access to c oracles $O_1, \ldots, O_c$. For a finite field $\mathbb{F}_p$ we let $\mathbb{F}_p^\times = \mathbb{F}_p \setminus \{0\}$.

Algorithm $\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,N}(m)$:
1: $L \leftarrow \mathsf{E}_{\mathsf{k}}(N)$
2: $\Delta \leftarrow i \cdot L$
3: $Y \leftarrow \mathsf{E}_{\mathsf{k}}(\Delta + m)$
4: **return** Y

Figure 7.1: XE-based tweakable pseudorandom function over $\mathbb{F}_p$.

7.3.1 Tweakable Pseudorandom Functions

A Pseudo-Random Function (PRF) is a keyed function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, where $\mathcal{K}$ is called the key space. The key $\mathsf{k} \in \mathcal{K}$ is typically chosen at random and the function keyed with k is denoted F_{k}.

A PRF is pseudorandom if an adversary cannot tell the difference between oracle access to F_{k}, for undisclosed k uniformly chosen at random from $\mathcal{K}$, on the one hand and oracle access to a function selected uniformly at random from the set $\mathrm{Rand}(\mathcal{X}, \mathcal{Y})$ of all functions which map $\mathcal{X}$ to $\mathcal{Y}$, on the other. More formally, for an adversary Adv the PRF advantage against F (or $F_{\mathsf{k}}(\cdot)$) is defined as

$$\mathsf{Adv}_F^{\mathsf{prf}} \stackrel{\mathrm{def}}{=} \left| \Pr\left[\mathsf{k} \xleftarrow{\$} \mathcal{K} : \mathsf{Adv}^{F_{\mathsf{k}}(\cdot)} \Rightarrow 1 \right] - \Pr\left[\rho \xleftarrow{\$} \mathrm{Rand}(\mathcal{X}, \mathcal{Y}) : \mathsf{Adv}^{\rho(\cdot)} \Rightarrow 1 \right] \right|$$

where we will informally say F is a PRF if this advantage is sufficiently small for all reasonably resourced adversaries. It is easy to formalize our work to an asymptotic setting where security equates to negligible advantages with respect to all probabilistic polynomial-time adversaries operating against function families (indexed by a security parameter).

In analogy with tweakable blockciphers, we shall also consider PRFs. A tweakable PRF (tPRF) takes as additional input a tweak T chosen from a set of tweaks $\mathcal{T}$, thus $\tilde{F} : \mathcal{K} \times \mathcal{T} \times \mathcal{X} \to \mathcal{Y}$. Security is defined in much the same way as for a PRF, except that the adversary can query the function on tweak–message pairs and the adversary's goal is to distinguish $\tilde{F}_{\mathsf{k}}$ from a random function $\widetilde{\rho} \in \mathrm{Rand}(\mathcal{T} \times \mathcal{X}, \mathcal{Y})$. More formally, for an adversary Adv the tPRF advantage against $\tilde{F}$ is defined as

$$\mathsf{Adv}_{\tilde{F}}^{\mathsf{tprf}} \stackrel{\mathrm{def}}{=} \left| \Pr\left[\mathsf{k} \xleftarrow{\$} \mathcal{K} : \mathsf{Adv}^{\tilde{F}_{\mathsf{k}}(\cdot,\cdot)} \Rightarrow 1 \right] - \Pr\left[\widetilde{\rho} \xleftarrow{\$} \mathrm{Rand}(\mathcal{T} \times \mathcal{X}, \mathcal{Y}) : \mathsf{Adv}^{\widetilde{\rho}(\cdot,\cdot)} \Rightarrow 1 \right] \right|.$$

When considering PRFs with domain and co-domain such that $\mathcal{X} = \mathcal{Y} = \mathbb{F}_p$, we shall write $\mathsf{E}_{\mathsf{k}}(m)$, without requiring $\mathsf{E}_{\mathsf{k}}(\cdot)$ being a permutation, and, in the tweakable setting, for the special case that $\mathcal{X} = \mathcal{Y} = \mathbb{F}_p$ we introduce the notation $\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,N}(m)$, with the tweak $(i, N) \in \mathbb{F}_p^{\times} \times \mathbb{F}_p = \mathcal{T}$. Given a PRF $\mathsf{E}_{\mathsf{k}}(\cdot)$ we can create a tweakable PRF $\widetilde{\mathsf{E}}_{\mathsf{k}}^{\cdot,\cdot}(\cdot)$ using Rogaway's XE framework [Rog04] adapted to $\mathbb{F}_p$ by setting $\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,N}(m) = \mathsf{E}_{\mathsf{k}}(m + (i \cdot \mathsf{E}_{\mathsf{k}}(N)))$, for $i \neq 0$, as in Figure 7.1.

Theorem 20. *Let* E *be any PRF with* $\mathcal{X} = \mathcal{Y} = \mathbb{F}_p$ *and let* $\widetilde{\mathsf{E}}$ *be the tweakable PRF with tweak space* $\mathbb{F}_p^{\times} \times \mathbb{F}_p$ *as defined in Figure 7.1. Let* Adv *be an arbitrary adversary against the PRF advantage of* $\widetilde{\mathsf{E}}$ *making at most* q *queries to its oracle, then there exists a similarly resourced adversary* $\mathcal{B}$ *against* E *satisfying*

$$\mathsf{Adv}_{\widetilde{\mathsf{E}}}^{\mathsf{tprf}}(\mathsf{Adv}) \leq \mathsf{Adv}_{\mathsf{E}}^{\mathsf{prf}}(\mathcal{B}) + 3q^2/2p\,.$$

Oracle $\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,N}(m)$:

1: **if** $N \notin \mathcal{N}$ **then**
2: $\quad \mathcal{N} \xleftarrow{\cup} N$
3: $\quad$ **if** $N \in \mathcal{X}$ **then**
4: $\quad\quad$ set $\mathsf{bad}_{\mathsf{xn}}$
5: $\quad\quad \boxed{L_N \xleftarrow{\$} \mathbb{F}_p}$
6: $\quad$ **else**
7: $\quad\quad L_N \xleftarrow{\$} \mathbb{F}_p$
8: $X \leftarrow m + i \cdot L_N$
9: **if** $X \in \mathcal{X}$ **then**
10: $\quad$ set $\mathsf{bad}_{\mathsf{xx}}$
11: $\quad \boxed{L_X \xleftarrow{\$} \mathbb{F}_p}$
12: **else if** $X \in \mathcal{N}$ **then**
13: $\quad$ set $\mathsf{bad}_{\mathsf{nx}}$
14: $\quad \boxed{L_X \xleftarrow{\$} \mathbb{F}_p}$
15: **else**
16: $\quad L_X \xleftarrow{\$} \mathbb{F}_p$
17: $\mathcal{X} \xleftarrow{\cup} X$
18: **return** L_X

Figure 7.2: Games G_2 and G_3, where only G_3 includes the boxed statements.

Proof. We closely follow Rogaway's original proof for XE [Rog04, Theorem 7], making only minimal changes to adapt from the $\mathbb{F}_2^n$ case to the more forgiving $\mathbb{F}_p$ case and to take advantage of operating on functions, as opposed to permutations. The latter allows us to avoid two PRP–PRF switches in the proof, resulting in a slightly tighter bound as a result. As is customary, without loss of generality we assume the adversary does not repeat queries.

Let game G_0 be the original game where an adversary has access to $\widetilde{\mathsf{E}}$ that calls E in the background and let G_1 be the game where the internal calls to E are replaced by calls to a random function. This standard hop incurs the tPRF advantage, that is

$$\Pr\left[\,\mathsf{Adv}^{G_0} \Rightarrow 1\,\right] - \Pr\left[\,\mathsf{Adv}^{G_1} \Rightarrow 1\,\right] \leq \mathsf{Adv}_{\mathsf{E}}^{\mathsf{prf}}(\mathcal{B})\,,$$

where $\mathcal{B}$ is the adversary that runs Adv and answers the latter's queries by evaluating $\widetilde{\mathsf{E}}$ using calls to its own oracle. The number of queries $\mathcal{B}$ makes is at most twice that of Adv and the runtime overhead is limited to a few finite field operations per query.

Next consider the games G_2 and G_3 as depicted in Figure 7.2. Game G_2 is identical to G_1 where the internal random function has been implemented using lazy sampling. By inspection, games G_2 and G_3 are identical until bad, and game G_3 is identical to providing access to a random tweakable function,

Oracle $(N, (i_j, m_j)_j)$:

1: $L_N \xleftarrow{\$} \mathbb{F}_p$
2: **for** j **do**
3: $\quad X \leftarrow m_j + i_j L_N$
4: $\quad$ **if** $X \in \mathcal{X}$ **then**
5: $\quad\quad$ set $\mathsf{bad_{xx}}$
6: $\quad$ **else if** $X \in \mathcal{N}$ **then**
7: $\quad\quad$ set $\mathsf{bad_{xn/nx}}$
8: $\quad \mathcal{X} \xleftarrow{\cup} X$

Figure 7.3: Bounding bad; here $\mathcal{N}$ is initialized to contain all N to be queried.

hence

$$\begin{aligned}\mathsf{Adv}^{\mathsf{tprf}}_{\widetilde{\mathsf{E}}}(\mathsf{Adv}) &\leq \Pr\left[\,\mathsf{Adv}^{G_0} \Rightarrow 1\,\right] - \Pr\left[\,\mathsf{Adv}^{G_3} \Rightarrow 1\,\right] \\ &= \Pr\left[\,\mathsf{Adv}^{G_0} \Rightarrow 1\,\right] - \Pr\left[\,\mathsf{Adv}^{G_1} \Rightarrow 1\,\right] + \Pr\left[\,\mathsf{Adv}^{G_2} \Rightarrow 1\,\right] - \Pr\left[\,\mathsf{Adv}^{G_3} \Rightarrow 1\,\right] \\ &= \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{E}}(\mathcal{B}) + \Pr\left[\,\mathsf{Adv\ sets\ bad\ in}\ G_3\,\right]\ .\end{aligned}$$

What remains to bound is the probability Adv sets bad in G_3. The first observation here is that in G_3 the oracle's output is independent of the input, which allows us to consider non-adaptive adversaries only: given a sequence of queries (N_j, i_j, m_j) what is the probability that the lazy sampling results in bad being set?

Without loss of generality, we assume that the queries are sorted on their first component. This allows us to track the probability that one of the bad events happens as L_N gets sampled (see Figure 7.3). Furthermore, we rely on $i_j \in \mathbb{F}_p^\times$ which means it has a multiplicative inverse so that, for a given triple (X, m_j, i_j), it holds that

$$\Pr\left[\ L_n \xleftarrow{\$} \mathbb{F}_p : X = m_j + i_j L_n\ \right] = \Pr\left[\ L_n \xleftarrow{\$} \mathbb{F}_p : L_n = i_j^{-1}(X - m_j)\ \right] = 1/p\ .$$

Bounding the probability that in the **for** loop $\mathsf{bad_{xx}}$ gets set is then easy: by using a union bound over $X \in \mathcal{X}$ this equals $|\mathcal{X}|/p$. Similarly, the probability that $\mathsf{bad_{xn/nx}}$ gets set is at most $|\mathcal{N}|/p$. The overall probability can then be bounded by union bound by

$$\Pr\left[\,\mathsf{Adv\ sets\ bad}\,\right] \leq \sum_{l=1}^{q} (q + l - 1)/p \leq \frac{3q^2}{2p}\ ,$$

where we used $|\mathcal{X}| \leq l - 1$ and $\mathcal{N} \leq q$.

□ □

Pseudorandom functions can double as message authentication codes (MACs). While it is possible to consider MACs in a more general context than PRFs (for instance allow probabilistic tagging and introduce a separate verification function) and with a weaker unforgeability security notion, we will

treat MACs as a deterministic keyed function $\mathsf{MacGen} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ whose security notion coincides with that of a PRF.

We are primarily interested in pPMAC, which is an adaptation of PMAC—or more accurately of PMAC1 [Rog04]—to $\mathbb{F}_p$. It can be considered a domain extension of $\mathsf{E}_\mathsf{k}(\cdot)$ with domain (and codomain) $\mathbb{F}_p$ to MacGen with domain $\mathcal{X} = \mathbb{F}_p^*$ and codomain $\mathcal{Y} = \mathbb{F}_p$, where $\mathbb{F}_p^*$ denotes the arbitrary length strings of $\mathbb{F}_p$ elements, though there will be an effective upper bound on the maximum length we can cope with.

7.3.2 Authenticated Encryption

For simplicity, we only consider AE schemes without associated data, although we are confident that the techniques we develop in later sections apply in equal measure to AEAD schemes. An AE scheme is defined by two algorithms $(\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}, \mathsf{AE}\text{-}\mathcal{D}_\mathbb{F})$, where we use the subscript $\mathbb{F}$ to denote that both the input and output will be vectors of elements in a finite field (typically $\mathbb{F} = \mathbb{F}_p$ due to their relevance to MPC applications).

The encryption $\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}$ always takes as input a key $\mathbf{k}$, a message $\mathbf{m} \in \mathbb{F}^*$, and an additional input $N \in \mathbb{F}$ or $\mathsf{IV} \in \mathbb{F}$, where the difference in notation refers to the distinction between nonce-based security (N) versus IV-based security (IV). The output consists of a ciphertext $\mathbf{c} \in \mathbb{F}^*$ and a separate tag $T \in \mathbb{F}$. Note that the bold notation $\mathbf{k}, \mathbf{m}$ represents a set which can possible have more than one field $\mathbb{F}$ element in it, for eg. to authenticate a message one needs $\mathbf{k} = (k, k')$ where k is used for encryption while k' is used to compute the tag. Thus we have that

$$(\mathbf{c}, T) \leftarrow \mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}(\mathbf{k}, N, \mathbf{m})$$

with N possibly replaced by IV depending on the context. Henceforth we will assume that the scheme is length-preserving, meaning that $|\mathbf{c}| = |\mathbf{m}|$ irrespective of $\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}$'s inputs. The decryption function $\mathsf{AE}\text{-}\mathcal{D}_\mathbb{F}$ receives as input a key $\mathbf{k}$, and $(N, \mathbf{c}, T)$ (or $(\mathsf{IV}, \mathbf{c}, T)$) and outputs a purported plaintext $\mathbf{m} \in \mathbb{F}^*$ or $\perp$ if the input is deemed invalid. We impose both correctness and tidiness [NRS14] on the pair $(\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}, \mathsf{AE}\text{-}\mathcal{D}_\mathbb{F})$, so that

- (correctness) for all inputs $\mathsf{AE}\text{-}\mathcal{D}_\mathbb{F}(\mathbf{k}, N, \mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}(\mathbf{k}, N, \mathbf{m})) = \mathbf{m}$ and
- (tidiness) for all inputs, if $\mathsf{AE}\text{-}\mathcal{D}_\mathbb{F}(\mathbf{k}, N, c) = m \neq \perp$, then $\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}(\mathbf{k}, N, m) = c$

which implies that as functions $\mathsf{AE}\text{-}\mathcal{D}_\mathbb{F}$ is completely defined by $\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}$.

Our choice for a separate tag in the syntax is customary in part of the literature and preempts later constructions where there is a clear authentication tag, although especially for encode-then-encipher constructions the split would be artificial.

Security for an AE scheme is defined by two notions: **PRIV** and **AUTH**. Informally, the first property defines what it means for a ciphertext to keep the message hidden, whereas the second defines what it means for the ciphertext to be authenticated. The PRIV adversary works as a basic IND-CPA adversary against the encryption scheme. In particular the adversary AE has access to an encryption oracle implementing either $\mathsf{AE}\text{-}\mathcal{E}_\mathbb{F}$ for the underlying AE scheme, or an oracle $\$$ which just outputs

random finite field elements of the correct length. The adversary will query this oracle with $(N_i, \mathbf{m}_i)$ (resp. just $\mathbf{m}_i$) in the nonce-based (resp. IV-based) setting, to obtain tuple $(\mathbf{c}_i, T_i)$ (resp. $(\mathbf{c}_i, T_i)$ plus the IV_i chosen by the experiment). The only constraint on the adversary's calls to this oracle come in the nonce-based setting, where the calls must be nonce-respecting, i.e. if $i \neq j$ then $N_i \neq N_j$. For an adversary Adv we let q denote the number of queries and σ_M the total length of all messages queried to the oracle, so $\sigma_M \stackrel{\text{def}}{=} \sum_{i=1}^{q} |\mathbf{m}_i|$.

The adversary's goal is to distinguish between a genuine encryption oracle (which also outputs the IV) and one that just outputs random values $(\mathbf{c}_i, T_i)$ (resp. $(\mathsf{IV}_i, \mathbf{c}_i, T_i)$) of the corresponding length. Thus we define the advantage of an adversary as follows:

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{AE}[\mathbb{F}]}(\mathsf{Adv}) \stackrel{\text{def}}{=} \left| \Pr\left[\mathbf{k} \stackrel{\$}{\leftarrow} \mathcal{K} : \mathsf{Adv}^{\mathsf{AE}\text{-}\mathcal{E}_{\mathbb{F}}} \Rightarrow 1 \right] - \Pr\left[\mathsf{Adv}^{\$} \Rightarrow 1 \right] \right| .$$

An AUTH adversary Adv can access both oracles $\mathsf{AE}\text{-}\mathcal{E}_{\mathbb{F}}$ and $\mathsf{AE}\text{-}\mathcal{D}_{\mathbb{F}}$, where it can make q encryption queries and q_v decryption queries. We denote the encryption queries by $(N_1, \mathbf{m}_1), \ldots, (N_q, \mathbf{m}_q)$ (resp. $\mathbf{m}_1, \ldots, \mathbf{m}_q$), and, as above, we require that they are nonce-respecting in the nonce-based setting. Decryption queries are denoted by $(N'_1, \mathbf{c}'_1, T'_1), \ldots, (N'_{q_v}, \mathbf{c}'_{q_v}, T'_{q_v})$, (resp. $(\mathsf{IV}'_1, \mathbf{c}'_1, T'_1), \ldots,$ $(\mathsf{IV}'_{q_v}, \mathbf{c}'_{q_v}, T'_{q_v})$); there are no restrictions on what can be passed to the decryption oracle by the adversary. We let σ_M be as above and additionally use σ_C to denote the total length of the ciphertexts passed to the decryption oracle, so $\sigma_{C'} \stackrel{\text{def}}{=} \sum_{i=1}^{q_v} |\mathbf{c}'_i|$. The adversary wins, or is said to have forged a message, if it passes a query to $\mathsf{AE}\text{-}\mathcal{D}_{\mathbb{F}}$ which does not return $\perp$ and which was not obtained from a query to $\mathsf{AE}\text{-}\mathcal{E}_{\mathbb{F}}$. Let this query be denoted by $(N'_{i^*}, \mathbf{c}'_{i^*}, T'_{i^*})$ (resp. $(\mathsf{IV}'_{i^*}, C'_{i^*}, T'_{i^*})$) for some $i^* \in \{1, \ldots, q_v\}$. In other words, the adversary wins if there is no $j \in \{1, \ldots, q\}$ for which $\mathbf{c}'_{i^*} = \mathbf{c}_j$, $T'_{i^*} = T_j$ and $N'_{i^*} = N_j$ (resp. $\mathsf{IV}'_{i^*} = \mathsf{IV}_j$). We define the adversary's advantage by

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{AE}[\mathbb{F}]}(\mathsf{Adv}) \stackrel{\text{def}}{=} \Pr\left[\mathbf{k} \stackrel{\$}{\leftarrow} \mathcal{K} : \mathsf{Adv}^{\mathsf{AE}\text{-}\mathcal{E}_{\mathbb{F}}, \mathsf{AE}\text{-}\mathcal{D}_{\mathbb{F}}} \text{ forges} \right] .$$

7.3.3 MPC Model

As most of this thesis, we focus on generic MPC with secret sharing with an arbitrary number of parties where the inputs are elements in a finite field $\mathbb{F}_p$. Recall that if parties hold a secret shared value of x, this is denoted as $[\![x]\!]$. This secret shared value can be revealed to all parties by a process called *opening* in which parties broadcast their shares to compute the value in clear.

We assume that the parties hold a sharing $[\![\mathbf{k}]\!]$ of some symmetric primitive's key as this can be generated cheaply by generating a random authenticated secret. In this chapter we will assume that we have access to some preprocessed shared random bits $[\![b]\!]$ where $b \in \{0, 1\}$ and shared random squares $([\![r]\!], [\![r^2]\!])$ where $r \in \mathbb{F}_p$. These random squares and bits can be obtained by calling **RandomSquare** or **RandomBit** within $\mathcal{F}_{\mathsf{ABB}}$ functionality.

To measure the MPC complexity of a function we concentrate on the online phase although we give metrics for the preprocessing phase as well. The function evaluation will require parties to both perform local computations and to communicate with one another (this holds both for the offline and

online phase by the way). The local computation is usually mostly ignored when considering MPC complexity, instead the focus is strongly on the communication. This communication is performed in *rounds*, where all parties can send as much data to any other party as they wish, based on the information they have received in previous rounds. The two main metrics for the communication are the round complexity and the number of openings (how many secret shared elements are opened to elements in the clear). Unless the amount of data communicated in a single round floods the network capacity, the round complexity strongly determines the latency required to compute the desired function securely. The number of openings is a strong indicator of throughput as it indicates how much data the network has to accommodate. Openings themselves take one round, but in one round many openings could potentially be performed in parallel.

The main operations over $\mathbb{F}_p$ are addition and multiplication. Both addition of secret shared values and scalar multiplication by clear values can be performed locally (i.e. without interaction) and are deemed efficient: neither contributes to the number of rounds or openings of the overall computation. On the other hand, multiplication of secret shared values does require a round of interaction between the players: it requires two *openings*, which can be done in parallel thus consuming one round. Additionally, the multiplication will consume one of the preprocessed Beaver Triples. A value can also be squared by consuming a shared random square; this only requires one opening, yet still takes one round of interaction.

To reduce the number of online rounds when optimizing MPC, the main techniques are moving input-independent computation to the preprocessing stage, parallelizing computations during the online stage, and performing early openings to allow cheaper, subsequent operations on clear instead of shared elements. We will see examples of all three techniques in what follows.

7.3.4 Two Candidate PRFs for MPC

7.3.4.1 MiMC

Minimal Multiplicative depth Cipher (MiMC) is a cipher which works in both binary and prime fields, though we will only consider the prime field variant $\mathsf{MiMC} : \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$ with $p \equiv 2 \bmod 3$ [AGR+16, GRR+16]. The cipher is a classical iterated Even–Mansour cipher using a simple algebraic round permutation inspired by a cipher by Nyberg and Knudsen [NK95]. When incorporating the key addition prior to applying the permutation, the round function is defined by

$$F_i(x) = (x + \mathbf{k} + c_i)^3 \,,$$

where the $c_i \in \mathbb{F}_p$ are randomly chosen round constants that "are fixed once and can be hard-coded into the implementation" [AGR+16]. This round function is iterated r times, with a final key addition for whitening purposes to yield

$$F_{\mathsf{MiMC}}(\mathbf{k}, x) = (F_{r-1} \circ F_{r-2} \circ \ldots \circ F_0)(x) + \mathbf{k} \,.$$

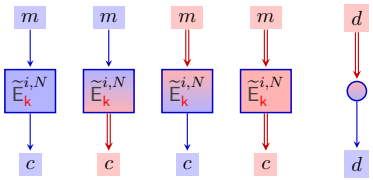

Figure 7.4: Pictorial notation to define processing of open versus shared data.

Originally, $r = \lceil \log_3 p \rceil$ rounds were suggested for security [AGR+16, Section 5]. For a prime p of 128 bits this would lead to $r = 82$ rounds for full keyed-permutation security. However, if the attacker only has access to a limited number $n \leq 2^{115}$ of plaintext/ciphertext pairs then the number of rounds can be reduced to $r = 73$ [AGR+16, Section 4.3].

7.3.4.2 Legendre Symbol Leg

In 1988 Damgård proposed the use of the Legendre symbol to yield a PRF with input and output in $\mathbb{F}_p$ [Dam90]. Although at that time there was no security proof that the resulting PRF is secure, several reductions were made later to the decision shifted Legendre symbol (DSLS) problem [vHI03, Cv07].

The PRF $\mathsf{Leg}_{\mathsf{bit}} : \mathbb{F}_p \times \mathbb{F}_p^* \longrightarrow \{0,1\}$ is initialized with a random key $k \xleftarrow{\$} \mathbb{F}_p$. To evaluate it on input x, we simply call the Legendre symbol on $k + x$ and normalise the output to be in $\{0,1\}$ as opposed to $\{-1,1\}$. It is known that $\mathsf{Leg}_{\mathsf{bit}}$ is a pseudorandom function if there is no probabilistic polynomial time adversary to solve DSLS efficiently [GRR+16].

This function can be extended to produce a field element by selecting a vector of keys $\mathsf{k} = (k_i) \in \mathbb{F}_p^L$ and by computing $\mathsf{Leg}(x) = \sum_{i=0}^{L-1} 2^i \cdot \mathsf{Leg}_{\mathsf{bit}}(k_i, x) \pmod p$, for some value L. Assuming Leg outputs an unbiased random bit, for general p one still needs to select $L = \lceil 2 \cdot \log_2 p \rceil$ to ensure statistical closeness to the uniform distribution over $\mathbb{F}_p$, however if p is chosen sufficiently close to a power of two then one can relax to $L = \lceil \log_2 p \rceil$ [GRR+16].

7.4 MPC Complexity of MiMC and Leg

When evaluating a tweakable PRF in an MPC setting, the key will always be secret and the tweak will always be in the clear, but whether the main input and output are held in the clear or are secret shared will depend on the application. Consequently, when optimizing MiMC and Leg we need to make a distinction between four cases, depending on whether the input and/or output is held in the clear or is secret shared. These four variants we will denote by the notation in Figure 7.4 in subsequent diagrams, with an opening operation denoted by a coloured circle (red denoting a shared data item, and blue a data item held in the clear).

In prior work on the MPC evaluation of MiMC and Leg, only the fourth and, for Leg only, the third variant were discussed [GRR+16]. As we will see, the other variants are more useful when defin-

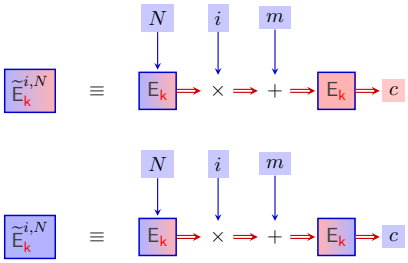

Figure 7.5: Composing a tweakable PRF from a non-tweakable PRF in the case of clear text message/shared output (resp. clear message and clear output).

ing modes of operation, and they can have a remarkably reduced MPC complexity. Another major consideration is whether one is interested in online times subject to standard preprocessing (in which multiplication triples, random squares and random bits are prepared ahead of time), or whether one is interested in key dependent preprocessing for the specific PRF in question, or even tweak (and key) dependent preprocessing for the specific tweakable PRF in question.

In the tweakable context, we can express the design in a similar pictorial way as in Figure 7.5. However, the distinction as to whether the actual message is in the clear disappears, as even in this case the input to the second PRF call is made on shared data due to the need to keep the output of the first PRF shared. Thus we really only have two cases to consider for general PRFs, although specific PRFs may have additional optimizations (see below for one such optimization in the case of the Leg PRF).

7.4.1 MiMC in MPC

Recall the MiMC PRF is defined by

$$\mathsf{E}_\mathsf{k}(x) = F_{\mathsf{MiMC}}(\mathbf{k}, x) = (F_{r-1} \circ F_{r-2} \circ \cdots \circ F_0)(x) + \mathbf{k}\ ,$$

where

$$F_i(x) = (x + k + c_i)^3\ .$$

Grassi et al. [GRR+16] consider two methods for computing MiMC in an MPC setting: $\mathsf{MiMC}^{\mathsf{basic}}$ and $\mathsf{MiMC}^{\mathsf{cube}}$. Given our focus on online times for latency and throughput, only $\mathsf{MiMC}^{\mathsf{cube}}$ is of interest to us; henceforth we will simply call it MiMC.

Using Standard Preprocessing. The computation of $[\![y]\!] \leftarrow [\![x]\!] + [\![k]\!] + c_i$ can always be performed locally, so of interest is the cubing $[\![y^3]\!]$. The standard MPC method to compute MiMC uses a special preprocessed tuple $([\![v]\!], [\![v^2]\!], [\![v^3]\!])$ for which $v \xleftarrow{\$} \mathbb{F}_p$. This preprocessed tuple itself could be computed using squaring and multiplication during the offline phase, or it can be done in the online phase. Given this tuple, to obtain $[\![y^3]\!]$ from $[\![y]\!]$ we open $z = y - v$ to all parties and then compute locally:

$$[\![y^3]\!] = 3 \cdot z \cdot [\![v^2]\!] + 3 \cdot z^2 \cdot [\![v]\!] + z^3 + [\![v^3]\!]\ .$$

Assuming the required r tuples $(\llbracket v \rrbracket, \llbracket v^2 \rrbracket, \llbracket v^3 \rrbracket)$ have been computed during the offline phase, the online phase reduces to one opening and one communication round per cipher round, for a total of r openings and r communication rounds for full evaluation of the cipher. If the tuple is produced in the online phase then we require $3 \cdot r$ openings and $r + 1$ rounds of communication (as all r tuples can be processed simultaneously). In the case where the output is in the clear we require an additional opening and round.

Using Key Dependent Preprocessing. If the input to MiMC is in the clear then a marginal improvement in performance results from the local evaluation of the first round function $F_0(\llbracket k \rrbracket, x) = (x + \llbracket k \rrbracket + c_0)^3$, where we need the values $(\llbracket k \rrbracket, \llbracket k^2 \rrbracket, \llbracket k^3 \rrbracket)$ to be precomputed. As the improvement is only minor over the general method above, we ignore this optimization in what follows.

Using Tweak Dependent Preprocessing. When evaluating the tweakable-PRF on a fixed nonce N known at preprocessing time, say $N = 0$ or $N = 1$, we could precompute the value of $\llbracket M \rrbracket = \mathsf{E}_{\llbracket \mathsf{k} \rrbracket}(N)$. We treat this case as tweak dependent preprocessing, as opposed to key dependent preprocessing, as it assumes knowledge of the application usage of the PRF at preprocessing time.

7.4.2 Leg in MPC

Recall the Leg PRF is defined by

$$\mathsf{E}_{\mathsf{k}}(x) = \mathsf{Leg}(x) = \sum_{i=0}^{L-1} 2^i \cdot \mathsf{Leg}_{\mathsf{bit}}(k_i, x) \pmod p$$

where $\mathsf{k} = \{k_i\}_{i=0}^{L-1}$. When evaluating Leg it suffices to compute the L invocations of $\mathsf{Leg}_{\mathsf{bit}}$ in parallel, followed by local computations for the linear combination of the $\mathsf{Leg}_{\mathsf{bit}}$ outputs into $\mathsf{Leg}(x)$ (after all, multiplications by public constants and additions can be done locally without any interaction between parties). If the final output of Leg should be in the clear, then the $\mathsf{Leg}_{\mathsf{bit}}$ already may be in the clear (implicitly this observation uses that the indistinguishability of Leg follows from that of $\mathsf{Leg}_{\mathsf{bit}}$). Thus the MPC complexity of Leg is equivalent to that of computing $\mathsf{Leg}_{\mathsf{bit}}$ in parallel.

Note, we could use a tweak to also define the extra keys needed in the extension of $\mathsf{Leg}_{\mathsf{bit}}$ to Leg, thus saving storage at the expense of the evaluation of the tweak. Thus the tweakable Leg, would be built out of a tweakable $\mathsf{Leg}_{\mathsf{bit}}$ with two tweak inputs (one for the domain extension to Leg and one for the actual tweak on the Leg function itself).

Using Standard Preprocessing. Grassi et al. [GRR+16] present an efficient method to compute $\mathsf{Leg}_{\mathsf{bit}}$ (Figure 6.6) when the input $\llbracket x \rrbracket$ and output $\llbracket y \rrbracket$ are both secret shared. Grassi et al. already observe that the two steps leading up to the computation of u can be preprocessed and that the step following the computation of u can be performed locally. The computation of u itself takes one round (containing two openings) to compute $\llbracket t \rrbracket \cdot (\llbracket k \rrbracket + \llbracket x \rrbracket)$ and one to open the result. Thus if a fixed quadratic non-residue

α and the data tuples $(\llbracket b \rrbracket, \llbracket t \rrbracket)$ are produced during the offline phase, then the online computation of the PRF $\mathsf{Leg}_{\mathsf{bit}}(\llbracket x \rrbracket)$ will require two rounds of communication and three openings. Without the special preprocessed tuples we would require an extra round and two extra openings.

Grassi et al. additionally suggest an alternative, conceptually easier evaluation when the input is shared but the output should be in the clear: on input $\llbracket x \rrbracket$ take a preprocessed square $\llbracket s^2 \rrbracket$, evaluate $\mathsf{Open}(\llbracket s^2 \rrbracket \cdot (\llbracket k \rrbracket + \llbracket x \rrbracket))$ and output the Legendre symbol of the result. This version still requires two rounds of interaction and three openings, but it only consumes standard preprocessed data.

Using Key Dependent Preprocessing. However, the implementation suggestions by Grassi et al. are not the end of the story. We first investigate what happens when the input x is provided in the clear, and we allow key dependent preprocessing. Our key observation is that if the input x is in the clear, then we can store $(\llbracket b \rrbracket, \llbracket t \rrbracket, \llbracket t \cdot k \rrbracket)$ in the offline phase. This allows simplification of Step 3 from Figure 6.6 to $u \leftarrow \mathsf{Open}(\llbracket t \cdot k \rrbracket + x \cdot \llbracket t \rrbracket)$, which requires only one round of interaction as multiplication by clear values is free. Step 4 proceeds (locally) as before, leading to a shared output.

If both input and output are in the clear, the product $\llbracket s^2 \cdot k \rrbracket$ can be preprocessed and the only online communication remaining is for $\mathsf{Open}(\llbracket s^2 \cdot k \rrbracket + x \cdot \llbracket s^2 \rrbracket)$, namely one round and one opening. The advantage of this method over the one with shared output is a reduction in the consumption of offline material. However, in our tweakable PRF setting we see this optimization is never used.

Figure 7.6 presents a method to compute Leg as a whole for key dependent preprocessing of the tweakable cipher when presented with a fresh value N. The method presented works for a shared input $\llbracket x \rrbracket$, requiring multiplications in Step 2b. These can be done in parallel with the openings of Step 2a, thus for a shared input, the online costs amounts to two rounds of interaction and $3L + 1$ openings. If x is clear, then Step 2b can be performed locally, reducing the total number of openings to $L + 1$; the number of rounds remains 2.

For more complicated calculations, such as re-use of the same N in a future *sequential* call to the tweakable PRF, some pipelining might be feasible. For instance, the respective Steps 2a can still be performed in parallel. However, the gains over a straightforward approach—treating the sequential composition of two tweakable PRF calls as three sequential PRF calls—are not worth the significant increase in consumption of preprocessed data. Whereas standard preprocessing only precomputes $O(L)$ elements, for Figure 7.6 we need to preprocess $O(L^2)$ elements instead. Due to the high preprocessing cost for relatively marginal on-line gains, we discard the method of Figure 7.6 for the remainder of this thesis.

Using Tweak Dependent Preprocessing. Recall that we adapted XE-tweaking of the form

$$\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,N}(m) = \mathsf{E}_{\mathsf{k}}(m + (i \cdot \mathsf{E}_{\mathsf{k}}(N))) \text{ , for } i \neq 0 \text{ .}$$

Due to the linearity of Leg as a function of $\mathsf{Leg}_{\mathsf{bit}}$, we are essentially interested in the evaluation of

$$\mathsf{Leg}_{\mathsf{bit}}(m + (i \cdot \mathsf{E}_{\mathsf{k}}(N))) \text{ ,}$$

Let α be a fixed, quadratic non-residue modulo p and $[\![k_i]\!]$ the shared secret key (for position i)
Preprocess: For each future evaluation prepare tuples as follows:

1. For $j \in \{1, \ldots, L\}$
   - Take random squares $[\![s_j^{(n)2}]\!]$ and random bits $[\![b_j^{(n)}]\!]$.
   - $[\![t_j^{(n)}]\!] \leftarrow [\![s_j^{(n)2}]\!] \cdot ([\![b_j^{(n)}]\!] + \alpha \cdot (1 - [\![b_j^{(n)}]\!]))$
   - $[\![(tk)_i^{(n)}]\!] \leftarrow [\![t_i^{(n)}]\!] \cdot [\![k_i]\!]$
2. For $i \in \{1, \ldots, L\}$
   - Take random squares $[\![s_i^{(x)2}]\!]$ and random bits $[\![b_i^{(x)}]\!]$.
   - $[\![t_i^{(x)}]\!] \leftarrow [\![s_i^{(x)2}]\!] \cdot ([\![b_i^{(x)}]\!] + \alpha \cdot (1 - [\![b_i^{(x)}]\!]))$
   - $[\![(tk)_i^{(x)}]\!] \leftarrow [\![t_i^{(x)}]\!] \cdot [\![k_i]\!]$
3. For $i, j \in \{1, \ldots, L\}$
   - $[\![(tb)_{ij}]\!] \leftarrow [\![t_i^{(x)}]\!] \cdot [\![b_j^{(n)}]\!]$
4. Output all the shares

Eval: To evaluate Leg on input $[\![x]\!]$ with key $[\![k]\!]$ and tweaks i and N, leading to shared output.

1. Retrieve a preprocessed tuple.
2. For $i = j \in \{1, \ldots, L\}$
   a) $v_j \leftarrow \mathsf{Open}([\![(tk)_j^{(n)}]\!] + N \cdot [\![t_j^{(n)}]\!])$
   b) $[\![xt_i^{(x)}]\!] \leftarrow [\![x]\!] \cdot [\![t_i^{(x)}]\!]$
3. For $i \in \{1, \ldots, L\}$
   - Locally compute $[\![(tL)_i]\!] \leftarrow \sum_{j=1}^{L} 2^{j-1} \cdot \left(\left(\frac{v_j}{p}\right) \cdot (2 \cdot [\![(tb)_{ij}]\!] - [\![t_i^{(x)}]\!]) + [\![t_i^{(x)}]\!]\right)$
   - $u_i \leftarrow \mathsf{Open}([\![(tk)_i^{(x)}]\!] + [\![xt_i^{(x)}]\!] + i \cdot [\![tL_i]\!])$
   - Locally compute $[\![y_i]\!] \leftarrow \left(\left(\frac{u_i}{p}\right) \cdot (2 \cdot [\![b_i^{(x)}]\!] - 1) + 1\right)/2$
4. Output $[\![y]\!] \leftarrow \sum_{i=1}^{L} 2^{i-1} \cdot [\![y_i]\!]$

Figure 7.6: Computing the tweakable Leg PRF with shared input, fresh N-tweak, and shared output.

in a number of scenarios, depending on whether the input m, resp. output, are clear or shared, and whether N is fixed or fresh (we will always assume i to be fresh and in the clear, and N to be in the clear).

For the scenario with a clear input m and a shared output, Figure 7.7 presents a method to compute $\mathsf{Leg}_{\mathsf{bit}}$, when the N part of the tweak is fixed (and hence can be preprocessed). This method requires in the online phase only a single round of openings. In the case where m is shared, one can save preprocessing $[\![t \cdot k]\!]$ and compute the second line of the evaluation method by $u \leftarrow \mathsf{Open}([\![t]\!] \cdot ([\![k]\!] + [\![m]\!]) + i \cdot [\![t \cdot M]\!]))$; this requires an additional round of interaction and an additional two openings.

7.4.3 Summary

It is clear the design choices for implementation depend very much on how much specialised preprocessing one wants to perform. In the rest of this paper we restrict ourselves to the case where we allow

Let α be a fixed, quadratic non-residue modulo p and $[\![k]\!]$ the shared secret key.

Preprocess: Assume $[\![M]\!] \leftarrow \mathsf{E}_{[\![k]\!]}(N)$ has already been computed. Then for each future evaluation prepare tuples as follows:

1. Take a random square $[\![s^2]\!]$ and a random bit $[\![b]\!]$.
2. $[\![t]\!] \leftarrow [\![s^2]\!] \cdot ([\![b]\!] + \alpha \cdot (1 - [\![b]\!]))$
3. $[\![t \cdot k]\!] \leftarrow [\![t]\!] \cdot [\![k]\!]$
4. $[\![t \cdot M]\!] \leftarrow [\![t]\!] \cdot [\![M]\!]$
5. Output $([\![b]\!], [\![t]\!], [\![t \cdot k]\!], [\![t \cdot M]\!])$

Eval: To evaluate $\mathsf{Leg}_{\mathsf{bit}}$ on input m with key $[\![k]\!]$ and tweaks i and N, leading to shared output.

1. Retrieve a preprocessed tuple $([\![b]\!], [\![t \cdot k]\!], [\![t \cdot M]\!])$
2. $u \leftarrow \mathsf{Open}([\![t \cdot k]\!] + m \cdot [\![t]\!] + i \cdot [\![t \cdot M]\!]))$
3. Output $[\![y]\!] \leftarrow \left(\left(\frac{u}{p}\right) \cdot (2 \cdot [\![b]\!] - 1) + 1\right)/2$

Figure 7.7: Computing the tweakable $\mathsf{Leg}_{\mathsf{bit}}$ PRF with clear input, fixed N-tweak, and shared output.

key-dependent, but not tweak dependent preprocessing. In this context our tweakable PRF this is then produced via our non-tweakable PRF via the methodology given in Figure 7.5. Note, when the message in this diagram is given in the clear, this makes *no difference* to the execution of the second PRF call, as the input is already in shared form.

In addition, any second call to the tweakable PRF with the same value N in the tweak can be done without the need to call the first PRF again. When the output of the tweakable PRF is to be returned in an open form, the second PRF call can be performed more efficiently in the case of Leg by using the key-dependent preprocessing variant. This leads to the online costs given in Table 7.1.

7.5 Encrypt-then-MAC in Characteristic p

In this section we examine an Encrypt-then-MAC paradigm to obtain AE for messages/ciphertexts consisting of vectors in $\mathbb{F}_p$. To enable the efficient computation, we select a nonce-based IND-CPA encryption mode which is highly parallel (specifically a modification of CTR mode). For the MAC algorithm we present two possibilities, a Hash-then-MAC method (which is suitable as we always MAC clear data), as well as a new MAC algorithm which we call pPMAC. Here pPMAC is the obvious port of PMAC from binary fields to the field $\mathbb{F}_p$, where we examine the PMAC proof to ensure that the scheme is still secure.

7.5.1 Encrypt-then-MAC

The encrypt-then-MAC paradigm originally applied probabilistic encryption followed by authentication of the resulting ciphertext [BN08]. The probabilistic encryption itself only needs to be PRIV or IND-CPA secure. Moving to a nonce-setting is relatively straightforward [NRS14]: assuming one has

	$\widetilde{\mathsf{E}}^{i,N}_{\mathsf{k}}$			$\widetilde{\mathsf{E}}^{i,N}_{\mathsf{k}}$		
	Rnds	Open	Prep	Rnds	Open	Prep
MiMC (SP)	$2 \cdot r + 1$	$6 \cdot r$	$4 \cdot r$	$2 \cdot r + 2$	$6 \cdot r + 1$	$4 \cdot r$
	r	$3 \cdot r$	$2 \cdot r$	$r + 1$	$3 \cdot r + 1$	$2 \cdot r$
MiMC (TP)	r	$3 \cdot r$	$2 \cdot r$	$r + 1$	$3 \cdot r + 1$	$2 \cdot r$
Leg (SP)	4	$10 \cdot L$	$8 \cdot L$	5	$8 \cdot L$	$6 \cdot L$
	2	$5 \cdot L$	$4 \cdot L$	2	$3 \cdot L$	$2 \cdot L$
Leg (KP)	3	$4 \cdot L$	$8 \cdot L$	3	$4 \cdot L$	$6 \cdot L$
	2	$3 \cdot L$	$4 \cdot L$	2	$3 \cdot L$	$2 \cdot L$
Leg (TP)	2	$3 \cdot L$	$2 \cdot L$	3	$3 \cdot L + 1$	$2 \cdot L$

Table 7.1: Summary of costs for our PRFs MiMC and Leg. The first line for each PRF is the cost of the first such tweakable PRF call, and the second is the cost of subsequent PRF tweakable calls with the same N component in the tweak (clearly their is no second line when we use tweakable preprocessing). The values SP, KP, and TP stand for standard preprocessing, key dependent preprocessing and tweak dependent preprocessing. Note the costs when the input message is in the clear are identical to when the input message is in shared form. The preprocessing costs are given in the number of data items needed to be preduced by the prepreprocessing.

a MAC function, one simply needs to combine a nonce based encryption $(\mathsf{Enc}, \mathsf{Dec})$ scheme which is just PRIV (i.e. IND-CPA) secure, and then authenticate the nonce and the obtained ciphertext with a tag generated from a secure MAC function MacGen. This composition corresponds to scheme 'N2' as studied by Namprempre et al. [NRS14]. This scheme is the only one of the four secure schemes (N1 up to N4) that feeds the ciphertext as opposed to the message to the MAC function. As in our context ciphertext is in the clear whereas messages is shared—and we do not believe that the slightly increased parallellism allowed by N1's encrypt-and-MAC approach outweighs this advantage—we opted for this N2 mode.

To obtain a nonce based scheme two variants of CTR mode are possible, either

$$c_i \leftarrow m_i + \widetilde{\mathsf{E}}^{1,N}_{\mathsf{k}}(i) = m_i + \mathsf{E}_{\mathsf{k}}(i + \mathsf{E}_{\mathsf{k}}(N))),$$
$$c_i \leftarrow m_i + \widetilde{\mathsf{E}}^{i,1}_{\mathsf{k}}(N) = m_i + \mathsf{E}_{\mathsf{k}}(N + i \cdot \mathsf{E}_{\mathsf{k}}(1))).$$

The latter variants is preferred as $\mathsf{E}_{\mathsf{k}}(1)$ can be precomputed when allowing key dependent preprocessing; it corresponds to a simplified variant of CTR-in-Tweak [PS16].

To this CTR mode nonce-based IND-CPA encryption we then add authentication via a MAC function. See Figure 7.8, where we use this CTR mode as the underlying encryption scheme and an arbitrary MAC function. In this figure we present the algorithm, making specific reference to what data is shared and what is open. The reader should note that in decryption we need to perform a secure comparison between the input tag (in the clear), and the computed tag (in shared form). This is easily accomplished, by opening the value $[\![r]\!] \cdot ([\![\mathsf{Tag}']\!] - \mathsf{Tag})$, for a random value r from the preprocessing, and comparing the value to zero.

Given a message $[\![\mathbf{m}]\!] = [\![m_1]\!], \ldots, [\![m_\ell]\!]$ for $m_i \in \mathbb{F}_p$ and a pair of keys $[\![\mathbf{k}]\!] = ([\![\mathsf{k}]\!], [\![\mathsf{k}']\!])$ for the PRF $\mathsf{E}_{[\![\mathsf{k}]\!]}(\cdot)$ we define the AE mode CTR+MAC as:

AE-$\mathcal{E}_{\mathbb{F}}([\![\mathbf{k}]\!], N, [\![\mathbf{m}]\!])$:
1: **for** $i = 1, \ell$ **do**
2: $\quad [\![c_i]\!] \leftarrow [\![m_i]\!] + [\![\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,1}(N)]\!]$
3: $\quad$ Open $[\![c_i]\!]$.
4: $\mathbf{c} \leftarrow c_1, \ldots, c_\ell$.
5: $\mathsf{Tag} \leftarrow \mathsf{MacGen}([\![\mathsf{k}']\!], N \| \mathbf{c})$.
6: Return $(\mathbf{c}, \mathsf{Tag})$.

AE-$\mathcal{D}_{\mathbb{F}}([\![\mathbf{k}]\!], N, \mathbf{c}, \mathsf{Tag})$:
1: **for** $i = 1, \ell$ **do**
2: $\quad [\![m_i]\!] \leftarrow c_i - [\![\widetilde{\mathsf{E}}_{\mathsf{k}}^{i,1}(N)]\!]$
3: $[\![\mathbf{m}]\!] \leftarrow [\![m_1]\!], \ldots, [\![m_\ell]\!]$.
4: $[\![\mathsf{Tag}']\!] \leftarrow \mathsf{MacGen}([\![\mathsf{k}']\!], N \| \mathbf{c})$.
5: **if** $[\![\mathsf{Tag}']\!] \neq \mathsf{Tag}$ **then return** $\perp$.
6: Return $[\![\mathbf{m}]\!]$.

Figure 7.8: AE mode CTR+MAC in the nonce-based setting.

The algorithm $\mathsf{pPMAC\text{-}Gen}(\mathsf{k}, \mathbf{m})$ is defined by:
1: Write $\mathbf{m}$ as ℓ finite field elements $m_1, \ldots, m_\ell$ where $m_i \in \mathbb{F}_p$.
2: **if** $\ell \geq p$ **then return** $\perp$.
3: **for** $i = 1, \ell - 1$ **do**
4: $\quad Y_i \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{i,\mathbf{0}}(m_i)$
5: $\Sigma \leftarrow Y_1 + \cdots + Y_{\ell-1} + m_\ell$
6: $\mathsf{Tag} \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{p-1,\mathbf{0}}(\Sigma)$.

Figure 7.9: pPMAC in $\mathbb{F}_p$

7.5.2 The PMAC Algorithm over $\mathbb{F}_p$

The original PMAC algorithm [BR02] operates (after suitable padding) on elements in the finite field $\mathbb{F}_{2^n}$. The algorithm makes use of various constants, which in the original PMAC are taken to be from a Gray code to enable efficient computation. In addition a "large" constant called Huge is defined, which is equal to $1/x$ for x being the formal root of the defining polynomial for the field. The tag is produced by utilizing an encryption function defined by $\mathsf{E}_{\mathsf{k}}(m) : \mathbb{F}_{2^n} \longrightarrow \mathbb{F}_{2^n}$.

PMAC1 [Rog04] is a conceptually simpler version of PMAC that recasts the masked blockcipher calls as direct, tweakable blockcipher ones instead. This abstraction is especially potent when moving to $\mathbb{F}_p$ and using a tweakable PRF. As we will be using $\mathbb{F}_p^\times$ as tweak space, we can set $\mathsf{Huge} = p - 1$ (to be used by the final $\widetilde{\mathsf{E}}$ call) and use tweak i to process message block m_i, for $i \in \{1, \ldots, p-2\}$. Hence our $\mathbb{F}_p$ variant of PMAC1, henceforth referred to as pPMAC, takes in a message which is at most $p-2$ finite field elements long and produces an element of the finite field $\mathbb{F}_p$ as final tag; the precise pPMAC algorithm is given in Figure 7.9.

While the security of PMAC over $\mathbb{F}_{2^n}$ has received ample attention [MM07, DY15, LPSY16], the security for our pPMAC version does not seem to follow directly from prior work. Hence we present Theorem 21 to bound an adversary's distinguishing advantage. Luckily, the proof is a fairly straightforward adaptation of Rogaway's [Rog04, Section 11], where the use of a tweakable PRF instead of a

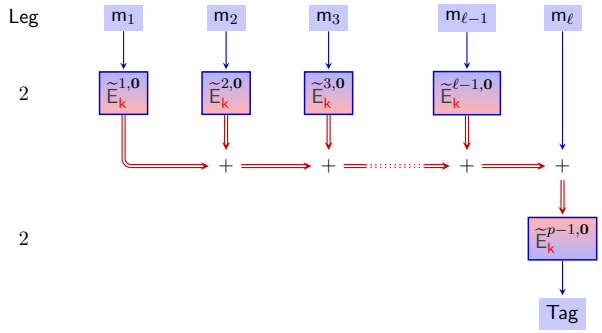

Figure 7.10: Implementing pPMAC in MPC for clear inputs and clear outputs. The number of rounds of interaction for the Leg tweakable PRF using key dependent preprocessing are given to the left.

tweakable blockcipher allows some simplifications and tightening of the bound.

Theorem 21. *Let* Adv *be a PRF-adversary against pPMAC making q queries having a total message length of σ finite field elements. Then there exists an adversary $\mathcal{B}$ attacking $\widetilde{\mathsf{E}}$ making at most $\sigma + q$ oracle queries and running in time comparable to that of* Adv *such that*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{pPMAC}[\widetilde{\mathsf{E}}]}(\mathsf{Adv}) \leq \mathsf{Adv}^{\mathsf{tprf}}_{\widetilde{\mathsf{E}}}(\mathcal{B}) + \frac{q(q-1)}{2p}.$$

Proof. Let G_0 be the original pPMAC game and let G_1 be the game with the keyed $\widetilde{\mathsf{E}}$ replaced by an ideal tweakable random function. Let $\mathcal{B}$ be the adversary against $\widetilde{\mathsf{E}}$ that runs Adv and uses its $\widetilde{\mathsf{E}}$ oracle to evaluate pPMAC for Adv, then

$$\Pr\left[\,\mathsf{Adv}^{G_0} \Rightarrow 1\,\right] - \Pr\left[\,\mathsf{Adv}^{G_1} \Rightarrow 1\,\right] = \mathsf{Adv}^{\mathsf{tprf}}_{\widetilde{\mathsf{E}}}(\mathcal{B})\,,$$

where the number of $\widetilde{\mathsf{E}}$ calls induced by Adv's queries is at most $\sigma + q$ and $\mathcal{B}$'s overhead otherwise is minimal.

Let G_2 be the game where bad is set if two inputs cause colliding final $\widetilde{\mathsf{E}}$ calls (with tweak Huge). As the tweak Huge cannot be used for any other $\widetilde{\mathsf{E}}$ calls, if no such collisions appear we can replace the tag output by a freshly drawn $\mathbb{F}_p$ elements in G_2. Then G_1 and G_2 are identical until bad. Morever, to analyse the probability that Adv sets bad in G_2 we may restrict without loss of generality to non-adaptive adversaries.

For any given pair of distinct queries, there has to be at least one $\widetilde{\mathsf{E}}$ call that is made with distinct inputs (if the messages are identical until the final message block, no collision is possible). For a collision to occur, fix the outputs for all the other message blocks (of this query pair) and one of the distinct message blocks of the colliding pair, then the $\widetilde{\mathsf{E}}$ value (for the corresponding distinct input) has to hit a specific value, which happens only with probability $1/p$. A union bound over all $\binom{q}{2}$ pairs results in the stated bound. □

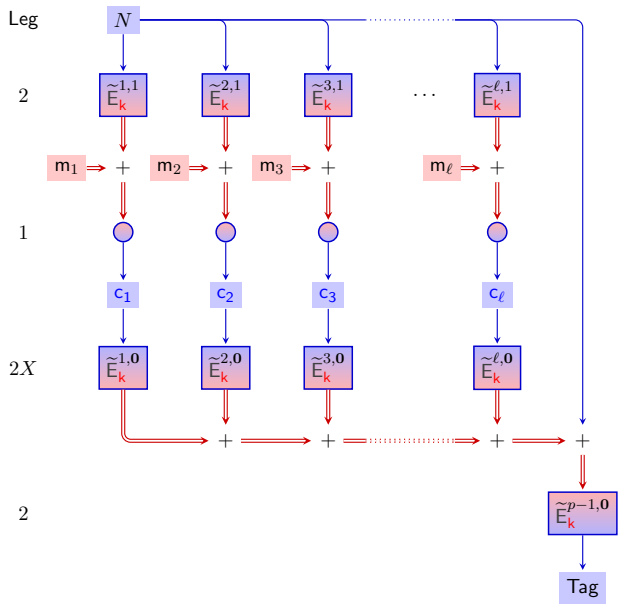

Figure 7.11: CTR+pPMAC Encryption Mode. The number of rounds of interaction for the Leg tweakable PRF using key dependent preprocessing are given to the left.

In an MPC context, we are primarily interested in an implementation where both the message and the tag are available in the clear, as our use case concentrates on the Encrypt-then-MAC setting where pPMAC will be applied on an already opened ciphertext. Figure 7.10 shows the implications for the underlying tweakable PRF calls, in the key dependent preprocessing setting. Note that the 'N'-tweak is fixed to $N = 0$ which allows preprocessing of $[\![M]\!] = \mathsf{E}_{[\![\mathsf{k}]\!]}(0)$ as required in each call to $\widetilde{\mathsf{E}}^{i,\mathbf{0}}_{[\![\mathsf{k}]\!]}(m)$. Also, notice that a naive implementation of the tweakable PRF will result that the remaining PRF applications will be on shared inputs even if m itself is clear, courtesy of $[\![M]\!]$ being shared. When combined with authenticated our CTR mode encryption we obtain an AE method given in Figure 7.11.

7.5.3 Hash-then-MAC

Whilst having pPMAC as a general MAC function might be useful in some other contexts, in terms of creating a MAC for use in an Encrypt-then-MAC AE scheme the pPMAC function is overkill. A simpler alternative, described in Figure 7.12 is to simply hash the clear ciphertext values c_i and then apply a single invocation of the PRF to the output. Note, the N-tweak value can be the same for this PRF call, as for the PRF calls in the CTR mode.

One has to convert the output of the hash function function H into an element modulo p, so it can be passed into our PRF. We require that the value passed to the PRF satisfies the collision resistance property. If H is chosen to be a standard hash function such as SHA-256 or SHA-3, then simply truncating the hash value to $\log_2 p$ bits and treating the result as an integer modulo p will suffice.

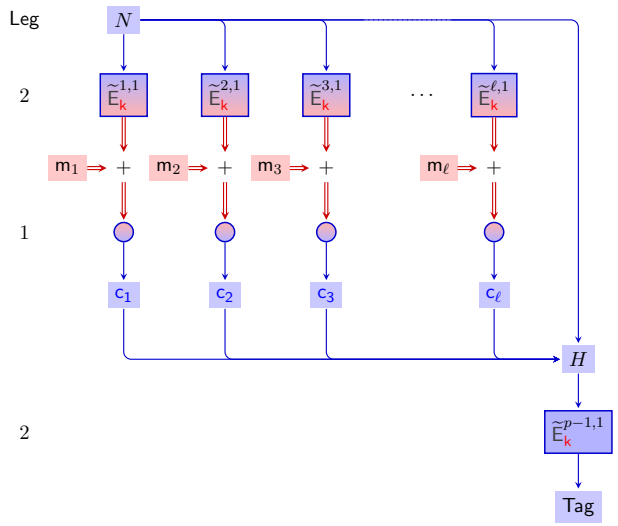

Figure 7.12: CTR and Hash-then-MAC Encryption Mode The number of rounds of interaction for the Leg tweakable PRF using key dependent preprocessing are given to the left.

7.6 OTR in Characteristic p

OTR is a nonce-based AE mode of operation for blockciphers [Min14]. It has a number of advantages that make it eminently suitable for adaptation to an MPC context, in particular its use of the forward direction of the blockcipher only (even for decryption) and its high level of parallellization for both encryption and decryption. The original OTR mode allows the encryption of arbitrary length bitstrings using arbitrary length bitstrings of associated data. In this section we will adapt Minematsu's OTR to encrypt arbitrary vectors of $\mathbb{F}_p$ elements based on a tweakable pseudorandom function, where we discard any associated data. Consequently, much of the complexity of the original OTR, for instance related to padding to some multiple of the blocklength, disappears. Although OTR strictly speaking is a blockcipher mode of operation, Minematsu already presents OTR as a tweakable blockcipher mode of operation instantiated with a specific tweakable blockcipher. Our version of $\mathbb{F}_p$ will be based on this perspective, making use of an $\mathbb{F}_p$ tweakable PRF $\widetilde{\mathsf{E}}$ (which need not need be invertible). The tweaks needed in our $\mathbb{F}_p$ variant are fairly straightforward. This contrast with a relatively complex tweak schedule in the original OTR to avoid colliding masks over the finite field $\mathbb{F}_{2^n}$ (cf. [BS16]). Finally, in order to present a cleaner implementation we removed the final block switch.

Our modified construction is presented in Figure 7.13 and Figure 7.14: encryption takes the key k as well as a nonce $N \in \mathbb{F}_p$ and a message $\mathbf{m} \in \mathbb{F}_p^*$, producing a ciphertext $\mathbf{c} \in \mathbb{F}_p^*$ and a tag $\mathsf{Tag} \in \mathbb{F}_p$, whereas decryption takes the key k as well as $N \in \mathbb{F}_p$, a ciphertext $\mathbf{c}$, and a tag Tag to produce a message $\mathbf{m}$ (or an invalid ciphertext symbol $\perp$). Encryption only works for messages with fewer than $p/2$ elements, with longer messages (and ciphertexts) rejected out of hand.

A diagramatic representation of encryption is given in Figure 7.15, where we additionally highlight

Write $\mathbf{m}$ as ℓ finite field elements $m_1, \ldots, m_\ell$
if $\ell \geq p/2$ **then return** $\perp$.
$\Sigma \leftarrow 0$ where $m_i \in \mathbb{F}_p$.
for $i = 1, \lfloor \ell/2 \rfloor$ **do**
    $c_{2\cdot i-1} \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{2\cdot i-1,N}(m_{2\cdot i-1}) + m_{2\cdot i}$
    $c_{2\cdot i} \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{2\cdot i,N}(c_{2\cdot i-1}) + m_{2\cdot i-1}$
    $\Sigma \leftarrow \Sigma + m_{2\cdot i}$
if ℓ **is odd then**
    $c_\ell \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{\ell,N}(0) + m_\ell$
    $\Sigma \leftarrow \Sigma + m_\ell$
$\mathbf{c} \leftarrow (c_1, \ldots, c_\ell)$
$\mathsf{Tag} \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{-\ell,N}(\Sigma)$
return $(\mathbf{c}, \mathsf{Tag})$

Figure 7.13: The Algorithm $\mathsf{OTR\text{-}E}(N, \mathbf{m})$.

Write $\mathbf{c}$ as ℓ finite field elements $c_1, \ldots, c_\ell$ where $c_i \in \mathbb{F}_p$.
if $\ell \geq p/2$ **then return** $\perp$.
$\Sigma \leftarrow 0$
for $i = 1, \lfloor \ell/2 \rfloor$ **do**
    $m_{2\cdot i-1} \leftarrow c_{2\cdot i} - \widetilde{\mathsf{E}}_{\mathsf{k}}^{2\cdot i,N}(c_{2\cdot i-1})$
    $m_{2\cdot i} \leftarrow c_{2\cdot i-1} - \widetilde{\mathsf{E}}_{\mathsf{k}}^{2\cdot i-1,N}(m_{2\cdot i-1})$
    $\Sigma \leftarrow \Sigma + m_{2\cdot i}$
if ℓ **is odd then**
    $m_\ell \leftarrow c_\ell - \widetilde{\mathsf{E}}_{\mathsf{k}}^{\ell,N}(0)$
    $\Sigma \leftarrow \Sigma + m_\ell$
$\mathbf{m} \leftarrow (m_1, \ldots, m_\ell)$
$\mathsf{Tag}' \leftarrow \widetilde{\mathsf{E}}_{\mathsf{k}}^{-\ell,N}(\Sigma)$
if $\mathsf{Tag}' = \mathsf{Tag}$ **then**
    return $\mathbf{m}$
return $\perp$

Figure 7.14: The Algorithm $\mathsf{OTR\text{-}D}(N, \mathbf{c}, \mathsf{Tag})$.

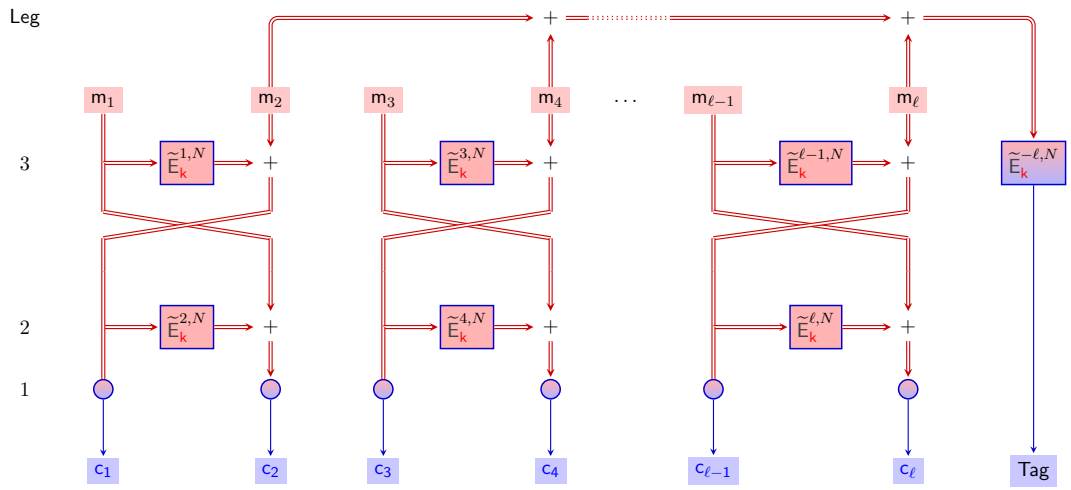

Figure 7.15: The OTR encryption mode. On the left hand side we present the number of rounds of interaction of each stage for the Leg PRF, assuming key dependent preprocessing.

some MPC implementation details. OTR's core encryption component is a two-round Feistel structure, Here one cannot use an output in the clear for the PRF—which would potentially be faster, especially for Leg—as this would be tantamount to using a public string as one-time pad and hence woefully insecure.

Decryption follows in a similar manner, see Figure 7.16. Note that, as for our previous MAC-then-Encrypt constructions, a secure comparison is needed to process the computed tag in the decryption algorithm.

7.6.1 Security of pOTR

Minematsu proved that the original (bit-oriented) OTR is a secure AEAD scheme against nonce-respecting adversaries. Our modified $\mathbb{F}_p$ largely inherits the original properties, but for completeness we provide the relevant theorems and proofs below, where we of course draw heavily on Minematsu's work. For OTR's security analysis Minematsu uses an alternative and conceptually cleaner mode, dubbed $\mathbb{OTR}$ [Min14, Fig. 5], that is based on a tweakable n-bit URF. This mode already matches ours a lot closer, as we use a tweakable PRF and the switch from a tweakable PRF to a tweakable URF is standard (incurring precisely the tweakable PRF advantage). We will ignore the parameter τ (in $\mathbb{OTR}[\tau]$) for the length of tags, as it becomes moot in our $\mathbb{F}_p$ setting. Minematsu additionally introduces $\mathbb{OTR}'$, but in the absence of associated data this mode collapses to $\mathbb{OTR}$. Thus we can safely refer to the security result for $\mathbb{OTR}'$ [Min14, Theorem 3] and its proof [Min14, Appendix A]. The proof for privacy is essentially unchanged (and still straightforward), whereas for authenticity we can simplify the proof considerably as there are fewer cases to consider due to our switch from bitstrings to elements of $\mathbb{F}_p$.

Theorem 22. *Let* Adv *be a PRIV adversary against* OTR *making* q *queries having a total message length of* σ *finite field elements. Then there exists an adversary* $\mathcal{B}$ *attacking* $\widetilde{\mathsf{E}}$ *making at most* $\sigma + q$ *oracle queries and running in time comparable to that of* Adv *such that*

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{OTR}}(\mathsf{Adv}) \leq \mathsf{Adv}^{\mathsf{tprf}}_{\widetilde{\mathsf{E}}}(\mathcal{B})$$

Proof. The first, standard step is to substitute the tweakable PRF with its ideal cousin, the tweakable URF, throughout. An adversary Adv that could distinguish between these two worlds can be turned into a reduction $\mathcal{B}$ that wins the PRF's security game by explicitly evaluating the OTR construction using a tweakable PRF/URF oracle. A counting exercise will show that Adv's queries to the construction induce exactly $\sigma + q$ queries to the underlying tweakable primitive.

With the tweakable URF in place, the key observation is that Adv is nonce-respecting and that, for the encryption of a single message, the tweaks count from $1, \ldots, \ell$ and, as we enforce $\ell < p/2$, the tweak $-\ell$ used for authentication will be distinct from these tweaks (modulo p). Consequently, each tweak (i, N) is used at most once and we can replace the outputs of the tweakable URF with independently and uniformly drawn $\mathbb{F}_p$ elements, ignoring the input. These random $\mathbb{F}_p$ elements act as a one-time pad; inspection shows that all ciphertext elements c_i as well as the Tag are thus affected, making them perfectly indistinguishable from independently and uniformly drawn $\mathbb{F}_p$ elements as desired. □

Theorem 23. *Let* Adv *be an AUTH adversary against* OTR *making* q_e *encryption queries and* q_v *decryption queries, jointly having a total message length of* σ *finite field elements. Then there exists an adversary* $\mathcal{B}$ *attacking* $\widetilde{\mathsf{E}}$ *making at most* $\sigma + q_e + q_v$ *oracle queries and running in time comparable to that of* Adv *such that*

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{OTR}[\widetilde{\mathsf{E}}]}(\mathsf{Adv}) \leq \mathsf{Adv}^{\mathsf{tprf}}_{\widetilde{\mathsf{E}}}(\mathcal{B}) + 3q_v/p.$$

Proof. Again, the first, standard step is to substitute the tweakable PRF with its ideal cousin, the tweakable URF $\widetilde{\mathsf{R}}$, throughout, incurring the same term as in the bound above.

With the tweakable URF in place, Minematsu's original security proof consists of a number of steps. Firstly, we only need to consider an adversary making a single forgery attempt using the decryption oracle, so $q_v = 1$, and then extend it to an arbitrary number of decryptions using a standard guessing argument [BGM04]. Furthermore, without loss of generality, we may assume that Adv makes all its encryption queries before the final decryption query.

We denote the adversary's forgery attempt by $(N', \mathbf{c}', \mathsf{Tag}')$. For the forgery to be counted, it needs to be fresh, that is $(N', \mathbf{c}', \mathsf{Tag}') \neq (N_j, \mathbf{c}_j, \mathsf{Tag}_j)$ for all encryption queries $j \in [1 \ldots q]$. As for each nonce and ciphertext vector there is one unique valid tag (by inspection of the decryption algorithm), we in fact need that $(N', \mathbf{c}') \neq (N_j, \mathbf{c}_j)$ for all j. For the forgery attempt $(N', \mathbf{c}')$, we will use Tag^* to denote the unique valid tag corresponding to it, whereas for all internal variables related to $(N', \mathbf{c}')$ we will use a prime, for instance m'_1 for the first tentative message block and Σ' for the unique input (used by decryption) to the tweakable URF that produces Tag^*.

The adversary's advantage is upper bounded by the maximum probability it can find a forgery $(N', \mathbf{c}', \mathsf{Tag}')$ given an transcript of encryption queries $\{(N_j, \mathbf{m}_j, \mathbf{c}_j, T_j)\}, j \in [1 \dots q]$. Here the maximum is over all possible transcript and the probability is over the 'residual' randomness of the tweakable URF, that is to sample the tweakable URF on values that are needed to evaluate Tag^* and have not yet been sampled during the encryption queries. As is customary, at this stage we can restrict to deterministic, computationally unbounded adversaries.

To upper bound this maximum probability $\mathsf{FP}_{\mathbf{z}}$, we will consider four cases (down from the original's 13): the forgery uses a fresh nonce; the forgery uses a nonce for an encryption query and matches the *even* message length; the forgery uses a past nonce and matches the *odd* message length; and finally the forgery uses a past nonce, but using a different message length.

Case 1: $N' \neq N_j$ *for all* $j \in [1 \dots q]$.

In this case, during decryption the tweak is fresh and hence the Tag^* will be an independent, uniformly random value, so the probability that Tag' is correct satisfies $\mathsf{FP}_{\mathbf{z}} = 1/p$.

Case 2: $N' = N_j$ *with* $|\mathbf{c}'| = |\mathbf{c}_j|$*, even, for some* $j \in [1 \dots q]$.

Let's write $(c'_1, \dots, c'_\ell)$ for $\mathbf{c}'$ and $(c_1, \dots, c_\ell)$ for $\mathbf{c}_j$, so dropping the j index. As $\mathbf{c}' \neq \mathbf{c}$ we know that for some i it holds that $c'_i \neq c_i$, where we will concentrate on the largest such i. As ℓ is even, all ciphertext blocks come with a 'twin' that is processed as part of the same Feistel structure. Let $h = \lceil (i+1)/2 \rceil$, then the indices of the two blocks (i.e. i and its twin) are $2h-1$ and $2h$. For the remainder of this case analysis, we will deal with this structure only, ignoring whether both of only one (and which) of the ciphertext blocks differ between $\mathbf{c}'$ and $\mathbf{c}_j$.

Figure 7.16 provides an overview of how decryption works, where we annotated three special collision events: e_3 corresponds to the event $\mathsf{Tag}^* = \mathsf{Tag}'$, e_2 corresponds to the event that $\Sigma' = \Sigma_j$, and finally e_1 corresponds to the event that $m'_{2\cdot h-1} = m_{2\cdot h-1}$. Our overall strategy will be to bound

$$\mathsf{FP}_{\mathbf{z}} \leq \Pr[e_3] \leq \Pr[e_3 | \neg e_2] + \Pr[e_2 | \neg e_1] + \Pr[e_1] \ ,$$

where all constituent three probabilities turn out to be at most $1/p$, so the sum is at most $3/p$.

Let's start with $\Pr[e_3 | \neg e_2]$. In this case, Tag^* is the result of a fresh query $\widetilde{\mathsf{R}}^{\langle \mathsf{N}, -\ell \rangle}(\Sigma')$, so the probability that it hits the adversary's Tag' is exactly $1/p$.

If, on the other hand, e_2 occurred, then $\mathsf{Tag}^* = \mathsf{Tag}_j$ so if the adversary had indeed set $\mathsf{Tag}' = \mathsf{Tag}_j$, the forgery attempt will be successful. To bound the probability of e_2 occurring, we go back to the point where $m'_{2\cdot h}$ gets added to the checksum. Let's denote with Σ_{h-1} the checksum so far (for the j-the query) and with Σ_h the checksum after adding $m_{2\cdot h}$, with similar primed notation for the values when running decryption on the forgery attempt. Then e_2 occurs iff $\Sigma_h = \Sigma'_h$.

Tracing through the decryption algorithm (and see Figure 7.16) tells us that

$$\begin{aligned} m'_{2\cdot h-1} &= c'_{2\cdot h} - \widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h} \rangle}(m'_{2\cdot h}) \text{ and} \\ m'_{2\cdot h} &= c'_{2\cdot h-1} - \widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h}-1 \rangle}(m'_{2\cdot h-1}) \end{aligned}$$

and therefore that

$$\begin{aligned}
\Sigma_h &= \Sigma'_h \\
\Sigma_{h-1} + m_{2\cdot h} &= \Sigma'_{h-1} + m'_{2\cdot h} \\
\Sigma_{h_1} + m_{2\cdot h} &= \Sigma'_{h-1} + c'_{2\cdot h-1} - \widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h}-1\rangle}(m'_{2\cdot h-1}) \\
\widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h}-1\rangle}(m'_{2\cdot h-1}) &= \Sigma'_{h-1} - \Sigma_{h_1} + c'_{2\cdot h-1} - m_{2\cdot h}
\end{aligned}$$

If e_1 didn't occur, the $\widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h}-1\rangle}(m'_{2'\cdot h-1})$ call is fresh, so the probability it hits the value on the right hand side is exactly $1/p$.

Finally, we are left with the event e_1, namely that $m'_{2\cdot h-1} = m_{2\cdot h-1}$. Although it is not a given that an adversary will be able to turn this event into a forgery, we are generous in granting a win regardless. We will assume that $c'_{2\cdot h-1} \neq c_{2\cdot h-1}$, because otherwise the event e_1 is not possible (by inspection). Our assumption implies that the $\widetilde{\mathsf{R}}^{\langle \mathsf{N}, 2\cdot \mathsf{h}\rangle}(c'_{2\cdot h-1})$ call is fresh, and since it needs to hit a unique value in order for e_1 to occur, e_1 happens with probability $1/p$.

Case 3: $N' = N_j$ *with* $|\mathbf{c}'| = |\mathbf{c}_j|$*, odd, for some* $j \in [1 \ldots q]$.
As before, we write $(c'_1, \ldots, c'_\ell)$ for $\mathbf{c}'$ and $(c_1, \ldots, c_\ell)$ for $\mathbf{c}_j$, so dropping the j index. As $\mathbf{c}' \neq \mathbf{c}$ we know that for some i it holds that $c'_i \neq c_i$, where we will concentrate on the largest such i, where we use a special ordering that makes the final, odd block ($i = \ell$) the smallest. If, under this ordering, "$i > \ell$" there is a difference in one of the blocks used in the Feistel structure and the analysis for ℓ even from above applies. Otherwise if $i = \ell$, the *only* difference occurs for the ℓ^{th} block, so $c'_\ell \neq c_\ell$. Observing that

$$m'_\ell = c'_\ell - \widetilde{\mathsf{R}}^{\langle \mathsf{N}, \ell\rangle}(0) \quad \text{and} \quad m_\ell = c_\ell - \widetilde{\mathsf{R}}^{\langle \mathsf{N}, \ell\rangle}(0)$$

we obtain that m'_ℓ and m_ℓ always differ, and as a consequence so will Σ' and Σ. This means that $\mathsf{Tag}^* = \widetilde{\mathsf{R}}^{\langle \mathsf{N}, -\ell\rangle}(\Sigma')$ is the result of a fresh call, hitting the adversary's Tag' with probability exactly $1/p$.

Case 4: $N' = N_j$ *with* $|\mathbf{c}'| \neq |\mathbf{c}_j|$ *for some* $j \in [1 \ldots q]$.
The length $\ell' = |\mathbf{c}'|$ is used as part of the tweak for the final $\widetilde{\mathsf{R}}$ call, as $\mathsf{Tag}^* = \widetilde{\mathsf{R}}^{\langle \mathsf{N}, -\ell'\rangle}(\Sigma')$. Irrespective of Σ', this $-\ell' \neq -\ell_j$ and therefore the tweak $(N, -\ell')$ is fresh and the output Tag^* is random and independent, hitting the adversary's Tag' with probability exactly $1/p$.

Overall we obtain that $\mathsf{FP}_{\mathbf{z}} \leq 3/p$ gives an AUTH bound for any number of queries q_v greater or equal than one, and so $\mathsf{Adv}^{\mathsf{auth}}_{\mathbb{OTR}}(\mathsf{Adv}) \leq 3q_v/p$.

□

7.7 Experimental Results

We consider two measurements latency and throughput, with various message lengths. Latency shows the total time required for a message to be encrypted and authenticated whereas throughput gives the

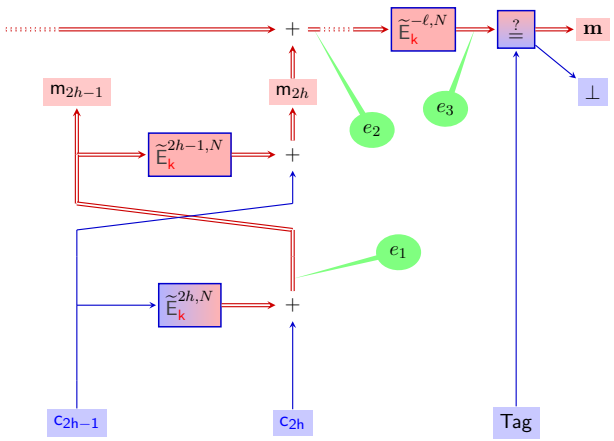

Figure 7.16: The OTR decryption case.

maximum number of executions which can be done in parallel. The experiments were ran between two machines each with Intel i7-4790 CPUs running at 3.60GHz, 16GB of RAM connected through a LAN network of 1Gbps with an average ping of 0.3ms (roundtrip) and implemented using the SPDZ software[1]. WAN experiments were simulated using Linux `tc` tool with an average ping latency of 100ms (roundtrip). To give precise timings, each experiment was averaged with at least 5 executions where each execution authenticated at least 1000 messages. We choose to exclude the times in the online phase for computing the key dependent preprocessing such as $\mathsf{E}_\mathsf{k}(0)$ or $\mathsf{E}_\mathsf{k}(1)$ since this is done just once before the start of authentication.

Table 7.2 contains the preprocessing costs for encryption (similar costs apply for decryption). For this we counted the number of triples and bits required to evaluate each mode of operation instantiated with different PRF's, these costs are given in terms of the message length ℓ, i.e. the number of finite field elements being encrypted. For Leg we assume a finite field size of $p \approx 2^{128}$, where p is chosen such that we can select $L = 128$ in the construction of the Leg PRF. The amount of data sent per party and computational cost is estimated, in the table, using the currently best-known method for producing triples and bits in $\mathbb{F}_p$ with active security [KOS16]. According to [KOS16] bits and triples have the same cost in arithmetic circuits $\mathbb{F}_p$ so we merge the costs into one column which is called Triples. As expected, OTR has a lower preprocessing cost, vs. using CTR+pPMAC, since the number of PRF calls is reduced by half compared to pPMAC; CTR+HtMAC is slightly better than OTR in terms of preprocessing costs.

For the case of CTR and Hash-then-MAC in Table 7.3 we give what these offline estimates would translate into in terms of MBytes of communication per party and throughput per second, for varying values of the number of message blocks ℓ for a LAN and WAN setting. These numbers are derived from the estimates in the MASCOT paper [KOS16], which is currently the most efficient offline processing

[1] https://github.com/bristolcrypto/SPDZ-2

PRF	Mode	Triples
Leg	CTR+pPMAC	$1024 \cdot \ell - 256$
MiMC	CTR+pPMAC	$292 \cdot \ell$
Leg	CTR+HtMAC	$512 \cdot \ell + 128$
MiMC	CTR+HtMAC	$146 \cdot \ell + 146$
Leg	OTR	$512 \cdot \ell + 728$
MiMC	OTR	$146 \cdot \ell + 292$

Table 7.2: Preprocessing costs for Encryption using OTR, CTR+pPMAC, and CTR+Hash-then-MAC (HtMAC) in MPC for an ℓ length message.

	PRF	$\ell = 1$	2	4	8	16	32
MBytes per party	Leg	14.42	25.95	49.02	95.16	187.43	371.98
	MiMC	6.58	9.87	16.45	29.60	55.91	108.54
LAN Throughput per second	Leg	7.57	4.20	2.23	1.15	0.58	0.29
	MiMC	16.58	11.05	6.63	3.68	1.95	1.00
WAN Throughput per second	Leg	0.38	0.21	0.11	0.06	0.03	0.01
	MiMC	0.82	0.55	0.33	0.18	0.10	0.05

Table 7.3: Preprocessing cost (MBytes) and throughput (seconds) for encrypting message blocks of size ℓ, with two parties over a LAN and a simulated WAN network using CTR+HtMAC and MASCOT [KOS16].

step for engines such as SPDZ. In Table 7.4 we present our results for the online phase, in terms of latency and throughput for CTR and Hash-then-MAC, in the LAN and WAN setting.

	PRF	$\ell = 1$	2	4	8	16	32
LAN Latency (ms)	Leg	1.17	1.97	2.75	4.61	8.20	15.68
	MiMC	6.63	13.27	13.42	13.74	14.25	15.35
WAN Latency (ms)	Leg	154	256	258	262	274	295
	MiMC	3760	7521	7521	7521	7521	7523
LAN Throughput per second	Leg	1389	895	527	285	149	76
	MiMC	8853	5697	3589	2010	1079	561
WAN Throughput per second	Leg	151	100	59	33	17	8
	MiMC	428	234	203	127	74	39

Table 7.4: Online phase latency (ms) and best throughput (seconds) for encrypting message blocks of size ℓ, with two parties over a LAN and a simulated WAN network, using CTR+HtMAC.

In Table 7.5 we present the online costs, as a function of ℓ for our various constructions. For each

variant we give the number of rounds and the number of openings. As we have selected highly parallel modes of operation, the round complexity does not depend on the message length. Intuitively, the online round complexity should define the latency of a protocol and the online opening complexity should define the throughput. However, due to the nature of actual physical networks we expect that as soon as we reach the maximum capacity of the network, in terms of data sent (i.e. openings) per round, the latency will drop off rapidly. Thus as ℓ increases we expect to see an increase in latency, despite latency "theoretically" being a constant. The key question is then how big does ℓ need to be before the latency for a specific PRF and mode decreases linearly in ℓ?

PRF	Mode	Online cost	
		Rounds (Enc/Dec)	Openings
Leg	CTR+pPMAC	7/6	$768 \cdot \ell + \ell$
MiMC	CTR+pPMAC	221/147	$146 \cdot \ell + \ell + 1$
Leg	CTR+HtMAC	5/4	$384 \cdot (\ell + 1) + \ell$
MiMC	CTR+HtMAC	148/75	$73 \cdot (\ell + 1) + \ell + 1$
Leg	OTR	6/9	$384 \cdot (\ell + 128) + \ell$
MiMC	OTR	220/295	$73 \cdot (\ell + 2) + \ell + 1$

Table 7.5: Online Costs for OTR and CTR+pPMAC in MPC.

To investigate this potential drop off in latency we carried out experiments in the LAN setting, the results of which are detailed in Figure 7.17 (for small messages) and Figure 7.18 (for long messages for the MiMC PRF). We see that despite ciphers based on the Leg PRF having lower round complexity, this does not translate into low latency as soon as the size of ℓ increases. For small values of ℓ we do benefit from using Leg, but not for larger values. This is because we reach network capacity for only a few parallel calls to Leg; as evaluting the PRF itself takes up a lot of network capacity. On the other hand with MiMC we require more rounds, but in each round we need to send much less data, so even as ℓ increases the latency does not increase that much. Eventually we see that for large messages MiMC ends up having the same growth as we experience with Leg for smaller messages.

In Figure 7.19 and Figure 7.20 we examine throughput for both Leg and MiMC in the LAN setting. Not surprisingly for all options throughput decreases as ℓ increases, and we get a better throughput if we select MiMC and use the CTR+HtMAC cipher. In this and in other figures in this section: OTR is marked in blue, CTR+pMAC is marked in red, and CTR+Hash-then-MAC is marked in Green. Use of the Leg PRF is marked with a dot on the line, and use of the MiMC PRF is marked with a cross.

Indeed contrary to the conclusion in [GRR+16] we conclude that MiMC is better than Leg for both throughput and latency. The primary reason for this conclusion is that, unlike the work in [GRR+16], we consider how these MPC-friendly PRFs work in a larger application and not in isolation.

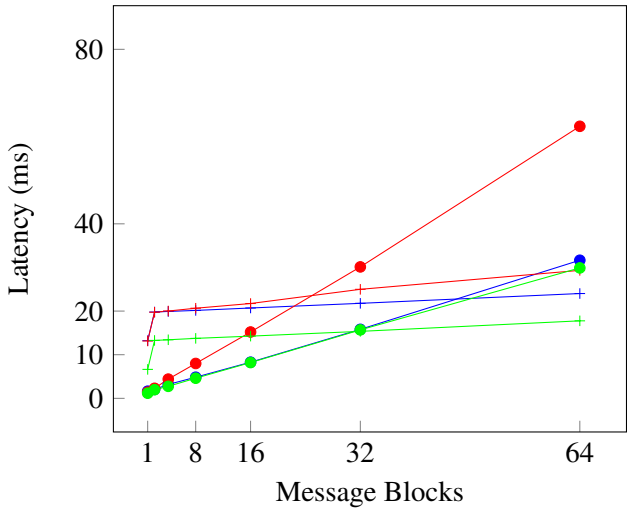

Figure 7.17: Latency of Encryption for OTR vs CTR+pPMAC vs CTR+Hash-then-MAC with MiMC and Leg.

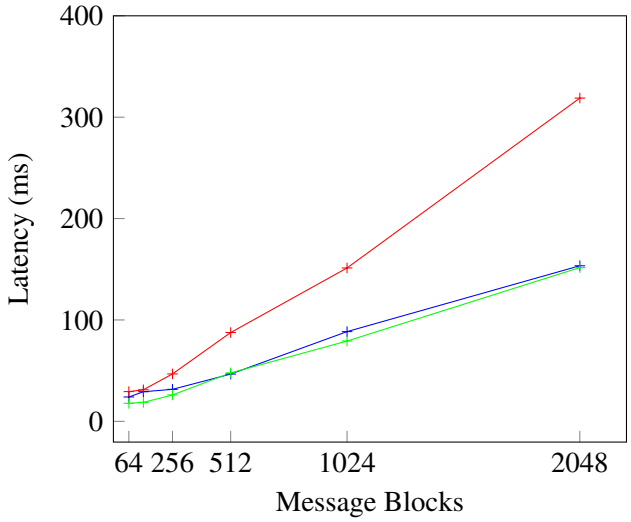

Figure 7.18: Latency of Encryption for OTR vs CTR+pPMAC vs CTR+HtMAC with MiMC, for large message sizes.

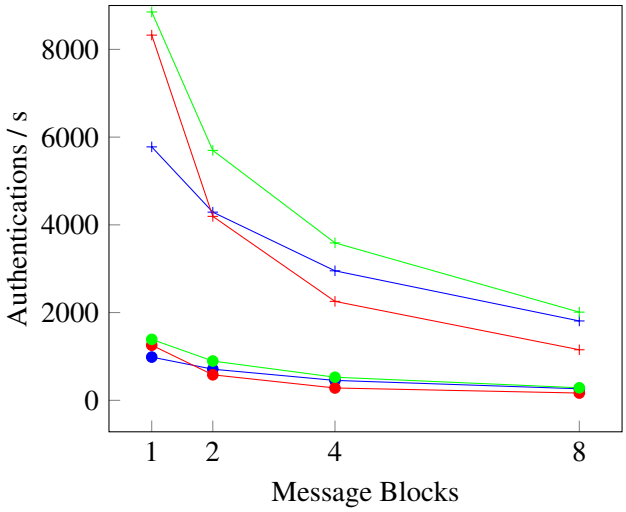

Figure 7.19: Throughput of OTR vs CTR+pPMAC vs CTR+HtMAC with MiMC and Leg.

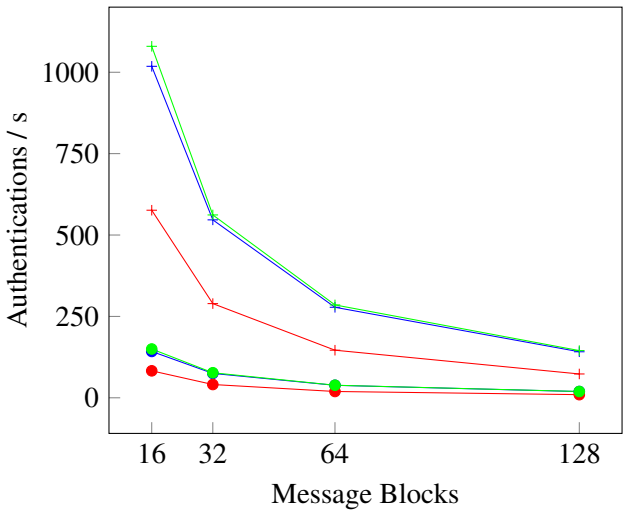

Figure 7.20: Throughput of OTR vs CTR+pPMAC vs CTR+HtMAC with MiMC and Leg.

Chapter 8

Towards an universal share conversion

This chapter is based on joint work with Tim Wood [RW19a] which was presented at INDOCRYPT 2019. Security proofs were done by the co-author of this work, we leave them here for completeness.

8.1 Contributions

In the previous chapter we have covered various circuits that support computations over fields of characteristic two or p. One major open problem is how to switch efficiently between different types of protocols: SPDZ over characteristic p fields and SPDZ over characteristic 2 fields; SPDZ over $\mathbb{F}_p$ and constant round protocols such as BMR. In this chapter, we show the first efficient construction of how to achieve share conversions for dishonest majority. For the case of moving between SPDZ and constant round protocol, our work reduces the cost of garbling over the folkore method by at least $100,000$ AND gates. More concretely, this chapter focuses on share conversions between SPDZ and BMR for dishonest majority.

We also shed some light on the landscape of share conversions for other dishonest majority protocols in Section 8.6 (which is unpublished work but given in the eprint version [RW19b]).

8.2 Overview

MPC over a finite field or a ring is used to emulate arithmetic over the integers, and consequently, non-linear operations such as comparisons between secrets (i.e. $<, >, =$) are an important feature of MPC protocols. One of the shortcomings of MPC based on secret-sharing is that these natural but more complicated procedures require special preprocessing and several rounds of communication.

One way to mitigate these costs would be to use *circuit-garbling* instead of secret-sharing for circuits involving lots of non-linear operations, since this method has low (in fact, constant) round complexity. Recent work has shown that multiparty Boolean circuit garbling with active security in the dishonest majority setting can be made very efficient [WRK17b, HSS17, KY18]. However, performing general arithmetic computations in Boolean circuits can be expensive since the arithmetic operations

must be accompanied with reduction modulo a prime inside the circuit. Moreover, efficient constructions of multiparty constant-round protocols for *arithmetic* circuits remain elusive. Indeed, the best-known optimisations for arithmetic circuits such as using a primorial modulus [BMR16] are expensive even for passive security in the two-party setting. The only work of which the authors are aware in the multiparty setting is the passively-secure honest-majority work by Ben-Efraim [Ben18].

So-called *mixed protocols* are those in which parties switch between secret-sharing (SS) and a garbled circuit (GC) mid-way through a computation, thus enjoying the efficiency of the basic addition and multiplication operations in any field using the former and the low-round complexity of GCs for more complex subroutines using the latter. One can think of mixed protocols as allowing parties to choose the most efficient field in which to evaluate different parts of a circuit.

There has been a lot of work on developing mixed protocols in the two-party passive security setting, for example [HKS+10, KSS13b, KSS14, BDK+18]. One such work was the protocol of Demmler et al. [DSZ15] known as ABY, that gave a method for converting between arithmetic, Boolean, and Yao sharings. For small subcircuits, converting arithmetic shares to Boolean shares (of the bit decomposition) of the same secret – i.e. without any garbling – was shown to give efficiency gains over performing the same circuits in with arithmetic shares; for large subcircuits, using garbling allows reducing online costs. Mohassel and Rindal [MR18] constructed a three-party protocol known as ABY^3 for mixing these three types of sharing in the malicious setting assuming at most one corruption.

For mixed protocols to be efficient, clearly the cost of switching between secret-sharing and garbling, performing the operation, and switching back must be more efficient than the method that does not require switching, perhaps achieved by relegating some computation to the offline phase.

8.2.1 Our approach

When considering mixed protocols in the active setting, the primary technical challenge is in maintaining authentication through the transition from secret-shared inputs and secret inputs inside the GC, and *vice versa*. The naïve way of obtaining authentication from SS to GC is for parties to bit-decompose the shares of their secrets and the MACs locally and use these as input bits to the circuit, and validating inside the GC. This solution would require $O(n \cdot \kappa \cdot \log |\mathbb{F}|)$ bits per party to be sent to switch inputs in the online phase, where n is the number of parties, κ is the computational security parameter, and $\mathbb{F}$ is the MPC field, since each party needs to broadcast a GC key for each bit of the input. This method also requires garbling several additions and multiplications inside the circuit to check the MAC. The advantage of this solution, despite these challenges, is that it requires no additional preprocessing, nor adaptations to the garbling procedure.

Contrasting this approach, our solution makes use of special preprocessing to speed up the conversion. This results in reducing the circuit size by approximately $100,000$ AND gates per conversion for a field with a 128-bit prime modulus (assuming Montgomery multiplication is used). Let $\mathbb{F}_q$ denote the finite field of order q. In this work we show how to convert between secret-shared data in $\mathbb{F}_p$, where p is a large prime and is the MPC modulus, and GCs in $\mathbb{F}_{2^k}$ through the use of "**d**ouble-shared"

authenticated **bits** which we dub *daBits*, following the nomenclature set out by [NNOB12]. These doubly-shared secrets are values in $\{0, 1\}$ shared and authenticated both in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$, where by 0 and 1 we mean the additive and multiplicative identity, respectively, in each field. In brief, the conversion of a secret shared input a into a GC involves constructing a random secret r in $\mathbb{F}_p$ using daBits, opening $a - r$ in MPC, bit decomposing this public value (requiring no communication) and using these as signal bits for the GC, and then in the circuit adding r and computing this modulo p, which is possible since the bit decomposition of r is doubly-shared. This keeps the authentication check mostly outside of the circuit instead requiring that the MAC on $a-r$ is correct. Going the other way around, the output of the circuit is a set of public signal bits whose masking bits are chosen to be daBits. To get the output, parties XOR the public signal bits with the $\mathbb{F}_p$ shares of the corresponding daBit masks, which can be done locally. These shares can then be used to reconstruct elements of $\mathbb{F}_p$ (or remain as bits if desired).

The only use of doubly-shared masks is at the two boundaries (input and output) between a garbled circuit and secret-sharing; all secrets used in evaluating arithmetic circuits (i.e. using standard SS-based MPC) are authenticated shares in $\mathbb{F}_p$ only; all wire masks "inside" the circuit (that is, for all wires that are *not* input or output wires) are authenticated shares of bits in $\mathbb{F}_{2^k}$ only. The *online* communication cost of our solution is that of each party broadcasting a single field element and then broadcasting $\log |\mathbb{F}|$ key shares per input, for a circuit of any depth. Thus the cost is $O(\kappa \cdot \log |\mathbb{F}|)$ per party, per field input to the circuit. The offline cost grows quadratically in n as generating daBits requires every party to communicate with every other party.

While the main focus of this work is to allow Boolean circuits to be evaluated on (the bits of) field elements of $\mathbb{F}_p$, our method gives a full arithmetic/Boolean/garbled circuit mixed protocol as once the bits of $a - r$ are public and the bit decomposition of r is known in $\mathbb{F}_{2^k}$, the parties can run the Boolean circuit computing $(a - r) + r \mod p$ to obtain the bits of a in the field $\mathbb{F}_{2^k}$ *with authentication*. Converting back to $\mathbb{F}_p$ involves XORing the public signal bits with shared daBits (which is free in $\mathbb{F}_{2^k}$). Our work is also compatible of converting classic SPDZ shares in $\mathbb{F}_p$ with the recent protocol SPDZ2k of Cramer et al [CDE+18].

We remark that several of the multiparty arithmetic garbling techniques of [Ben17] require the use of "multifield shared bits", which precisely correspond to our daBits (albeit in an unauthenticated honest-majority setting). In fact, this special preprocessed material lead to more efficient multiparty arithmetic garbling for dishonest majority [MW19].

Our construction involves two steps: the first extends the MPC functionality to allow for the same bits to be generated in two independent $\mathcal{F}_{\mathsf{Prep}}$ sessions in two different fields; the second uses this extended MPC functionality to perform the garbling SPDZ-BMR-style [LPSY15], which we explained in Chapter 3 Thus, while replacing the garbling is not an entirely black-box procedure, the necessary modifications to existing protocols are modest.

Our implementation shows that in some cases switching between arithmetic and Boolean circuits gives more efficient protocols by an order of magnitude than executing plain-SPDZ while increasing the preprocessing costs by a factor of two. More recent work shows that the preprocessing cost can be

reduced by approximately a factor of five [AOR+19, RST+19].

Active security beyond bounded inputs. While essentially all of the basic actively-secure MPC protocols enable the evaluation of additions and multiplications, for more complicated non-linear functions the only solutions that exist are those that require additional assumptions on the input data. For example, comparison requires bit decomposition, which itself requires that secrets be bounded by some constant. Since the bits of each input are directly inserted into the circuit, we can avoid this additional assumption. We refer the reader to [DFK+06] or the documentation for the SCALE-MAMBA project [ACK+19, §10 Advanced Protocols] for an overview of implementations of other functions in MPC.

8.3 Preliminaries

Our protocol makes use of MPC as a black box, with functionality outlined in Figure 3.2. The functionality $\mathcal{F}_{\mathsf{Prep}}$ over a field $\mathbb{F}$ is realised using protocols with statistical security sec if $|\mathbb{F}| = \Omega(2^{\mathsf{sec}})$ and computational security κ depending on the computational primitives being used. We will describe MPC as executed in the SPDZ-family of protocols [DPSZ12, DKL+13, KOS16, KPR18, CDE+18].

8.3.1 Secret-sharing

As in the previous chapters we will asume that values are additively shared. but in this case we need three different types of shared values in our scheme, over two different fields, always additively shared along with their MAC shares. A secret $a \in \mathbb{F}_p$ is shared amongst the parties by additively sharing the secret a in $\mathbb{F}_p$ along with a linear MAC $\gamma_p(a)$ defined as $\gamma_p(a) \leftarrow \alpha \cdot a$, where $\alpha \in \mathbb{F}_p$ is a global MAC key, which is also additively shared. By "global" we mean that every MAC in the protocol uses this MAC key, rather than each party holding their own key and authenticating every share held by every other party. Similarly, a secret $c \in \mathbb{F}_{2^k}$ and its MAC $\gamma_{2^k}(c) = \Delta \cdot c$, where $\Delta \in \mathbb{F}_{2^k}$ is an additively-shared global MAC key, are additively shared in $\mathbb{F}_{2^k}$ amongst the parties.

We denote shared, authenticated secrets in the following ways:

Sharing in $\mathbb{F}_p$	$[\![a]\!]_p = (a^{(i)}, \gamma_p(a)^{(i)}, \alpha^{(i)})_{i=1}^n$ where $a \in \mathbb{F}_p$ and $a^{(i)}, \gamma_p(a)^{(i)}, \alpha^{(i)} \in \mathbb{F}_p$ for all $i \in [n]$.
Sharing in $\mathbb{F}_{2^k}$	$[\![c]\!]_{2^k} = (c^{(i)}, \gamma_{2^k}(c)^{(i)}, \Delta^{(i)})_{i=1}^n$ where $c \in \mathbb{F}_{2^k}$ and $c^{(i)}, \gamma_{2^k}(c)^{(i)}, \Delta^{(i)} \in \mathbb{F}_{2^k}$ for all $i \in [n]$.
Sharing in both	$[\![b]\!]_{p,2^k} = ([\![b]\!]_p, [\![b]\!]_{2^k})$ where $b \in \{0, 1\}$.

The shares are considered correct if

$$\left(\sum_{i=1}^n \gamma_p(a)^{(i)}\right) = \left(\sum_{i=1}^n a^{(i)}\right) \cdot \left(\sum_{i=1}^n \alpha^{(i)}\right)$$

and

$$\left(\sum_{i=1}^{n} \gamma_{2^k}(c)^{(i)}\right) = \left(\sum_{i=1}^{n} c^{(i)}\right) \cdot \left(\sum_{i=1}^{n} \Delta^{(i)}\right)$$

and party P_i holds every value indexed by i. Moreover, secret $[\![b]\!]_{p,2^k}$ is considered correct if the bit is the same in both fields, by which we mean they are either both the additive identity or are both the multiplicative identity, in their fields. Creating these bits efficiently is one of the main contributions of this work. Notice that the superscript on the MACs is outside the bracket: the parties each hold one share of the MAC $\alpha \cdot a$ on a, not MACs on the shares $a^{(i)}$. The security of the MACs comes from the fact that, any adversary learns at most $n-1$ values and so does not know the global MAC key and hence can only alter the secret and its MAC correctly with probability at most $1/|\mathbb{F}|$.

As explained in the previous chapters, additions of secrets / public values or multiplication by public values can be done locally by each party. The main difficulty is in performing secret shared multiplications using the Beaver's trick [Bea92].

Active security. Since all operations in the online phase are linear, if we assume a secure preprocessing (offline) phase, in the online phase only *additive* errors need to be detected. This is where the MACs are used: if the MAC on the final output of the circuit being computed is incorrect, then an additive error has been introduced on the MAC or the secret, and in this case the parties abort. If there is no error, then either the output is correct, or (it can be shown that) the adversary must have learnt enough information to guess the global MAC key. If p is $O(2^{\mathsf{sec}})$ then the chance of the adversary doing so is already negligible, and otherwise parties can generate $\lceil \mathsf{sec}/\log p \rceil$ independent global MAC keys, hold this number of MACs on each secret and require that all MACs on the final output be correct. We refer the reader to [DKL+13] for details on the MAC checking procedure.

8.3.2 Conditions on the secret-sharing field

Let $l = \lfloor \log p \rfloor$. Throughout, we assume the MPC is over $\mathbb{F}_p$ where p is some large prime, but we require that one must be able to generate uniformly random field elements by sampling bits uniformly at random $\{[\![r_j]\!]_p\}_{j=0}^{l-1}$ and summing them to get $[\![r]\!]_p \leftarrow \sum_{j=0}^{l-1} 2^j \cdot [\![r_j]\!]_p$. For this to hold in $\mathbb{F}_p$, we require that $\frac{p-2^l}{p} = O(2^{-\mathsf{sec}})$. Roughly speaking this says that p is slightly larger than a power of 2. (By symmetry of this argument we can require that p be close to a power of 2.) Recall that sampling a uniform element of $\{0,1\}^l$ produces the same distribution as sampling l bits independently by standard measure theory. It follows from Lemma 24 that the statistical distance between the uniform distribution over $\mathbb{F}_p$ and the same over $\{0,1\}^l$ is negligible.

Lemma 24. *Let $l = \lfloor \log p \rfloor$, let P be the probability mass function for the uniform distribution $\mathcal{P}$ over $[0,p) \cap \mathbb{Z}$ and let Q be the probability mass function for the uniform distribution $\mathcal{Q}$ over $[0,2^l) \cap \mathbb{Z}$. Then the statistical distance between distributions is negligible in the security parameter if $\frac{p-2^l}{p} = O(2^{-\mathsf{sec}})$.*

Protocol Π^{+}_{Rand}

This protocol is in the $\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}}$-hybrid model. Let $\mathsf{RShuffle}(\mathsf{seed}, s)$ denote any deterministic algorithm that takes a random seed seed and a vector s and outputs a permutation of components of s. Recall κ is the computational security parameter.

Initialise Parties agree on a session identifier sid and call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\mathsf{Initialise}, \{0,1\}^{\kappa}, \mathsf{sid})$.

Random subset To compute a random subset of size t of a set X, parties run **Random** to obtain a random seed seed for a PRG and then do the following:

1. Let $X = \{x_i\}_{i=1}^{|X|}$. Parties set the vector $s = (s_1, \ldots, s_{|X|}) \leftarrow (1, \ldots, 1, 0, \ldots, 0) \in \{0,1\}^{|X|}$, where the first t bits are set to 1 and the remaining bits set to 0.
2. Each P_i locally computes $s' = (s'_1, \ldots, s'_{|X|}) \leftarrow \mathsf{RShuffle}(\mathsf{seed}, s)$ and outputs the set $S \leftarrow \{x_i : s'_i = 1\}$.

Random buckets To put a set of items indexed by a set X into buckets of size t where t divides $|X|$, parties run **Random** to obtain a seed seed for a PRG and then do the following:

1. Let $X = \{x_i\}_{i=1}^{|X|}$. Each P_i locally computes $s' \leftarrow \mathsf{RShuffle}(\mathsf{seed}, s)$ where $s \leftarrow (i)_{i=1}^{|S|}$.
2. For each $i = 1$ to $|S|/t$, let $S_i \leftarrow \{x_{s'_j} : (i-1) \cdot t < j \leq i \cdot t\}$.

Figure 8.1: Protocol Π^{+}_{Rand}.

Proof. By definition of statistical distance,

$$\begin{aligned}\Delta(\mathcal{P}, \mathcal{Q}) &= \frac{1}{2} \cdot \sum_{x=0}^{p-1} |P(x) - Q(x)| = \frac{1}{2} \cdot \sum_{x=0}^{2^l-1} \left|\frac{1}{p} - \frac{1}{2^l}\right| + \frac{1}{2} \cdot \sum_{x=2^l}^{p-1} \left|\frac{1}{p} - 0\right| \\ &= \frac{1}{2} \cdot 2^l \cdot \frac{p - 2^l}{p \cdot 2^l} + \frac{1}{2} \cdot p - 2^l) \cdot \frac{1}{p} \\ &= \frac{p - 2^l}{p} = O(2^{-\mathsf{sec}}).\end{aligned}$$

□

8.3.3 Extending $\mathcal{F}_{\mathsf{Rand}}$

In order to place items into random buckets and shuffle them around, we need access to certain functionalities which given a set X: i) compute a random subset of X and ii) place all items of X randomly into t separate buckets. Details on how to realize the protocols are given in Figure 8.1.

8.3.4 Arbitrary Rings vs Fields

Our protocol uses actively-secure MPC as black box, so there is no reason the MPC cannot take place over any ring $\mathbb{Z}/m\mathbb{Z}$ where m is possibly composite, as long as m is (close to) a power of 2. The

security of our procedure for generating daBits can tolerate zero-divisors in the ring, so computation may, for example, take place over the ring $\mathbb{Z}/2^l\mathbb{Z}$ for any l, for which actively-secure $\mathcal{F}_{\mathsf{Prep}}$ can be realised using [CDE+18].

Note on XOR. In our context, we will require heavy use of the (generalised) XOR operation. This can be defined in any field as the function

$$\begin{aligned} f : \mathbb{F}_p \times \mathbb{F}_p &\to \mathbb{F}_p \\ (x, y) &\mapsto x + y - 2 \cdot x \cdot y, \end{aligned} \tag{8.1}$$

which coincides with the usual XOR function for fields of characteristic 2. In SS-based MPC, addition requires no communication, so computing XOR in $\mathbb{F}_{2^k}$ is for free; the cost in $\mathbb{F}_p$ (char$(p) > 2$) is one multiplication, which requires preprocessed data and some communication. This operation is the main cost associated with our offline phase, since generating daBits with active security requires generating lots of them and then computing several XORs in both fields.

8.3.5 Garbled Circuits

Throughout this chapter we will use BMR garbling where the preprocessing material for the garbling is done using MASCOT protocol over $\mathbb{F}_{2^k}$ as described in the introductory Section 3.6. In Section 8.4.2 we describe the modifications necessary to this standard garbling technique to provide inputs from get outputs to $\mathbb{F}_p$.

8.4 Protocol

In our protocol, one instance of $\mathcal{F}_{\mathsf{Prep}}$ over $\mathbb{F}_p$ is used to perform addition and multiplication in the field, and one instance of $\mathcal{F}_{\mathsf{Prep}}$ over $\mathbb{F}_{2^k}$ is used to perform the garbling. Note that since the keys for the PRF live in the field $\mathbb{F}_{2^k}$ in the garbling protocol, the instance of $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ must be over a field with $k = O(\kappa)$ for computational security. Indeed, we emphasise that in our protocol k is not directly related to $\log p$. Once the garbling is completed, the full MPC engine in $\mathbb{F}_{2^k}$ is no longer required: the parties only maintain the $\mathbb{F}_p$ instance of $\mathcal{F}_{\mathsf{Prep}}$ and retain the garbled circuits in memory, and will additionally need to make sure they can still perform the procedure **Check** in $\mathcal{F}_{\mathsf{Prep}}$ on values opened in the evaluation of the GC.

In summary, our protocol requires a single opening of a secret-shared value and then locally bit-decomposing this public value to obtain the input wire signal bits to the garbled circuit. Once the parties have these, they open the appropriate keys for circuit evaluation, and the rest of the protocol (including retrieving outputs in secret-shared form) is local. The key challenge in creating the garbled circuit is that for some wire masks, namely a certain set of input masks and all the output wires, we need wire masks which are the same value in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ (i.e. both the additive identity or both the multiplicative identity in each field), which then must be used in the garbling stage of the preprocessing.

We construct the functionality $\mathcal{F}_{\mathsf{CABB}}$ by first showing how to generate daBits, and then showing how this procedure coupled with two instances of the standard $\mathcal{F}_{\mathsf{Prep}}$ functionality gives a preprocessing phase which we call $\mathcal{F}^{+}_{\mathsf{Prep}}$, given in Figure 8.3 which can be used to realise $\mathcal{F}_{\mathsf{CABB}}$.

It would be straightforward to instantiate $\mathcal{F}_{\mathsf{CABB}}$ directly in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, using two independent instances of $\mathcal{F}_{\mathsf{Prep}}$ over the fields $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$. However, we choose to build up to $\mathcal{F}_{\mathsf{CABB}}$ via the functionality $\mathcal{F}^{+}_{\mathsf{Prep}}$ for three reasons:

1. This approach more faithfully resembles the execution of the protocols in our implementation, where daBits are generated and the "extended" $\mathcal{F}_{\mathsf{ABB}}$ functionality $\mathcal{F}_{\mathsf{Prep}}$ is used to run the extended BMR protocol.
2. The daBits are "raw" preprocessed data, so $\mathcal{F}^{+}_{\mathsf{Prep}}$ really forms a complete "offline" phase from which garbling and secret-sharing can be done.
3. Any future work giving a better protocol for creating daBits for two independent $\mathcal{F}_{\mathsf{Prep}}$ instances does not require reproving the security of the extended BMR protocol that makes use of daBits.

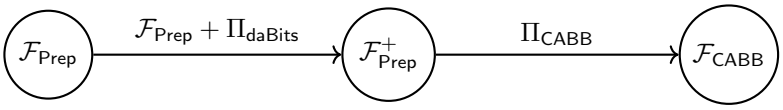

Figure 8.2: Functionality dependencies

8.4.1 Generating daBits using Bucketing

Any technique for generating daBits require some form of checking procedure to ensure consistency between the two fields. Checking consistency often means checking random linear combinations of secrets produce the same result in both cases. Unfortunately, in our case such comparisons are meaningless since the fields have different characteristics, so shares are uniform in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ and so multiplications in the field are not compatible. We can, however, check XORs of bits, which in $\mathbb{F}_p$ involves multiplication. (See Equation 8.1 in Section 8.3.) It is therefore necessary to use a protocol that minimises (as far as possible) the number of multiplications. Consequently, techniques using oblivious transfer (OT) such as [WRK17b] to generated authenticated bits require a lot of XORs for checking correctness, so are undesirable for generating daBits.

Our chosen solution uses $\mathcal{F}_{\mathsf{Prep}}$ as a black box. In order to generate the same bit in both fields, each party samples a bit and calls the $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ instances of $\mathcal{F}_{\mathsf{Prep}}$ with this same input and then the parties compute the n-party XOR. To ensure all parties provided the same inputs in both fields, cut-and-choose and bucketing procedures are required, though since the number of bits it is necessary to generate is a multiple of $\log p \approx \mathsf{sec}$ and we can batch-produce daBits, the parameters are modest.

We use similar cut-and-choose and bucketing checks to those described by Frederiksen et al. [FKOS15, App. F.3], in which "triple-like" secrets can be efficiently checked. The idea behind these checks is the following. One first opens a random subset of secrets so that with some probability all unopened bits are correct. This ensures that the adversary cannot cheat on too many of the daBits. One

Functionality $\mathcal{F}^{+}_{\mathsf{Prep}}$

This functionality extends the reactive functionality $\mathcal{F}_{\mathsf{Prep}}$ with commands to generate the same bits in two independent sessions.

Instances of $\mathcal{F}_{\mathsf{Prep}}$

Independent copies of $\mathcal{F}_{\mathsf{Prep}}$ are identified via session identifiers sid;
Additional command

daBits: On receiving $(\mathsf{daBits}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, \mathsf{sid}_1, \mathsf{sid}_2)$, from all parties where $\mathsf{id}_i \notin \mathtt{Reg}.\mathsf{Keys}$ for all $i \in \ell$, await a message OK or Abort from the adversary. If the message is OK, then sample $\{b_j\}_{j\in[\ell]} \xleftarrow{\$} \{0,1\}$ and for each $j \in [\ell]$, set $\mathtt{Reg}_{\mathsf{sid}_1}[\mathsf{id}_j] \leftarrow b_j$ and $\mathtt{Reg}_{\mathsf{sid}_2}[\mathsf{id}_j] \leftarrow b_j$ and insert the set $\{\mathsf{id}_i\}_{i\in[\ell]}$ into $\mathtt{Reg}_{\mathsf{sid}_1}.\mathsf{Keys}$ and $\mathtt{Reg}_{\mathsf{sid}_2}.\mathsf{Keys}$; otherwise send the messages $(\mathsf{Abort}, \mathsf{sid}_1)$ and $(\mathsf{Abort}, \mathsf{sid}_2)$ to all honest parties and the adversary and ignore all further messages to $\mathcal{F}_{\mathsf{Prep}}$ with session identifier sid_1 or sid_2.

Figure 8.3: Functionality $\mathcal{F}^{+}_{\mathsf{Prep}}$.

then puts the secrets into buckets, and then in each bucket designates one secret as the one to output, uses all other secrets in bucket to check the last, and discards all but the designated secret. For a single bucket, the check will only pass (by construction) if either all secrets are correct or all are incorrect. Thus the adversary is forced to corrupt whole multiples of the bucket size and hope they are grouped together in the same bucket. Fortunately, (we will show that) there is no leakage on the bits since the parameters required for the parts of the protocol described above already preclude it. The protocol is described in Figure 8.4; we prove that this protocol securely realises the functionality $\mathcal{F}^{+}_{\mathsf{Prep}}$ in Figures 8.3 and in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model. To do this, we require Proposition 25.

Proposition 25. *For a given $\ell > 0$, choose $B > 1$ and $C > 1$ so that $C^{-B} \cdot \binom{B\ell}{B}^{-1} < 2^{-\mathsf{sec}}$. Then the probability that one or more of the ℓ daBits output after **Consistency Check** by $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ in Figure 8.4 is different in each field is at most $2^{-\mathsf{sec}}$.*

Proof. Using $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ as black boxes ensures the adversary can only possibly cheat in the input stage. We will argue that:

1. If both sets of inputs from corrupt parties to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ are bits (rather than other field elements), then the bits are consistent in the two different fields with overwhelming probability.
2. The inputs in $\mathbb{F}_{2^k}$ are bits with overwhelming probability.
3. The inputs in $\mathbb{F}_p$ are bits with overwhelming probability.

We will conclude that the daBits are bits in the two fields, and are consistent. We now start with the argument for the first item:

Protocol $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$

This protocol is in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

Initialise:

1. Call an instance of $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Initialise}, \mathbb{F}_p, 0)$; denote it by $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$.
2. Call an instance of $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$; denote it by $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_2^k]$.

To generate ℓ bits, all of the following procedures are performed, in order.

Generate daBits:

1. Let $m \leftarrow CB\ell$ where $C > 1$ and $B > 1$ are chosen so that $C^B \cdot \binom{B\ell}{B} > 2^{\mathsf{sec}}$.
2. For each $i \in [n]$,
   a) Party P_i samples a bit string $(b_1^i, \ldots, b_m^i) \xleftarrow{\$} \{0,1\}^m$.
   b) Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ where P_i has input $(\mathsf{Input}, \mathsf{sid}_p, \mathsf{id}_{b_j^i}, i, b_j^i)_{j=1}^m$ and P_j $(j \neq i)$ has input $(\mathsf{Input}, \mathsf{sid}_p, \mathsf{id}_{b_j^i}, i, \perp)_{i=1}^m$.
   c) Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ where P_i has input $(\mathsf{Input}, \mathsf{sid}_{2^k}, \mathsf{id}_{b_j^i}, i, b_j^i)_{j=1}^m$ and P_j $(j \neq i)$ has input $(\mathsf{Input}, \mathsf{sid}_{2^k}, \mathsf{id}_{b_j^i}, i, \perp)_{i=1}^m$.
   d) Store the outputs of $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ as $[\![b_j^i]\!]_p$ and $[\![b_j^i]\!]_{2^k}$ respectively where $j \in [m]$.

Cut and Choose:

1. Call $\mathcal{F}_{\mathsf{Rand}}^{+}$ with input $(\mathsf{RSubset}, [CB\ell], (C-1)B\ell)$ to obtain a set S.
2. Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ with inputs $(\mathsf{Open}, \mathsf{sid}_p, [\![b_j^i]\!]_p, 0)_{j \in S}$ for all $i \in [n]$.
3. Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ with inputs $(\mathsf{Open}, \mathsf{sid}_{2^k}, [\![b_j^i]\!]_{2^k}, 0)_{j \in S}$ for all $i \in [n]$.
4. If any party sees daBits which are not in $\{0,1\}$ or not the same in both fields, they send the message Abort to all parties and halt.

Combine: For all $j \in S$, compute the XOR sum of parties' bit shares.

1. Set $[\![b_j]\!]_p \leftarrow [\![b_j^1]\!]_p$ and then for i from 2 to n compute: $[\![b_j]\!]_p \leftarrow \mathsf{XOR}([\![b_j]\!]_p, [\![b_j^i]\!]_p)$.
2. Compute $[\![b_j]\!]_{2^k} \leftarrow \bigoplus_{i=1}^n [\![b_j^i]\!]_{2^k}$.

Consistency Check:

1. Call $\mathcal{F}_{\mathsf{Rand}}^{+}$ with input $(\mathsf{RBucket}, [B\ell], B)$ and use the returned sets $(S_i)_{i=1}^{\ell}$ to put the $B\ell$ daBits into ℓ buckets of size B.
2. For each bucket S_i,
   a) Relabel the bits in this bucket as $b^1, \ldots, b^B$.
   b) For $j = 2$ to B, compute $[\![c^j]\!]_p \leftarrow [\![b^1]\!]_p + [\![b^j]\!]_p - 2 \cdot [\![b^1]\!]_p \cdot [\![b^j]\!]_p$ and $[\![c^j]\!]_{2^k} \leftarrow [\![b^1]\!]_{2^k} \oplus [\![b^j]\!]_{2^k}$.
   c) Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ with inputs $(\mathsf{Open}, \mathsf{sid}_p, [\![c^j]\!]_p, 0)_{j=2}^B$. If check passes call $\mathcal{F}_{\mathsf{Prep}}[2^k]$ with inputs $(\mathsf{Open}, \mathsf{sid}_{2^k}, [\![c^j]\!]_{2^k}, 0)_{j=2}^B$.
   d) Parties send Abort and halt if they see dabits which are not in $\{0,1\}$.
   e) Set $[\![b_i]\!]_{p,2^k} \leftarrow [\![b^1]\!]_{p,2^k}$.
3. Call $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ with input $(\mathsf{Check}, \mathsf{sid}_p)$. Then call $\mathcal{F}_{\mathsf{Prep}}\mathbb{F}_{2^k}$ with input $(\mathsf{Check}, \mathsf{sid}2^k)$.
4. If the checks pass without aborting, output $\{[\![b_i]\!]_{p,2^k}\}_{i=1}^{\ell}$ and discard all other bits.

Figure 8.4: Protocol $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$.

Claim 1. *If both sets of inputs from corrupt parties to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ are bits (rather than other field elements), then the bits are consistent in the two different fields with overwhelming probability.*

Proof. Let c be the number of inconsistent daBits generated by a given corrupt party. If $c > B\ell$ then every set of size $(C-1)B\ell$ contains an incorrect daBit so the honest parties will always detect this in **Cut and Choose** and abort. Since $(C-1)B\ell$ out of $CB\ell$ daBits are opened, on average the probability that a daBit is not opened is $1-(C-1)/C = C^{-1}$, and so if $c < B\ell$ then we have:

$$\Pr[\text{None of the } c \text{ corrupted daBits is opened}] = C^{-c}. \tag{8.2}$$

At this point, if the protocol has not yet aborted, then there are $B\ell$ daBits remaining of which exactly c are corrupt.

Suppose a daBit $[\![b]\!]_{p,2^k}$ takes the value $\tilde{b}$ in $\mathbb{F}_p$ and $\hat{b}$ in $\mathbb{F}_{2^k}$. If the bucketing check passes then for every other daBit $[\![b']\!]_{p,2^k}$ in the bucket it holds that $\tilde{b}\oplus\tilde{b'} = \hat{b}\oplus\hat{b'}$, so $\tilde{b'} = (\hat{b}\oplus\hat{b'})\oplus\tilde{b}$, and so $\tilde{b} = \hat{b}\oplus 1$ if and only if $\tilde{b'} = \hat{b'}\oplus 1$. (Recall that we are assuming the inputs are certainly bits at this stage.) In other words, within a single bucket, the check passes if and only if either all daBits are inconsistent, or if none of them are. Thus the probability **Consistency Check** passes without aborting is the probability that all corrupted daBits are placed into the same buckets. Moreover, this implies that if the number of corrupted daBits, c, is not a multiple of the bucket size, this stage never passes, so we write $c = Bt$ for some $t > 0$. Then we have:

$$\begin{aligned}\Pr[\text{All corrupted daBits are placed in the same buckets}] = \\ \frac{\binom{Bt}{B}\cdot\binom{B(t-1)}{B}\cdots\binom{B}{B}\cdot\binom{B\ell-Bt}{B}\cdot\binom{B\ell-Bt-B}{B}\cdots\binom{B}{B}}{\binom{B\ell}{B}\cdot\binom{B\ell-B}{B}\cdots\binom{B}{B}} \\ = \frac{(Bt)!}{B!^t}\cdot\frac{(B\ell-Bt)!}{B!^{\ell-t}}\cdot\frac{B!^\ell}{(B\ell)!} = \binom{B\ell}{Bt}^{-1}.\end{aligned} \tag{8.3}$$

Since the randomness for **Cut and Choose** and **Check Correctness** is independent, the event that both checks pass after the adversary corrupts c daBits is the product of the probabilities. To upper-bound the adversary's chance of winning, we compute the probability by maximising over t: thus we need C and B so that

$$\max_t \left\{ C^{-Bt}\cdot\binom{B\ell}{Bt}^{-1} \right\} < 2^{-\mathsf{sec}} \tag{8.4}$$

The maximum occurs when t is small, and $t \geq 1$ otherwise no cheating occurred; thus since the proposition stipulates that $C^{-B}\cdot\binom{B\ell}{B}^{-1} < 2^{-\mathsf{sec}}$, the daBits are consistent in both fields, if they are indeed bits in both fields. □

Claim 2. *The inputs in $\mathbb{F}_{2^k}$ are bits with overwhelming probability.*

Proof. Next, we will argue that the check in **Cut and Choose** ensures that the inputs given to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ are indeed bits. It follows from Equation 8.2 that the step **Cut and Choose** aborts with probability C^{-c} if any element of either field is not a bit, as well as if the element in the two fields does not match. Moreover, in **Consistency Check**, in order for the check to pass in $\mathbb{F}_{2^k}$ for a given bucket, the secrets' higher-order bits must be the same for all shares so that the XOR is always zero when the pairwise XORs are opened. Thus the probability that this happens is the same as the probability above in Equation 8.4 since again this can only happen when the adversary is not detected in **Cut and Choose**, that he cheats in some multiple of B daBits, and that these cheating bits are placed in the same buckets in **Consistency Check**. □

Now we proceed to the last claim:

Claim 3. *The inputs in $\mathbb{F}_p$ are bits with overwhelming probability.*

Proof. We now show that all of the $\mathbb{F}_p$ components are bits. To do this, we will show that if the $\mathbb{F}_p$ component of a daBit is not a bit, then the bucket check passes only if all other daBits in the bucket are also not bits in $\mathbb{F}_p$.

If the protocol has not aborted, then in every bucket B, for every $2 \leq j \leq B$, it holds that

$$b^1 + b^j - 2 \cdot b^1 \cdot b^j = c^j \tag{8.5}$$

where $c^j \in \{0,1\}$ are determined by the XOR in $\mathbb{F}_{2^k}$. Note that since $c^j = \bigoplus_{i=1}^n b_i^1 \oplus \bigoplus_{i=1}^n b_i^j$ and at least one b_i^j is generated by an honest party, this value is uniform and unknown to the adversary when he chooses his inputs at the beginning.

Suppose $b^1 \in \mathbb{F}_p \setminus \{0,1\}$. If $b^1 = 2^{-1} \in \mathbb{F}_p$ then by Equation 8.5 we have $b^1 = c^j$; but c^j is a bit, so the "XOR" is not the same in both fields and the protocol will abort. Thus we may assume $b^1 \neq 2^{-1}$ and so we can rewrite the equation above as

$$b^j = \frac{b^1 - c^j}{2 \cdot b^1 - 1}. \tag{8.6}$$

Now if b^j is a bit then it satisfies $b^j(b^j - 1) = 0$, and so

$$0 = \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1}\right) \cdot \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1} - 1\right) = -\frac{(b^1 - c^j)(b^1 - (1 - c^j))}{(2 \cdot b^1 - 1)^2}$$

so $b^1 = c^j$ or $b^1 = 1 - c^j$; thus $b^1 \in \{0,1\}$, which is a contradiction. Thus we have shown that if b^1 is not a bit then b^j is not a bit for every other b^j in this bucket. Moreover, for each $j = 2, \ldots, B$, there are two distinct values $b^j \in \mathbb{F}_p \setminus \{0,1\}$ solving Equation 8.6 corresponding to the two possible values of $c^j \in \{0,1\}$, which means that if the bucket check passes then the adversary must *also* have guessed the bits $\{c^j\}_{j=1}^B$, which he can do with probability 2^{-B} since they are constructed using at least one honest party's input. Thus the chance of cheating without detection in this way is at most $2^{-Bt} \cdot C^{-Bt} \cdot \binom{B\ell}{Bt}^{-1}$.

Thus we have shown that the probability that $b^1 \in \mathbb{F}_p \setminus \{0,1\}$ is given as output for the $\mathbb{F}_p$ component is at most the probability that the adversary corrupts a multiple of B daBits, that these daBits

are placed in the same buckets, and that the adversary correctly guesses c bits from honest parties (in the construction of the bits $\{b^j\}_{j\in B}$) so that the appropriate equations hold in the corrupted buckets. Indeed, needing to guess the bits ahead of time only *reduces* the adversary's chance of winning from the same probability in the $\mathbb{F}_{2^k}$ case. □

We conclude that the daBits are bits in both fields and are the same in both fields with probability except with probability at most $2^{-\mathsf{sec}}$. □

Theorem 26. *The protocol $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ securely realises $\mathcal{F}^{+}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}^{+}_{\mathsf{Rand}})$-hybrid model against an active adversary corrupting up to $n-1$ out of n parties.*

Proof. To prove security in the UC framework we must show that to any environment $\mathcal{Z}$, for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that the execution of an idealised version of the protocol run by a trusted third party $\mathcal{F}$ with the simulator is indistinguishable from a real execution of the protocol Π between the honest parties and the adversary. The environment specifies the code run by the adversary as well as the inputs of all parties, honest and dishonest. Additionally, the environment sees all outputs of all parties; it does not see the intermediate interactions in subroutines of the honest parties' executions, otherwise distinguishing would be trivial as honest parties either perform Π or interact with $\mathcal{F}$. In the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}^{+}_{\mathsf{Rand}})$-hybrid model, the adversary is allowed to make oracle queries to these functionalities and $\mathcal{S}$ must generate the responses.

Note the functionality does *not* have access to the random tapes honest parties as this would make distinguishing between worlds trivial: it would be impossible for the simulator to emulate honest parties to the real-world adversary indistinguishably since for any random tape sampled by the simulator, the environment would always be able to execute the protocol internally, using its knowledge of the random tapes of honest parties to execute the entire protocol deterministically, and compare it to the output of the simulator.

Following standard practice, and as described in [Can00, §4.2.2], we define a simulator which interacts with the adversary $\mathcal{A}$ as a black box. This allows us to make the claim that the simulator works regardless of the code run by the adversary and hence prove the claim.

Suppose the adversary corrupts $t < n$ parties in total, indexed by a set A. We define a sequence of hybrid worlds $(\textbf{Hybrid } h)_{h=0}^{n-t}$ and show that each is indistinguishable from the previous. **Hybrid** h is defined as follows:

Definition 27. *(**Hybrid** $h)_{h=0}^{n-t}$ game. The simulator has the actual input of $n-t-h$ honest parties and must simulate the remaining h honest parties towards the adversary.*

The simulator is described in Figure 8.5.

Claim 4. *The $\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}^{+}_{\mathsf{Rand}}$-hybrid world is indistinguishable from **Hybrid** 0.*

Proof. Correctness of the simulation holds as follows. The simulator emulates $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$, $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ and $\mathcal{F}^{+}_{\mathsf{Rand}}$, so all calls made to these oracles are dealt with as in an execution of the protocol. Indeed,

Simulator $\mathcal{S}^h_{\mathsf{Prep}^+}$

The simulator is (vacuously) parameterised by h, which means the simulator knows the actual inputs of $n-t-h$ honest parties, and must simulate for the remaining h. We denote the adversary by $\mathcal{A}$.

Initialise: On receiving the call to $\mathcal{F}_{\mathsf{Prep}}$ with inputs $(\mathsf{Initialise}, \mathbb{F}_p, \mathsf{sid}_p)$ and $(\mathsf{Initialise}, \mathbb{F}_{2^k}, \mathsf{sid}_{2^k})$, initialise corresponding internal copies.

Calls to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ All calls for producing preprocessing, other than what is described below, sent from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ should be forwarded to $\mathcal{F}_{\mathsf{Prep}}$. All response messages from $\mathcal{F}_{\mathsf{Prep}}$ are sent directly to $\mathcal{A}$.

Calls to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ All calls for producing preprocessing, other than what is described below, sent from the $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ should be forwarded to $\mathcal{F}_{\mathsf{Prep}}$. All response messages from $\mathcal{F}_{\mathsf{Prep}}$ are sent directly to $\mathcal{A}$.

For the following procedures, send the calls to the *internal* copies of $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$, $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ and $\mathcal{F}^+_{\mathsf{Rand}}$ as described in the protocol.

[Start] Call $\mathcal{F}^+_{\mathsf{Prep}}$ with input $(\mathsf{daBits}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, \mathsf{sid}_p, \mathsf{sid}_{2^k})$.

Generate daBits: Run **Generate daBits** from $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ with $\mathcal{A}$, sampling inputs for all honest parties.

Cut and Choose: Run **Cut and Choose** from $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ with $\mathcal{A}$.

Combine: Run **Combine** from $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ with $\mathcal{A}$.

Check Correctness: Run **Check Correctness** from $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$ with $\mathcal{A}$.

[Finish] If the protocol aborted, send Abort to $\mathcal{F}_{\mathsf{Prep}}$, and otherwise send OK.

Figure 8.5: Simulator $\mathcal{S}^h_{\mathsf{Prep}^+}$.

for all calls to $\mathcal{F}_{\mathsf{Prep}}$ in either field which are outside of the daBits generation procedure, the commands are forwarded to $\mathcal{F}_{\mathsf{Prep}}$ and relayed back to $\mathcal{A}$, and since $\mathcal{F}_{\mathsf{Prep}}$ has the same interface as $\mathcal{F}_{\mathsf{Prep}}$ by definition, there is no difference between the worlds. As for the daBit generation, when the adversary makes calls to provide (random) inputs and then perform **Cut and Choose**, the simulator does not forward the messages through to $\mathcal{F}_{\mathsf{Prep}}$ since all bits used in the protocol except the final output bits are discarded. Instead the command $(\mathsf{daBits}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, \mathsf{sid}_p, \mathsf{sid}_{2^k})$ is sent to $\mathcal{F}_{\mathsf{Prep}}$ and the simulator executes the daBit routines honestly with the adversary, making random choices for honest parties by sampling in the same way as in the protocol.

Now we argue indistinguishability between executions: we must show that for any algorithm $\mathcal{A}$ specified by the environment $\mathcal{Z}$, it holds that

$$\textsc{Exec}(\mathcal{Z}, \mathcal{A}^{\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Rand}}}, \mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}) \sim \textsc{Exec}(\mathcal{Z}, \mathcal{S}^0_{\mathsf{Prep}^+}, \mathcal{F}^+_{\mathsf{Prep}})$$

where $\sim$ denotes statistical indistinguishability of distributions, and the randomness of these distribu-

tions is taken over the random tapes of honest parties and the adversary and simulator.

First, note that the oracles $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}^{+}_{\mathsf{Rand}}$ are executed honestly by $\mathcal{S}^{0}_{\mathsf{Prep}^{+}}$ so the contribution to the distributions is the same in both executions.

Second, since the inputs of honest parties are sampled during the protocol, they are not specified or known by the environment. However, if the adversary performs a *selective-failure attack*, then the environment may learn information. A selective failure attack is where the environment can learn some information if the protocol does not detect cheating behaviour. For example, if the environment guesses an entire bucket of bits and chooses inputs for the adversary's input so that the bucket check would pass based on these guesses, then if the protocol does not abort then the environment learns that its guesses were correct. Then if the final output bit is *not* the XOR of all parties' inputs then the execution must have happened in **Hybrid** 0 since in this world the output depends on the random tape of $\mathcal{F}_{\mathsf{Prep}}$ and is independent of the adversary's and honest parties' random tapes, contrasting the output in the $\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}^{+}_{\mathsf{Rand}}$-hybrid world in which the final output is an XOR of bits on these tapes (which were guessed by the environment). Since this happens with probability $\frac{1}{2}$, in expected 2 executions, the environment can distinguish. However, by Proposition 25, the environment can only mount a selective failure attack with success with probability at most $2^{-\mathsf{sec}}$ by the choice of parameters.

Thus the only way to distinguish between worlds is if the transcript leaks information on the honest parties' inputs. In **Check Correctness**, XORs are computed in both fields and the result is opened; however, this reveals no information on the final daBit outputs as the linear dependence between the secret and the public values is broken by discarding all secrets in each bucket except the designated (i.e. first) bit. We conclude that the overall distributions of the two executions are statistically indistinguishable in sec. □

Claim 5. ***Hybrid*** h *is indistinguishable from* ***Hybrid*** $h+1$ *for* $h = 0, \ldots, n-t-1$.

Proof. There is no difference between these worlds since honest parties' (random) inputs are sampled the same way in both cases. □

Since $\mathcal{F}_{\mathsf{Prep}}$ is secure up to $t = n-1$, the result follows. □

8.4.2 Garbling and Switching

In this section we give a high-level description of how our approach can be used to provide input to a garbled circuit from secret-shared data, and convert garbled-circuit outputs into sharings of secrets in $\mathbb{F}_p$.

8.4.2.1 From SS to GC

In brief, the parties input a secret-shared $[\![x]\!]_p$ by computing $[\![x-r]\!]_p$ and opening it to reveal $x-r$ where $r = \sum_{j=0}^{\lfloor \log p \rfloor - 1} 2^j \cdot [\![r_j]\!]_p$ is constructed from daBits $\{[\![r_j]\!]_{p,2^k}\}_{j=0}^{\lfloor \log p \rfloor - 1}$, and $\mathcal{F}_{\mathsf{Prep}}$ is called with input $(\mathsf{Check}, 0)$ either at this point or later on, and then these public values are taken to be input bits to

the garbled circuit. To correct the offset r, the circuit $(x-r)+r \bmod p$ is computed inside the garbled circuit. This is possible since the bits of r can be hard-wired into the circuit using the $\mathbb{F}_{2^k}$ sharings of its bit-decomposition.

Note that typically for a party to provide input bit b on wire w in a garbled circuit, the parties reveal the secret-shared *wire mask* $[\![\lambda_w]\!]_{2^k}$ to this party, which broadcasts $\Lambda_w \leftarrow b \oplus \lambda_w$, called the associated *signal bit*; then the parties communicate further to reveal keys required for ciphertext decryptions, which is how the circuit is evaluated. This mask thus hides the actual input (and is removed inside the garbled circuit). Since the inputs here are the bits of the public value $x-r$, there is no need mask inputs here, and thus it suffices to set all the corresponding wire mask bits to be 0.

8.4.2.2 From GC to SS

In standard BMR-style garbling protocols, the outputs of the circuit are a set of public signal bits. These are equal to the actual Boolean outputs XORed with circuit output wire masks, which are initially secret-shared, concealing the actual outputs. Typically in multi-party circuit garbling, the wire masks for output wires are revealed immediately after the garbling stage so that all parties can learn the final outputs without communication after locally evaluating the garbled circuit. When garbling circuits using SS-based techniques, and aiming for computation in which parties can continue to operate on private outputs of a GC, a simple way of obtaining shared output is for the parties not to reveal the secret-shared wire masks for output wires after garbling and instead, after evaluating, to compute the XOR of the secret-shared mask with the public signal bit, in MPC.

In other words, for output wire w they obtain a sharing of the secret output bit b by computing

$$[\![b]\!]_{2^k} \leftarrow \Lambda_w \oplus [\![\lambda_w]\!]_{2^k}.$$

In our case, we want the shared output of the circuit to be in $\mathbb{F}_p$, and to do this it suffices for the masks on circuit output wires to be daBits (instead of random bits shared only in $\mathbb{F}_{2^k}$ as would be done normally) and for the parties to compute (locally)

$$[\![b]\!]_p \leftarrow \Lambda_w + [\![\lambda_w]\!]_p - 2 \cdot \Lambda_w \cdot [\![\lambda_w]\!]_p.$$

To avoid interfering with the description of the garbling subprotocol, we can define an additional layer to the circuit after the output layer which converts output wires with masks only in $\mathbb{F}_{2^k}$ to output wires with masks as daBits, without changing the real values on the wire. To do this, for every output wire w, let $[\![\lambda_w]\!]_{2^k}$ be the associated secret-shared wire mask. Then,

- In the garbling stage take a new daBit $[\![\lambda_{w'}]\!]_{p,2^k}$,
  1. Set $[\![\Lambda_{w_0}]\!]_{2^k} \leftarrow [\![\lambda_w]\!]_{2^k} \oplus [\![\lambda_{w'}]\!]_{2^k}$.
  2. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, \mathsf{id}\Lambda_{w_0}, 1)$ to obtain Λ_{w_0}.
- In the evaluation stage, upon obtaining Λ_w,
  1. Compute $\Lambda_{w'} \leftarrow \Lambda_w \oplus \Lambda_{w_0}$.
  2. Compute the final ($\mathbb{F}_p$-secret-shared) output as $[\![b]\!]_p \leftarrow \Lambda_{w'} + [\![\lambda_{w'}]\!]_p - 2 \cdot \Lambda_{w'} \cdot [\![\lambda_{w'}]\!]_p$.

Observe that $\Lambda_{w_0} \equiv \lambda_{w_0}$ so this procedure is just adding a layer of XOR gates where the masking bits are daBits and the other input wire is always 0 (so the gate evaluation doesn't change the real wire value). Note that since the signal bits for XOR gates are determined from input signal bits and not the output key, there is no need to generate an output key for wire w_0.
For correctness, observe that

$$\begin{aligned}\Lambda_{w'} \oplus \lambda_{w'} &= (\Lambda_w \oplus \Lambda_{w_0}) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= ((b \oplus \lambda_w) \oplus (0 \oplus \lambda_{w_0})) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= b.\end{aligned}$$

8.5 Implementation

We have implemented daBit generation and the conversion between arithmetic shares and garbled circuits. Our code is developed on top of the MP-SPDZ framework [Ana19] and experiments were run on computers with commodity hardware connected via a 1 Gb/s LAN connection with an average round-trip ping time of 0.3ms. The $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ functionality is implemented using LowGear, one of the two variants of Overdrive [KPR18]; the $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$ functionality is implemented using MASCOT [KOS16]. In our experiments, $\mathbb{F}_{2^k}$ is always taken with $k = \kappa = 128$ since this is the security of PRF keys used in SPDZ-BMR. The daBits are always generated with $\kappa = 128$ and the same statistical security sec as the protocol for $\mathcal{F}_{\mathsf{Prep}}$.

8.5.1 Primes.

We require that p be close to a power of 2 so that $a - r$ is indistinguishable from a uniform element of the field, as discussed in Section 8.3. Since we use LowGear in our implementation, for a technical reason we also require that p be congruent to $1 \mod N$ where $N = 32768$. (This is the amount of packing in the ciphertexts.) Consequently, using LowGear means we always lose $15 = \log 32768$ bits of security if $p > 65537$ since then the k-bit prime must be of the form $2^{k-1} + t \cdot 2^{15} + 1$ for some t where $1 \leq t \leq 2^{k-16} - 1$, so the secret masks r constructed from a sequence of bits "miss" at least this much of the field.

8.5.2 Cut and choose optimisation

One key observation that enables reduction of the preprocessing overhead in $\mathbb{F}_{2^k}$ is that parties only need to input bits (instead of full $\mathbb{F}_{2^k}$ field elements) into $\mathcal{F}_{\mathsf{Prep}}$ during $\mathcal{F}_{\mathsf{Prep}}||\Pi_{\mathsf{daBits}}$. For a party to input a secret x in MASCOT, the parties create a random authenticated mask r and open opened it to the party, and the party then broadcasts $x + r$. Since the inputs are just bits, it suffices for the random masks also to be bits. Generating authenticated bits using MASCOT is extremely cheap and comes with a small communication overhead (see Table 8.3).

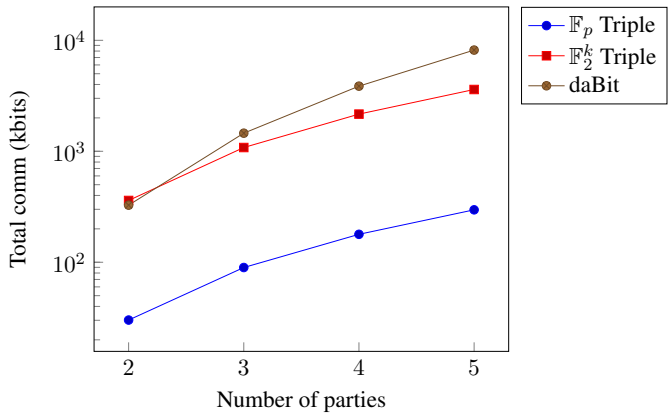

Figure 8.6: Total communication costs for all parties per preprocessed element.

8.5.3 More efficient packing for MAC Check

Instead of a set of k secret bits being opened as full $\mathbb{F}_{2^k}$ field elements $(0,\ldots,0,b_1),\ldots,(0,\ldots,0,b_t) \in \mathbb{F}_2^k \cong \mathbb{F}_{2^k}$, we can save on all the redundant 0's being sent by sending a single field element $(b_k,\ldots,b_1) \in \mathbb{F}_{2^k}$. This optimisation reduces by a factor 2 the amount of data sent for the online phase of daBit generation.

8.5.4 Complexity analysis.

In LowGear (Overdrive) and MASCOT the authors choose to avoid reporting any benchmarks for random bit masks in $\mathbb{F}_{2^k}$ or random input masks in $\mathbb{F}_p$ since they focused on the entire triple generation protocol. Fortunately their code is open source and easy to modify so we micro-benchmarked their protocols in order to get concrete costs for the procedure **Input** for $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_{2^k}]$. For example, in the two-party case, to provide an input bit costs overall 0.384kbits with MASCOT in $\mathbb{F}_{2^k}$. For LowGear providing bits as input is equivalent to providing an entire $\mathbb{F}_p$ field element, strongly contrasting the case for $\mathbb{F}_{2^k}$; thus the cost for an input is 2.048kb. Hence, with the current state of protocols, inputs are cheap in a binary field whereas triples are cheap in a prime field.

8.5.5 Bucketing parameters.

Recall that our goal is to minimise the total amount of communication and time spent by parties generating each daBit. After examining the input and triple costs for LowGear and MASCOT (see Table 8.1) we observed that the optimal communication for statistical security $\mathsf{sec} = 64$ and a $p \approx 2^{128}$ is achieved with a generation of $l = 8192$ daBits per loop, a cut-and-choose parameter and $C = 5$ and a bucket size $B = 4$. Then we ran the daBit generation along with LowGear and MASCOT for multiple parties on the same computer configuration to get the total communication cost in order to see how communication scales in terms of number of parties. Results are given in Figure 8.6. Although MASCOT triples

# Parties	MASCOT $\mathbb{F}_{2^k}$		LowGear $\mathbb{F}_p$	
	Input (bit)	Triple	Input	Triple
2	0.384	360.44	2.048	30.146
3	1.024	1081.32	5.888	89.67
4	1.92	2162.64	11.520	178.572
5	3.072	3604.4	18.94	296.85

Table 8.1: Communication costs (kbits) for fields with different characteristic.

are never used during the daBit production, we believe that comparing the cost of a daBit to the best triple generation in $\mathbb{F}_{2^k}$ helps to give a rough idea of how expensive a single daBit is.

	$\mathsf{sec} > 40$			$\mathsf{sec} > 64$			$\mathsf{sec} > 80$		
# daBits	128	1024	8192	128	1024	8192	128	1024	8192
Calls to $\mathcal{F}_{\mathsf{Prep}}[\{\mathbb{F}_p, \lvert\mathbb{F}_{2^k}\}]$.**Input**	40	16	12	42	40	40	36	28	24
Calls to $\mathcal{F}_{\mathsf{Prep}}[\mathbb{F}_p]$.**Multiply**	7	7	5	13	9	7	17	13	11
Achieved sec	40	47	44	67	64	64	82	84	90

Table 8.2: Two parties preprocessing cost per daBit while varying the number of daBits per batch and statistical security. Parameters minimize for total communication given by LowGear and MASCOT.

To see how efficiency scales when the statistical security parameter sec is increased, we record the fewest numbers of calls to $\mathcal{F}_{\mathsf{Prep}}$, optimising for total (actual) communication cost in Table 8.2. Since the numbers are dependent on integers (number of parties, size of buckets, and cut and choose parameter), several of the numbers in the table give far better security than the minimum stated. Note that since we optimise for the total communication cost and not for the smallest **Cut and Choose** and **Bucketing** parameters that achieve each level of security, in the cost for $\mathsf{sec} = 64$ the number of calls to $\mathcal{F}_{\mathsf{Prep}}$.**Input** is larger than for $\mathsf{sec} = 80$. The bucket size, correlated with the number of calls to $\mathcal{F}_{\mathsf{Prep}}$.**Multiply**, is therefore is smaller than for $\mathsf{sec} = 80$.

8.5.6 Share conversion

To reduce the amount of garbling when converting an additive share to a GC one, if we assume the $\mathbb{F}_p$ input to the garbled circuit is bounded by $p/2^{\mathsf{sec}}$, then a uniform r in $\mathbb{F}_p$ is 2^{sec} times larger than a so $a - r$ is statistically-indistinguishable from a uniform element of $\mathbb{F}_p$; consequently, one need only garble $a + r$ and not $a + r \mod p$, which makes the circuit marginally smaller – 379 AND gates for a 128 bit prime rather than ≈ 1000 AND gates for an addition mod p circuit.

In Table 8.4 we split the conversion into two phases: the cost of generating 127 daBits for doing a full conversion (including the preprocessing triples from LowGear) and the online of SPDZ-BMR.

sec	$\log p$	k	Comm. (kb)			Total (kb)	Time (ms)			Total(ms)
			$\mathcal{F}^{p}_{\mathsf{Prep}}$	$\mathcal{F}^{2^k}_{\mathsf{Prep}}$	daBitgen		$\mathcal{F}^{p}_{\mathsf{Prep}}$	$\mathcal{F}^{2^k}_{\mathsf{Prep}}$	daBitgen	
40	128	128	76.60	2.30	6.94	85.84	0.159	$<$ 10ns	0.004	0.163
64	128	128	146.47	7.68	9.39	163.54	0.303	$<$ 10ns	0.010	0.313
80	128	128	192.95	4.60	7.32	204.88	0.485	$<$ 10ns	0.008	0.493

Table 8.3: 1 Gb/s LAN experiments for two-party daBit generation per party. For all cases, the daBit batch has length 8192.

Conversion	daBit (total)		SPDZ-BMR	
	Comm. (kbits)	Time (ms)	ANDs	Online (ms)
SPDZ $\mapsto$ GC	20769	39.751	379	0.106
GC $\mapsto$ SPDZ	10303	19.719	0	0.005

Table 8.4: Two parties 1 Gb/s LAN experiments converting a 63 bit field element with 64 statistical security. BMR online phase times are amortized over 1000 executions in parallel (single-threaded).

8.5.7 Comparison to semi-honest conversion.

When benchmarked with 40 bit statistical security, the online phase to convert 1000 field elements of size 32 bits takes 193ms. Our solution benefits from merging multiple conversions at once due to the SIMD nature of operations and that we can perform a single MAC-Check to compute the signal bits for the GC. Note that our conversion from an arithmetic SPDZ share to a SPDZ-BMR GC share takes about 14 times more than the semi-honest arithmetic to an Yao GC conversion in ABY or Chameleon on an identical computer configuration [RWT+18, DSZ15].

8.5.8 Multiple class Support Vector Machine

A support vector machine (SVM) is a machine learning algorithm that uses training data to compute a matrix A and a vector $\mathbf{b}$ such that for a so-called *feature vector* $\mathbf{x}$ of a new input, the index of the largest component of the vector $A \cdot \mathbf{x} + \mathbf{b}$ is defined to be its class. We decided to benchmark this circuit using actively-secure circuit marbling as it is clear that there is an operation best suited to arithmetic circuits (namely, $[\![A]\!] \cdot [\![\mathbf{x}]\!] + [\![b]\!]$) and another better for a Boolean circuit (namely, argmax, which computes the index of the vector's largest component).

We have benchmarked the online phase of a multi-class Linear SVM with 102 classes and 128 features over a simulated WAN network (using the Linux `tc` command) with a round-trip ping time of 100ms and 50Mb/s bandwidth with two parties. The SVM structure is the same used by Makri et al. [MRSV19] to classify the Caltech-101 dataset which contains 102 different categories of images

such as aeroplanes, dolphins, helicopters and others [FFFP04]. In this dataset, $\mathbf{x} \in \mathbb{F}_p^{128}$, $A \in \mathbb{F}_p^{102\times 128}$ and $\mathbf{b} \in \mathbb{F}_p^{102}$, and it requires 102 conversions from $\mathbb{F}_p$ to $\mathbb{F}_2^k$ – one for each SVM label. The particular SVM used by Makri et al. has bounded inputs x where $\log|x| \leq 25$, a field size $\log p = 128$ and statistical security $\mathsf{sec} = 64$.

We have implemented a special instruction in MP-SPDZ which loads a secret integer modulo p (a SPDZ share) into the SPDZ-BMR machine. To merge all modulo p instructions of SPDZ shares into SPDZ-BMR to form an universal Virtual Machine requires some extra engineering effort: this is why we chose to micro-benchmark in Table 8.5 the different stages of the online phase: doing $[\![\mathbf{y}]\!]_p \leftarrow [\![A]\!]_p \cdot [\![\mathbf{x}]\!]_p + [\![\mathbf{b}]\!]_p$ with SPDZ, then the instruction converting $[\![\mathbf{y}]\!]_p = ([\![y_1]\!]_p, \ldots, [\![y_{102}]\!]_p)$ to $(\{[\![(y_1)_j]\!]_{2^k}\}_{j=0}^{\log p-1}, \ldots, \{[\![(y_{102})_j]\!]_{2^k}\}_{j=0}^{\log p-1})$, ending with the evaluation stage of SPDZ-BMR on

$$\mathsf{argmax}((([\![(y_1)_j]\!]_{2^k})_{j=0}^{\log p-1}, \ldots, ([\![(y_{102})_j]\!]_{2^k})_{j=0}^{\log p-1})).$$

We name this construction Marbled-SPDZ.

Online cost. The online phase (Table 8.5) using Marbled-SPDZ is more than 10 times faster than SPDZ-BMR and about 10 times faster than SPDZ.

Preprocessing cost. The preprocessing effort for the garbling (in AND gates) is reduced by a factor of almost 400 times using our construction. We chose to express the preprocessing costs of Table 8.5 in terms of AND gates, random triples and bits mainly for the reason that SPDZ-BMR requires much more work for an AND gate than WRK. Based on the concrete preprocessing costs we have in Table 8.5 we give estimations on the communication where the preprocessing of the garbling is done via WRK: performing an SVM evaluation using i) WRK alone would require 6.6GB sent per party (3.8kb per AND gate), ii) SPDZ alone (with LowGear) would require 54MB per party (15kb per triple/random bit), iii) Marbled-SPDZ would take 160MB per party.

Nevertheless, the main cost in Marbled SPDZ is the daBit generation (119 MB) which is more than 70% of the preprocessing effort. If one chooses $\mathrm{sec} = 40$ then we need five triples per daBit and 65 daBits per conversion which amounts to only 119MB for the entire SVM evaluation (twice the cost of plain SPDZ). A detailed cost can be found in Table 8.6 where the column daBitC represents the conversion cost whereas the GC column is the SPDZ-BMR protocol.

8.6 Generality of daBits

In Diagram 8.7 we show how our daBit generation bridges different MPC protocols for dishonest majority. Our inspiration is drawn from Keller and Yanai [KY18] which can convert between SPDZ-BMR and SPDZ over $\mathbb{F}_2^k$ by setting the global difference used in the free-XOR as the global MAC-key in $\mathsf{SPDZ}[\mathbb{F}_2^k]$. Their main idea is to sample a secret random bit authenticated in $\mathbb{F}_2^k$ and use that random bit to do a share conversion. This lends nicely because the authentication key has the same representation in both engines whereas we need more involved techniques (eg: use cut and choose) to generate such a preprocessed authenticated random bit between $\mathsf{SPDZ}[\mathbb{F}_p]$ and SPDZ-BMR (or $\mathsf{BMR}[\mathbb{F}_2^k]$).

Protocol	Sub-Prot	Online cost			Preprocessing cost		
		Comm. rounds	Time (ms)	Total (ms)	$\mathbb{F}_p$ triples	$\mathbb{F}_p$ bits	AND gates
SPDZ		54	2661	2661	19015	9797	-
SPDZ-BMR		0	2786	2786	-	-	14088217
Marbled-SPDZ	SPDZ	1	133		13056	0	-
	daBitC	2	137	271.73	63546	0	27030
	GC	0	1.73		-	-	8383

Table 8.5: Two-party linear SVM: single-threaded (non-amortized) online phase costs and preprocessing costs with $\mathsf{sec} = 64$.

Circuit type	Sub-Protocol	Preprocessing protocol (comm.)			Total
		LowGear	WRK (indep.)	WRK (dep.)	
SPDZ		49.4 MB	-	-	49.4 MB
GC		-	4917 MB	1768 MB	6685 MB
Marbled	SPDZ	24.48 MB	-	-	
	daBit convert	71.13 MB	6.83 MB	2.45 MB	108.87 MB
	GC	-	2.92 MB	1.05 MB	

Table 8.6: Two-party linear SVM communication cost for preprocessing in MBytes and statistical security $\mathsf{sec} = 40$.

BMR and TinyOT. This is done by converting the pairwise MAC to a global one and it is explained in several papers [HSS17,WRK17b]. Going from a global MAC to a pairwise one is slightly more difficult but could be achieved using daBits or a similar consistency check by Damgård et al. [DEF+19] to go from a bit share in $\mathsf{SPDZ}[\mathbb{Z}_{2^k}]$ to a TinyOT sharing.

TinyOT and SPDZ2k. Recently Damgård et al. [DEF+19] introduced a method of switching back and forth between $\mathsf{SPDZ}[\mathbb{Z}_{2^k}]$ to TinyOT. They use a lightweight batch-check to ensure input consistency in both engines. Although they show how to switch a random bit $[\![b]\!]_{2^k} \in \mathbb{Z}_{2^k}$ to $[\![b]\!]_2 \in \mathbb{F}_2^k$ their subroutines can be used to convert a full input $[\![x]\!]_{2^k} \in \mathbb{Z}_{2^k}$ by bit-decomposing it and then translate each bit into the TinyOT family of protocols.

SPDZ2k and SPDZ. One can use daBits to convert from a $\mathsf{SPDZ}[\mathbb{F}_p]$ share to a $\mathsf{SPDZ}[\mathbb{Z}_{2^k}]$ share. The high level idea of converting between a field and a ring is to generate the same correlated randomness $[\![r]\!]_p \in \mathbb{F}_p$ and $[\![r]\!]_{2^k} \in \mathbb{Z}_{2^k}$ by bit-composing the daBits. Then the conversion from $[\![x]\!]_p$ to $[\![x]\!]_{2^k}$ becomes trivial: parties open $[\![x]\!]_p - [\![r]\!]_p$, assign this to a public y then perform the reduction modulo p in $\mathsf{SPDZ}[\mathbb{Z}_{2^k}]$ using the public constant y i.e. $[\![x]\!]_{2^k} \leftarrow (y + [\![r]\!]_{2^k}) \mod p$. This procedure can be

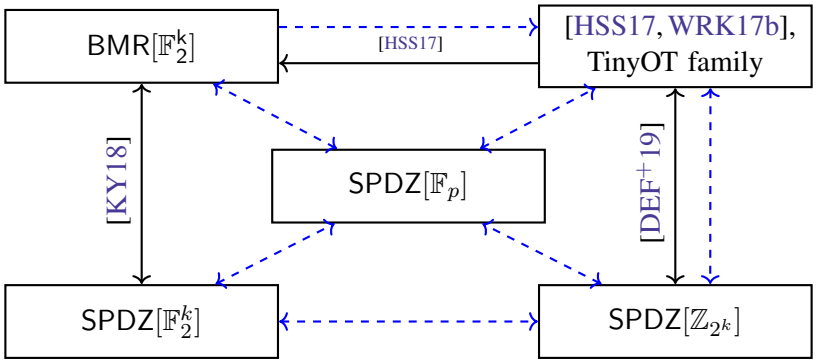

Figure 8.7: Share conversions for dishonest majority protocols. Dashed lines use our daBits as an inner subroutine.

adapted to allow conversions from SPDZ[$\mathbb{Z}_{2^k}$] to SPDZ[$\mathbb{F}_p$]: parties open $[\![x]\!]_{2^k} - [\![r]\!]_{2^k}$ in $\mathbb{Z}_{2^k}$ then add the randomness back in $\mathbb{F}_p$ and truncate the result modulo $\mathbb{Z}_{2^k}$. A similar idea can be applied to convert from SPDZ[$\mathbb{F}_p$] to SPDZ[$\mathbb{F}_2^k$] with the exception that in the last step parties now truncate their shares modulo $\mathbb{F}_2^k$.

SPDZ and TinyOT family. This conversion can be done again with daBits and works in the same way we describe it in our paper for SPDZ[$\mathbb{F}_p$] and BMR[$\mathbb{F}_2^k$]. Recently Aly et al. [AOR+19] improve and fully integrate the conversion between SPDZ and WRK/HSS garbling into SCALE-MAMBA. Their protocol improvements come from a slightly modified check of Damgård et al. [DEF+19] with a twist in how parties extract the least significant bit of a SPDZ share by tweaking their shares locally in the two-party case. They achieve an amortized cost of just one $\mathbb{F}_p$ triple per daBit.

Chapter 9

Future work

Throughout my PhD I have noticed that one does not need to be a genius in order to come up with new ideas (although being clever would be of benefit). It also turned out that being able to write a bit of code can get you in all sorts of collaboration. That or being brave enough to dive into undocumented code and try to make some sense of it - so much fun though frustrating at some times.

The more I think about research the more I see it as a two-step problem solving algorithm: 1) find an interesting problem which can help many others and many would be thankful if it would have been solved then 2) dig deep enough into the problem, understand all the available literature and then start combining ideas, test them or ask more experienced people. The second step can be avoided by choosing a problem which requires a large amount of engineering which no one wants to do it but everyone would like to have this extra tool created. Sadly, sometimes a lot of engineering might not get you any publications due to its lack of "novelty" but would help the community. On the bright side, most likely there will always be a venue which would accept your work if it is useful enough.

Another path that one might take to come up with new ideas is to think hard of a problem X. Then they could stumble upon some adjacent problem Y and realise that it would be really useful to have a solution for Y. As a personal anecdote this is perhaps how the "Marbled Circuits" paper was born: one day I was thinking how to write AES in SPDZ over modulo p circuits and then I realised that writing AES is much easier over $\mathbb{F}_{2^k}$ rather than $\mathbb{F}_p$. Then I have started to ask around about share conversions in SPDZ and turns out no one had any clue how to solve this issue. One year later I have asked Marcel again about this and he pointed me to one of his papers where they needed to convert between SPDZ over $\mathbb{F}_{2^k}$ and BMR garbled circuits. That was a good enough start for me to take Tim on board and start thinking more deeply about this problem. In retrospective "Marbled Circuits" would have not been possible without Tim and I owe him a great deal for accepting to collaborate with me.

My point here is that perhaps there is no perfect recipe on how to solve research problems but some of these ideas worked out well for me: find people who are more clever than you and work together.

Now we have arrived to a list of problems which I consider to be somehow cool if someone would solve them:

Applications of Lookup Tables in $\mathbb{F}_p$. In Section 5.8 we presented an improved protocol for evaluating look-up tables over arithmetic circuits when the inputs are from a prime finite field $\mathbb{F}_p$. One immediate question is how to generate efficiently the preprocessing material required for the online phase of such protocol as well as finding good applications where this could be useful: computing high degree functions for floating and fixed point arithmetic other privacy preserving scientific operations such as logistic regression.

Performance metrics platform for MPC systems. Probably the work I have done that had the most impact during my PhD was creating the Awesome-MPC list [Rot19] where people can find a list with introductory papers to MPC and categorizes briefly existent MPC software based on their description. As simple as this list might be, it sparked some interested into creating an SoK paper by Hastings et al. [HHNZ19] which looked at various frameworks and tested whether their description matched the implementation and classified the software in terms of usability. The next obvious step to do is to take all these frameworks and benchmark their performance on the same type of machines and figure out some approximate "universal" cost metric such as communication data, network type, circuit depth for running some simple programs to then run them on all frameworks. This seems to be a huge engineering effort as the researchers have to integrate different experimental code but the end result we envision might matter: being able to choose a framework based on the (proven in real-time) performance for specific tasks.

Improving compilers for MPC. Due to the rise of mixed protocols or the ability to switch between garbled circuits and secret sharing based frameworks for the two-party semi-honest case [DSZ15], recently for three-parties honest majority [MR18] and as presented in this thesis multiparty with dishonest majority [RW19a] there is a high need for designing compilers which have in mind the costs of switching between different frameworks. There is some work done in this direction called HyCC [BDK+18] and by Ishaq et al. [IMZ19] but the former works only for semi-honest two-party computations used by ABY while the latter is not integrated yet into any MPC framework. Although we split the circuit manually for the SVM in Section 8.5.8 it would be very interesting to decide automatically which parts of the circuit to be evaluated in GC and which using linear secret sharing.

Random bit generation in $\mathbb{F}_p$. This is by far one of the most interesting theoretical problem we consider. In [AOR+19, RST+19] we reduce the problem of generating a daBit to roughly generate a random shared bit $b \xleftarrow{\$} \{0,1\} \in \mathbb{F}_p$. As opposed to GC frameworks where generating random shared bits $b \xleftarrow{\$} \{0,1\} \in \mathbb{F}_2$ is cheap, in the secret shared domain to generate one bit of randomness shared in a large field $\mathbb{F}_p$ has approximately the cost of a full random triple. Perhaps newly introduced techniques by Boyle at al. [BCG+19a, BCG+19b] might be helpful to realize random shared bits "silently".

Bibliography

[AA92] Larry C Andrews and Larry C Andrews.
Special functions of mathematics for engineers.
McGraw-Hill New York, 1992.

[AAUC18] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti.
A survey on homomorphic encryption schemes: Theory and implementation.
ACM Comput. Surv., 51(4):79:1–79:35, July 2018.

[AB09] Sanjeev Arora and Boaz Barak.
Computational complexity: a modern approach.
Cambridge University Press, 2009.

[ABH10] Martin Albrecht, Gregory Bard, and William Hart.
Algorithm 898: Efficient multiplication of dense matrices over GF(2).
ACM Transactions on Mathematical Software (TOMS), 37(1):9, 2010.

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele.
Secure computation on floating point numbers.
In *NDSS 2013*. The Internet Society, February 2013.

[ACK+19] A Aly, D Cozzo, M Keller, E Orsini, D Rotaru, P Scholl, N Smart, and T Wood.
Scale-mamba v1.6 : Documentation, 2019.
https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe.
NewHope without reconciliation.
Cryptology ePrint Archive, Report 2016/1157, 2016.
http://eprint.iacr.org/2016/1157.

[AGP+19] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger.
Feistel structures for MPC, and more.
In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part II*, volume 11736 of *LNCS*, pages 151–171. Springer, Heidelberg, September 2019.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen.

MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity.
In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 191–219. Springer, Heidelberg, December 2016.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar.
A sieve algorithm for the shortest lattice vector problem.
In *33rd ACM STOC*, pages 601–610. ACM Press, July 2001.

[AMMR18] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal.
DiSE: Distributed symmetric-key encryption.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1993–2010. ACM Press, October 2018.

[Ana19] N1 Analytics.
MP-SPDZ, 2019.
https://github.com/n1analytics/MP-SPDZ.

[AOR+19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood.
Zaphod: Efficiently combining lsss and garbled circuits in scale, 2019.

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott.
On the concrete hardness of learning with errors.
Journal of Mathematical Cryptology, 9(3):169–203, 2015.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner.
Ciphers for MPC and FHE.
In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015.

[BBUV19] Ward Beullens, Tim Beyne, Aleksei Udovenko, and Giuseppe Vitto.
Cryptanalysis of the Legendre PRF and generalizations.
Cryptology ePrint Archive, Report 2019/1357, 2019.
https://eprint.iacr.org/2019/1357.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft.
Secure multiparty computation goes live.
In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin.
PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract.
In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 208–225. Springer, Heidelberg, December 2012.

[BCG+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl.
Efficient two-round OT extension and silent non-interactive secure computation.
In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[BCG+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl.
Efficient pseudorandom correlation generators: Silent OT extension and more.
In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth.
Succinct non-interactive arguments via linear interactive proofs.
In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 315–333. Springer, Heidelberg, March 2013.

[BCS19] Carsten Baum, Daniele Cozzo, and Nigel P. Smart.
Using TopGear in overdrive: A more efficient ZKPoK for SPDZ.
In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 274–302. Springer, Heidelberg, August 2019.

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider.
HyCC: Compilation of hybrid protocols for practical secure computation.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 847–861. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias.
Semi-homomorphic encryption and multiparty computation.
In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[Bea92] Donald Beaver.
Efficient multiparty protocols using circuit randomization.
In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[Ben94] Josh Benaloh.
Dense probabilistic encryption.
In *Proceedings of the workshop on selected areas of cryptography*, pages 120–128, 1994.

[Ben17] Aner Ben-Efraim.
On multiparty garbling of arithmetic circuits.
Cryptology ePrint Archive, Report 2017/1186, 2017.
https://eprint.iacr.org/2017/1186.

[Ben18] Aner Ben-Efraim.
On multiparty garbling of arithmetic circuits.
In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2018.

[Ber14] David Bernhard.
Zero-knowledge proofs in theory and practice.
PhD thesis, University of Bristol, 2014.

[BG93] Mihir Bellare and Oded Goldreich.
On defining proofs of knowledge.
In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 390–420. Springer, Heidelberg, August 1993.

[BGKS12] Jean Bourgain, Moubariz Z Garaev, Sergei V Konyagin, and Igor E Shparlinski.
On the hidden shifted power problem.
SIAM Journal on Computing, 41(6):1524–1557, 2012.

[BGM04] Mihir Bellare, Oded Goldreich, and Anton Mityagin.
The power of verification queries in message authentication and authenticated encryption.
Cryptology ePrint Archive, Report 2004/309, 2004.
http://eprint.iacr.org/2004/309.

[BGR95] Mihir Bellare, Roch Guérin, and Phillip Rogaway.
XOR MACs: New methods for message authentication using finite pseudorandom functions.
In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 15–28. Springer, Heidelberg, August 1995.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan.
(Leveled) fully homomorphic encryption without bootstrapping.
In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky.
Circular-secure encryption from decision Diffie-Hellman.

In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 108–125. Springer, Heidelberg, August 2008.

[BHJL17] Fabrice Benhamouda, Javier Herranz, Marc Joye, and Benoît Libert.
Efficient cryptosystems from 2^k-th power residue symbols.
Journal of Cryptology, 30(2):519–549, April 2017.

[BIB89] Judit Bar-Ilan and Donald Beaver.
Non-cryptographic fault-tolerant computing in constant number of rounds of interaction.
In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

[BISW17] Dan Boneh, Yuval Ishai, Amit Sahai, and David J. Wu.
Lattice-based SNARGs and their application to more efficient obfuscation.
In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 247–277. Springer, Heidelberg, April / May 2017.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent.
On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 456–467. ACM Press, October 2016.

[BLP+13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé.
Classical hardness of learning with errors.
In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 575–584. ACM Press, June 2013.

[Blu82] Manuel Blum.
Coin flipping by telephone.
In *Proc. IEEE Spring COMPCOM*, pages 133–137, 1982.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson.
Sharemind: A framework for fast privacy-preserving computations.
In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway.
The round complexity of secure protocols (extended abstract).
In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek.
Garbling gadgets for Boolean and arithmetic circuits.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016.

[BN06] Mihir Bellare and Gregory Neven.

Multi-signatures in the plain public-key model and a general forking lemma.
In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BN08] Mihir Bellare and Chanathip Namprempre.
Authenticated encryption: Relations among notions and analysis of the generic composition paradigm.
Journal of Cryptology, 21(4):469–491, October 2008.

[Bog15] Dan Bogdanov.
Smarter decisions with no privacy breaches, 2015.
https://rwc.iacr.org/2015/Slides/RWC-2015-Bogdanov-final.pdf.

[BOO10] Amos Beimel, Eran Omri, and Ilan Orlov.
Protocols for multiparty coin toss with dishonest majority.
In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 538–557. Springer, Heidelberg, August 2010.

[BOS16] Carsten Baum, Emmanuela Orsini, and Peter Scholl.
Efficient secure multiparty computation with identifiable abort.
In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, October / November 2016.

[BR02] John Black and Phillip Rogaway.
A block-cipher mode of operation for parallelizable message authentication.
In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 384–397. Springer, Heidelberg, April / May 2002.

[BS16] Raphael Bost and Olivier Sanders.
Trick or tweak: On the (in)security of OTR's tweaks.
In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 333–353. Springer, Heidelberg, December 2016.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan.
Fully homomorphic encryption from ring-LWE and security for key dependent messages.
In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Heidelberg, August 2011.

[Can00] Ran Canetti.
Universally composable security: A new paradigm for cryptographic protocols.
Cryptology ePrint Archive, Report 2000/067, 2000.
http://eprint.iacr.org/2000/067.

[Can01] Ran Canetti.
Universally composable security: A new paradigm for cryptographic protocols.

In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey.
Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression.
In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 313–333. Springer, Heidelberg, March 2016.

[CD09] Ronald Cramer and Ivan Damgård.
On the amortized complexity of zero-knowledge protocols.
In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 177–191. Springer, Heidelberg, August 2009.

[Cd10a] Octavian Catrina and Sebastiaan de Hoogh.
Improved primitives for secure multiparty integer computation.
In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

[Cd10b] Octavian Catrina and Sebastiaan de Hoogh.
Secure multiparty linear programming using fixed-point arithmetic.
In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS 2010*, volume 6345 of *LNCS*, pages 134–150. Springer, Heidelberg, September 2010.

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing.
SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority.
In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[CDFG20] Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli.
Mon$\mathbb{Z}_{2^k}$a: Fast maliciously secure two party computation on $\mathbb{Z}_{2^k}$.
In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 357–386. Springer, Heidelberg, May 2020.

[CDXY17] Ronald Cramer, Ivan Damgård, Chaoping Xing, and Chen Yuan.
Amortized complexity of zero-knowledge proofs revisited: Achieving linear soundness slack.
In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 479–500. Springer, Heidelberg, April / May 2017.

[CF01] Ran Canetti and Marc Fischlin.
Universally composable commitments.
In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.

[CGP+12] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain.
Higher-order masking schemes for S-boxes.
In Anne Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 366–384. Springer, Heidelberg, March 2012.

[CKR+19] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh.
Maliciously secure matrix multiplication with applications to private deep learning.
in submission, 2019.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai.
Universally composable two-party and multi-party secure computation.
In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[CLWW16] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu.
Practical order-revealing encryption with limited leakage.
In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 474–493. Springer, Heidelberg, March 2016.

[CP19] Benjamin R. Curtis and Rachel Player.
On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption.
Cryptology ePrint Archive, Report 2019/1148, 2019.
https://eprint.iacr.org/2019/1148.

[CRV14] Jean-Sébastien Coron, Arnab Roy, and Srinivas Vivek.
Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures.
In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 170–187. Springer, Heidelberg, September 2014.

[CS10] Octavian Catrina and Amitabh Saxena.
Secure computation with fixed-point numbers.
In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010.

[CS16] Ana Costache and Nigel P. Smart.
Which ring based somewhat homomorphic encryption scheme is best?
In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 325–340. Springer, Heidelberg, February / March 2016.

[Cv07] Andrew M. Childs and Wim van Dam.
Quantum algorithm for a generalized hidden shift problem.

In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *18th SODA*, pages 1225–1232. ACM-SIAM, January 2007.

[Dam90] Ivan Damgård.
On the randomness of Legendre and Jacobi sequences.
In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 163–172. Springer, Heidelberg, August 1990.

[DEF+19] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev.
New primitives for actively-secure MPC over rings with applications to private machine learning.
In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft.
Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation.
In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, March 2006.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen.
Asynchronous multiparty computation: Theory and implementation.
In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, Heidelberg, March 2009.

[DGN+17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti.
TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation.
In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2263–2276. ACM Press, October / November 2017.

[DJ01] Ivan Damgård and Mats Jurik.
A generalisation, a simplification and some applications of Paillier's probabilistic public-key system.
In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

[DK10] Ivan Damgård and Marcel Keller.
Secure multiparty AES.
In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 367–374. Springer, Heidelberg, January 2010.

[DKL+12] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart.
Implementing AES via an actively/covertly secure dishonest-majority MPC protocol.

In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 241–263. Springer, Heidelberg, September 2012.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart.
Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits.
In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DKS+17] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner.
Pushing the communication barrier in secure computation using lookup tables.
In *NDSS 2017*. The Internet Society, February / March 2017.

[DLR16] Sébastien Duval, Virginie Lallemand, and Yann Rotella.
Cryptanalysis of the FLIP family of stream ciphers.
In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 457–475. Springer, Heidelberg, August 2016.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft.
An empirical study and some improvements of the MiniMac protocol for secure computation.
In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 398–415. Springer, Heidelberg, September 2014.

[DNNR16] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci.
Gate-scrambling revisited - or: The TinyTable protocol for 2-party secure computation.
Cryptology ePrint Archive, Report 2016/695, 2016.
http://eprint.iacr.org/2016/695.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci.
The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited.
In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias.
Multiparty computation from somewhat homomorphic encryption.
In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DPVAR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen.
Nessie proposal: Noekeon.
In *First Open NESSIE Workshop*, pages 213–230, 2000.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner.

ABY - A framework for efficient mixed-protocol secure two-party computation.
In *NDSS 2015*. The Internet Society, February 2015.

[DY15] Nilanjan Datta and Kan Yasuda.
Generalizing PMAC under weaker assumptions.
In Ernest Foo and Douglas Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 433–450. Springer, Heidelberg, June / July 2015.

[DZ13] Ivan Damgård and Sarah Zakarias.
Constant-overhead secure computation of Boolean circuits using preprocessing.
In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 621–641. Springer, Heidelberg, March 2013.

[DZ16] Ivan Damgård and Rasmus Winther Zakarias.
Fast oblivious AES: A dedicated application of the MiniMac protocol.
In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 245–264. Springer, Heidelberg, April 2016.

[EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek.
A pragmatic introduction to secure multi-party computation.
Foundations and Trends® in Privacy and Security, 2(2-3):70–246, 2018.

[Fac19] Facebook.
Crypten library, 2019.
https://crypten.ai/.

[FFFP04] Li Fei-Fei, R Fergus, and P Perona.
Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories.
In *CVPR*, pages 178–178. IEEE, 2004.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl.
A unified approach to MPC with preprocessing using OT.
In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[Gen09] Craig Gentry.
Fully homomorphic encryption using ideal lattices.
In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart.
Homomorphic evaluation of the AES circuit.

In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

[GKWY19] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu.
Efficient and secure multiparty computation from fixed-key block ciphers.
Cryptology ePrint Archive, Report 2019/074, 2019.
https://eprint.iacr.org/2019/074.

[GLR+19] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger.
On a generalization of substitution-permutation networks: The hades design strategy.
Cryptology ePrint Archive, Report 2019/1107, 2019.
https://eprint.iacr.org/2019/1107.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff.
The knowledge complexity of interactive proof-systems (extended abstract).
In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson.
How to play any mental game or A completeness theorem for protocols with honest majority.
In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[Gol95] Oded Goldreich.
Foundations of cryptography:(fragments of a book), 1995.

[Goo19] Google.
Google Trends: crypto vs cryptography, 2019.
https://trends.google.com/trends/explore?date=2016-01-01%202019-12-09&q=cryptography,crypto.

[GRR+16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart.
MPC-friendly symmetric key primitives.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 430–443. ACM Press, October 2016.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic.
SoK: General purpose compilers for secure multi-party computation.
In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

[HIMV19] Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam.
LevioSA: Lightweight secure arithmetic computation.

In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 327–344. ACM Press, November 2019.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg.
TASTY: tool for automating secure two-party computations.
In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010*, pages 451–462. ACM Press, October 2010.

[HM97] Martin Hirt and Ueli M. Maurer.
Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract).
In James E. Burns and Hagit Attiya, editors, *16th ACM PODC*, pages 25–34. ACM, August 1997.

[HM04] Dennis Hofheinz and Jörn Müller-Quade.
Universally composable commitments using random oracles.
In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, Heidelberg, February 2004.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez.
Low cost constant round MPC combining BMR and oblivious transfer.
In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[IMZ19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas.
Efficient MPC via program analysis: A framework for efficient optimal mixing.
In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1539–1556. ACM Press, November 2019.

[JK97] Thomas Jakobsen and Lars R. Knudsen.
The interpolation attack on block ciphers.
In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 28–40. Springer, Heidelberg, January 1997.

[Kho19] Dmitry Khovratovich.
Key recovery attacks on the Legendre PRFs within the birthday bound.
Cryptology ePrint Archive, Report 2019/862, 2019.
https://eprint.iacr.org/2019/862.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas.
Universally composable synchronous computation.
In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[KOR+17] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek.
Faster secure multi-party computation of AES and DES using lookup tables.
In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 229–249. Springer, Heidelberg, July 2017.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl.
Actively secure OT extension with optimal overhead.
In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl.
MASCOT: Faster malicious arithmetic secure computation with oblivious transfer.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru.
Overdrive: Making SPDZ great again.
In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[KRSW18] Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood.
Reducing communication channels in MPC.
In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 181–199. Springer, Heidelberg, September 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider.
Improved garbled circuit: Free XOR gates and applications.
In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[KSS13a] Marcel Keller, Peter Scholl, and Nigel P. Smart.
An architecture for practical actively secure MPC with dishonest majority.
In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.

[KSS13b] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider.
A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design.
Journal of Computer Security, 21(2):283–315, 2013.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer.
Automatic protocol selection in secure two-party computations.

In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14*, volume 8479 of *LNCS*, pages 566–584. Springer, Heidelberg, June 2014.

[KTX07] Akinori Kawachi, Keisuke Tanaka, and Keita Xagawa.
Multi-bit cryptosystems based on lattice problems.
In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 315–329. Springer, Heidelberg, April 2007.

[KY18] Marcel Keller and Avishay Yanai.
Efficient maliciously secure multiparty computation for RAM.
In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 91–124. Springer, Heidelberg, April / May 2018.

[Lak19] Ravie Lakshmanan.
Google open-sources cryptographic tool to keep data sets private, 2019.
https://thenextweb.com/security/2019/06/20/google-open-sources-cryptographic-tool-to-keep-data-sets-private/.

[LDDA12] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran.
Efficient lookup-table protocol in secure multiparty computation.
In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 189–200, 2012.

[Lit19] Dwayne C Litzenberger.
Pycrypto-the python cryptography toolkit.
URL: https://www.dlitz.net/software/pycrypto, 2019.

[LJA+18] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer 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*, COMPASS '18, pages 48:1–48:5, New York, NY, USA, 2018. ACM.

[LK06] Chae Hoon Lim and Tymur Korkishko.
mCrypton - a lightweight block cipher for security of low-cost RFID tags and sensors.
In Jooseok Song, Taekyoung Kwon, and Moti Yung, editors, *WISA 05*, volume 3786 of *LNCS*, pages 243–258. Springer, Heidelberg, August 2006.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart.
Dishonest majority multi-party computation for binary circuits.
In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, Heidelberg, August 2014.

[LP04] Yehuda Lindell and Benny Pinkas.

A proof of Yao's protocol for secure two-party computation.
Cryptology ePrint Archive, Report 2004/175, 2004.
http://eprint.iacr.org/2004/175.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev.
On ideal lattices and learning with errors over rings.
In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart.
Implementing two-party computation efficiently with security against malicious adversaries.
In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 08*, volume 5229 of *LNCS*, pages 2–20. Springer, Heidelberg, September 2008.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai.
Efficient constant round multi-party computation combining BMR and SPDZ.
In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[LPSY16] Atul Luykx, Bart Preneel, Alan Szepieniec, and Kan Yasuda.
On the influence of message length in PMAC's security bounds.
In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 596–621. Springer, Heidelberg, May 2016.

[LR15] Yehuda Lindell and Ben Riva.
Blazing fast 2PC in the offline/online setting with security for malicious adversaries.
In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 579–590. ACM Press, October 2015.

[Lyu09] Vadim Lyubashevsky.
Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures.
In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Heidelberg, December 2009.

[Mar19] AbdelKarim Mardini.
Better password protections in Chrome, 2019.
https://blog.google/products/chrome/better-password-protections/.

[Mau06] Ueli Maurer.
Secure multi-party computation made simple.
Discrete Applied Mathematics, 154(2):370–381, 2006.

[Mic19] Microsoft.
EzPC, 2019.

https://www.microsoft.com/en-us/research/project/ezpc-easy-secure-multi-party-computation/.

[Min14] Kazuhiko Minematsu.
Parallelizable rate-1 authenticated encryption from pseudorandom functions.
In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 275–292. Springer, Heidelberg, May 2014.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet.
Towards stream ciphers for efficient FHE with low-noise ciphertexts.
In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 311–343. Springer, Heidelberg, May 2016.

[MM07] Kazuhiko Minematsu and Toshiyasu Matsushima.
New bounds for PMAC, TMAC, and XCBC.
In Alex Biryukov, editor, *FSE 2007*, volume 4593 of *LNCS*, pages 434–451. Springer, Heidelberg, March 2007.

[MP13] Daniele Micciancio and Chris Peikert.
Hardness of SIS and LWE with small parameters.
In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 21–39. Springer, Heidelberg, August 2013.

[MPI19] MPIR team.
Multiple precision integers and rationals.
https://www.mpir.org, 2019.
Online; accessed September 2019.

[MR18] Payman Mohassel and Peter Rindal.
ABY^3: A mixed protocol framework for machine learning.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[MRSV19] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren.
EPIC: Efficient private image classification (or: Learning from the masters).
In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 473–492. Springer, Heidelberg, March 2019.

[MW19] Eleftheria Makri and Tim Wood.
Full-threshold actively-secure multiparty arithmetic circuit garbling.
Cryptology ePrint Archive, Report 2019/1098, 2019.
https://eprint.iacr.org/2019/1098.

[MZ17] Payman Mohassel and Yupeng Zhang.
SecureML: A system for scalable privacy-preserving machine learning.

In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[NISa] Nist: Aes validation list.
http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html.
Online; accessed 17-November-2019.

[NISb] Nist: Triple des validation list.
http://csrc.nist.gov/groups/STM/cavp/documents/des/tripledesnewval.html.
Online; accessed 17-November-2019.

[NK95] Kaisa Nyberg and Lars R. Knudsen.
Provable security against a differential attack.
Journal of Cryptology, 8(1):27–37, December 1995.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra.
A new approach to practical active-secure two-party computation.
In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner.
Privacy preserving auctions and mechanism design.
EC, 99:129–139, 1999.

[NR97] Moni Naor and Omer Reingold.
Number-theoretic constructions of efficient pseudo-random functions.
In *38th FOCS*, pages 458–467. IEEE Computer Society Press, October 1997.

[NRS14] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton.
Reconsidering generic composition.
In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 257–274. Springer, Heidelberg, May 2014.

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti.
Constant round maliciously secure 2PC with function-independent preprocessing using LEGO.
In *NDSS 2017*. The Internet Society, February / March 2017.

[NWI+13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft.
Privacy-preserving ridge regression on hundreds of millions of records.
In *2013 IEEE Symposium on Security and Privacy*, pages 334–348. IEEE Computer Society Press, May 2013.

[O'D19] Lindsey O'Donnell.
Google Releases Open Source Tool For Computational Privacy, 2019.
https://threatpost.com/google-computational-privacy/145835/.

[Pai99] Pascal Paillier.
Public-key cryptosystems based on composite degree residuosity classes.
In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[PS00] David Pointcheval and Jacques Stern.
Security arguments for digital signatures and blind signatures.
Journal of Cryptology, 13(3):361–396, June 2000.

[PS16] Thomas Peyrin and Yannick Seurin.
Counter-in-tweak: Authenticated encryption modes for tweakable block ciphers.
In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 33–63. Springer, Heidelberg, August 2016.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams.
Secure two-party computation is practical.
In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[PV16] Jürgen Pulkus and Srinivas Vivek.
Reducing the number of non-linear multiplications in masking schemes.
In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 479–497. Springer, Heidelberg, August 2016.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters.
A framework for efficient and composable oblivious transfer.
In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2008.

[Reg05] Oded Regev.
On lattices, learning with errors, random linear codes, and cryptography.
In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[Rog04] Phillip Rogaway.
Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC.
In Pil Joong Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 16–31. Springer, Heidelberg, December 2004.

[Rot19] Dragos Rotaru.

Awesome-MPC, 2019.
https://github.com/rdragos/awesome-mpc.

[RP10] Matthieu Rivain and Emmanuel Prouff.
Provably secure higher-order masking of AES.
In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Heidelberg, August 2010.

[RR16] Peter Rindal and Mike Rosulek.
Faster malicious 2-party secure computation with online/offline dual execution.
In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 297–314. USENIX Association, August 2016.

[RS04] Alexander Russell and Igor E Shparlinski.
Classical and quantum function reconstruction via character evaluation.
Journal of Complexity, 20(2-3):404–422, 2004.

[RSS17] Dragos Rotaru, Nigel P. Smart, and Martijn Stam.
Modes of operation suitable for computing on encrypted data.
IACR Trans. Symm. Cryptol., 2017(3):294–324, 2017.

[RST+19] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood.
Actively secure setup for SPDZ.
Cryptology ePrint Archive, Report 2019/1300, 2019.
https://eprint.iacr.org/2019/1300.

[RSW18] Miruna Rosca, Damien Stehlé, and Alexandre Wallet.
On the ring-LWE and polynomial-LWE problems.
In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 146–173. Springer, Heidelberg, April / May 2018.

[RV13] Arnab Roy and Srinivas Vivek.
Analysis and improvement of the generic higher-order masking scheme of FSE 2012.
In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 417–434. Springer, Heidelberg, August 2013.

[RW19a] Dragos Rotaru and Tim Wood.
MArBled circuits: Mixing arithmetic and Boolean circuits with active security.
In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

[RW19b] Dragos Rotaru and Tim Wood.
MArBled circuits: Mixing arithmetic and Boolean circuits with active security.
Cryptology ePrint Archive, Report 2019/207, 2019.
https://eprint.iacr.org/2019/207.

[RWT+18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar.
Chameleon: A hybrid secure computation framework for machine learning applications.
In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, April 2018.

[Sav98] John E Savage.
Models of computation, volume 136.
Addison-Wesley Reading, MA, 1998.

[SC19] Yongha Son and Jung Hee Cheon.
Revisiting the hybrid attack on sparse secret LWE and application to HE parameters.
In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 11–20. ACM, 2019.

[Sch87] Claus-Peter Schnorr.
A hierarchy of polynomial time lattice basis reduction algorithms.
Theoretical computer science, 53(2-3):201–224, 1987.

[Sch91] Claus-Peter Schnorr.
Factoring integers and computing discrete logarithms via Diophantine approximations.
In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 281–293. Springer, Heidelberg, April 1991.

[sec] secureSCM.
Deliverable D9.2.
https://www1.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf.

[SGRP19] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas.
Make some ROOM for the zeros: Data sparsity in secure distributed machine learning.
In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1335–1350. ACM Press, November 2019.

[Sha79] Adi Shamir.
How to share a secret.
Communications of the Association for Computing Machinery, 22(11):612–613, November 1979.

[Soc04] American Mathematical Society.
The Culture of Research and Scholarship in Mathematics: Joint Research and Its Publication, 2004.
http://www.ams.org/profession/leaders/CultureStatement04.pdf.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa.
Efficient public key encryption based on ideal lattices.
In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635. Springer, Heidelberg, December 2009.

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas.
Path ORAM: an extremely simple oblivious RAM protocol.
In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[Tec19] Unbound Tech.
Unbound Tech, 2019.
https://www.unboundtech.com/.

[Uni19] BU University.
JIFF, JavaScript library for building web-based applications that employ secure multi-party computation (MPC)., 2019.
https://github.com/multiparty/jiff.

[VD02] Wim Van Dam.
Quantum algorithms for weighing matrices and quadratic residues.
Algorithmica, 34(4):413–428, 2002.

[Ver08] Frederik Vercauteren.
The hidden root problem.
In Steven D. Galbraith and Kenneth G. Paterson, editors, *PAIRING 2008*, volume 5209 of *LNCS*, pages 89–99. Springer, Heidelberg, September 2008.

[vHI03] Wim van Dam, Sean Hallgren, and Lawrence Ip.
Quantum algorithms for some hidden shift problems.
In *14th SODA*, pages 489–498. ACM-SIAM, January 2003.

[Vic61] William Vickrey.
Counterspeculation, auctions, and competitive sealed tenders.
The Journal of Finance, 16(1):8–37, 1961.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz.
Authenticated garbling and efficient maliciously secure two-party computation.
In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz.
Global-scale secure multiparty computation.

In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

[Yao82] Andrew Chi-Chih Yao.
Protocols for secure computations (extended abstract).
In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[Yao86] Andrew Chi-Chih Yao.
How to generate and exchange secrets (extended abstract).
In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[Yao19] Andrew Chi-Chih Yao.
Yao Turing Award, 2019.
http://amturing.acm.org/bib/yao_1611524.cfm#bib_6.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans.
Two halves make a whole - reducing data transfer in garbled circuits using half gates.
In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.