Zero-Knowledge Protocols and Multiparty Computation

A PhD Thesis by Valerio Pastro

Supervisor: Ivan Damgård

July 31, 2013

Department of Computer Science

Aarhus University

Contents

Acknowledgements

I would like to thank Ivan Damgård for his guidance and wise advice; he has always been helpful in all sorts of problems, both work-related and otherwise. My thanks also go to Nigel Smart, for many valuable discussions and suggestions on my research; and I would like to express my gratitude to Ronald Cramer, for stimulating my interest on secret sharing schemes and for the opportunity to visit CWI. I am grateful to all my coauthors, who contributed to the publication of the projects I was involved in during my years at Aarhus University.

I wish to thank Yevgeniy Dodis and Oded Regev, for hosting me at New York University, and for our collaboration on cryptography and lattice theory, which has much influenced the development of my academic interests. During my stay abroad, I had the chance to interact with several people from the research groups at IBM Research and Boston University; I would like to thank Daniel Wichs and Ran Canetti for the organisation of those visits, the interesting conversations, and the useful feedback they gave on my work.

Thanks also to the non-academic staff who helped me on countless occasions; assistance provided by Dorthe Haagen Nielsen, Ellen Kjemtrup Lindstrøm, Stephanie Lo, and Rosemary Amico was crucial for the smooth progress of my studies.

Finally, many thanks go to the past and current members and visitors of the cryptography group at Aarhus University, who have been a dependable source of advice and camaraderie throughout my graduate studies, both in research and in wider spheres of life.

Summary

This thesis presents results in two branches of cryptography.

In the first part we construct two general multiparty computation protocols that can evaluate any arithmetic circuit over a finite field. Both are built in the preprocessing model and achieve active security in the setting of a dishonest majority, in which all players but one are controlled by the adversary. In Chapter 5 we present both the preprocessing and the online phase of [**DKL+13**], while in Chapter 2 we describe only the preprocessing phase of [**DPSZ12**] since the combination of this preprocessing phase with the online phase of [**DKL+13**] yields a more efficient protocol than that originally proposed in [**DPSZ12**]. Our preprocessing phases make use of a somewhat homomorphic encryption scheme, and significantly improve on the previous state of the art, both asymptotically and in practice. The online phase we present relies on information-theoretic message authentication codes, requires only a linear amount of data from the preprocessing, and improves on the number of field multiplications needed to perform one secure multiplication (linear, instead of quadratic as in earlier work). The preprocessing phase in Chapter 5 comes in an actively secure flavour and in a covertly secure one, both of which compare favourably to previous work in terms of efficiency and provable security. Moreover, the covertly secure solution includes a key generation protocol that allows players to obtain a public key and shares of a corresponding secret key. In previous work this task was assumed to be performed by an ideal functionality.

In the second part we shift our focus to a task that is related to multiparty computation in an indirect way: we propose a zero-knowledge protocol that allows a prover to show a verifier that he holds a tuple of three values in a finite field, in which the third one is the product of the first two. We extend this protocol in two ways: first we consider the case where the values are integers, and then we consider tuples of values that satisfy more general algebraic relations. Our basic scheme achieves optimal amortized communication complexity, while the construction over the integers improves the state of the art both in terms of communication complexity and in the security requirements (it requires factoring instead of the strong RSA assumption).

Resumé

Denne afhandling præsenterer resultater i to områder i kryptografi.

I den første del konstrueres to generelle multiparty computation protokoller som kan evaluere ethvert aritmetisk kredsløb over et endeligt legeme. Begge protokoller er i præprocesseringsmodellen og opnår sikkerhed mod en uærlig majoritet, hvor alle spillere undtagen n er under kontrol af en modstander. I kapitel 5 præsenterer vi både præprocessering og on-line fasen af [**DKL+13**] mens kapitel 2 kun præsenterer præprocesseringsfasen af [**DPSZ12**], eftersom kombinationen af denne med online fasen fra [**DKL+13**] giver en mere effektiv protokol end den der først var beskrevet i [**DPSZ12**]. Præprocesseringen anvender et somewhat homomorphic kryptosystem der giver væsentlige forbedringer over tidligere resultater, både asymptotisk og i praksis. On-line fasen anvender informationsteoretiske autentificeringskoder, kræver kun arbejde der er lineært i antal spillere (i stedet for kvadratisk som i tidligere resultater). Præprocesseringen fra kapitel 5 findes i en en version med aktiv sikkerhed og en version med covert sikkerhed, begger versioner forbedrer tidligere resultater mht. effektivitet og bevislig sikkerhed. Versionen med covert sikkerhed indeholder også en protokol for nøglegenerering, der producerer en fælles offentlig nøgle og en del til hver spiller af den hemmelige nøgle. I tidligere arbejde var det antaget at denne opgave blev løst af en ideel funktionalitet.

I den anden del flyttes fokus til et primitiv der er relateret indirekte til til multiparty computation: vi foreslår en zero-knowledge protokol der tillader en prover at overbevise en verifier om at han har committet til tre værdier i et endeligt legeme, hvor den tredje værdi er produktet af de to første. Vi udvider denne protokol på to måder. Vi betragter først tilfældet hvor værdierne er heltal, og dernæst tilfældet hvor værdierne opfylder mere generelle algebraiske relationer. Protokollen opnår optimal amortiseret kommunikations kompleksitet mens protokollen for heltalsværdier forbedrer state of the art både mht.kompleksitet og sikkerhedsantagelser (vi antager faktorisering er svært, i stedet for den stærke RSA antagelse).

Introduction

0.1. Prologue

New York City, 1913. The Brooklyn Rapid Transit Company (BRT) is a flourishing transportation company having control of virtually every rapid transit and streetcar operation in the areas of Brooklyn, Queens, and Long Island. In Manhattan the main transportation company is the Interborough Rapid Transit Company (IRT), whose extensive network serves every commuter, from South Ferry to Harlem.

Each of these companies has an incentive to spread their business into the neighbour's area of operation, and the City has an incentive to let both companies serve its residents, since competition improves service and lowers cost for the end users. What are, however, the chances that BRT can provide successful competition to IRT in Manhattan?

One century later, we know what happened: both companies and the City signed the "Dual Contracts", which allowed BRT to operate in Manhattan and IRT to expand to Queens. It was a successful move at first, but eventually led BRT to bankruptcy in 1919, as it struggled to match IRT's turnover amidst the turmoil of World War I. IRT continued its business until 1940, when it was acquired by the NYC Board of Transportation.

Thus, despite the City's efforts a monopoly situation was realised. Couldn't it have been prevented? Perhaps not at the time; however, before the agreement was made, the two companies could have benefited from a little insider information on each other – and with modern tools, such as those described herein, they could have exchanged it securely in a way serving the interest of both.

0.2. The Problem of Secure Function Evaluation

We just described a scenario where two entities want to know which one has the highest turnover, without revealing their revenue to the other. This problem is known in the cryptographic community as the "millionaires' problem", introduced by Yao in [Yao82]. In mathematical terms, it can be rephrased as follows: two parties P_1 and P_2 hold private values x_1 and x_2 respectively (x_i represents P_i's revenue) and they want to compute $j \in \{1, 2\}$ such that $x_j = \max\{x_1, x_2\}$. Moreover, j has to be computed and revealed in such a way that (x_i, j) is the only information that P_i may gain from the procedure.

In general, one can think of a scenario where many players want to compute securely a function that depends on their private inputs. More specifically, let n be a positive integer, and let K be a set (usually K is a finite field, or ring or Abelian group). Let $P_1, \ldots, P_n$ denote the players and let $f : K^n \to K^n$ be the function to be computed. We think of f as a function mapping the private inputs of all players to their respective outputs as follows: if P_i has private input $x_i \in K$, then P_i obtains y_i, where $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$; moreover, y_i is correctly computed and (x_i, y_i) is the only information available to P_i. Protocols achieving these properties are the main topic of this thesis.

0.3. Applications

The mathematical description we gave above is not only an abstraction of the millionaires' problem, but it also neatly characterises many other problems of pertinence in our modern society. Some common examples are provided below:

Set Intersection: Alice's investigation agency wants to check whether any suspects from a secret list appear in the list of passengers in a flight operated by Bob's company. Alice does not want to disclose the list of suspects to Bob, who does not want to reveal the list of passengers to Alice.

Keyword Search: Alice needs some medical data from Bob's database to understand the statistics of a certain disease without telling Bob what kind of data she wants. Bob, on the other hand, cannot give Alice his database of confidential data about his patients.

Auctions: Alice wants to maximise the profits of her sugar-producing company when buying sugar-beets from the farmers. Each farmer wants to sell certain quantities of beets at a certain price, to maximise their own profits, but without revealing the pricing strategy to Alice, nor to any other farmer.[1]

Social Activities: Alice and Bob want to find out whether both of them like each other, but also avoid the disclosure of unrequited love, for fear of ruining their friendship.

Benchmarking: Alice runs a company and wants to know the performance of her company compared to other competitors according to some metric (e.g.: Alice wants to know whether her company's revenue is below or above the average in a set of companies competing in the same market). Neither Alice nor her competitors want to disclose their data to others.[2]

Voting: Alice, Bob, Charlie, and their class mates want to find out who to elect as class president. They only want to reveal who among them gets the most votes (e.g. without disclosing the number of votes per candidate).

0.4. The Ideal World

The above mathematical problem, and all the examples provided so far, can be solved if players have access to a common trusted party T as follows: Each player P_i sends his private input x_i to T, who then computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and sends y_i back to P_i (see Figure 0.1). This solution is called the "ideal world", since it guarantees the required properties in all circumstances.

In principle, it would be possible to drop the requirement that the party computing the function is trustworthy: with the advent of fully homomorphic encryption (FHE) by Gentry [**Gen09**], any party can compute functions on the input by manupulating the encrypted forms directly, without gaining information on the unencrypted input values. More precisely, a fully homomorphic encryption scheme is an encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$[3] equipped with an extra algorithm Eval that allows the computation of any function f on encrypted data such that the resulting ciphertext decrypts to the evaluation of the function on the corresponding plaintexts: if $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^{\mathsf{sec}})$ is a pair consisting of a public key and its associated secret key, then

$$\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Eval}_f(\mathsf{Enc}_{\mathsf{pk}}(m_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(m_n))) = f(m_1, \ldots, m_n). \tag{0.1}$$

Using an FHE scheme, if P_i needs $f_i(x_1, \ldots, x_n)$, where x_j is P_j's private input, then each player P_j can generate $c_j \leftarrow \mathsf{Enc}_{\mathsf{pk}_i}(x_j)$, i.e. an encryption under P_i's public key pk_i, and send it to a

[1] Since 2008, a multiparty computation protocol has been used every year in Denmark to find the clearance price of the sugar-beets market [**BCD+09**].

[2] Notice that this scenario includes the one mentioned in the Prologue.

[3] We assume the reader is familiar with the concept of public key encryption schemes, and refer the curious reader to [**Gol04**], Chapter 5, and [**KL07**], Chapter 10 for details.

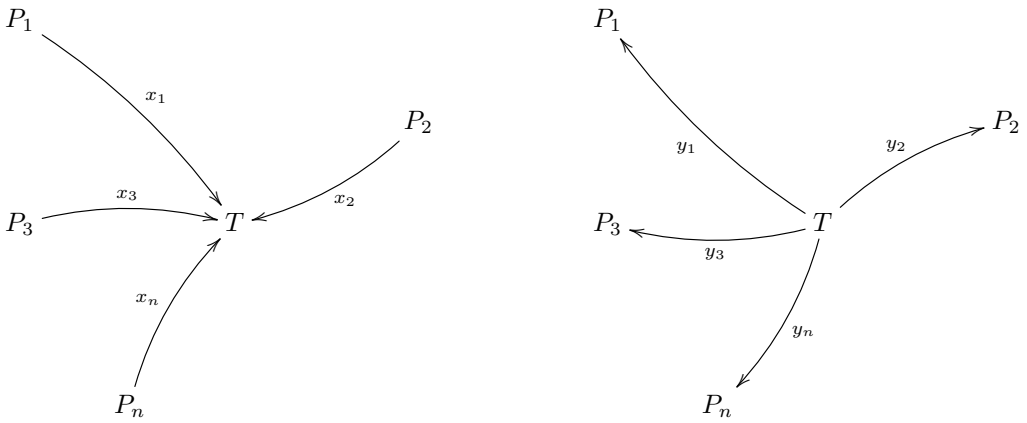


FIGURE 0.1. In the ideal world, the trusted party receives the inputs, computes the function, and returns the corresponding outputs.

designated player $P \neq P_i$ who gathers all encryptions and computes $c^* \leftarrow \mathsf{Eval}_f(c_1, \ldots, c_n)$ and sends c^* to P_i, who can then decrypt it to $f_i(x_1, \ldots, x_n)$.

Even if this approach is theoretically possible, there are some drawbacks, in that this solution is not (currently) practical and does not cover all the possible scenarios. Therefore, we prefer to take a more feasible and classical approach; we also use some ideas from the FHE scenario, but in a less pervasive fashion: we make use of somewhat homomorphic encryption (SHE) schemes, which resemble FHE schemes, but allow a restricted class of functions to be computed (more efficiently) on encrypted data – further details shall be presented later.

0.5. The Real World

In a setting where trust is not always an option, or or may only guaranteed at prohibitive cost, one must find a solution to the problem of secure function evaluation without the need for a trusted party.

One possibility is to let the players communicate via a multiparty computation (MPC) protocol, which is designed to prevent players from exploiting its specifications in order to sabotage security (e.g. by learning more information than they are supposed to, or by making the protocol output incorrect values).

In order to model the players' (mis)behaviour, we introduce an external entity, called "adversary", who has control of a set of players, called "corrupt players" (see Figure 0.3). The players who are not under the control of the adversary are called "honest players". The adversary communicates with a player – and with any agent (i.e. an interactive Turing machine), as we will see – using two special ports:

- the "leakage port", which is an output port that allows the adversary to obtain information and
- the "influence port", which is an input port that allows the adversary to give commands to the player.

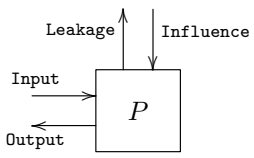


FIGURE 0.2. An agent P and its ports: usually we depict regular input and output port on the side of an agent, and leakage and influence ports on top.

The simplest type of adversary is called "passive adversary", and it acts as follows: it uses a player's influence port to corrupt the player and from that point on all the player's memory and communications are also sent to the adversary via the leakage port. This adversary has the ability to read the memory and to listen to the input and output ports of the corrupt players in order to gain information on the honest players' data. A passive adversary models a coalition of players who follow the protocol and want to share their data in order to gain information on some other players. This is not the only way a protocol can be attacked: more secure protocols take into account a more sophisticated adversary called "active adversary". An active adversary has the same properties as a passive one, but it can also use the influence port of a player to make that player deviate from the protocol. We say furthermore that the adversary is "static" if the set of corrupt players is determined before the execution of the protocol, while we say that it is "adaptive" otherwise.

During the development of multiparty computation, researchers concentrated at first on designing protocols in the presence of passive adversaries (Yao's garbling technique) and later focused on the case of active adversaries [**GMW87**, **CCD88**, **BOGW88**].

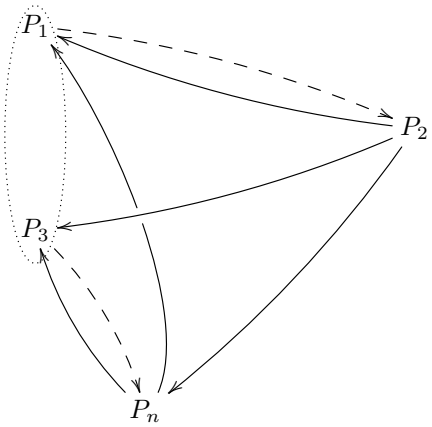


FIGURE 0.3. In the real world players communicate between each other following a protocol (solid lines), and, depending on the corruption model, it may be possible for them to misuse the protocol in order to try to get more information than they are supposed to (dashed lines). In this example P_1 and P_3 are under the control of an adversary (dotted ellipse).

0.6. Security

So far we described what can happen in a protocol, but we never mentioned a method to determine whether a protocol is secure. We assume the reader is familiar with the concept of Universally Composable (UC) Security [**Can01**], and delineate its essential properties here.

First, one has not only to fix a goal for the protocol, but also to design a solution in the ideal world that achieves the goal, i.e. a series of actions performed by the players and the trusted party, where players may only communicate with the trusted party (not between themselves). This solution in the ideal world is the "ideal functionality" that is used as a model for the real protocol. Ideal functionalities are thought of as agents and therefore have leakage and influence ports as well, and their specifications allow limited action from the adversary (e.g. the ideal functionality that models a secure channel between two players leaks the length of each message that either player sends; see Figure 0.4).

Second, one builds a protocol in the real world, where players have access to some resource (i.e. basic functionality assumed to exist, e.g. an authenticated channel), and checks that the protocol acts exactly as does to the corresponding ideal functionality. For this check to be performed, we need to introduce two more agents: the "environment" and the "simulator".

FIGURE 0.4. The ideal functionality modelling a secure channel between two players: upon receiving a message m from one of the two players, it leaks (only) the length $|m|$ and forwards the message to the other player.

We think of the simulator as an agent which compiles the leakage and influence ports of the ideal functionality into the leakage and influence ports of players and resources that appear in the protocol, and of the environment as an agent that tests whether the protocol indeed mimics the actions that the functionality admits. More in detail, the environment has to try to distinguish between runs of the protocol (i.e.: the real world), and runs of the simulator attached to the ideal functionality (i.e.: the ideal world). Formally, we state the security notion as follows: Let Π be a protocol using resource R, let F be an ideal functionality, let $\mathcal{Z}$ be a class of environments (e.g.: computationally bounded agents) and let $\mathcal{S}$ be a class of simulators. We say that Π securely implements F against environments in the class $\mathcal{Z}$ with simulator in $\mathcal{S}$, if for all environments $Z \in \mathcal{Z}$ there exists a simulator $S \in \mathcal{S}$ such that

$$\Pi \diamond R \equiv_Z F \diamond S,$$

where $\diamond$ denotes attachment of agents via ports with the same name, and $\equiv_Z$ means that the environment Z cannot distinguish between two distributions on either side of the equivalence sign (see Figure 0.5).

The two notions of classes of environments and simulators exist in order to specify the type of security one is trying to achieve: usually simulators are picked from a class of computationally bounded agents [**Can01**], while environments may or may not be (in the former case only computational security is guaranteed [**GMW87**, **CLOS02**], whereas in the latter, the stronger notion of statistical security is achieved [**BOGW88**, **CCD88**]).

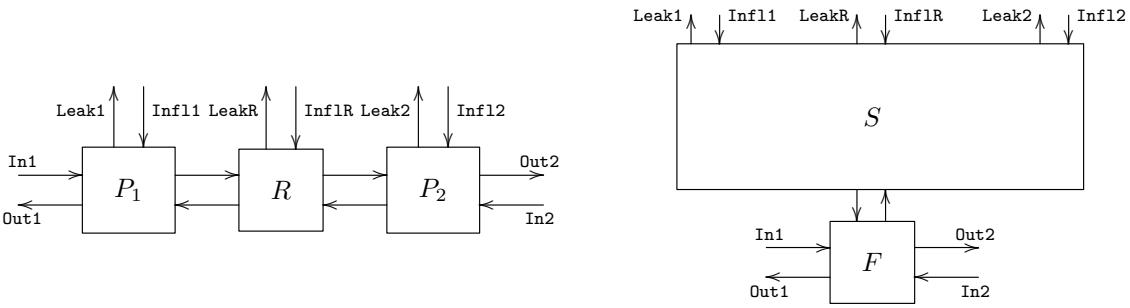


FIGURE 0.5. A protocol Π between P_1 and P_2 using resource R on the left; A simulator S accessing the ideal functionality F on the right. If Π securely implements F, there exist a simulator S such that, for all environments, $\Pi \diamond R$ looks indistinguishable from $F \diamond S$.

0.7. Composability

Protocols satisfying the security notion introduced above are called "Universally Composable". The term comes from a property that it guarantees: specifically, if a protocol Π_G uses a resource F to securely implement the functionality G and the resource F is securely implemented by another protocol Π_F with resource R, then the composite protocol $\Pi_F \diamond \Pi_G$ securely

implements G with access to R. In mathematical terms, if for all environments $Z \in \mathcal{Z}$ there exist simulators $S, T \in \mathcal{S}$ such that

$$\begin{cases} G \diamond T \equiv_Z \Pi_G \diamond F \\ F \diamond S \equiv_Z \Pi_F \diamond R, \end{cases}$$

then the simulator $S \diamond T$[4] is such that

$$G \diamond T \diamond S \equiv_Z \Pi_G \diamond \Pi_F \diamond R.$$

0.8. The SPDZ Protocols

The first part of this thesis is devoted to the construction of a general MPC protocol, i.e. a protocol that allows players to compute any arithmetic circuit f over a finite field. We concentrate on the setting of active security and "dishonest majority", in which all players but one are controlled by the adversary. In this scenario one has to make use of encryption schemes, and therefore design protocols that are only computationally secure, since there cannot exist statistically secure protocols when the number of corrupt players exceeds $(n-1)/2$ (in the case of passive corruption [**CCD88**, **BOGW88**]), or $(n-1)/3$ (in the case of active corruption [**LSP82**]).

Herein we extend a long line of work started with the definition of two party computation by Yao [**Yao82**] and multiparty computation by Goldreich, Micali and Wigderson [**GMW87**] [5], which led to the seminal work of Canetti et al. [**CLOS02**] – formally, the first UC secure MPC protocol – and gave rise to a variety of recent paradigms, such as the "MPC in the head" [**IKOS07**, **IPS08**], and the preprocessing model [**DO10**, **BDOZ11**, **NNOB12**].

Our MPC protocols are in the preprocessing model, which is an approach that divides the players' computation into two phases, described as follows:

The Preprocessing: In this part of the protocol players generate correlated randomness, which is essentially independent of the function f to be computed (it only depends on a lower bound given by the number of multiplication gates in the circuit representing f). Here (and only here!) the players make use of a public key encryption scheme, which implies that the protocol is only computationally secure. The use of an encryption scheme also has an impact on the speed of this phase; on the other hand, the runtime of the preprocessing is only of second importance, given that this stage may be executed at any time before the actual computation of the function.

The Online Phase: This part is free from public key operations and carries out the actual computation of the function f in a statistically secure fashion. The absence of public key operations makes this stage extremely fast to compute.

This thesis shall treat two protocols we designed: the SPDZ [**DPSZ12**] and the SPDZ2 [**DKL+13**]: the more recent one can be thought of as a heavily optimised, tuned-up, and refined version of the first. In the online phase of both of our protocols the arithmetic circuit f is processed gate by gate, and in order to maintain privacy, the computation of each gate is performed on "additively secret shared" values, i.e. for a secret input x player P_i has a uniform value x_i with the constraint $x_1 + \cdots + x_n = x$. Moreover, to preserve correctness, for a secret value x, P_i has a share $\gamma(x)_i$ of an additively secret shared message authentication code (MAC) $\alpha \cdot x =: \gamma(x) = \gamma(x)_1 + \cdots + \gamma(x)_n$ of x, where α is a secret key unknown to any proper subset of players. We defer to later sections the details of how α is distributed and the approach used in order to check MACs later, since the approaches differ between SPDZ and SPDZ2.

[4] In order for the reader to understand the reasons why the class of computationally bounded simulators contains the composition of two simulators, we refer to [**CDN12**], Chapter 2 and 4.

[5] the first information theoretically secure solutions were introduced independently by Ben-Or, Goldwasser, Wigderson [**BOGW88**], and Chaum, Crépeau, Damgård [**CCD88**]

Throughout the thesis, we refer to the above method of distributing values as "angle representation"; more precisely, the angle representation of x is defined as

$$\langle x \rangle := (x_1, \ldots, x_n, \gamma(x)_1, \ldots, \gamma(x)_n),$$ [6]

where P_i has $(x_i, \gamma(x)_i)$,

$$x_1 + \cdots + x_n = x \qquad \text{and} \qquad \gamma(x)_1 + \cdots + \gamma(x)_n = \gamma(x).$$

The choice of additive secret sharings is not only natural in the case of dishonest majority (since all the players must coordinate in order to reconstruct a value shared additively) but also convenient for computational purposes: in fact, since additive secret sharings are linear, the processing of an addition gate can be performed locally by adding the respective shares:

$$\langle x \rangle + \langle y \rangle := (x_1 + y_1, \ldots, x_n + y_n, \gamma(x)_1 + \gamma(y)_1, \ldots, \gamma(x)_n + \gamma(y)_n) = \langle x + y \rangle.$$

This both simplifies the protocol for secure computation of a circuit, and yields better performance when implemented. To process a multiplication gate, on the other hand, we borrow a technique introduced by Beaver [**Bea91**] in the context of circuit randomisation; to compute $\langle x \cdot y \rangle$ given $\langle x \rangle$ and $\langle y \rangle$, if players have $\langle a \rangle, \langle b \rangle, \langle a \cdot b \rangle$ for uniform $a, b \in K$, they run the following protocol:

- Reconstruct $\varepsilon \leftarrow \langle x - a \rangle$ and $\delta \leftarrow \langle y - b \rangle$.
- Compute $\langle z \rangle \leftarrow \langle a \cdot b \rangle + \delta \cdot \langle a \rangle + \varepsilon \cdot \langle b \rangle + \delta \cdot \varepsilon$.

We give a formal discussion of the security of this protocol in Section 5.6, and here we point out only that in a correct run of the protocol, $\langle z \rangle$ is indeed an angle representation of $x \cdot y$:

$$\begin{aligned} z &= a \cdot b + \delta \cdot a + \varepsilon \cdot b + \delta \cdot \varepsilon \\ &= a \cdot b + (y - b) \cdot a + (x - a) \cdot b + (y - b) \cdot (x - a) \\ &= a \cdot b + y \cdot a - a \cdot b + x \cdot b - a \cdot b + x \cdot y - x \cdot b - y \cdot a + a \cdot b \\ &= x \cdot y. \end{aligned}$$

The above sums up the main idea of the online phases for both SPDZ and SPDZ2; in this thesis we present the details of the SPDZ2 online phase only, since it gives several advantages to and generalises the one in SPDZ. The reader will find all the details of the protocols, functionalities and security proofs in Section 5.6.

The approach described above for the online phase requires the players to be able to obtain a "multiplicative triple" $(\langle a \rangle, \langle b \rangle, \langle a \cdot b \rangle)$ for uniform $a, b \in K$ for each multiplication gate that has to be processed. Producing this material is the main goal of the preprocessing phase. The two protocols share some ideas in their preprocessing phase, too: most importantly, the use of a somewhat homomorphic encryption (SHE) scheme that allows the evaluation of quadratic polynomials on encrypted data. Formally, an SHE scheme is an FHE scheme where Equation 0.1 holds only for f in a certain class of functions: as mentioned above, we are interested on is the class of polynomials of degree 2, i.e. circuits of multiplicative depth 1. This relaxation gives us not only more freedom in the design of SHE schemes, but also better performance compared to FHE schemes, which is a crucial factor in our approach. We describe the details of preprocessing and the SHE scheme for SPDZ in Sections 2.1 (abstract properties for SHE scheme), 2.3 (preprocessing phase), and 2.5 (concrete instantiation of SHE scheme), and those for the SPDZ2 in Sections 5.1 (concrete instantiation of the SHE scheme), 5.2 (key generation), and 5.5 (preprocessing phase).

0.9. Contribution

Both our MPC protocols carry important theoretical innovations (see Sections 1.1 and 4.1 for more details):

[6] A reader familiar with [**DPSZ12**] will notice that the above definition does not include the public constant δ used to add and multiply by public scalars; we refer to Section 4.4 for details.

Optimal Online Phase: the online phase of SPDZ (and SPDZ2) requires a linear (in the number of players) amount of data per secret value, as opposed to quadratic, which was the case for the previous state of the art [BDOZ11] and earlier constructions [CLOS02].

A Rational Use of Homomorphic Encryption: Both SPDZ and SPDZ2 make use of a somewhat homomorphic encryption scheme in the preprocessing phase (Sections 2.1, 2.3, 2.5, and Sections 5.1, 5.2, 5.5); this was a novel introduction, since previous constructions were based on more general schemes, like semi-homomorphic schemes in [BDOZ11]. Even if our construction seems specialised relative to previous results (since fewer encryption schemes fit in the category of somewhat homomorphic schemes), the particular properties of these schemes enable extensive simplification of the structure of the preprocessing phase (more precisely, there is no need for an expensive zero-knowledge proof of correct multiplication).

SIMD Optimised Preprocessing: The concrete encryption scheme we introduced, based on [BV11], is designed to take advantage of SIMD operations, which allows a gain in performance (in terms of public key operations) proportional to the batch size.

Recyclable Preprocessed Data: The online protocol of SPDZ2 incorporates an innovative method to check the validity of the computation, which does not affect the security of the unused preprocessed data: this allows players to compute reactive functionalities using a single run of the preprocessing phase, instead of having to perform further runs of the preprocessing phase each time a value is outputted.

Covert Security: In SPDZ2 we fully specify a protocol that is secure against covert adversaries, whose purpose is mainly practical, and its performance is around 10 times better than the actively secure implementation (see Section 6.2 for details).

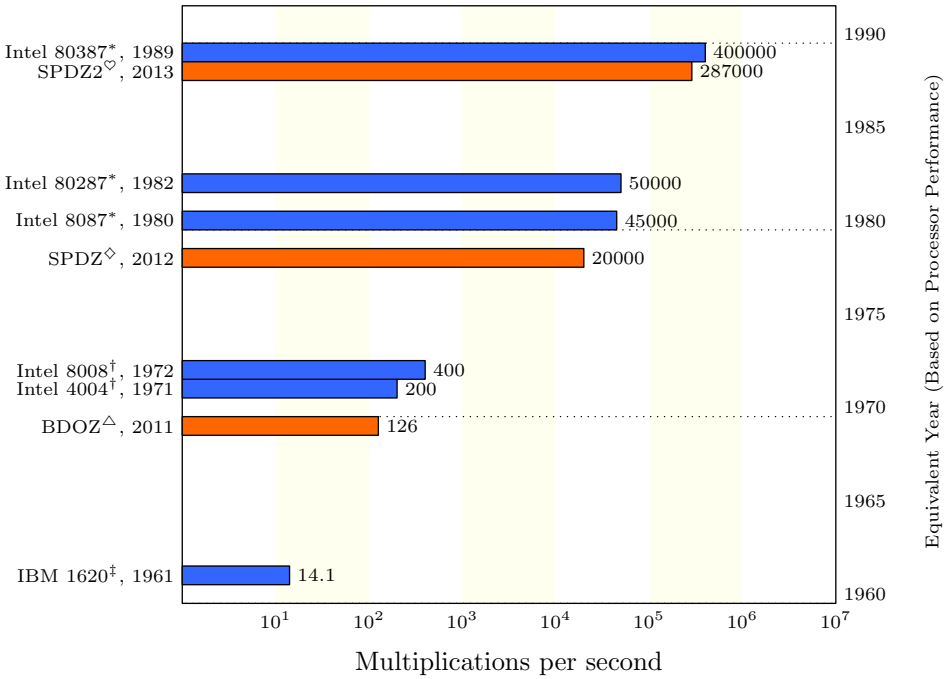
Active Security – Revisited: In SPDZ2 we also propose an approach to achieve active security in the offline phase, based on a technique similar to the one introduced in [NNOB12], rather than on the one presented in SPDZ and [BDOZ11], (which are based on zero-knowledge proofs of knowledge). The alternative method allows better parameters for the cryptosystem while maintaining similar security guarantees compared to SPDZ.

Moreover, our work improves the practical performance of MPC protocols by a significant factor. Since the online approach of [BDOZ11], SPDZ, and SPDZ2 is such that additions are performed locally, their performance depends almost exclusively on the hardware that the players use; multiplications constitute the bottleneck, as they are interactive and may be influenced by network delays. Therefore, we measure performance by the speed achieved by our protocols in terms of multiplications, in a single scenario of three players connected via a LAN, who are performing operations over a 64-bit prime field. In order to give the reader a better idea of the pace at which MPC is evolving, we fit the performance figures retrieved in [BDOZ11] SPDZ and SPDZ2 in a bar diagram that represents the evolution of CPUs and coprocessors in the second half of the 20th century. It turns out that SPDZ exceeds the performance (in computing multiplications) of [BDOZ11] by a measure of around 10 years of computer history, and SPDZ2 by around 20 years (see Figure 0.6).

Our experimental results show significant performance gains also in the preprocessing phase (see Figure 0.7):

0.10. Extra

Even though multiparty computation is the main topic in this thesis, we consider it pertinent to introduce the reader to another result achieved in the course of our studies, which is related to multiparty computation in an indirect way. We present a zero-knowledge protocol that allows a prover P to show a verifier V that P knows three values $x, y, z \in K$ (where K is a finite field or $\mathbb{Z}$) such that $z = x \cdot y$. We construct a basic protocol that achieves optimal amortised



Notes:

- ∗ maths coprocessors, whose quoted performance numbers are evaluated as an average (over the various models) of the nominal performance of 64-bit `FMUL`, cited in the respective production sheets.
- † data extrapolated from the peak MIPS count of the processors, adjusted to match 64-bit operations in the best possible scenario.
- ‡ performance obtained from the data sheet of the processor, adjusted as †.
- ♡ numbers from our implementation of SPDZ2, see [DKL+13] and Section 6.2.
- ◇ numbers from our implementation of SPDZ, see [DPSZ12] and Section 1.1.
- △ numbers obtained in [BDOZ11].

FIGURE 0.6. Online phase performance comparison, measured in 64-bit multiplications per second: blue bars denote the data of CPUs or coprocessors and are vertically spread according to their year of introduction in the market. Red bars denote data of MPC protocols and are placed on the histogram according to the comparative performance to non-distributed solutions. MPC data represents the scenario of 3 players, connected via a LAN.

	2 Players	3 Players
[BDOZ11]†	2.0	N/A
[BDOZ11]‡	4.0	N/A
SPDZ	0.008	0.013
SPDZ2	0.003	0.004

Notes:

- † data obtained from the best preliminary estimate given in [BDOZ11].
- ‡ data obtained from the worst preliminary estimate given in [BDOZ11].

FIGURE 0.7. Preprocessing phase: time spent (in seconds) per triple generated.

communication complexity in the field scenario, and extend it to the integers, where we improve the previous state of the art both in terms of communication complexity, and in the security requirements (our construction over the integers is based on factoring, while previous results

were based on the strong RSA assumption). Moreover, we construct a more general scheme that builds upon the basic technique to prove algebraic relations, rather than multiplicative relations, over finite fields. We give a full introduction to this result in a dedicated section in the second part of the thesis (see Section 7.1).

0.11. Notation

We conclude the introduction by covering some basic notation which is used throughout this thesis. For a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ we denote by $\|\mathbf{x}\|_\infty := \max_{1 \leq i \leq n} |x_i|$, $\|\mathbf{x}\|_1 := \sum_{1 \leq i \leq n} |x_i|$ and $\|\mathbf{x}\|_2 := \sqrt{\sum |x_i|^2}$.

Let $\varepsilon(\kappa)$ denote an unspecified negligible function of κ.

If S is a set, $x \leftarrow S$ denotes assignment to the variable x with respect to a uniform distribution on S; $x \leftarrow s$ for a value s is used as shorthand for $x \leftarrow \{s\}$.

If A is an algorithm, $x \leftarrow A$ denotes assignment to x with respect to the distribution of the output of A (over the random coins of A).

The notation $x := y$ means "x is defined to be y".

Part 1

Multiparty Computation

CHAPTER 1

SPDZ – Overview

1.1. Introduction to SPDZ

In this section we give an overview of the innovations brought by the SPDZ protocol [DPSZ12]. For completeness we give the full description of achievements of the original protocol, but in this thesis we describe the details of the preprocessing phase only, since the online phase can be replaced by the more efficient online phase of SPDZ2 (described in Section 5.6) to yield a more efficient protocol.

1.1.1. Optimal Online Phase. The following italics text concerns all those results of [DPSZ12] not detailed in this thesis.

We propose an MPC protocol in the preprocessing model that computes securely an arithmetic circuit C over any finite field $\mathbb{F}_{p^k}$. The protocol is statistically UC-secure against active and adaptive corruption of up to $n-1$ of the n players, and we assume synchronous communication and secure point-to-point channels. Measured in elementary operations in $\mathbb{F}_{p^k}$ the total amount of work done is $O(n \cdot |C| + n^3)$ where $|C|$ is the size of C. All earlier work in this model had complexity $\Omega(n^2 \cdot |C|)$. A similar improvement is attained for communication complexity and for the storage requirements of the preprocessing. Hence, the work done by each player in the online phase is essentially independent of n. Moreover, it is only a small constant factor larger than what one would need to compute the circuit in the clear. This is the first protocol in the preprocessing model with these properties[1].

We prove a lower bound implying that w.r.t. the amount of data required from the preprocessing, our protocol is optimal up to a constant factor. We also obtain a similar lower bound on the number of bit operations required, and hence the computational work done in our protocol is optimal up to poly-logarithmic factors.

We concern ourselves primarily with the case of large fields, i.e. where the desired error probability is $(1/p^k)^c$, for a small constant c. Note that many applications of MPC need integer arithmetic, modular reductions, conversion to binary, etc., which we can emulate by computing in $\mathbb{F}_p$ with p large enough to avoid overflow. This naturally leads to computing with large fields. Our protocol works for all fields, but like earlier work in this model, it is less efficient for small fields by a factor of essentially $\lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ for error probability $2^{-\Theta(\mathsf{sec})}$. See Section 3.3 for details.

Our result requires innovative ideas beyond those of [DPSZ12], the previous state of the art, which was based on additive secret sharing, where each share of a secret is authenticated using an information theoretic Message Authentication Code (MAC). Since each player needs to have his own key, each of the n shares need to be authenticated with n MACs, so this approach is inherently quadratic in n. Our idea is to authenticate the secret value itself instead of the shares, using a single global key. This seems to lead to a "chicken and egg" problem since to check a MAC requires possession of the key, but knowledge of the key yields the ability to forge MACs. Our solution to this involves secret sharing the key as well, and we optimise by means of various tricks to reduce the amortised cost of checking a set of MACs.

[1] With dishonest majority, successful termination cannot be guaranteed, so our protocols simply abort if cheating is detected. We do not, however, identify *who* cheated; indeed the standard definition of secure function evaluation does not require this. Identification of cheaters is possible but we do not know how to do this while maintaining complexity linear in n.

1.1.2. Efficient Use of FHE for MPC. As a conceptual contribution we propose what we believe is "the right" way to use FHE/SHE for *computationally* efficient MPC, namely, to use it for implementing a preprocessing phase. The observation is that since such preprocessing is typically based on the classic circuit randomisation technique of Beaver [Bea91], it can be done by evaluating in parallel many small circuits of small multiplicative depth (in fact, depth 1 in our case). Thus, SHE suffices; we do not need bootstrapping, and we can use the SHE SIMD approach of [SV11] to handle many values in parallel in a single ciphertext.

To capitalise on this idea, we apply the SIMD approach to the cryptosystem from [BV11] (see also [GHS12a], which employs a similar technique). To optimise performance, we need to do a non-trivial analysis of the parameter values we can use, and we prove some results on norms of embeddings of a cyclotomic field for this purpose. We also design a distributed decryption procedure for our cryptosystem, which is only robust against passive attacks. Nevertheless, this is sufficient for the overall protocol to be actively secure. Intuitively, this is the case because the only damage the adversary can do is to add a known error term to the decryption result obtained. The effect in the online protocol is that certain shares of secret values may be incorrect, but such anomalies will be caught when checking the MACs. Finally, we adapt a zero-knowledge proof of plaintext knowledge from [BDOZ11] for our purpose and in particular we improve the analysis of the soundness guarantees it offers. This influences the choice of parameters for the cryptosystem and improves overall performance.

1.1.3. An Efficient Preprocessing Protocol. As a result of the above, we obtain a constant-round preprocessing protocol that is, UC-secure against active and static corruption of $n-1$ players assuming the underlying cryptosystem is semantically secure, which follows from the Polynomial Learning with Errors (PLWE) assumption. UC-security for dishonest majority cannot be obtained without a set-up assumption. We assume that a key pair for our cryptosystem has been generated and the secret key has been shared among the players.

Whereas previous work in the preprocessing/online model [BDOZ11, DO10] use $\Omega(n^2)$ public-key operations per secure multiplication, we only need $O(n^2/s)$ operations, where s is a number that grows with the security parameter of the SHE scheme (we have $s \approx 12000$ in our concrete instantiation for computing in $\mathbb{F}_p$ where $p \approx 2^{64}$). We stress that our adapted scheme is exactly as efficient as the basic version of [BV11] that does not allow this optimisation, so the improvement is indeed "genuine".

Relative to the case where FHE is used throughout the protocol, our combined preprocessing and online phase is incomparable from a theoretical point of view, but much more practical: we need more communication and rounds, but the computational overhead is much smaller – we need $O(n^2/s \cdot |C|)$ public key operations compared to $O(n \cdot |C|)$ for the FHE approach, where for realistic values of n and s, we have $n^2/s \ll n$. Furthermore, we only need a low depth SHE which is much more efficient than FHE. And finally, we can push all the work using SHE into a (function independent) preprocessing phase.

1.1.4. Performance in Practice. Both the preprocessing and online phase have been implemented and tested for 3 players on up-to-date machines connected on a LAN. The preprocessing takes about 13 ms amortised time to prepare one multiplication in $\mathbb{F}_p$ for a 64-bit p, with security level corresponding roughly to 1024 bit RSA and an error probability of 2^{-40} for the zero-knowledge proofs (the error probability can be lowered to 2^{-80} by repeating the ZK proofs which will at most double the time). This is 2-3 orders of magnitude faster than preliminary estimates for the most efficient instantiation of [BDOZ11]. The online phase executes a secure 64-bit multiplication in 0.05 ms amortised time. These rough orders of magnitude, and the ability to deal with a non-trivial number of players, are obtained in a recent implementation of the protocols, described in [DKL+12].

1.1.5. Concurrent Related Work. In recent independent work [MSas11, AJLA+12, GHS12a], Meyers at al., Asharov et al. and Gentry et al. also use an FHE scheme for multiparty computation. They follow the pure FHE approach mentioned above, using a threshold

decryption protocol tailored to the specific FHE scheme they use. They focus primarily on round complexity, whereas our purpose is on minimising the computational overhead. We note that in [**GHS12a**], Gentry et al. obtain small overhead by showing a way to use the FHE SIMD approach for computing any circuit homomorphically. However, this requires full FHE with bootstrapping (to work on arbitrary circuits) and does not (currently) lead to a practical protocol.

In [**NNOB12**], Nielsen et al. consider secure computing for Boolean Circuits. Their online phase is similar to that of [**BDOZ11**], while the preprocessing is a very efficient construction based on Oblivious Transfer (OT). This result is complementary to ours in the sense that we target computations over large fields which are good for some applications whereas for other cases, Boolean Circuits are the most compact way to express the desired computation. It would be possible to use the preprocessing from [**NNOB12**] to set up data for our online phase, but current benchmarks indicate that our approach is faster for large fields, say of size 64 bits or more.

CHAPTER 2

SPDZ – Preprocessing

2.1. The Abstract Somewhat Homomorphic Encryption Scheme

In this section we specify the abstract properties needed for our cryptosystem. A concrete instantiation is presented in Section 2.5.

We first define the plaintext space M, given by a direct product $(\mathbb{F}_{p^k})^s$ of finite fields of characteristic p. Component-wise addition and multiplication of elements in M are denoted by $+$ and $\cdot$. We assume there is an injective encoding function encode which takes elements in $(\mathbb{F}_{p^k})^s$ to elements in a ring R, equal to $\mathbb{Z}^N$ (as a $\mathbb{Z}$-module) for some integer N. We also assume a decode function which takes *arbitrary* elements in $\mathbb{Z}^N$ and returns elements in $(\mathbb{F}_{p^k})^s$. We require that for all $\mathbf{m} \in M$ that $\mathsf{decode}(\mathsf{encode}(\mathbf{m})) = \mathbf{m}$, that the decode operation is compatible with the characteristic of the field, i.e. for any $\mathbf{x} \in \mathbb{Z}^N$ we have $\mathsf{decode}(\mathbf{x}) = \mathsf{decode}(\mathbf{x} \pmod p)$, and finally, that the encoding function produces "short" vectors: more precisely, that for all $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq \tau$ where $\tau = p/2$.

The two operations in R are denoted by $+$ and $\cdot$. The addition operation in R is assumed to be component-wise addition, whereas we make no assumption on multiplication. All we require is that the following properties hold, for all elements $\mathbf{m}_1, \mathbf{m}_2 \in M$:

$$\mathsf{decode}(\mathsf{encode}(\mathbf{m}_1) + \mathsf{encode}(\mathbf{m}_2)) = \mathbf{m}_1 + \mathbf{m}_2,$$
$$\mathsf{decode}(\mathsf{encode}(\mathbf{m}_1) \cdot \mathsf{encode}(\mathbf{m}_2)) = \mathbf{m}_1 \cdot \mathbf{m}_2.$$

From now on, when we discuss the plaintext space M we assume it comes implicitly with the encode and decode functions for some integer N. If an element in M has the same component in each of the s coordinates, then we call it a "diagonal" element. For $x \in \mathbb{F}_{p^k}$ we let $\mathsf{Diag}(x)$ denote the element $(x, x, \ldots, x) \in (\mathbb{F}_{p^k})^s$.

Our cryptosystem consists of a tuple $(\mathsf{ParamGen}, \mathsf{KeyGen}, \mathsf{KeyGen}^*, \mathsf{Enc}, \mathsf{Dec})$ of algorithms, defined below, and parametrised by a security parameter κ.

$\mathsf{ParamGen}(1^\kappa, M)$: This parameter generation algorithm outputs an integer N (as above), definitions of the encode and decode functions, and a description of a randomised algorithm D^d_ρ, which outputs vectors in $\mathbb{Z}^d$. We assume that D^d_ρ outputs $\mathbf{r}$ with $\|\mathbf{r}\|_\infty \leq \rho$, except with negligible probability. The algorithm D^d_ρ is used by the encryption algorithm to select the randomness needed during encryption. The algorithm $\mathsf{ParamGen}$ also outputs an additive Abelian group G, also provided with a (not necessarily closed) multiplicative operator, which is commutative and distributes over the addition in G. The group G is the group in which the ciphertexts are assumed to lie. We write $\boxplus$ and $\boxtimes$ for the operations on G, and extend these in the natural way to vectors and matrices of elements of G. Finally, $\mathsf{ParamGen}$ outputs a set C of allowable arithmetic SIMD circuits over $(\mathbb{F}_{p^k})^s$, these are the set of functions which our scheme is able to evaluate ciphertexts over. We can think of C as a subset of $\mathbb{F}_{p^k}[X_1, X_2, \ldots, X_n]$, where we evaluate a function $f \in \mathbb{F}_{p^k}[X_1, X_2, \ldots, X_n]$ a total of s times in parallel on inputs from $(\mathbb{F}_{p^k})^n$. We assume that all other algorithms take as implicit input the output $P \leftarrow (1^\kappa, N, \mathsf{encode}, \mathsf{decode}, \mathsf{D}^d_\rho, G, C)$ of $\mathsf{ParamGen}$.

$\mathsf{KeyGen}()$: This algorithm outputs a public key pk and a secret key sk.

$\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$: On input of $\mathbf{x} \in \mathbb{Z}^N$, $\mathbf{r} \in \mathbb{Z}^d$, this deterministic algorithm outputs a ciphertext $c \in G$. When applying this algorithm one would obtain $\mathbf{x}$ from the application of the encode function, and $\mathbf{r}$ by calling D^d_ρ. This is what we mean when we write $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$, where $\mathbf{m} \in M$. However,

it is convenient for us to define Enc on the intermediate state, $\mathbf{x} = \mathsf{encode}(\mathbf{m})$. To ease notation we write $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x})$ if the value of the randomness $\mathbf{r}$ is not important for our discussion. To make our zero-knowledge proofs below work, we require that addition of V "clean" ciphertexts (for "small" values of V), of plaintext $\mathbf{x}_i$ in $\mathbb{Z}^N$, using randomness $\mathbf{r}_i$, results in a ciphertext which could be obtained by adding the plaintexts and randomness, as integer vectors, and then applying $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$, i.e.

$$\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_1 + \cdots + \mathbf{x}_V, \mathbf{r}_1 + \cdots + \mathbf{r}_V) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_1, \mathbf{r}_1) \boxplus \cdots \boxplus \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_V, \mathbf{r}_V).$$

$\mathsf{Dec}_{\mathsf{sk}}(c)$: On input the secret key and a ciphertext c it returns either an element $\mathbf{m} \in M$, or the symbol $\perp$.

We are now able to define various properties of the above abstract scheme that we require. WE begin with some notation: for a function $f \in C$ let $n(f)$ denote the number of variables in f, and let $\widehat{f}$ denote the function on G induced by f. That is, given f, replace every $+$ operation with a $\boxplus$, every $\cdot$ operation is replaced with a $\boxtimes$ and every constant c is replaced by $\mathsf{Enc}_{\mathsf{pk}}(\mathsf{encode}(c), \mathbf{0})$. Also, given a set of $n(f)$ vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{n(f)}$, we define $f(\mathbf{x}_1, \ldots, \mathbf{x}_{n(f)})$ in the natural way by applying f in parallel on each coordinate.
Correctness: Intuitively correctness means that if one decrypts the result of a function $f \in C$ applied to $n(f)$ encrypted vectors of variables, then this should return the same value as applying the function to the $n(f)$ plaintexts. However, to apply the scheme in our protocol, we need to be somewhat more liberal, namely, the decryption result should be correct, even if the ciphertexts we start from were not necessarily generated by the normal encryption algorithm. They only need to "contain" encodings and randomness that are not too large, such that the encodings decode to legal values. Formally, the scheme is said to be (B_{plain}, B_{rand}, C)-*correct* if

$$\begin{aligned}\Pr\Big[& P \leftarrow \mathsf{ParamGen}(1^\kappa, M), \quad (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(), \quad \text{for any } f \in C, \\ & \text{any } \mathbf{x}_i, \mathbf{r}_i, \text{ with } \|\mathbf{x}_i\|_\infty \le B_{plain}, \ \|\mathbf{r}_i\|_\infty \le B_{rand}, \ \mathsf{decode}(\mathbf{x}_i) \in (\mathbb{F}_{p^k})^s, \\ & i = 1, \ldots, n(f), \text{ and } c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i), \ c \leftarrow \widehat{f}(c_1, \ldots, c_{n(f)}) : \\ & \mathsf{Dec}_{\mathsf{sk}}(c) \neq f(\mathsf{decode}(\mathbf{x}_1), \ldots, \mathsf{decode}(\mathbf{x}_{n(f)})) \Big] < \varepsilon(\kappa).\end{aligned}$$

We say that a ciphertext is (B_{plain}, B_{rand}, C)-*admissible* if it can be obtained as the ciphertext c in the above experiment, i.e. by applying a function from C to ciphertexts generated from (legal) encodings and randomness that are bounded by B_{plain} and B_{rand}.
$\mathsf{KeyGen}^*()$: This is a randomised algorithm that outputs a *meaningless public key* $\widetilde{\mathsf{pk}}$. We require that an encryption of any message $\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(\mathbf{x})$ is statistically indistinguishable from an encryption of 0. Furthermore, if we set $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ and $\widetilde{\mathsf{pk}} \leftarrow \mathsf{KeyGen}^*()$, then pk and $\widetilde{\mathsf{pk}}$ are computationally indistinguishable. This implies the scheme is IND-CPA secure in the usual sense.
Distributed Decryption: We assume that a common public key has been set up where the secret key has been secret-shared among the players in such a way that they must collaborate to decrypt a ciphertext. We assume throughout that only (B_{plain}, B_{rand}, C)-admissible ciphertexts are to be decrypted, and this constraint is guaranteed by our main protocol.

We note that some set-up assumption is always required to show UC security, which is our present goal. Concretely, we assume that a functionality $\mathcal{F}_{\text{KeyGen}}$ is available, as specified in Figure 2.1. It basically generates a key pair and secret-shares the secret key among the players using a secret-sharing scheme that is assumed to be given as part of the specification of the cryptosystem. Since we want to allow corruption of all but one player, the maximal unqualified sets must be all sets of $n - 1$ players.

We note that it is possible to make a weaker set-up assumption, such as a common reference string (CRS), and using a general UC secure multiparty computation protocol for the CRS model

Functionality $\mathcal{F}_{\text{KeyGen}}$

(1) When receiving "start" from all honest players, run $P \leftarrow \mathsf{ParamGen}(1^\kappa, M)$, and then, using the parameters generated, run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ (recall P, and hence 1^κ, is an implicit input to all functions we specify). Send pk to the adversary.
(2) We assume a secret sharing scheme is given with which sk can be secret-shared. Receive from the adversary a set of shares s_j for each corrupted player P_j.
(3) Construct a complete set of shares $(s_1, \ldots, s_n)$ consistent with the adversary's choices and sk. Note that this is always possible since the corrupted players form an unqualified set. Send pk to all players and s_i to each honest P_i.

FIGURE 2.1. The Ideal Functionality for Distributed Key Generation

to implement $\mathcal{F}_{\text{KeyGen}}$. While this may not be very efficient, this protocol needs to be run only once in the life-time of the system.

We also want our cryptosystem to implement the functionality $\mathcal{F}_{\text{KeyGenDec}}$ specified in Figure 2.2, which essentially specifies that players can cooperate to decrypt a (B_{plain}, B_{rand}, C)-admissible ciphertext, but the protocol is only secure against a passive attack: the adversary gets the correct decryption result, but can decide which result the honest players should learn.

Functionality $\mathcal{F}_{\text{KeyGenDec}}$

(1) When receiving "start" from all honest players, run $\mathsf{ParamGen}(1^\kappa, M)$, and then, using the parameters generated, run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$. Send pk to the adversary and to all players, and store sk.
(2) Hereafter on receiving "decrypt c" for (B_{plain}, B_{rand}, C)-admissible c from all honest players, send c and $m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$ to the adversary. On receiving m' from the adversary, send "Result m'" to all players, Both m and m' may be a special symbol $\perp$ indicating that decryption failed.
(3) On receiving "decrypt c to P_j" for admissible c, if P_j is corrupt, send $c, m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$ to the adversary. If P_j is honest, send c to the adversary. On receiving δ from the adversary, if $\delta \notin M$, send $\perp$ to P_j, if $\delta \in M$, send $\mathsf{Dec}_{\mathsf{sk}}(c) + \delta$ to P_j.

FIGURE 2.2. The Ideal Functionality for Distributed Key Generation and Decryption

We are now finally ready to define the basic set of properties that the underlying cryptosystem should satisfy, in order to be used in our protocol. Here we use an "information theoretic" security parameter sec that controls the errors in our ZK proofs below.

DEFINITION 2.1 (Admissible Cryptosystem). We say that a cryptosystem is *admissible* if the following two properties hold:

Correctness: If C is a set of formulas containing circuits of multiplicative depth one, that is, of form

$$(x_1 + \cdots + x_n) \cdot (y_1 + \cdots + y_n) + z_1 + \cdots + z_n,$$

as well as all "smaller" formulas , i.e. with a smaller number of additions and possibly no multiplication, then the cryptosystem is (B_{plain}, B_{rand}, C)-correct, where

$$B_{plain} = N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}}, \quad B_{rand} = d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}};$$

and where $\nu > 0$ can be an arbitrary constant.

Distributed Decryption: There exists a secret sharing scheme as required in $\mathcal{F}_{\text{KeyGen}}$ and a protocol Π_{DDec} with the property that when composed with $\mathcal{F}_{\text{KeyGen}}$ it securely implements the functionality $\mathcal{F}_{\text{KeyGenDec}}$.

The set C is defined to contain all computations on ciphertext that we need in our main protocol. We assume that B_{plain}, B_{rand} are defined as here in terms of τ, ρ and sec. This is because these are the bounds we can force corrupt players to respect via our zero-knowledge protocol, as shall be seen.

2.2. Zero-Knowledge Proof of Plaintext Knowledge

This section presents a zero-knowledge protocol that takes as input sec ciphertexts $c_1, \ldots, c_{\mathsf{sec}}$ generated by one of the players in our protocol, who acts as the prover. If the prover is honest then $c_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i)$, where $\mathbf{x}_i$ has been obtained from the encode function, i.e. $\|\mathbf{x}_i\|_\infty \leq \tau$, and $\mathbf{r}_i$ has been generated from D_ρ^d (so we may assume that $\|\mathbf{r}_i\|_\infty \leq \rho$). Our protocol is a zero-knowledge proof of plaintext knowledge (ZKPoPK) for the following relation:

$$\begin{aligned} R_{\mathsf{PoPK}} = \big\{(x,w) \,|\, & x = (\mathsf{pk}, \mathbf{c}), w = ((\mathbf{x}_1, \mathbf{r}_1), \ldots, (\mathbf{x}_{\mathsf{sec}}, \mathbf{r}_{\mathsf{sec}})) : \\ & \mathbf{c} = (c_1, \ldots, c_{\mathsf{sec}}), c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i), \\ & \|\mathbf{x}_i\|_\infty \leq B_{plain}, \mathsf{decode}(\mathbf{x}_i) \in (\mathbb{F}_{p^k})^s, \|\mathbf{r}_i\|_\infty \leq B_{rand} \big\}. \end{aligned}$$

The zero-knowledge and completeness properties hold only if the ciphertexts c_i satisfy $\|\mathbf{x}_i\|_\infty \leq \tau$ and $\|\mathbf{r}_i\|_\infty \leq \rho$.

In our preprocessing protocol players are required to give such a ZKPoPK for all ciphertexts they provide. By admissibility of the cryptosystem, every ciphertext occurring in the protocol is (B_{plain}, B_{rand}, C)-admissible and can therefore be decrypted correctly. The ZKPoPK can also be called with a flag diag which modifies the proof so that it additionally proves that $\mathsf{decode}(\mathbf{x}_i)$ is a diagonal element.

The protocol is not meant to implement an ideal functionality, but we can still use it and prove UC security for the main protocol, since we always generate the challenge $\mathbf{e}$ by calling the $\mathcal{F}_{\text{RAND}}$ ideal functionality given in Figure 2.3.

Functionality $\mathcal{F}_{\text{RAND}}$

Random Sample: When receiving "rand" from all parties, it samples a uniform $r \leftarrow \{0,1\}^u$ and outputs r to all parties.

Random modulo p: When receiving "rand, p" from all parties, it samples a uniform value $e \leftarrow \mathbb{F}_{p^k}$ and outputs e to all parties.

FIGURE 2.3. The ideal functionality for coin-flipping.

Hence the honest-verifier ZK property implies straight-line simulation[1]. As for knowledge extraction, the UC simulator for the security proof knows the secret key for the cryptosystem and can therefore extract a dishonest prover's witness simply by decrypting. In the reduction to show that the simulator works, we do not know the secret key, but here we are allowed to do extraction by rewinding.

We give two versions of the protocol: the first one is a standard 3-move protocol, while the second one uses an "abort" technique to optimise the parameter values. The latter one makes use of the Fiat-Shamir heuristic, and may be the best option for a practical implementation.

2.2.1. Interactive Zero-Knowledge Proof. For the protocol, set $\tau = p/2$, so that $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq \tau = p/2$. This means that each entry in $\mathsf{encode}(\mathbf{m})$ corresponds to a uniquely determined residue mod p and conversely each such residue is uniquely determined by $\mathbf{m}$. Note that if there exists an $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ such that $\mathbf{x} \bmod p = \mathsf{encode}(\mathbf{m})$ then $\mathsf{decode}(\mathbf{x}) = \mathbf{m}$. Therefore, the verifier explicitly checks whether the encodings the prover sends him decode to legal values, so that the ciphertexts in question also decode to legal values.

Let R denote the matrix in $\mathbb{Z}^{\mathsf{sec} \times d}$ whose ith row is $\mathbf{r}_i$. Our protocol makes use of a matrix $M_{\mathbf{e}}$ defined as follows. Let $V := 2 \cdot \mathsf{sec} - 1$. For $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ we define $M_{\mathbf{e}} \in \mathbb{Z}^{V \times \mathsf{sec}}$ to be the matrix whose (i,k)-th entry is given by $\mathbf{e}_{i-k+1}$, for $1 \leq i - k + 1 \leq \mathsf{sec}$ and 0 otherwise.

THEOREM 2.2. *The protocol* Π_{ZKPoPK} *(Figure 2.4) is an honest-verifier zero-knowledge proof of knowledge for the relation* R_{PoPK}.

[1] $\mathcal{F}_{\text{RAND}}$ can be implemented by standard methods, and the complexity of this is not significant for the main protocol since we may use the same challenge for many instances of the proof, and each proof handles sec ciphertexts.

Protocol Π_{ZKPoPK}

- For $i = 1, \ldots, V$, the prover sets $\mathbf{y}_i \leftarrow \mathbb{Z}^N$ and $\mathbf{s}_i \leftarrow \mathbb{Z}^d$, such that $\|\mathbf{y}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$ and $\|\mathbf{s}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. For $\mathbf{y}_i$, this is done as follows: choose a random message $\mathbf{m}_i \in (\mathbb{F}_{p^k})^s$ and set $\mathbf{y}_i = \mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$, where each entry in $\mathbf{u}_i$ is a multiple of p, chosen uniformly at random, subject to $\|\mathbf{y}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. If diag is set to true, then the $\mathbf{m}_i$ are chosen to be diagonal elements.
- The prover computes $a_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i)$, for $i = 1, \ldots, V$, and defines $S \in \mathbb{Z}^{V \times d}$ to be the matrix whose ith row is $\mathbf{s}_i$ and sets $\mathbf{y} \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_V)$, $\mathbf{a} \leftarrow (a_1, \ldots, a_V)$.
- The prover sends $\mathbf{a}$ to the verifier.
- The verifier selects $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ and sends it to the prover.
- The prover sets $\mathbf{z} \leftarrow (\mathbf{z}_1, \ldots, \mathbf{z}_V)$, such that $\mathbf{z}^\mathsf{T} = \mathbf{y}^\mathsf{T} + M_\mathbf{e} \cdot \mathbf{x}^\mathsf{T}$, and $T = S + M_e \cdot R$. The prover sends $(\mathbf{z}, T)$ to the verifier.
- The verifier computes $d_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$, for $i = 1, \ldots, V$, where $\mathbf{t}_i$ is the ith row of T and sets $\mathbf{d} \leftarrow (d_1, \ldots, d_V)$.
- The verifier checks that $\mathsf{decode}(\mathbf{z}_i) \in \mathbb{F}_{p^k}^s$ and whether the following three conditions hold; he rejects if not

$$\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus \left(M_\mathbf{e} \boxtimes \mathbf{c}^\mathsf{T}\right), \quad \|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}, \quad \|\mathbf{t}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}.$$

- If diag is set to true the verifier also checks whether $\mathsf{decode}(\mathbf{z}_i)$ is a diagonal element, and rejects if it is not.

FIGURE 2.4. The ZKPoPK Protocol, interactive version.

Proof. For completeness: assume the prover is honest. For $i = 1, \ldots, V$ the verifier checks if $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ equals $a_i \boxplus M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}$, since $M_{\mathbf{e},i}$ is a scalar matrix we write multiplication with $\cdot$ as opposed to $\boxtimes$. The check passes because of the following relation:

$$\begin{aligned}
a_i \boxplus \left(M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}\right) &= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot c_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{x}_k, \quad \mathbf{s}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{r}_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}, \quad \mathbf{s}_i + M_{\mathbf{e},i} \cdot \mathbf{r}^\mathsf{T}\right) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i).
\end{aligned}$$

Moreover, given that $\mathbf{z}_i = \mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ and that all ciphertexts in $\mathbf{c}$ are (τ, ρ)-ciphertexts, we get that each single coordinate in $M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ is numerically at most $\mathsf{sec} \cdot \tau$. Each coordinate of $\mathbf{y}_i$ was chosen from an interval that is a factor $N \cdot \mathsf{sec} \cdot 2^{\nu\mathsf{sec}-1}$ larger. By a union bound over the $N \cdot \mathsf{sec}$ coordinates involved, each coordinate in $\mathbf{z}_i$ fails to be in the required range with probability exponentially small in sec. A similar argument shows that the check $\|\mathbf{t}_i\|_\infty$ also fails with negligible probability. Finally, each $\mathbf{y}_i$ was constructed to be congruent mod p to the encoding of a value in $\mathbb{F}_{p^k}^s$. Since this is also the case for the $\mathbf{x}_i$'s if the prover is honest, the same is true for the $\mathbf{z}_i$'s, and they therefore decode to values in $\mathbb{F}_{p^k}^s$. If diag is set to true, all $\mathbf{x}_i, \mathbf{y}_i$ contain diagonal plaintexts, and then $\mathbf{z}_i$ contains diagonal values as well.

For soundness: we consider a prover making a verifier accept both $(x, \mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and $(x, \mathbf{a}, \mathbf{e}', (\mathbf{z}', T'))$ with $\mathbf{e} \neq \mathbf{e}'$. Since both checks $\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$ and $\mathbf{d}'^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_{\mathbf{e}'} \cdot \mathbf{c}^\mathsf{T})$ passed, one can subtract the two equalities and obtain

$$(M_\mathbf{e} - M_{\mathbf{e}'}) \boxtimes \mathbf{c}^\mathsf{T} = \left(\mathbf{d} \boxminus \mathbf{d}'\right)^\mathsf{T} \tag{2.1}$$

In order to find $\mathbf{x}$ and R such that $c_k = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)$ for $k = 1, \ldots, \mathsf{sec}$, we first solve (2.1) as a linear system in $\mathbf{c}$. Let j be the highest index such that $\mathbf{e}_j \neq \mathbf{e}'_j$. The $\mathsf{sec} \times \mathsf{sec}$ submatrix of $M_\mathbf{e} - M_{\mathbf{e}'}$, consisting of the rows of $M_\mathbf{e} - M_{\mathbf{e}'}$ between j and $j + \mathsf{sec} - 1$ both included, is upper triangular with entries in $\{-1, 0, 1\}$ and its diagonal consists of the non-zero value $\mathbf{e}_j - \mathbf{e}'_j$ (so it is possible to find a solution for $\mathbf{c}$). Since the verifier has values $\mathbf{z}_i, \mathbf{t}_i, \mathbf{z}'_i, \mathbf{t}'_i$

such that $d_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ and $d'_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}'_i, \mathbf{t}'_i)$, and given that $c_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i)$, it is possible to directly solve the linear system in $\mathbf{x}$ and R (since the cryptosystem is additively homomorphic), from the bottom equation to the one "in the middle" with index $\mathsf{sec}/2$. Since $\|\mathbf{z}_i\|_\infty, \|\mathbf{z}'_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$ and $\|\mathbf{t}_i\|_\infty, \|\mathbf{t}'_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$, we conclude that $c_{\mathsf{sec}-i}$ is a $(s \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}+i}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}+i})$-ciphertext (by induction on i). To solve for $c_1, \ldots c_{\mathsf{sec}/2}$, we consider the *lowest* index j such that $\mathbf{e}_j \neq \mathbf{e}'_j$, construct an lower triangular matrix in a similar way as above, and solve from the first equation downwards. We conclude that $\mathbf{c}$ contains $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}})$-ciphertexts.

We note that since the verifier accepted, each $\mathbf{z}_i$ has small norm and decodes to a value in $(\mathbb{F}_{p^k})^s$. Since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, it follows from correctness of the cryptosystem that the $\mathbf{x}_i$ also decode to values in $(\mathbb{F}_{p^k})^s$. Finally, if diag was set to true, the verifier only accepts if all $\mathbf{z}_i$ decode to diagonal values. Again, since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, the $\mathbf{x}_i$ also decode to diagonal values.

For Zero-Knowledge: we give an honest-verifier simulator for the protocol that outputs accepting conversations. In order to simulate one repetition, the simulator samples $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ uniformly and $\mathbf{z}, T$ uniformly with the constraint that $\mathbf{d}$ contains random ciphertexts satisfying the verifier's check, i.e. $\mathbf{z}_i, \mathbf{t}_i$ are uniform, subject to $\|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}, \|\mathbf{t}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$, where moreover $\mathbf{z}_i$ is generated as $\mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$ where $\mathbf{m}_i$ is a random plaintext (diagonal if diag is set to true) and $\mathbf{u}_i$ contains multiples of p that are uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. Finally, $\mathbf{a}$ is computed as $\mathbf{a}^\mathsf{T} \leftarrow \mathbf{d}^\mathsf{T} \boxminus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$. In the real conversation, the prover's choice of values in $\mathbf{z}_i$ and $\mathbf{t}_i$ are statistically close to the distribution used by the simulator. This is because the prover uses the same method to generate these values, except that he adds in some vectors of exponentially smaller norm which leads to a statistically close distribution. Since $\mathbf{e}$ has the correct distribution and $\mathbf{a}$ follows deterministically from the last two messages, the simulation is statistically indistinguishable.

□

2.2.2. Zero-Knowledge Protocol – Fiat-Shamir Heuristic. We now give a protocol that leads to smaller values of the parameters and hence also allows better parameters for the underlying cryptosystem. This version, however, is better suited for use with the Fiat-Shamir heuristic. The idea is to let the prover choose his randomness in a smaller interval, and abort if the last message would reveal too much information. This is an idea from [Lyu09]. When using the Fiat-Shamir heuristic, this is not a problem as the prover only needs to show a successful attempt to the verifier. Let h be a suitable hash function that outputs sec-bit strings.

Protocol Π_{ZKPoPK} – Fiat-Shamir heuristic

- For $i = 1, \ldots, V$, the prover generates $\mathbf{y}_i \leftarrow \mathbb{Z}^N$ and $\mathbf{s}_i \leftarrow \mathbb{Z}^d$, such that $\|\mathbf{y}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$ and $\|\mathbf{s}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2$. For $\mathbf{y}_i$, this is done as follows: choose a random message $\mathbf{m}_i \in (\mathbb{F}_{p^k})^s$ and set $\mathbf{y}_i = \mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$, where each entry in $\mathbf{u}_i$ is a multiple of p, chosen uniformly at random, subject to $\|\mathbf{y}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$. If diag is set to true then the $\mathbf{m}_i$ are additionally chosen to be diagonal elements.
- The prover computes $a_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i)$, for $i = 1, \ldots, V$, and defines $S \in \mathbb{Z}^{V \times d}$ to be the matrix whose ith row is $\mathbf{s}_i$ and sets $\mathbf{y} \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_V)$, $\mathbf{a} \leftarrow (a_1, \ldots, a_V)$.
- The prover sends $\mathbf{a}$ to the verifier.
- The prover computes $\mathbf{e} = h(\mathbf{a}, \mathbf{c})$.
- The prover sets $\mathbf{z} \leftarrow (\mathbf{z}_1, \ldots, \mathbf{z}_V)$, such that $\mathbf{z}^\mathsf{T} = \mathbf{y}^\mathsf{T} + M_\mathbf{e} \cdot \mathbf{x}^\mathsf{T}$, and $T = S + M_e \cdot R$. Let $\mathbf{t}_i$ be the ith row of T. If for any i, it is the case that $\|\mathbf{z}_i\|_\infty > 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ or $\|\mathbf{t}_i\|_\infty > 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$, the prover aborts and the protocol is restarted. Otherwise the prover sends $(\mathbf{a}, \mathbf{z}, T)$ to the verifier.
- The verifier computes $\mathbf{e} = h(\mathbf{a}, \mathbf{c}), d_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$, for $i = 1, \ldots, V$, where $\mathbf{t}_i$ is the ith row of T and sets $\mathbf{d} \leftarrow (d_1, \ldots, d_V)$.
- The verifier checks $\mathsf{decode}(\mathbf{z}_i) \in \mathbb{F}_{p^k}^s$ and whether the following three conditions hold

$$\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus \left(M_\mathbf{e} \boxtimes \mathbf{c}^\mathsf{T}\right), \quad \|\mathbf{z}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2, \quad \|\mathbf{t}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2.$$

If diag is set to true the verifier also checks whether $\mathsf{decode}(\mathbf{z}_i)$ is a diagonal element, and rejects if it is not.

FIGURE 2.5. The ZKPoPK Protocol, version for Fiat-Shamir heuristic.

We claim that the Fiat-Shamir based protocol is a proof of knowledge for the relation in question in the random oracle model. In this case, however, we can guarantee that the adversarially generated ciphertexts are $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8})$-ciphertexts.

Proof. For completeness: assume the prover is honest. Note first that each $\mathbf{y}_i$ was constructed to be congruent mod p to the encoding of a value in $(\mathbb{F}_{p^k})^s$. Since this is also the case for the $\mathbf{x}_i$'s if the prover is honest, the same is true for the $\mathbf{z}_i$'s, and they therefore always decode to a value in $(\mathbb{F}_{p^k})^s$. If diag was set to true, all $\mathbf{x}_i, \mathbf{y}_i$ contain diagonal plaintexts, and then the same is true for the $\mathbf{z}_i$.

Next, for $i = 1, \ldots, V$ the verifier checks if $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ equals $a_i \boxplus M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}$, since $M_{\mathbf{e},i}$ is a scalar matrix we write multiplication with $\cdot$ as opposed to $\boxtimes$. The check passes because of the following relation:

$$\begin{aligned}
a_i \boxplus \left(M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}\right) &= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot c_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{x}_k, \quad \mathbf{s}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{r}_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}, \quad \mathbf{s}_i + M_{\mathbf{e},i} \cdot \mathbf{r}^\mathsf{T}\right) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i).
\end{aligned}$$

Moreover, given that $\mathbf{z}_i = \mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ and that all ciphertexts in $\mathbf{c}$ are (τ, ρ)-ciphertexts, we get that each single coordinate in $M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ is numerically at most $\mathsf{sec} \cdot \tau$. Each coordinate of $\mathbf{y}_i$ was chosen from an interval that is a factor $128 \cdot N \cdot \mathsf{sec}$ larger. Therefore each coordinate in

$\mathbf{z}_i$ fails to be in the required range with probability $1/(128 \cdot N \cdot \mathsf{sec})$. Note that this probability does not depend on the concrete values of the coordinates in $M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$, only on the bound on the numeric value.

By a union bound over the N coordinates of $\mathbf{z}_i$ we get that $\|\mathbf{z}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ fails with probability at most $1/(128 \cdot \mathsf{sec})$, and by a final union bound over the $2\,\mathsf{sec} - 1$ ciphertexts that all checks on the $\mathbf{z}_i$'s are OK except with probability at most $1/64$. A similar argument shows that the check $\|\mathbf{t}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$ fails also with probability at most $1/64$. The conclusion is that the prover aborts with probability at most $1/32$, so we expect to only have to repeat the protocol once to have success.

For soundness: by a standard argument, a prover who can efficiently produce a valid proof is able to produce $(x, \mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and $(x, \mathbf{a}, \mathbf{e}', (\mathbf{z}', T'))$ with $\mathbf{e} \neq \mathbf{e}'$ that the verifier would accept. Since both checks $\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$ and $\mathbf{d}'^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_{\mathbf{e}'} \cdot \mathbf{c}^\mathsf{T})$ passed, one can subtract the two equalities and obtain

$$(M_\mathbf{e} - M_{\mathbf{e}'}) \boxtimes \mathbf{c}^\mathsf{T} = \left(\mathbf{d} \boxminus \mathbf{d}'\right)^\mathsf{T} \tag{2.2}$$

In order to find $\mathbf{x}$ and R such that $c_k = \mathsf{Enc}_\mathsf{pk}(\mathbf{x}_k, \mathbf{r}_k)$ for $k = 1, \ldots, \mathsf{sec}$, we first solve (2.2) as a linear system in $\mathbf{c}$. Let j be the highest index such that $\mathbf{e}_j \neq \mathbf{e}'_j$. The $\mathsf{sec} \times \mathsf{sec}$ submatrix of $M_\mathbf{e} - M_{\mathbf{e}'}$, consisting of the rows of $M_\mathbf{e} - M_{\mathbf{e}'}$ between j and $j + \mathsf{sec} - 1$ both included, is upper triangular with entries in $\{-1, 0, 1\}$ and its diagonal consists of the non-zero value $\mathbf{e}_j - \mathbf{e}'_j$ (so it is possible to find a solution for $\mathbf{c}$). Since the verifier has values $\mathbf{z}_i, \mathbf{t}_i, \mathbf{z}'_i, \mathbf{t}'_i$ such that $d_i = \mathsf{Enc}_\mathsf{pk}(\mathbf{z}_i, \mathbf{t}_i)$ and $d'_i = \mathsf{Enc}_\mathsf{pk}(\mathbf{z}'_i, \mathbf{t}'_i)$, and given that $c_i = \mathsf{Enc}_\mathsf{pk}(\mathbf{x}_i, \mathbf{r}_i)$, it is possible to directly solve the linear system in $\mathbf{x}$ and R (since the cryptosystem is additively homomorphic), from the bottom equation to the one "in the middle" with index $\mathsf{sec}/2$.
Since $\|\mathbf{z}_i\|_\infty, \|\mathbf{z}'_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$ and $\|\mathbf{t}_i\|_\infty, \|\mathbf{t}'_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2$, we conclude that $c_{\mathsf{sec}-i}$ must be a $(256 \cdot N \cdot \tau \cdot 2^i \cdot \mathsf{sec}^2, 256 \cdot d \cdot \rho \cdot 2^i \cdot \mathsf{sec}^2)$-ciphertext (by induction on i). To solve for $c_1, \ldots c_{\mathsf{sec}/2}$, we consider the *lowest* index j such that $\mathbf{e}_j \neq \mathbf{e}'_j$, construct an lower triangular matrix in a similar way as above, and solve from the first equation downwards. We conclude that $\mathbf{c}$ contains $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{sec/2+8}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{sec/2+8})$-ciphertexts.

We note that since the verifier accepted, each $\mathbf{z}_i$ has small norm and decodes to a value in $(\mathbb{F}_{p^k})^s$. Since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, it follows from correctness of the cryptosystem that the $\mathbf{x}_i$ also decode to values $(\mathbb{F}_{p^k})^s$. Finally, if diag was set to true, the verifier only accepts if all $\mathbf{z}_i$ decode to diagonal values. Again, since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, the $\mathbf{x}_i$ also decode to diagonal values.

For Zero-Knowledge: we give an honest-verifier simulator for the protocol that outputs an accepting conversation (that does not abort).

In order to simulate one repetition, the simulator samples $\mathbf{e} \in \{0,1\}^\mathsf{sec}$ uniformly and $\mathbf{z}, T$ uniformly with the constrain that $\mathbf{d}$ contains random $(8 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}, 8 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec})$-ciphertexts. where moreover $\mathbf{z}_i$ is generated as $\mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$ where $\mathbf{m}_i$ is a random plaintext (a diagonal one if diag is set to true) and $\mathbf{u}_i$ contains multiples of p that are uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq 8N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$. Finally, $\mathbf{a}$ is computed as $\mathbf{a}^\mathsf{T} \leftarrow \mathbf{d}^\mathsf{T} \boxminus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$. Define the random oracle to output $\mathbf{e}$ on input $\mathbf{a}, \mathbf{c}$, output $(\mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and stop.

We argue that this simulation is perfect: the distribution of a simulated $\mathbf{e}$ is the same as a real one. Also, it is straightforward to see that in a real conversation, given that the prover does not abort, the vectors $\mathbf{z}_i, \mathbf{t}_i$ are uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq 8 \cdot s \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ and $\|\mathbf{t}_i\|_\infty \leq 8 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$. So the simulator chooses $\mathbf{z}_i, \mathbf{t}_i$ with exactly the right distribution. Since the value of $\mathbf{a}$ follows deterministically from the $\mathbf{e}, \mathbf{z}_i, \mathbf{t}_i$, we have what we wanted.

□

Doing without random oracles. The above protocol can also be executed without using the Fiat-Shamir heuristic. In this case, the prover starts $\mathsf{sec}/5$ instances of the protocol, computing $\mathbf{a}_1, \ldots, \mathbf{a}_{\mathsf{sec}/5}$. We choose this number of instances because it ensures that the prover fails on all of them with probability only $(1/32)^{\mathsf{sec}/5} = 2^{-\mathsf{sec}}$. The prover commits to all these values, which can be done, for instance, with a Merkle hash tree, in which case the commitment is very short,

and any of the **a**'s can be opened by sending a piece of information that is only logarithmic in sec.

The verifier selects **e**, the prover finds an instance where he would not abort the protocol with this **e**, opens the corresponding **a** and completes that instance.

This is complete and zero-knowledge by the same argument as above plus the hiding property of the commitment scheme used. Soundness follows from the fact that if the prover succeeds with probability significantly greater that $2^{-\mathsf{sec}} \cdot \mathsf{sec}/5$ he must be able to answer different challenges correctly for some fixed instance out of the $\mathsf{sec}/5$ we have. Such answers can be extracted by rewinding, and then the rest of the argument is the same as above.

2.3. The Preprocessing Phase

2.3.1. A Small Description of the Online Phase. In the online phase each shared value $a \in \mathbb{F}_{p^k}$ is represented as follows

$$\langle a \rangle := (a_1, \ldots, a_n, \gamma(a)_1, \ldots, \gamma(a)_n)$$

where $a = a_1 + \cdots + a_n$ and $\gamma(a)_1 + \cdots + \gamma(a)_n = \alpha \cdot a$. Player P_i holds $a_i, \gamma(a)_i$, and the interpretation is that $\gamma(a) = \gamma(a)_1 + \cdots + \gamma(a)_n$ is the MAC authenticating a under the global key α.

Using the natural component-wise addition of representations, and suppressing the underlying choices of $a_i, \gamma(a)_i$ for readability, we clearly have for secret values a, b and public constant e that

$$\langle a \rangle + \langle b \rangle = \langle a + b \rangle \quad \text{and} \quad e \cdot \langle a \rangle = \langle ea \rangle.$$

To check the MACs players need the global key α, which is given from the preprocessing in the following representation:

$$[\![\alpha]\!] := ((\alpha_1, \ldots, \alpha_n), (\beta_i, \gamma(\alpha)_1^i, \ldots, \gamma(\alpha)_n^i)_{i=1,\ldots,n})),$$

where $\alpha = \sum_i \alpha_i$ and $\sum_j \gamma(\alpha)_i^j = \alpha\beta_i$. Player P_i holds $\alpha_i, \beta_i, \gamma(\alpha)_1^i, \ldots, \gamma(\alpha)_n^i$. The idea is that $\gamma(\alpha)_i \leftarrow \sum_j \gamma(\alpha)_i^j$ is the MAC authenticating α under P_i's private key β_i. To open $[\![\alpha]\!]$ each P_j sends to each P_i his share α_j of α and his share $\gamma(\alpha)_i^j$ of the MAC on α made with P_i's private key and then P_i checks that $\sum_j \gamma(\alpha)_i^j = \alpha\beta_i$. (To open the value to only one party P_i, the other parties simply send their shares only to P_i, who will do the checking. Only shares of α and $\alpha\beta_i$ are needed.)

Notice that this representation of the global key allows the players to add or subtract public values to already shared values by first creating an angle representation of the public value as follows: for a public constant e, each party P_i computes $\gamma(e)_i \leftarrow e \cdot \alpha_i$ and therefore players obtain $\langle e \rangle$.[2]

Multiplications are not straightforward: here we use the preprocessing. We would like the preprocessing to output triples $\langle a \rangle, \langle b \rangle, \langle c \rangle$, for uniform a, b and where $c = ab$. However, the SPDZ preprocessing produces triples which satisfy $c = ab + \Delta$, where Δ is an error that can be introduced by the adversary. Players therefore need to check the validity of each triple before using it. The check can be done by "sacrificing" another triple $\langle f \rangle, \langle g \rangle, \langle h \rangle$, where the same multiplicative equality should hold. Given such a valid triple, multiplications can be performed as follows: to compute $\langle xy \rangle$ players first open $\langle x \rangle - \langle a \rangle$ to get ε, and $\langle y \rangle - \langle b \rangle$ to get δ. Then $xy = (a + \varepsilon)(b + \delta) = c + \varepsilon b + \delta a + \varepsilon\delta$. Thus, the new representation can be computed as

$$\langle x \rangle \cdot \langle y \rangle = \langle c \rangle + \varepsilon\langle b \rangle + \delta\langle a \rangle + \varepsilon\delta.$$

An important note is that during the online protocol there is no guarantee that players are working with the correct results, since players do not immediately check the MACs of the opened values. During the first part of the protocol, parties only do what we define as a *partial opening*,

[2]The reader familiar with SPDZ will notice that the $\langle \cdot \rangle$ representation that we presented differed from the one in [DPSZ12], in that we dropped the public constant δ. The observation on how to compute MACs on public values makes the usage of δ superfluous.

meaning that for a value $\langle a \rangle$, each party P_i sends a_i to P_1, who computes $a = a_1 + \cdots + a_n$ and broadcasts a to all players. We assume that this is done via P_1, for simplicity, whereas in practice, one would balance the workload over the players.

Finally, the preprocessing also outputs n pairs of a random value r in both of the presented representations $\langle r \rangle, [\![r]\!]$. These pairs are used in the Input phase of the protocol.

2.3.2. The Preprocessing Protocol. In this section we construct the protocol Π_{PREP} which securely implements the functionality $\mathcal{F}_{\text{PREP}}$ (specified in Figure 2.6) in the presence of functionalities $\mathcal{F}_{\text{KEYGENDEC}}$ (Figure 2.2) and $\mathcal{F}_{\text{RAND}}$ (Figure 2.3).

The preprocessing uses the above abstract cryptosystem with $M = (\mathbb{F}_{p^k})^s$, but the online phase is designed for messages in $\mathbb{F}_{p^k}$. Therefore, we extend the notation $\langle \cdot \rangle$ and $[\![\cdot]\!]$ to messages in M: since addition and multiplication on M are component-wise, for $\mathbf{m} = (m_1, \ldots, m_s)$, we define $\langle \mathbf{m} \rangle = (\langle m_1 \rangle, \ldots, \langle m_s \rangle)$ and similarly $[\![\mathbf{m}]\!]$. Conversely, once a representation (or a pair, triple) on vectors is produced in the preprocessing, it is disassembled into its coordinates, so that it can be used in the online phase. In Figures 2.7,2.8 and 2.9, we introduce subprotocols that are accessed by the main preprocessing protocol in several steps. Note that the subprotocols are not meant to implement ideal functionalities: their purpose is merely to summarise parts of the main protocol that are repeated in various occasions.

THEOREM 2.3. *The protocol Π_{PREP} (Figure 2.10) implements $\mathcal{F}_{\text{PREP}}$ with computational security against any static, active adversary corrupting up to $n-1$ parties, in the $\mathcal{F}_{\text{KEYGEN}}, \mathcal{F}_{\text{RAND}}$-hybrid model when the underlying cryptosystem is admissible*[3].

Proof.

Recall first that we assume the cryptosystem has an alternative key generation algorithm $\mathsf{KeyGen}^*()$ which is a randomised algorithm that outputs a *meaningless public key* $\widetilde{\mathsf{pk}}$ with the property that an encryption of any message $\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(\mathbf{x})$ is statistically indistinguishable from an encryption of 0. Furthermore, if we set $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ and $\widetilde{\mathsf{pk}} \leftarrow \mathsf{KeyGen}^*()$, then pk and $\widetilde{\mathsf{pk}}$ are computationally indistinguishable.

We construct a simulator $\mathcal{S}_{\text{PREP}}$ for Π_{PREP}. In a nutshell, the simulator runs a copy of the protocol. Here, it plays the honest players' part while the environment $\mathcal{Z}$ plays for the corrupt players. The simulator also internally runs copies of $\mathcal{F}_{\text{KEYGEN}}$ and $\mathcal{F}_{\text{RAND}}$, in order to simulate calls to these functionalities. Note that in the following we say that the simulator executes or performs some part of the protocol as shorthand for the simulator going through that part with $\mathcal{Z}$. During the protocol execution, whenever $\mathcal{Z}$ sends ciphertexts on behalf of corrupt players, the simulator can obtain the plaintexts, since it knows the secret key. These values are then used to generate input to $\mathcal{F}_{\text{PREP}}$. A precise description is provided in Figure 2.11.

We now need to show that no $\mathcal{Z}$ can distinguish between the simulated and the real process. For the sake of contradiction, we assume that there exists $\mathcal{Z}$ that can distinguish these two cases with significant advantage ε. The output of $\mathcal{Z}$ is a single bit, thought of as a guess at one of the two cases. Concretely, we assume

$$\begin{aligned}\mathsf{A}(\mathcal{Z}) &:= \Pr\left[\text{“Real”} \leftarrow \mathcal{Z}(\text{Real process})\right] - \Pr\left[\text{“Real”} \leftarrow \mathcal{Z}(\text{Simulated process})\right] \\ &\geq \varepsilon.\end{aligned}$$

We show that such a $\mathcal{Z}$ can be used to distinguish between a normally generated public key and a meaningless one with basically the same advantage. This leads to a contradiction, since a key generated by the normal key generator is computationally indistinguishable from a meaningless one.

More in detail, we construct an algorithm B that takes as input a public key pk^* (randomly chosen as either a normal public key or a meaningless one), sets up a copy of $\mathcal{Z}$, goes through the protocol with $\mathcal{Z}$ and uses its output to guess the type of key it got as input. During the process B

[3]The definition of admissible cryptosystem demands a decryption protocol that implements $\mathcal{F}_{\text{KEYGENDEC}}$ based on $\mathcal{F}_{\text{KEYGEN}}$, hence the theorem only assumes $\mathcal{F}_{\text{KEYGEN}}$.

Functionality $\mathcal{F}_{\text{PREP}}$

Usage: We first describe two macros, one to produce $[\![\mathbf{v}]\!]$ representations and one to produce $\langle\mathbf{v}\rangle$ representations. We denote by A the set of players controlled by the adversary.

$\mathsf{Bracket}(\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$**:**

where $\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$, $\beta_1,\ldots,\beta_n \in \mathbb{F}_{p^k}$

(1) Let $\mathbf{v} = \sum_{i=1}^n \mathbf{v}_i$.
(2) For $i = 1,\ldots,n$
  (a) The functionality computes the MAC $\gamma(\mathbf{v})_i \leftarrow \mathbf{v}\cdot\beta_i$ and sets $\gamma_i \leftarrow \gamma(\mathbf{v})_i + \Delta_i$.
  (b) For every corrupt player P_j, $j \in A$ the environment specifies a share γ_i^j.
  (c) The functionality sets each share γ_i^j, $j \notin A$, uniformly such that $\sum_{j=1}^n \gamma_i^j = \gamma_i$.
(3) The functionality sends $(\mathbf{v}_i, (\beta_i, \gamma_1^i, \ldots, \gamma_n^i))$ to each honest player P_i (dishonest players already have the respective data).

$\mathsf{Angle}(\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta,\alpha)$**:**

where $\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta \in (\mathbb{F}_{p^k})^s$, $\alpha \in \mathbb{F}_{p^k}$

(1) Let $\mathbf{v} = \sum_{i=1}^n \mathbf{v}_i$.
(2) The functionality computes the MAC $\gamma(\mathbf{v}) \leftarrow \alpha\cdot\mathbf{v}$ and sets $\gamma \leftarrow \gamma(\mathbf{v}) + \Delta$.
(3) For every corrupt player P_i, $i \in A$ the environment specifies a share γ_i.
(4) The functionality sets each share γ_i $i \notin A$ uniformly such that $\sum_{i=1}^n \gamma_i = \gamma$.
(5) The functionality sends $(\mathbf{v}_i, \gamma_i)$ to each honest player P_i (dishonest players already have the respective data).

Initialise: On input $(init, p, k, s)$ from all players, the functionality stores the prime p and the integers k, s. It then waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

(1) For each corrupt player P_i, $i \in A$, the environment specifies a share α_i.
(2) The functionality sets each share α_i, $i \notin A$ uniformly.
(3) For each corrupt player P_i, $i \in A$, the environment specifies a key β_i.
(4) The functionality sets each key β_i $i \notin A$ uniformly.
(5) The environment specifies $\Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$.
(6) The functionality runs $\mathsf{Bracket}(\mathsf{Diag}(\alpha_1),\ldots,\mathsf{Diag}(\alpha_n),\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$.

Pair: On input $(pair)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

(1) For each corrupt player P_i, $i \in A$, the environment specifies a share $\mathbf{r}_i$.
(2) The functionality sets each share $\mathbf{r}_i$, $i \notin A$ uniformly.
(3) The environment specifies $\Delta, \Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$.
(4) The functionality runs:
  $\mathsf{Bracket}(\mathbf{r}_1,\ldots,\mathbf{r}_n,\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$, $\mathsf{Angle}(\mathbf{r}_1,\ldots,\mathbf{r}_n,\Delta,\alpha)$.

Triple: On input $(triple)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

(1) For each corrupt player P_i, $i \in A$, the environment specifies shares $\mathbf{a}_i, \mathbf{b}_i$.
(2) The functionality sets each share $\mathbf{a}_i, \mathbf{b}_i$, $i \notin A$ uniformly.
  Let $\mathbf{a} := \sum_{i=1}^n \mathbf{a}_i$, $\mathbf{b} := \sum_{i=1}^n \mathbf{b}_i$.
(3) The environment specifies $\Delta_{\mathbf{a}}, \Delta_{\mathbf{b}}, \delta \in (\mathbb{F}_{p^k})^s$.
(4) It sets $\mathbf{c} \leftarrow \mathbf{a}\cdot\mathbf{b} + \delta$.
(5) For each corrupt player P_i, $i \in A$, the environment specifies shares $\mathbf{c}_i$.
(6) The functionality sets each share $\mathbf{c}_i$, $i \notin A$ uniformly with the constrain $\sum_{i=1}^n \mathbf{c}_i = \mathbf{c}$.
(7) The environment specifies $\Delta_{\mathbf{c}} \in (\mathbb{F}_{p^k})^s$.
(8) The functionality runs:
  $\mathsf{Angle}(\mathbf{a}_1,\ldots,\mathbf{a}_n,\Delta_{\mathbf{a}},\alpha)$, $\mathsf{Angle}(\mathbf{b}_1,\ldots,\mathbf{b}_n,\Delta_{\mathbf{b}},\alpha)$, $\mathsf{Angle}(\mathbf{c}_1,\ldots,\mathbf{c}_n,\Delta_{\mathbf{c}},\alpha)$.

FIGURE 2.6. The functionality generating the key $[\![\alpha]\!]$, pairs $[\![\mathbf{r}]\!], \langle\mathbf{r}\rangle$ and triples $\langle\mathbf{a}\rangle, \langle\mathbf{b}\rangle, \langle\mathbf{c}\rangle$.

uniformly chooses a bit (that can be thought as a switch between "Real" and "Simulation"): in

Protocol $\mathsf{Reshare}$

Usage: The input consists of $e_{\mathbf{m}}$, where $\mathsf{Dec}_{\mathsf{sk}}(e_{\mathbf{m}}) = \mathbf{m}$ is a public ciphertext and a parameter enc, where $enc = \mathsf{NewCiphertext}$ or $enc = \mathsf{NoNewCiphertext}$. The output is a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; and if $enc = \mathsf{NewCiphertext}$, a ciphertext $e'_{\mathbf{m}}$. The idea is that $e_{\mathbf{m}}$ could be a product of two ciphertexts, which $\mathsf{Reshare}$ converts to a "fresh" ciphertext $e'_{\mathbf{m}}$. Since $\mathsf{Reshare}$ uses distributed decryption (that may return an incorrect result), it is not guaranteed that $e_{\mathbf{m}}$ and $e'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $e'_{\mathbf{m}}$.

$\mathsf{Reshare}(e_{\mathbf{m}}, enc)$**:**

(1) P_i samples a uniform $\mathbf{f}_i \in (\mathbb{F}_{p^k})^s$. Define $\mathbf{f} := \sum_{i=1}^n \mathbf{f}_i$.
(2) P_i computes and broadcasts $e_{\mathbf{f}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_i)$.
(3) P_i runs Π_{ZKPoPK} acting as a prover on $e_{\mathbf{f}_i}$. The protocol aborts if any proof fails.
(4) The players compute $e_{\mathbf{f}} \leftarrow e_{\mathbf{f}_1} \boxplus \cdots \boxplus e_{\mathbf{f}_n}$, and $e_{\mathbf{m}+\mathbf{f}} \leftarrow e_{\mathbf{m}} \boxplus e_{\mathbf{f}}$.
(5) The players invoke $\mathcal{F}_{\text{KeyGenDec}}$ to decrypt $e_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m} + \mathbf{f}$.
(6) P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i $(i \neq 1)$ sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
(7) If $enc = \mathsf{NewCiphertext}$, all players set $e'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f}) \boxminus e_{\mathbf{f}_1} \boxminus \cdots \boxminus e_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f})$.

FIGURE 2.7. The protocol to share $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ on input $e_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$.

Protocol $\mathsf{PBracket}$

Usage: On input shares $\mathbf{v}_1, \ldots, \mathbf{v}_n$ privately held by the players and public ciphertext $e_{\mathbf{v}}$, this protocol generates $[\![\mathbf{v}]\!]$. It is assumed that $\sum_i \mathbf{v}_i$ is the plaintext contained in $e_{\mathbf{v}}$.

$\mathsf{PBracket}(\mathbf{v}_1, \ldots, \mathbf{v}_n, e_{\mathbf{v}})$**:**

(1) For $i = 1, \ldots, n$
  (a) All players set $e_{\gamma_i} \leftarrow e_{\beta_i} \boxtimes e_{\mathbf{v}}$ (note that e_{β_i} is generated during the initialisation process, and known by every player).
  (b) Players generate $(\gamma_i^1, \ldots \gamma_i^n) \leftarrow \mathsf{Reshare}(e_{\gamma_i}, \mathsf{NoNewCiphertext})$, so each player P_j gets a share γ_i^j of $\mathbf{v} \cdot \beta_i$.
(2) Output the representation $[\![\mathbf{v}]\!] = (\mathbf{v}_1, \ldots, \mathbf{v}_n, (\beta_i, \gamma_1^i, \ldots, \gamma_n^i)_{i=1,\ldots,n})$.

FIGURE 2.8. The sub-protocol for generating $[\![\mathbf{v}]\!]$.

Protocol PAngle

Usage: On input shares $\mathbf{v}_1, \ldots, \mathbf{v}_n$ privately held by the players and public ciphertext $e_{\mathbf{v}}$, this protocol generates $\langle \mathbf{v} \rangle$. It is assumed that $\sum_i \mathbf{v}_i$ is the plaintext contained in $e_{\mathbf{v}}$.

$\mathsf{PAngle}(\mathbf{v}_1, \ldots, \mathbf{v}_n, e_{\mathbf{v}})$**:**

(1) All players set $e_{\mathbf{v}\cdot\alpha} \leftarrow e_{\mathbf{v}} \boxtimes e_{\alpha}$ (note that e_{α} is generated during the initialisation process, and known by every player).
(2) Players generate $(\gamma_1, \ldots, \gamma_n) \leftarrow \mathsf{Reshare}(e_{\mathbf{v}\cdot\alpha}, \mathsf{NoNewCiphertext})$, so each player P_i gets a share γ_i of $\alpha \cdot \mathbf{v}$.
(3) Output representation $\langle \mathbf{v} \rangle = (\mathbf{v}_1, \ldots, \mathbf{v}_n, \gamma_1, \ldots, \gamma_n)$.

FIGURE 2.9. The sub-protocol for generating $\langle \mathbf{v} \rangle$.

case pk^* is correctly computed, if the bit is set to "Real", $\mathcal{Z}$'s view is indistinguishable from a real execution of the protocol, while if the bit is set to "Simulation", $\mathcal{Z}$'s view is indistinguishable from a simulated run. However, in case pk^* is meaningless, both choices of the bit lead to statistically indistinguishable views. Hence, if $\mathcal{Z}$ guesses correctly whether B chose "Real" or "Simulation", B guesses that pk^* was a standard public key; otherwise B guesses that pk^* was meaningless.

For simplicity we describe the algorithm B for the two-party setting, where there is a corrupt party P_1 and an honest party P_2: on input pk^*, where pk^* is a public key (either meaningless

Protocol Π_{PREP}

Usage: The Triple-step is always executed sec times in parallel. This ensures that when calling Π_{ZKPoPK}, players can always give it the sec ciphertexts it requires as input. In addition both Π_{ZKPoPK} and Π_{PREP} can be executed in a SIMD fashion, i.e. they are data-oblivious bar when they detect an error. Thus players can execute Π_{ZKPoPK} and Π_{PREP} on the packed plaintext space $(\mathbb{F}_{p^k})^s$. Thereby, players generate $s \cdot \mathsf{sec}$ elements in one go and then buffer the generated triples, outputting the next unused one on demand.

Initialise: This step generates the global key α and "personal keys" β_i.

(1) The players call "start" on $\mathcal{F}_{\text{KeyGenDec}}$ to obtain the public key pk.
(2) P_i generates a MAC-key $\beta_i \in \mathbb{F}_{p^k}$.
(3) P_i generates $\alpha_i \in \mathbb{F}_{p^k}$. Let $\alpha := \sum_{i=1}^{n} \alpha_i$.
(4) P_i computes and broadcasts $e_{\alpha_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{Diag}(\alpha_i))$, $e_{\beta_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{Diag}(\beta_i))$.
(5) P_i invokes Π_{ZKPoPK} (with diag set to true) acting as prover on input $(e_{\alpha_i}, \ldots, e_{\alpha_i})$ and on input $(e_{\beta_i}, \ldots, e_{\beta_i})$, where $e_{\alpha_i}, e_{\beta_i}$ are repeated sec times, which is the number of ciphertexts Π_{ZKPoPK} requires as input. (This is not very efficient, but only needs to be done once for each player.)
(6) All players compute $e_\alpha \leftarrow e_{\alpha_1} \boxplus \cdots \boxplus e_{\alpha_n}$, and generate $[\![\mathsf{Diag}(\alpha)]\!] \leftarrow \mathsf{PBracket}(\mathsf{Diag}(\alpha_1), \ldots, \mathsf{Diag}(\alpha_n), e_\alpha)$.

Pair: This step generates a pair $[\![\mathbf{r}]\!], \langle \mathbf{r} \rangle$, and can be used to generate a single value $[\![\mathbf{r}]\!]$, by not performing the call to Pangle.

(1) P_i generates $\mathbf{r}_i \in (\mathbb{F}_{p^k})^s$. Let $\mathbf{r} := \sum_{i=1}^{n} \mathbf{r}_i$.
(2) P_i computes and broadcasts $e_{\mathbf{r}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{r}_i)$. Let $e_{\mathbf{r}} = e_{\mathbf{r}_1} \boxplus \cdots \boxplus e_{\mathbf{r}_n}$.
(3) P_i invokes Π_{ZKPoPK} acting as prover on the ciphertext he generated.
(4) Players generate $[\![\mathbf{r}]\!] \leftarrow \mathsf{PBracket}(\mathbf{r}_1, \ldots, \mathbf{r}_n, e_{\mathbf{r}})$, $\langle \mathbf{r} \rangle \leftarrow \mathsf{PAngle}(\mathbf{r}_1, \ldots, \mathbf{r}_n, e_{\mathbf{r}})$.

Triple: This step generates a multiplicative triple $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$.

(1) P_i generates $\mathbf{a}_i, \mathbf{b}_i \in (\mathbb{F}_{p^k})^s$. Let $\mathbf{a} := \sum_{i=1}^{n} \mathbf{a}_i$, $\mathbf{b} := \sum_{i=1}^{n} \mathbf{b}_i$.
(2) P_i computes and broadcasts $e_{\mathbf{a}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{a}_i)$, $e_{\mathbf{b}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{b}_i)$.
(3) P_i invokes Π_{ZKPoPK} acting as prover on the ciphertexts he generated.
(4) The players set $e_{\mathbf{a}} \leftarrow e_{\mathbf{a}_1} \boxplus \cdots \boxplus e_{\mathbf{a}_n}$ and $e_{\mathbf{b}} \leftarrow e_{\mathbf{b}_1} \boxplus \cdots \boxplus e_{\mathbf{b}_n}$.
(5) Players generate $\langle \mathbf{a} \rangle \leftarrow \mathsf{PAngle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, e_{\mathbf{a}})$, $\langle \mathbf{b} \rangle \leftarrow \mathsf{PAngle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, e_{\mathbf{b}})$.
(6) All players compute $e_{\mathbf{c}} \leftarrow e_{\mathbf{a}} \boxtimes e_{\mathbf{b}}$.
(7) Players set $(\mathbf{c}_1, \ldots, \mathbf{c}_n, e'_{\mathbf{c}}) \leftarrow \mathsf{Reshare}(e_{\mathbf{c}}, \mathsf{NewCiphertext})$.
(8) Players generate $\langle \mathbf{c} \rangle \leftarrow \mathsf{PAngle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, e'_{\mathbf{c}})$.

FIGURE 2.10. The protocol generating the key $[\![\alpha]\!]$, pairs $[\![\mathbf{r}]\!], \langle \mathbf{r} \rangle$ and triples $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$.

or standard), B starts executing the protocol Π_{PREP}, playing for P_2, while $\mathcal{Z}$ plays for P_1. B does exactly what the simulator would do, with some exceptions:

(1) It uses the public key it got as input, instead of generating a key pair initially.
(2) B cannot decrypt ciphertexts from P_1 since it does not know the secret key (e.g. at step 4 of Initialise, step 2 of Pair, step 2 of Triple, etc.). Instead, B exploits that P_1 and P_2 ran the protocol Π_{ZKPoPK} with P_1 as prover. That is, P_1 proved that he knows encodings of appropriate size corresponding to the plaintext inside the ciphertexts broadcast in the previous step. This means B can use the knowledge extractor of the protocol Π_{ZKPoPK} followed by decoding to extract the shares from P_1 (e.g. α_i, β_i at step 4 of Initialise, etc). At this point B continues the protocol as if it had decrypted. Note that the knowledge extractor requires rewinding of the prover (which here effectively is $\mathcal{Z}$). B can do this as it runs its own copy of $\mathcal{Z}$ and since it also controls the copy of $\mathcal{F}_{\text{Rand}}$ used in the protocol, it can issue challenges of its choice to $\mathcal{Z}$.
(3) When P_2 gives a ZK proof for a set of ciphertexts, B simulates the proof. This is done by running the honest verifier simulator to get a transcript $(\mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and letting the copy of $\mathcal{F}_{\text{Rand}}$ output $\mathbf{e}$ that occurs in the simulate transcript.

In the end B uniformly chooses to generate a real or a simulated view. In the first case, B outputs to $\mathcal{Z}$ exactly those values for P_2 that were used in the execution of the protocol.

Simulator $\mathcal{S}_{\text{PREP}}$

$\mathsf{SReshare}(e_{\mathbf{m}})$**:** This is a subroutine the simulator uses while executing the main steps of the protocol described below. At any point in Π_{PREP} when there is a call to $\mathsf{Reshare}(e_{\mathbf{m}})$, the simulator proceeds as the protocol does, but it also performs the following extra tasks in order to retrieve the quantity $\Delta_{\mathbf{m}}$:

- On step 2 the simulator decrypts $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_n)$ and obtains the values $\mathbf{f}_1, \ldots, \mathbf{f}_n$.
- On step 5 the simulator performs step 2 of $\mathcal{F}_{\text{KeyGenDec}}$, and thereby obtains $\mathbf{m}+\mathbf{f}$ decrypting $e_{\mathbf{m}+\mathbf{f}}$, and $(\mathbf{m}+\mathbf{f})'$ from the adversary.
- The simulator sets $\Delta_{\mathbf{m}} \leftarrow (\mathbf{m}+\mathbf{f})' - (\mathbf{m}+\mathbf{f})$, that is, $\Delta_{\mathbf{m}}$ is the difference between the output chosen by the adversary for the decryption of $e_{\mathbf{m}+\mathbf{f}}$ and the decryption itself.
- The simulator computes and stores $\mathbf{m}_1 \leftarrow (\mathbf{m}+\mathbf{f})' - \mathbf{f}_1$, and $\mathbf{m}_i \leftarrow -\mathbf{f}_i$ for $i \neq 1$.

Initialise:

- The simulator performs the initialisation steps of Π_{PREP}. The call to $\mathcal{F}_{\text{KeyGenDec}}$ in step 1 is simulated by running KeyGen to generate the key pair $(\mathsf{pk}, \mathsf{sk})$. The simulator then sends pk to the players and stores sk.
- Steps 2–5 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$.
- Step 6 is performed according to the protocol, but the simulator gets $\Delta_1 \leftarrow \mathsf{SReshare}(e_{\gamma(\alpha\cdot\beta_1)}), \ldots, \Delta_n \leftarrow \mathsf{SReshare}(e_{\alpha\cdot\beta_n})$.
- The simulator calls Initialise on $\mathcal{F}_{\text{PREP}}$ with input $\{\alpha_i\}_{i\in A}$ at step 1, $\{\beta_i\}_{i\in A}$ at step 3 and $\Delta_1, \ldots, \Delta_n$ at step 5.

Pair:

- The simulator performs step 1 according to the protocol.
- Steps 2–3 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\mathbf{r}_1, \ldots, \mathbf{r}_n$.
- Step 4 is performed according to the protocol, but the simulator gets $\Delta \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\alpha})$, $\Delta_1 \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\beta_1}), \ldots, \Delta_n \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\beta_n})$.
- The simulator calls Pair on $\mathcal{F}_{\text{PREP}}$ with input $\{\mathbf{r}_i\}_{i\in A}$ at step 1, and $\Delta, \Delta_1, \ldots, \Delta_n$ at step 3.

Triple:

- The simulator performs step 1 according to the protocol.
- Steps 2–3 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\mathbf{a}_1, \ldots, \mathbf{a}_n$, $\mathbf{b}_1, \ldots, \mathbf{b}_n$.
- Steps 4–5 are performed according to the protocol, but the simulator gets $\Delta_{\mathbf{a}} \leftarrow \mathsf{SReshare}(e_{\mathbf{a}\cdot\alpha})$, $\Delta_{\mathbf{b}} \leftarrow \mathsf{SReshare}(e_{\mathbf{b}\cdot\alpha})$.
- Steps 6–7 are performed according to the protocol, but the simulator gets $\mathbf{c}_1, \ldots \mathbf{c}_n$ and $\delta \leftarrow \mathsf{SReshare}(e_{\mathbf{c}})$.
- Step 8 is performed according to the protocol, but the simulator gets $\Delta_{\mathbf{c}} \leftarrow \mathsf{SReshare}(e_{\mathbf{c}\cdot\alpha})$.
- The simulator calls Triple on $\mathcal{F}_{\text{PREP}}$ with input $\{\mathbf{a}_i\}_{i\in A}$, $\{\mathbf{b}_i\}_{i\in A}$ at step 1, $\Delta_{\mathbf{a}}, \Delta_{\mathbf{b}}, \delta$ at step 3, $\{\mathbf{c}_i\}_{i\in A}$ in step 5, and $\Delta_{\mathbf{c}}$ at step 7.

FIGURE 2.11. The simulator for $\mathcal{F}_{\text{PREP}}$.

In the other case, B generates the output for P_2 as $\mathcal{F}_{\text{PREP}}$ would do. That means that P_2's shares $\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2$ of a triple $\langle\mathbf{a}\rangle, \langle\mathbf{b}\rangle, \langle\mathbf{c}\rangle$ will be determined by choosing $\mathbf{a}, \mathbf{b}$ at random, setting $\mathbf{c} \leftarrow \mathbf{a}\cdot\mathbf{b}$ and then letting $\mathbf{a}_2 \leftarrow \mathbf{a} - \mathbf{a}_1^{Real}$, $\mathbf{b}_2 \leftarrow \mathbf{b} - \mathbf{b}_1^{Real}$, $\mathbf{c}_2 \leftarrow \mathbf{c} - \mathbf{c}_1^{Real}$.

It can now be seen that if pk^* is a normal key, then the view generated by B corresponds statistically to either a real or a simulated execution: if B chooses the simulation case, the only differences to the actual simulator are 1) the simulator executes the ZK proofs given by P_2 according to the protocol while B simulates them; and 2) the simulator opens the ciphertexts using the secret key to decrypt, while B uses the extractor for Π_{ZKPoPK} and computes the plaintexts from its results. As for 1) the ZK proof is statistical ZK so this leads to a statistically

indistinguishable distribution. As for 2), note that for every ciphertext $e_{\mathbf{x}}$ generated by P_1, the extractor for Π_{ZKPoPK} is able (with overwhelming probability) to find an encoding $\mathbf{x}$ (resp. randomness r) smaller than B_{plain} (resp. B_{rand}), with $e_{\mathbf{x}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$. This follows from soundness of Π_{ZKPoPK} and admissibility of the cryptosystem. Then, by correctness of the cryptosystem, computing the plaintexts as B does gives the same result as decrypting, except with negligible probability. If B chooses the real case, a similar argument shows that the resulting view is statistically indistinguishable from a real run of the protocol. Hence if pk^* is a normal key, $\mathcal{Z}$ can guess B's choice of "Real" or "Simulation" with advantage essentially ε.

On the other hand if pk^* is a meaningless key, the encryptions contain statistically no information about the values inside. Moreover, all messages sent in the zero-knowledge protocols where P_2 acts as prover do not depend on the specific values that P_2 has, since the proofs are simulated. We conclude that essentially no information on any value held by P_2 is revealed. This is the case also for step 5 of $\mathsf{Reshare}(e_{\mathbf{m}})$: $\mathbf{m} + \mathbf{f}$ is retrieved, but no information on $\mathbf{m}$ is revealed, since $\mathbf{f}$ is uniform.

The view $\mathcal{Z}$ sees consists of the view of the corrupt player(s) and the output of the honest player(s). We just argued that the view of the corrupt player is essentially independent of the internal values B uses for P_2, and hence also independent of whether B chooses the real or the simulated case. Therefore, the output generated for the honest player(s) that is seen by $\mathcal{Z}$ is in both cases a set of (essentially) uniformly and independently chosen shares and MAC keys. As a result, using a meaningless key, a real execution and a simulated one are statistically indistinguishable, and the guess of $\mathcal{Z}$ equals B's random choice of "Real" or "Simulation" with probability essentially $1/2$.

An easy calculation now shows that the advantage of B is

$$\begin{aligned}\mathsf{A}(B) &:= \Pr\left[\text{"Standard Key"} \leftarrow B(\mathsf{pk})\right] - \Pr\left[\text{"Standard Key"} \leftarrow B(\widetilde{\mathsf{pk}})\right] \\ &\geq \mathsf{A}(\mathcal{Z})/2 - \delta \\ &= \varepsilon/2 - \delta,\end{aligned}$$

for some negligible δ that accounts for the differences between the involved distributions. However, if ε is non-negligible, then $\varepsilon/2 - \delta$ is also non-negligible, which contradicts the assumption on that meaningless keys are statistically indistinguishable from standard ones. □

2.4. A Lower Bound for the Preprocessing

In this section we show that any preprocessing matching the properties we have must output the same amount of data as we do, up to a constant factor. We use the following theorem for 2-party computation from [WW10], in a setting where the parties A, B have access to a functionality that gives a random variable U to A and V to B with some guaranteed joint distribution P_{UV} of U, V. Given this, the parties compute securely a function $f : \mathcal{X} \times \mathcal{Y} \mapsto \mathcal{Z}$, where A holds $x \in \mathcal{X}$, and B holds $y \in \mathcal{Y}$. This function should have the property that there exist inputs y_1, y_2 such that

$$\text{for all } x \neq x',\ f(x, y_1) \neq f(x', y_1) \tag{2.3}$$

$$\text{for all } x, x',\ f(x, y_2) = f(x', y_2). \tag{2.4}$$

In other words, for some inputs B learns all of A's input, but for other inputs B learns nothing new.

THEOREM 2.4 (Theorem 2 in [WW10]). *Let $f : \mathcal{X} \times \mathcal{Y} \mapsto \mathcal{Z}$ be a function with property (2.3) and (2.4). If there exists a protocol that computes f securely with access to P_{UV} and with error probability ε in the semi-honest model, then*

$$H(V) \geq I(U; V) \geq \log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)).$$

We also need the following technical lemma:

LEMMA 2.5. *Let R be a random variable defined over the natural numbers. Then there exists a constant C such that $E(R) \geq H(R) - 1 - C$.*

Proof. Let

$$I := \left\{ i \mid i \geq \log\left(\frac{1}{Pr[R=i]}\right) \right\}.$$

Under this definition, the following holds

$$H(R) = \sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) + \sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right)$$

By the construction of I, the first summand is bounded as follows

$$\begin{aligned}\sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) &\leq \sum_{i \in I} Pr[R=i] \cdot i \\ &\leq \sum_{i} Pr[R=i] \cdot i \\ &= E(R).\end{aligned}$$

For the second summand a little more work is needed. Let $q(i) := \log(1/Pr[R=i])$. Then

$$\sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) = \sum_{i \notin I} 2^{-q(i)} \cdot q(i).$$

We now claim that

$$2^{-q(i)} \cdot q(i) \leq 2^i \cdot i, \text{ for all } 0 \neq i \notin I.$$

This happens if and only if

$$2^{-q(i)} \cdot 2^{\log(q(i))} \leq 2^{-i} \cdot 2^{\log(i)}.$$

Taking the logarithm of this relation one gets $-q(i) + \log(q(i)) \leq -i + \log(i)$, which is equivalent to $q(i) - \log(q(i)) \geq i - \log(i)$.
Since $q(i) = \log(1/Pr[R=i]) \geq i$ for all $i \notin I$, and $i \geq 1$, the latter relation is always satisfied. Therefore, the second summand is bounded by $C + \sum_{i \geq 1} 2^{-i} \cdot i$, where $C = 2^{-q(0)} \cdot q(0)$.
Moreover, $\sum_{i \geq 1} 2^{-i} \cdot i$ converges to 1, so the second summand can be bounded by $1 + C$.

Finally, from all of the above it follows that

$$\sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) + \sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) \leq E(R) + C + 1.$$

The last inequality implies that $H(R) \leq E(R) + 1 + C$ □

We can now show a lower bound on the amount of preprocessing data and work required for a protocol satisfying our specification.

THEOREM 2.6. *Assume a protocol π in the preprocessing model can compute any circuit over $\mathbb{F}_{p^k}$ of size at most S, with security against active corruption of at most $n-1$ players. We assume that the players supply roughly the same number of inputs ($O(S/n)$ each), and that any any player may receive output. Then, the preprocessing must output $\Omega(S \log p^k)$ bits to each player, and for any player P_i, there exists a circuit C satisfying the conditions above, where secure computation of C requires P_i to execute an expected number of bit operations that is $\Omega(S \log p^k)$.*

Proof. Suppose there exists an online protocol π that satisfies the assumptions in the theorem. Consider any player P_i and suppose we want to compute the function

$$f_T((\vec{x}, \vec{x}'), y) = y\vec{x} + (1-y)\vec{x}'.$$

Here $y \in \mathbb{F}_{p^k}$ and $\vec{x}, \vec{x}'$ are vectors over $\mathbb{F}_{p^k}$ of length T. P_i has input y and each P_j, $j \neq i$ has as input substrings $\vec{x}_j, \vec{x}'_j$ such that the concatenation of all $\vec{x}_j$ ($\vec{x}'_j$) is $\vec{x}$ ($\vec{x}'$). Finally, only P_i learns the output $f_T((\vec{x}, \vec{x}'), y)$.

Clearly, f_T can be computed using a circuit of size $O(T)$, and this is the circuit promised in the theorem. Note that our assumed protocol π can handle circuits of size S and can therefore compute f_T securely where T is $\Theta(S)$.

One can transform π to a two-party protocol π' for parties A and B. A has input $\vec{x}, \vec{x}'$, B has input y and B is supposed to learn $f_T((\vec{x}, \vec{x}'), y)$. Now, π' simply consists of running π where B emulates P_i and A emulates all other players. B is then given whatever P_i gets from the preprocessing and A gets whatever the other players receive, so this defines the random variables U and V. Since π is secure if P_i is corrupt and also if all other players are corrupt, it trivially means that π' is an actively secure two-party protocol for computing f_T.

This implies that π' also computes f_T with passive security. As noted in [WW10], this is actually not necessarily the case for all functions. The problem is that if the adversary is passive, then active security guarantees that there is a simulator for this case, but such a simulator is allowed to change the inputs of corrupted parties. A simulator for the passive case is not allowed to do this. However, [WW10] observes that for some functions, an active simulator cannot get away with changing the inputs, as this would make it impossible to simulate correctly. They show this is the case for Oblivious Transfer, which is essentially f_T in the 2-party case. We may therefore assume π' is also passively secure.

Finally, we define $f'_T(\vec{x}, y) = f_T((\vec{x}, \vec{0}), y) = y\vec{x}$. Obviously π' can be used to compute f'_T securely: A just sets her second input to be $\vec{0}$. Moreover, f'_T satisfies the conditions in Theorem 2.4. So we get that $H(V) \geq \log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon))$. If we adopt the standard convention that the security parameter grows linearly with the input size $\log|\mathcal{X}|$ then because ε is negligible in the security parameter, we have that the "error term" $7(\varepsilon \log|\mathcal{X}| + h(\varepsilon))$ is $o(\log|\mathcal{X}|)$.

So we get that $H(V)$ is $\Omega(\log|\mathcal{X}|) = \Omega(T \log p^k) = \Omega(S \log p^k)$, since T is $\Theta(S)$. Recalling that $H(V)$ is actually the entropy of the variable P_i received in the original protocol π, we get the first conclusion of the Theorem.

For the second conclusion about the computational work done, it is tempting to simply claim that B has to at least read the information he is given and so $H(V)$ is a lower bound on the expected number of bit operations. But this is not enough, as it is conceivable that in any given execution, B might only have to read a small part of the information.

It turns out that this does not happen, however, which can be argued as follows: let $B(V)$ be the random variable representing the bits of V that B actually reads. By inspection of the proof of Theorem 2.4, one sees that replacing V by $B(V)$ the proof still applies. So, we have $H(B(V)) \geq \log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon))$. Now let R be the random variable representing the number of bits B reads from V.

Conditioning on R, the entropy of $B(V)$ cannot drop by more than $H(R)$, so

$$H(B(V)|R) \geq H(B(V)) - H(R) \geq \log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)) - H(R).$$

Moreover,

$$H(B(V)|R) = \sum_r Pr(R = r) H(B(V)|R = r) \leq \sum_r Pr(R = r) r = E(R)$$

Putting these two inequalities together, we obtain that

$$E(R) + H(R) \geq \log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)).$$

Now, either $E(R) \geq (\log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)))/2$, or $H(R) \geq (\log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)))/2$. In the latter case Lemma 2.5 implies that $E(R)$ is much larger than $H(R)$, so we can certainly conclude that $E(R) \geq (\log|\mathcal{X}| - 7(\varepsilon \log|\mathcal{X}| + h(\varepsilon)))/2$ in any case. As above, the error term depending on ε becomes negligible as the security parameter increases, so we get that $E(R)$ is $\Omega(S \log p^k)$ as desired. □

2.5. Concrete Instantiation of the Abstract Scheme Based on LWE

We now describe the concrete scheme, which is based on the somewhat homomorphic encryption scheme of Brakerski and Vaikuntanathan (BV) [BV11]. The main differences are that we are only interested in evaluation of circuits of multiplicative depth one, we are interested in performing operations in parallel on multiple data items, and we require a distributed decryption procedure. In this section we detail the scheme and the distributed decryption procedure; in Section 3.2 we discuss security of the scheme, and present some sample parameter sizes and performance figures.

$\underline{\mathsf{ParamGen}(1^\kappa, M)}$: Recall the message space is given by $M = (\mathbb{F}_{p^k})^s$ for two integers k and s, and a prime p, i.e. the message space is s copies of the finite field $\mathbb{F}_{p^k}$. To map this to our scheme below, one first finds a cyclotomic polynomial $F(X) := \Phi_m(X)$ of degree $N := \phi(m)$, where N is lower bounded by some function of the security parameter κ. The polynomial $F(X)$ needs to be such that modulo p it factors into l' irreducible factors of degree k' where $l' \geq s$ and k divides k'. We then define an algebra A_p as $A_p := \mathbb{F}_p[X]/F(X)$ and an embedding of M into A_p, $\phi : M \to A_p$. "Lifting" modulo p gives a natural inclusion $\iota : A_p \to \mathbb{Z}^N$, which maps the polynomial of degree less than N with coefficients in $\mathbb{F}_p$ into the integer vector of length N with coefficients in the range $(-p/2, \ldots, p/2]$. The encode function is then defined by $\iota(\phi(\mathbf{m}))$ for $\mathbf{m} \in (\mathbb{F}_{p^k})^s$, with decode defined by $\phi^{-1}(\mathbf{x} \pmod p)$ for $\mathbf{x} \in \mathbb{Z}^N$. It is clear, by choice of the natural inclusion ι, that $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq p/2 = \tau$.

We choose a large integer q, whose size will be determined later, and define the algebra $A_q := (\mathbb{Z}/q\mathbb{Z})[X]/F(X)$, i.e. the ring of integer polynomials modulo reduction by $F(X)$ and q. In practice we consider the image of encode to lie in A_q, and thus we abuse notation, by writing addition and multiplication in A_q by $+$ and $\cdot$. Note that this means that applying decode to elements obtained from encode followed by a series of arithmetic operations may not result in the value in M which one would expect. This corresponds to where our scheme can only evaluate circuits from a given set C.

The ciphertext space G is defined to be A_q^3, with addition $\boxplus$ defined component-wise. The multiplicative operator $\boxtimes$ is defined as follows

$$(\mathbf{a}_0, \mathbf{a}_1, 0) \boxtimes (\mathbf{b}_0, \mathbf{b}_1, 0) := (\mathbf{a}_0 \cdot \mathbf{b}_0, \mathbf{a}_1 \cdot \mathbf{b}_0 + \mathbf{a}_0 \cdot \mathbf{b}_1, -\mathbf{a}_1 \cdot \mathbf{b}_1),$$

i.e. multiplication is only defined on elements whose third coefficient is zero.

We define D_ρ^d as follows: the discrete Gaussian $D_{\mathbb{Z}^N,s}$, with *Gaussian parameter* s, is defined to be the random variable on $\mathbb{Z}_q^N$ (centred around the origin) obtained from sampling $\mathbf{x} \in \mathbb{R}^N$, with probability proportional to $\exp(-\pi \cdot \|\mathbf{x}\|_2/s^2)$, and then rounding the result to the nearest lattice point and reducing it modulo q. Note that sampling from the distribution with probability density function proportional to $\exp(-\pi \cdot \|\mathbf{x}\|_2/s^2)$ means using a normal variate with mean zero, and standard deviation $r := s/\sqrt{2 \cdot \pi}$. In our concrete scheme we set $d := 3 \cdot N$ and define D_ρ^d to be the distribution defined by $(D_{\mathbb{Z}^N,s})^3$. Note, that in the notation D_ρ^d the implicit dependence on q has been suppressed to ease readability. We will specify q and r (as functions of all the other parameters) when we discuss security of the scheme.

$\underline{\mathsf{KeyGen}()}$: We use the public key version of the Brakerski-Vaikuntanathan scheme [BV11]. Given the above set up, key generation proceeds as follows: first, sample elements $\mathbf{a} \leftarrow A_q$ and $\mathbf{s}, \mathbf{e} \leftarrow D_{\mathbb{Z}^N,s}$. Then, treating $\mathbf{s}$ and $\mathbf{e}$ as elements of A_q, compute $\mathbf{b} \leftarrow (\mathbf{a} \cdot \mathbf{s}) + (p \cdot \mathbf{e})$. The public and private key are then set to be $\mathsf{pk} \leftarrow (\mathbf{a}, \mathbf{b})$ and $\mathsf{sk} \leftarrow \mathbf{s}$.

$\underline{\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})}$: Given a message $\mathbf{x} \leftarrow \mathsf{encode}(m)$ where $m \in M$, and $\mathbf{r} \in \mathsf{D}_\rho^d$, the encryptor parses $\mathbf{r}$ as $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in (\mathbb{Z}^N)^3$, computes $\mathbf{c}_0 \leftarrow (\mathbf{b} \cdot \mathbf{v}) + (p \cdot \mathbf{w}) + \mathbf{x}$ and $\mathbf{c}_1 \leftarrow (\mathbf{a} \cdot \mathbf{v}) + (p \cdot \mathbf{u})$. and finally returns the ciphertext $(\mathbf{c}_0, \mathbf{c}_1, 0)$.

$\underline{\mathsf{Dec}_{\mathsf{sk}}(c)}$: Given a secret key $\mathsf{sk} = \mathbf{s}$ and a ciphertext $c = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$ this algorithm computes the element in A_q satisfying $\mathbf{t} = \mathbf{c}_0 - (\mathbf{s} \cdot \mathbf{c}_1) - (\mathbf{s} \cdot \mathbf{s} \cdot \mathbf{c}_2)$. On reduction by q the value of $\|\mathbf{t}\|_\infty$ is bounded by a relatively small constant B, assuming that the "noise" within a ciphertext has not grown too large. We shall refer to the value $\mathbf{t} \mod q$ as the "noise", despite it also containing

the message to be decrypted. At this point the decryptor simply reduces $\mathbf{t}$ modulo p to obtain the desired plaintext in A_q, which can then be decoded via the decode algorithm.

$\underline{\mathsf{KeyGen}^*()}$: This simply samples $\widehat{\mathbf{a}}, \widehat{\mathbf{b}} \leftarrow A_q$ and returns $\widehat{\mathsf{pk}} := (\widehat{\mathbf{a}}, \widehat{\mathbf{b}})$.

Following the discussion in [BV11] we see that with this *fixed* ciphertext space, our scheme is somewhat homomorphic. It can support a relatively large number of addition operations, and a single multiplication.

Distributed Version: We now extend the scheme above to enable distributed decryption. We first set up the distributed keys as follows. After invoking the functionality for key generation, each player obtains a share $\mathsf{sk}_i = (\mathbf{s}_{i,1}, \mathbf{s}_{i,2})$. These are chosen uniformly such that the master secret is written as

$$\mathbf{s} = \mathbf{s}_{1,1} + \cdots + \mathbf{s}_{n,1}, \quad \mathbf{s} \cdot \mathbf{s} = \mathbf{s}_{1,2} + \cdots + \mathbf{s}_{n,2}.$$

As remarked earlier this one-time setup procedure can be accomplished by standard UC-secure multiparty computation protocols such as that described in [BDOZ11]. In Section 3.2 we determine the value of B when the input ciphertext is (B_{plain}, B_{rand}, C)-admissible, and show how to choose parameters for the cryptosystem such that the required bound on B is satisfied.

Protocol Π_{DDEC}

Initialise: Each party P_i, on being given the ciphertext $c = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$, and an upper bound B on the infinity norm of $\mathbf{t}$ above, computes

$$\mathbf{v}_i \leftarrow \begin{cases} \mathbf{c}_0 - (\mathbf{s}_{i,1} \cdot \mathbf{c}_1) - (\mathbf{s}_{i,2} \cdot \mathbf{c}_2) & \text{if } i = 1 \\ -(\mathbf{s}_{i,1} \cdot \mathbf{c}_1) - (\mathbf{s}_{i,2} \cdot \mathbf{c}_2) & \text{if } i \neq 1 \end{cases}$$

and sets $\mathbf{t}_i \leftarrow \mathbf{v}_i + p \cdot \mathbf{r}_i$ where $\mathbf{r}_i$ is a random element with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$.

Public Decryption: All the players are supposed to learn the message.

(1) Each party P_i broadcasts $\mathbf{t}_i$

(2) All players compute $\mathbf{t}' \leftarrow \mathbf{t}_1 + \cdots + \mathbf{t}_n$ and obtain a message $m' \leftarrow \mathsf{decode}(\mathbf{t}' \bmod p)$.

Private Decryption: Only player P_j is supposed to learn the message.

(1) Each party P_i sends $\mathbf{t}_i$ to P_j

(2) P_j computes $\mathbf{t}' \leftarrow \mathbf{t}_1 + \cdots + \mathbf{t}_n$ and obtain a message $m' \leftarrow \mathsf{decode}(\mathbf{t}' \bmod p)$.

FIGURE 2.12. The distributed decryption protocol.

THEOREM 2.7. *In the $\mathcal{F}_{\text{KEYGEN}}$-hybrid model, the protocol Π_{DDEC} (Figure 2.12) implements $\mathcal{F}_{\text{KEYGENDEC}}$ with statistical security against any static active adversary corrupting up to $n-1$ parties if $B + 2^{\mathsf{sec}} \cdot B < q/2$.*

Proof. The requirement $B + 2^{\mathsf{sec}} \cdot B < q/2$ implies that $\mathbf{t}' = \mathbf{t} \bmod p$, since $\|\mathbf{r}_i\|_\infty < 2^{\mathsf{sec}} \cdot B/(n \cdot p)$ for $i = 1, \ldots, n$. Therefore the protocol allows players to retrieve the correct message if all the players are honest.

We now build a simulator $\mathcal{S}_{\text{DDEC}}$ to work on top of $\mathcal{F}_{\text{KEYGENDEC}}$, such that the adversary cannot distinguish whether it is playing with the decryption protocol and $\mathcal{F}_{\text{KEYGEN}}$ or the simulator and $\mathcal{F}_{\text{KEYGENDEC}}$. Let A denote the set of players controlled by the adversary.

In a simulated decryption the adversary receives pk and $(\mathbf{t}_i)_{i \notin A, i \neq j}, \widetilde{\mathbf{t}}_j$ from $\mathcal{S}_{\text{DDEC}}$. The distribution of pk is the same as in a real conversation, since it was sampled using the same algorithm as in a real conversation. The distribution of simulated $\mathbf{t}_i$, $i \neq j$ is statistically close to the real one, since $\mathbf{t}_i$ was computed correctly using shares of a possible secret key. We therefore focus on the case where all the players but one are dishonest. We first analyse the simulation of public decryption, introducing a hybrid machine, and prove its output is statistically indistinguishable from P_j's output (in the real protocol) and perfectly indistinguishable from P_j's simulated output.

Hybrid: On input $(\mathsf{sk}_i)_{i=1,\ldots,n}, c$, reconstruct sk, compute $\mathsf{Dec}_{\mathsf{sk}}(c)$, sample $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ and output $\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i \neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j + \mathsf{encode}(m)$.

Simulator $\mathcal{S}_{\text{DDEC}}$

Key Generation: This stage is needed to distribute shares of a secret key.

- Upon "start", the simulator sends "start" to $\mathcal{F}_{\text{KEYGENDEC}}$ and obtains pk. Moreover, the simulator obtains $(\mathsf{sk}_i)_{i\in A}$ from the adversary.
- The simulator (internally) sets random $(\mathsf{sk}_i)_{i\notin A}$ such that $(\mathsf{sk}_i)_{i=1,\dots,n}$ is a full vector of shares of 0.
- The simulator sends pk to A.

Public Decryption: This stage simulates a public decryption.

- Upon "decrypt c, B", the simulator sends "decrypt c" to $\mathcal{F}_{\text{KEYGENDEC}}$ and obtains $m = \mathsf{Dec}_{\mathsf{sk}}(c)$.
- It then computes the value $\mathbf{v}_i$ for all players except for an honest player P_j.
- It then samples $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ and computes
$$\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i\neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j + \mathsf{encode}(m).$$
- For each other honest player P_i, it computes $\mathbf{t}_i$ honestly (using c, sk_i).
- The simulator broadcasts the values $(\mathbf{t}_i)_{i\notin A, i\neq j}, \widetilde{\mathbf{t}}_j$ and obtains $(\mathbf{t}_i^*)_{i\in A}$ from the adversary.
- It then sends $m' \leftarrow \mathsf{decode}\left(\left(\widetilde{\mathbf{t}}_j + \sum_{i\in A} \mathbf{t}_i^* + \sum_{i\notin A, i\neq j} \mathbf{t}_i\right) \bmod p\right)$ to $\mathcal{F}_{\text{KEYGENDEC}}$ so that the ideal functionality sends "Result m'" to all the players.

Private Decryption: This stage simulates a private decryption.

- Upon "decrypt c, B to P_j", the simulator sends "decrypt c to P_j" to $\mathcal{F}_{\text{KEYGENDEC}}$.
- If P_j is corrupt, the simulator obtains $c, m = \mathsf{Dec}_{\mathsf{sk}}(c)$ from $\mathcal{F}_{\text{KEYGENDEC}}$ and acts as in the simulated public decryption.
- If P_j is honest, the simulator receives c from $\mathcal{F}_{\text{KEYGENDEC}}$, $\mathbf{t}_i^*$ from each corrupt player P_i and $\mathbf{t}_i$ from each honest player.
  - The simulator samples $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$.
  - It evaluates $\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i\neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j$.
  - It computes $\varepsilon \leftarrow \left(\widetilde{\mathbf{t}}_j + \sum_{i\in A} \mathbf{t}_i^* + \sum_{i\notin A, i\neq j} \mathbf{t}_i\right) \bmod p$
  - , it sends $\delta \leftarrow \mathsf{decode}(\varepsilon)$ to $\mathcal{F}_{\text{KEYGENDEC}}$ in order to get $\mathsf{Dec}_{\mathsf{sk}}(c) + \delta$ to P_j.

FIGURE 2.13. The simulator for Π_{DDEC}.

Notice that $\widetilde{\mathbf{t}}_j = \mathbf{v}_j - \mathbf{t} + \mathsf{encode}(m) + p \cdot \mathbf{r}_j$. Now, for a distribution X, define $\varphi(X) := p \cdot X + \mathbf{v}_j$. Notice that $\mathbf{t}_j = \varphi(U)$, where U denotes the uniform distribution over vectors of integral entries bounded with infinity norm $2^{\mathsf{sec}} \cdot B/(n \cdot p)$; moreover, since $\mathbf{t} - \mathsf{encode}(m)$ is a multiple of p, one can write $\widetilde{\mathbf{t}}_j = \varphi(U + (\mathsf{encode}(m) - \mathbf{t})/p)$. Notice that $\|(\mathsf{encode}(m) - \mathbf{t})/p\|_\infty \leq (B+p)/p$, so the distribution $U + (\mathsf{encode}(m) - \mathbf{t})/p$ is statistically close to U, since the probability of distinguishing $U + (\mathsf{encode}(m) - \mathbf{t})/p$ and U is bounded by the ratio

$$\begin{aligned}\frac{N \cdot \|(\mathsf{encode}(m) - \mathbf{t})/p\|_\infty}{2^{\mathsf{sec}} \cdot B/(n \cdot p) + (B+p)/p} &\leq \frac{N \cdot (B+p)/p}{2^{\mathsf{sec}} \cdot B/(n \cdot p) + (B+p)/p} \\ &= O(N \cdot n \cdot 2^{-\mathsf{sec}}),\end{aligned}$$

which is negligible. Therefore $\widetilde{\mathbf{t}}_j$ is statistically close to $\mathbf{t}_j$.

What is left to prove is that the simulation of private decryption to an honest player P_j is statistically indistinguishable from the real protocol. In the real protocol P_j computes $\mathbf{t}_j$ and

$$m' \leftarrow \mathsf{decode}\left(\sum_{i\in A} \mathbf{t}_i^* + \sum_{i\notin A} \mathbf{t}_i\right).$$

In that case the error $m' - m$ introduced by the adversary depends only on the value

$$\varepsilon' := \left(\sum_{i \in A} (\mathbf{t}_i^* - \mathbf{t}_i) \right) \bmod p$$

computed using the actual secret key. In the simulation the error introduced by the adversary is

$$\varepsilon = \left(\widetilde{\mathbf{t}}_j + \sum_{i \in A} \mathbf{t}_i^* + \sum_{i \notin A, i \neq j} \mathbf{t}_i \right) \bmod p = \left(\sum_{i \in A} (\mathbf{t}_i^* - \mathbf{t}_i) \right) \bmod p,$$

computed using secret shares of 0. Since the secret sharing scheme has privacy threshold n and the sums involve at most $n - 1$ shares, the quantities ε and ε' are statistically indistinguishable. □

CHAPTER 3

SPDZ – Addenda

3.1. Canonical Embeddings of Cyclotomic Fields

Our concrete instantiation uses some basic results of cyclotomic fields which we now describe; these results are needed for the main result of this section which is a proof of a "folklore" result about the relationship between norms in the canonical and polynomial embeddings of a cyclotomic field. This result is used repeatedly in our main construction to produce estimates on the sizes of parameters needed.

We first recap on some basic facts about number fields and their canonical embeddings, focusing particularly on the case of cyclotomic fields.

3.1.1. Number Fields. An algebraic number (resp. algebraic integer) $\theta \in \mathbb{C}$ is the root of a polynomial (resp. monic polynomial) with coefficients in $\mathbb{Q}$ (resp. $\mathbb{Z}$). The minimal polynomial of θ is the unique monic irreducible polynomial $F(X) \in \mathbb{Q}[X]$ which has θ as a root.

A number field $K = \mathbb{Q}(\theta)$ is the field obtained by adjoining powers of an algebraic number θ to $\mathbb{Q}$. If θ has minimal polynomial $F(X)$ of degree N, then K can be considered as a vector space over $\mathbb{Q}$ of dimension N, with basis $\{1, \theta, \ldots, \theta^{N-1}\}$. Note that this "coefficient embedding" is relative to the defining polynomial $F(X)$. Equivalently, $K \cong \mathbb{Q}[X]/F(X)$, i.e. the field of rational polynomials with degree less than N, modulo the polynomial $F(X)$. Without loss of generality we can assume that K is defined by a monic irreducible integral polynomial of degree N. The ring of integers $\mathcal{O}_K$ of K is defined to be the subring of K consisting of all elements whose minimal polynomial has integer coefficients.

3.1.2. Canonical Embedding. There are N field morphisms $\sigma_i : K \longrightarrow \mathbb{C}$ which fix every element of $\mathbb{Q}$. Such a morphism is called a complex *embedding* and it takes θ to each distinct complex root of $F(X)$. The number field K is said to have signature (s_1, s_2) if the defining polynomial has s_1 real roots and s_2 non-real, complex conjugate pairs of roots; clearly $N = s_1 + 2 \cdot s_2$. The roots are numbered in the standard way so that $\sigma_i(\theta) \in \mathbb{R}$ for $1 \leq i \leq s_1$ and $\sigma_{i+s_1+s_2}(\theta) = \overline{\sigma_{i+s_1}(\theta)}$ for $1 \leq i \leq s_2$. We define $\sigma = (\sigma_1, \ldots, \sigma_N)$, which determines the *canonical embedding* of K into $\mathbb{R}^{s_1} \times \mathbb{C}^{2 \cdot s_2}$, where the field operations in K are mapped into component-wise addition and multiplication in $\mathbb{R}^{s_1} \times \mathbb{C}^{2 \cdot s_2}$. For ease of notation we often write $\alpha^{(i)} = \sigma_i(\alpha)$, for $\alpha \in K$.

Let $\|\alpha\|_p$ for $p \in \{1, \ldots, \infty\}$ denote the p-norm of α in the coefficient embedding (i.e. the p-norm of the vector of coefficients) and let $\|\sigma(\alpha)\|_p$ denote norms in the canonical embedding.

3.1.3. Cyclotomic Fields. We are mainly interested in cyclotomic number fields. The mth cyclotomic polynomial is given by $\Phi_m(X)$, which is an irreducible polynomial (over $\mathbb{Z}$) of degree $N = \phi(m)$. The number field defined by $\Phi_m(X)$ is said to be a cyclotomic number field, and is defined by $K = \mathbb{Q}(\zeta_m)$, where ζ_m is an mth root of unity, i.e. a root of $\Phi_m(X)$. The ring of integers of K is equal to $\mathbb{Z}[\zeta_m]$. The number field K is Galois, and hence (importantly for us) the polynomial splits modulo p (for any prime p not dividing m) into a product of distinct irreducible polynomials all of the same degree.

The key fact is that if $\Phi_m(X)$ has factors of degree d modulo the prime p then m divides $p^d - 1$. To see this, notice that if $\Phi_m(X)$ factors into N/d factors each of degree d, then the finite field $\mathbb{F}_{p^d}$ must contain the mth roots of unity and so m divides $p^d - 1$. In the other direction,

if d is the smallest integer such that m divides $p^d - 1$ then $\Phi_m(X)$ has a degree d factor since the decomposition group of the prime p in the Galois group has order d.

3.1.4. Relating Norms Between Canonical and Polynomial Embeddings. There is a distinct difference between the canonical and polynomial embeddings of a number field. In particular, notice the following expansions upon multiplication, for $x, y \in \mathcal{O}_K$,

$$\|x \cdot y\|_\infty \leq \delta_\infty \cdot \|x\|_\infty \cdot \|y\|_\infty.$$
$$\|\sigma(x \cdot y)\|_p \leq \|\sigma(x)\|_\infty \cdot \|\sigma(y)\|_p.$$

where

$$\delta_\infty = \sup \left\{ \frac{\|a(X) \cdot b(X) \pmod{F(X)}\|_\infty}{\|a(X)\|_\infty \cdot \|b(X)\|_\infty} : a, b \in \mathbb{Z}[X], \deg(a), \deg(b) < N \right\}.$$

In this section we show that the expansion factors of elements in the polynomial representation can be more tightly controlled, as long as they are drawn randomly with a discrete Gaussian distribution. In particular we prove the following theorem; this result is well known among researchers in ideal lattice theory, but proofs have not yet appeared in any paper.

THEOREM 3.1. *Let K denote a cyclotomic number field then there is a constant C_m, depending only on m, such that for all $\alpha \in \mathcal{O}_K$ we have*

- $\|\sigma(\alpha)\|_\infty \leq \|\alpha\|_1$.
- $\|\alpha\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty$.

We recall some facts about various matrices associated with roots of unity, see [PM08] and the full version of [LPR11]. First some notation: for any integer $m \geq 2$, set $\zeta_m = \exp(2 \cdot \pi \cdot \sqrt{-1}/m)$ to be a root of unity for an integer m. As usual we let $N = \phi(m)$ and we define $\mathbb{Z}_m^* = \{a_{m,i} : 0 \leq i < N\}$ to be a complete set of representatives for $\mathbb{Z}_m^*$ with $1 \leq a_{m,i} < m$. We let $A \otimes B$, for matrices A and B, denote the Kronecker product. We let I_t denote the $t \times t$ identity matrix. All $a \times b$ matrices M in this section will have elements $m_{i,j}$ indexed by $0 \leq i < a$ and $0 \leq j < b$. Note that we index from zero: this is to make some of the expressions neater. The infinity norm for a matrix $M = (m_{i,j})$ is defined by

$$\|M\|_\infty := \max \left\{ \sum_{j=0}^{N-1} |m_{i,j}| \right\}_{i=0}^{N-1}.$$

We define the $N \times N$ CRT matrix as follows:

$$\mathsf{CRT}_m := \left(\zeta_m^{a_{m,i} \cdot j} \right)_{0 \leq i,j < N}.$$

Then we define the constant C_m in the above theorem as $C_m = \|\mathsf{CRT}_m^{-1}\|_\infty$. From this, the proof immediately follows, as below:

Proof. [Theorem 3.1] For a cyclotomic field, the canonical embedding is given by the map $\sigma(\alpha) = \mathsf{CRT}_m \cdot \alpha$, where α is thought of as the vector of the coefficient embedding of α, i.e. α considered as a polynomial in θ (a root of $F(X) = \Phi_m(X)$), and CRT_m is the matrix defined earlier, i.e. it satisfies the following equality

$$\mathsf{CRT}_m = \begin{pmatrix} 1 & \theta^{(1)} & \dots & \theta^{(1)N-1} \\ \vdots & \vdots & & \vdots \\ 1 & \theta^{(N)} & \dots & \theta^{(N)N-1} \end{pmatrix}.$$

For the first part of the theorem we note that, since $\alpha = \sum_{i=0}^{N-1} x_i \cdot \theta^i$, we have

$$\left|\alpha^{(i)}\right| = \left| \sum_{j=0}^{N-1} x_j \cdot \theta^{(i)j} \right| \leq \sum_{j=0}^{N-1} |x_j| \cdot |\theta^{(i)j}| = \sum_{j=0}^{N-1} |x_j| = \|\mathbf{x}\|_1 = \|\alpha\|_1.$$

For the second part we note that for all $\beta \in \mathcal{O}_K$,

$$\|\beta\|_\infty = \|\mathsf{CRT}_m^{-1} \cdot \sigma(\beta)\|_\infty \leq \|\mathsf{CRT}_m^{-1}\|_\infty \cdot \|\sigma(\beta)\|_\infty$$

from which the result follows. □

The key question then is how large can C_m become. So we now turn to this problem, and we shall give a partial answer.

The $m \times m$ DFT matrix is defined by:

$$\mathsf{DFT}_m := \left(\zeta_m^{i \cdot j}\right)_{0 \leq i,j < m}.$$

Let m' be a divisor of m. For $i \in \{0, \ldots, m-1\}$ we write $i_0 = i \bmod m'$ and $i_1 = (i - i_0)/m'$. We then define the $m \times m$ "twiddle matrix" to be the diagonal matrix defined by

$$T_{m,m'} := \mathsf{Diag}\left\{\zeta_m^{i_0 \cdot i_1}\right\}_{i=0,\ldots,m-1}.$$

Finally, we define $L_{m'}^m$ to be the permutation matrix which fixes the row with index $m-1$, but sends all other rows i, for $0 \leq i < m-1$, to row $i \cdot m' \bmod m-1$. Following [PM08] we use these matrices to decompose the matrix D_m into D'_m and D_k, where $m = m' \cdot k$, via the following identity

$$\mathsf{DFT}_m = L_{m'}^m \cdot (I_k \otimes \mathsf{DFT}_{m'}) \cdot T_{m,m'} \cdot (\mathsf{DFT}_k \otimes I_{m'}), \tag{3.1}$$

This is simply the general Cooley-Tukey decomposition of the DFT for composite m. Consider the Vandermonde matrix

$$V(x_1, \ldots, x_m) := \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{m-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^{m-1} \end{pmatrix}.$$

It is clear that $\mathsf{DFT}_m = V(1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1})$.

Lemma 3.2. *We have, for any m,*

$$\mathsf{DFT}_m^{-1} = \frac{1}{m} \cdot V(1, \zeta_m^{-1}, \zeta_m^{-2}, \ldots, \zeta_m^{1-m})$$

Proof. Let $\delta_{i,j}$ be defined so that $\delta_{i,j} = 0$ if $i \neq j$ and $\delta_{i,i} = 1$. We have

$$\begin{aligned} \left(V(1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}) \cdot V(1, \zeta_m^{-1}, \zeta_m^{-2}, \ldots, \zeta_m^{1-m})\right)_{i,j} & \\ = \sum_{0 \leq k < m} \zeta_n^{i \cdot k} \cdot \zeta_m^{-k \cdot j} & \\ = \sum_{0 \leq k < m} \zeta_m^{k \cdot (i-j)} = m \cdot \delta_{i,j}. & \end{aligned}$$

□

This leads to the following lemma which gives shows that the infinity norm of the inverse of the DFT matrix is always equal to one.

Lemma 3.3. *For any m we have $\|\mathsf{DFT}_m^{-1}\|_\infty = 1$.*

Proof. If ζ_m is an m-th root of unity, it is clear that $\|V(1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1})\|_\infty = m$. In addition we have that $\psi_m = 1/\zeta_m$ is also an m-th root of unity, and thus

$$\|\mathsf{DFT}_m^{-1}\|_\infty = \frac{1}{m} \cdot \|V(1, \psi_m, \psi_m^2, \ldots, \psi_m^{m-1})\|_\infty = \frac{m}{m} = 1.$$

□

Let $m = p_1^{e_1} \cdots p_k^{e_s}$, $r = p_1 \cdots p_s$, $m_1 = m/r$. Hence $N = \phi(m) = \phi(r) \cdot m_1$. In [LPR11] the authors specialise the decomposition (3.1) (by selecting appropriate rows and columns) in

the case $m' = m_1$ and $k = r$, to show that up to a permutation of the rows, the matrix CRT_m is equal to

$$\left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}\right) \cdot T^*_{m,m_1} \cdot \left(\mathsf{CRT}_r \otimes I_{m_1}\right)$$

where T^*_{m,m_1} is another diagonal matrix consisting of roots of unity. We then have that

LEMMA 3.4. *For an integer $m \geq 2$ such that $m = p_1^{e_1} \cdots p_k^{e_k}$ we write $r = p_1 \cdots p_k$, we then have $C_m \leq C_r$.*

Proof. As above we write $m_1 = m/r$. First note that $\|A \otimes I_t\|_\infty = \|I_s \otimes A\|_\infty = \|A\|_\infty$ for any matrix A and any integers s and t. Then also note that since CRT_m is given (up to a permutation of the rows) by the above decomposition, we have that CRT_m^{-1} is given (up to a permutation of the rows) by the decomposition

$$\left(\mathsf{CRT}_r^{-1} \otimes I_{m_1}\right) \cdot T^{-1} \cdot \left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\right).$$

So we have

$$\begin{aligned}
\|\mathsf{CRT}_m^{-1}\|_\infty &= \| \left(\mathsf{CRT}_r^{-1} \otimes I_{m_1}\right) \cdot T^{-1} \cdot \left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\right) \|_\infty, \\
&\leq \|\mathsf{CRT}_r^{-1} \otimes I_{m_1}\|_\infty \cdot \|T^{-1}\|_\infty \cdot \|I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\|_\infty, \\
&= \|\mathsf{CRT}_r^{-1}\|_\infty \cdot \|T^{-1}\|_\infty \cdot \|\mathsf{DFT}_{m_1}^{-1}\|_\infty = \|\mathsf{CRT}_r^{-1}\|_\infty.
\end{aligned}$$

□

This result means that we can bound C_m for infinite families of values of m, by simply deducing a bound on C_r, where r is the product of all primes dividing m. For example, notice that $\mathsf{CRT}_r = (1)$ and hence $C_{2^e} = C_2 = 1$ for all values of e. Indeed it is relatively straightforward to determine the exact value of C_p for a prime p:

LEMMA 3.5. *If p is a prime then*

$$C_p = \frac{2 \cdot \sin(\pi/p)}{p \cdot (\cos(\pi/p) - 1)}.$$

Proof. [1] First note that it is a standard fact from algebra (by considering inverses of Vandermonde matrices, for example) that the entries of a row of the matrix CRT_p^{-1} are given by the coefficients of the polynomial

$$\frac{\Phi_p(X)}{\Phi_p'(\zeta_p) \cdot (X - \zeta_p)}, \tag{3.2}$$

where each row uses a different root of unity ζ_p. We then note that

$$\begin{aligned}
\Phi_p'(\zeta_p) &= (\zeta_p - \zeta_p^2) \cdot (\zeta_p - \zeta_p^3) \cdots (\zeta_p - \zeta_p^{p-1}) \\
&= \zeta_p^{-2} \cdot (1 - \zeta_p) \cdot (1 - \zeta_p^2) \cdots (1 - \zeta_p^{p-2}) \cdot \frac{(1 - \zeta_p^{p-1})}{(1 - 1/\zeta_p)} \\
&= \frac{\zeta_p^{-2} p}{1 - 1/\zeta_p} = \frac{p}{\zeta_p^2 - \zeta_p}.
\end{aligned}$$

Thus the coefficients of the polynomial in (3.2) are given by $\zeta_p \cdot (\zeta_p^r - 1)/p$ for $r = 1, \ldots, p-1$, where each row of our matrix is given by a different pth root ζ_p.

Thus to determine the infinity norm of CRT_p^{-1} we simply need to sum the absolute values of these coefficients for the first row, since all other rows are equal:

$$\begin{aligned}
C_p &= \sum_{r=1}^{p-1} |\zeta_p(\zeta_p^r - 1)/p| = \frac{1}{p} \sum_{r=1}^{p-1} \sqrt{2 - 2 \cdot \cos(2r\pi/p)} \\
&= \frac{1}{p} \sum_{r=1}^{p-1} 2 \cdot \sin(r\pi/p) = \frac{2 \cdot \sin(\pi/p)}{p \cdot (\cos(\pi/p) - 1)}
\end{aligned}$$

[1] This proof was provided to us by Robin Chapman.

□

In practice this result means that $C_p \approx 4/\pi \approx 1.2732$ for all $p \geq 11$.

If m is odd then we see that, subject to a permutation of the rows, the matrix CRT_{2m} and CRT_m are identical up to a multiple of -1 for every second column. Thus we have

$$C_{2m} = C_m \text{ for odd values of } m.$$

We find that $C_r \leq 8.6$ for square-free $r \leq 400$, which provides a relatively small upper bound on C_m for an infinite family of cyclotomic fields K. It appears that the size of C_m depends crucially on the number of prime factors of m. Thus it is an interesting open question to provide a tight upper bound on C_m. Indeed the growth in C_m seems to be closely related to the growth in the coefficients of the polynomial $\Phi_m(X)$, which also depends on the number of prime factors of m.

3.1.5. Application of the above bounds. An immediate consequence of Theorem 3.1 is to provide an upper bound on the value δ_∞ for cyclotomic number fields. Let $\alpha \in \mathcal{O}_K$. Then, by the standard inequalities between norms, that $\|\alpha\|_1 \leq N \cdot \|\alpha\|_\infty$. Thus we have, for $\alpha, \beta \in \mathcal{O}_K$,

$$\begin{aligned}\|\alpha \cdot \beta\|_\infty &\leq C_m \cdot \|\sigma(\alpha \cdot \beta)\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty \cdot \|\sigma(\beta)\|_\infty \\ &\leq C_m \cdot \|\alpha\|_1 \cdot \|\beta\|_1 \\ &\leq C_m \cdot N^2 \cdot \|\alpha\|_\infty \cdot \|\beta\|_\infty,\end{aligned}$$

i.e. $\delta_\infty \leq C_m \cdot N^2$. When m is a power of two, since $C_m = 1$ we find the bound $\delta_\infty \leq \phi(m)^2$; however, in this case it is known that $\delta_\infty = \phi(m)$, thus the above bound is not tight.

A more interesting application, for our purposes, is to bound the infinity norm in the polynomial embedding of the product of two elements which have been selected with a discrete Gaussian.

To demonstrate this result we first need to introduce the following standard tailbound:

LEMMA 3.6. *Let $c \geq 1$ and $C = c \cdot \exp(\frac{1-c^2}{2}) < 1$ then for any integer $N \geq 1$ and real $r > 0$ we have*

$$\Pr_{\mathbf{x} \leftarrow D_{\mathbb{Z}^N, s}}\left[\, \|\mathbf{x}\|_2 \geq c \cdot s \cdot \sqrt{\frac{N}{2 \cdot \pi}} \,\right] \leq C^N.$$

Note that this implies that

$$\Pr_{\mathbf{x} \leftarrow D_{\mathbb{Z}^N, s}}\left[\, \|\mathbf{x}\|_2 \geq 2 \cdot r \cdot \sqrt{N} \,\right] \leq 2^{-N},$$

where $r = s/\sqrt{2 \cdot \pi}$. If we therefore select $\alpha, \beta \in D_{\mathbb{Z}^N, s}$, consider them as elements of $\mathcal{O}_K$, we then have, with overwhelming probability that $\|\alpha\|_2, \|\beta\|_2 \leq 2 \cdot r \cdot \sqrt{N}$. We then apply the standard inequality between the 2- and the 1-norm to deduce $\|\alpha\|_1, \|\beta\|_1 \leq 2 \cdot r \cdot N$. We then have that

$$\begin{aligned}\|\alpha \cdot \beta\|_\infty &\leq C_m \cdot \|\sigma(\alpha \cdot \beta)\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty \cdot \|\sigma(\beta)\|_\infty \\ &\leq C_m \cdot \|\alpha\|_1 \cdot \|\beta\|_1 \\ &\leq 4 \cdot C_m \cdot r^2 \cdot N^2.\end{aligned}$$

3.2. Security, Parameter Choice and Performance

In this section we show that our concrete SHE scheme meets all the security requirements needed by our MPC protocol, i.e. that it is an admissible cryptosystem. On the way we derive parameter settings, and finally we present some implementation results for the core operations.

Recall that a cryptosystem is admissible if:

- It is IND-CPA secure.
- It includes function KeyGen^* as in Section 2.1.

- It is (B_{plain}, B_{rand}, C)-correct, where $B_{plain} = N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, $B_{rand} = d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, and where C, the set of functions we can evaluate on ciphertexts, contains all formulas evaluated in the protocol $\Pi_{\text{PREP}'}$ (including the identity function). Note that here we choose the values for B_{plain}, B_{rand} that correspond to the most efficient variant of the ZK proofs.

Recall in the expressions for B_{plain} and B_{rand} d is the dimension of the randomness space, i.e. $d = 3 \cdot N$, τ is a bound on the infinity norm of valid plaintexts, i.e. $p/2$; and ρ is a bound on the infinity norm of the randomness in validly generated ciphertexts, i.e. $\rho \approx 2 \cdot r \cdot \sqrt{N}$, by the tailbound of Lemma 3.6.

3.2.1. IND-CPA and KeyGen^*'s properties: We first turn to discussing security. Since our scheme is identical (bar the distributed decryption functionality) to that of [BV11], security can be reduced to the hardness of the following problem.

DEFINITION 3.7 (PLWE Assumption). For all $\mathsf{sec} \in \mathbb{N}$, let $f(X) = f_{\mathsf{sec}}(X) \in \mathbb{Z}[X]$ be a polynomial of degree $N = N(\mathsf{sec})$, let $q = q(\mathsf{sec}) \in \mathbb{Z}$ be a prime integer, let $R = \mathbb{Z}[X]/f(X)$ and $R = R/qR$, and let χ denote a distribution over the ring R. The *polynomial LWE assumption* $\mathsf{PLWE}_{f,q,\chi}$ states that for any $l = \text{poly}(\mathsf{sec})$ it holds that

$$\{(a_i, a_i \cdot s + e_i)\}_{i \in [l]} \approx \{(a_i, u_i)\}_{i \in [l]}$$

where s is sampled from the distribution χ, and a_i, u_i are uniformly random in R_q. We require computational indistinguishability to hold given only l samples, for some $l = \text{poly}(\mathsf{sec})$.

In particular our scheme is semantically secure if the $\mathsf{PLWE}_{\Phi_m(X),q_0,\mathrm{D}_\rho^N(s)}$-problem is hard. The hardness of the same problem also implies that the output from $\mathsf{KeyGen}()$ is computationally indistinguishable from that of $\mathsf{KeyGen}^*()$.

Thus, our first task is to derive relationships between the parameters so as to ensure the first two properties of being admissible are satisfied, i.e. the PLWE problem is actually hard to solve. The basic parameters of our scheme are the degree of the associated number field $N = \phi(m)$, the standard deviation r of the used Gaussian distribution, and the modulus q. We first turn to estimating r; we do this by using the "standard" analysis of the underlying LWE problem.

We first ensure that r is chosen to avoid combinatorial style attacks. Consider the underlying LWE problem as being given by $\mathbf{s} \cdot A + \mathbf{e} = \mathbf{v}$, where $\mathbf{e}$ is the LWE error vector, and A is a random $N \times t$ matrix over $\mathbb{F}_q$. In [AG11] the authors present a combinatorial attack which breaks LWE in time $2^{O(\|\mathbf{e}\|_\infty^2)}$ with high probability. Since $\mathbf{e}$ is chosen by the discrete Gaussian with standard deviation r, if we pick r large enough then this attack should be prevented. Thus choosing r such that $r > 3.2$ will ensure that r is large enough to avoid combinatorial attacks, i.e. $s \geq 8$.

We now turn to the distinguishing problem, namely: Given $\mathbf{v}$, can we determine whether it arises from an LWE sample, or from a uniform sample? We determine a lower bound on N. The natural "attack" against the decision LWE problem is to first find a short vector $\mathbf{w}$ in the dual lattice $\Lambda_q(A^\mathsf{T})^*$ and then check whether $\mathbf{w} \cdot \mathbf{v}^\mathsf{T}$ is close to an integer. If it is then the input vector is an LWE sample, if not it is random. Thus to ensure security, following the argument in [MR08][Section 5.4.1], we require

$$r \geq \frac{1.5}{\|\mathbf{w}\|_2}.$$

Following the work of [GN08] we can estimate, for $t \gg N$, the size of the output of a lattice reduction algorithm operating on the lattice $\Lambda_q(A^\mathsf{T})^*$. In particular if the algorithm tries to find a vector with root Hermite factor δ (thus δ measures the difficulty in breaking the underlying SHE system, typically one may select $\delta \approx 1.005$, but see later for other choices) then we expect to find a vector $\mathbf{w}$ of size

$$\frac{1}{q} \min(q, \delta^t \cdot q^{N/t}).$$

Following the analysis of [MR08] the above quantity is minimised when we select $t = t' := \sqrt{N\log(q)/\log(\delta)}$. This leads us to the deduce the lower bound

$$r \geq 1.5 \cdot \max(1\ ,\ \delta^{-t'} \cdot q^{1-N/t'}).$$

3.2.2. Noise of a Clean Ciphertext: We now turn to determining the bound, in the infinity norm, of the value obtained in decrypting valid ciphertexts. Consider what happens when we decrypt a clean ciphertext, encrypted via $(\mathbf{c}_0, \mathbf{c}_1) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$, with $\mathbf{r} = (\mathbf{u}, \mathbf{v}, \mathbf{w})$. This looks like a PLWE sample $(\mathbf{c}_1, \mathbf{c}_0)$ where the "noise" term, for a validly generated clean ciphertext, is given by

$$\begin{aligned}\mathbf{t} &= \mathbf{c}_0 - \mathbf{s}\cdot\mathbf{c}_1\\ &= \mathbf{x} + p\cdot(\mathbf{e}\cdot\mathbf{v} + \mathbf{w} + \mathbf{s}\cdot\mathbf{u})\end{aligned}$$

By our estimates in Section 3.1.5 we can bound the infinity norm of $\mathbf{t}$ by

$$\|\mathbf{t}\|_\infty \leq \frac{p}{2} + p\cdot\left(4\cdot C_m\cdot r^2\cdot N^2 + 2\cdot\sqrt{N}\cdot r + 4\cdot C_m\cdot r^2\cdot N^2\right) =: Y.$$

3.2.3. (B_{plain}, B_{rand}, C)-correctness: Whilst IND-CPA is about security in relation to validly created ciphertexts, our distributed decryption functionality must be secure even when some ciphertexts are not completely valid. This was the reason we introduced the notion of (B_{plain}, B_{rand}, C)-correctness. We need to pick B_{plain} and B_{rand} so that $B_{plain} \geq N\cdot\tau\cdot\mathsf{sec}^2\cdot 2^{\mathsf{sec}/2+8}$ and $B_{rand} \geq d\cdot\rho\cdot\mathsf{sec}^2\cdot 2^{\mathsf{sec}/2+8}$. Since $B_{plain} \ll B_{rand}$ we estimate the noise term associated to such a "clean" ciphertext will be bounded by $Y' = (B_{rand}/\rho)^2\cdot Y = 9\cdot N^2\cdot \mathrm{sec}^4\cdot 2^{\mathsf{sec}+16}\cdot Y$. In our MPC protocol we only need to be able to evaluate functions of the form

$$(x_1 + \cdots + x_n)\cdot(y_1 + \cdots + y_n) + (z_1 + \cdots + z_n).$$

We can, via the results in Section 3.1.5, crudely estimate the size of B, from Section 2.5, that are needed to ensure valid decryption. Our crude (over-) estimate therefore comes out as

$$\begin{aligned}B &\leq \delta_\infty\cdot(n\cdot Y')\cdot(n\cdot Y') + (n\cdot Y')\\ &\leq C_m\cdot N^2\cdot n^2\cdot Y'^2 + n\cdot Y'\\ &\leq C_m\cdot N^2\cdot n^2\cdot c_{\mathsf{sec}}^2\cdot Y^2 + n\cdot c_{\mathsf{sec}}\cdot Y =: Z\end{aligned}$$

where $c_{\mathsf{sec}} = 9\cdot N^2\cdot\mathsf{sec}^4\cdot 2^{\mathsf{sec}+8}$. We take Z as the bound, which we then need to scale by $1+2^{\mathsf{sec}}$ to ensure we have sufficient space to enable the distributed decryption algorithm. Hence, the value of q needs to be selected so that $Z\cdot(1+2^{\mathsf{sec}}) < q/2$.

In summary, we need to choose parameters such that

$$\begin{aligned}q &> 2\cdot Z\cdot(1+2^{\mathsf{sec}}),\\ r &> \max\left\{3.2, 1.5\cdot\delta^{-t'}\cdot q^{1-N/t'}\right\},\end{aligned}$$

where sec is the statistical security parameter, δ is a measure of how hard it is to break the underlying SHE scheme, and $t' = \sqrt{N\log(q)/\log(\delta)}$. This leads to a degree of circularity in the dependency of the parameters, but valid parameter sets can be found by a simple search technique.

3.2.4. Specific Parameter Sets: To determine parameters for fixed values of $(\mathbb{F}_{p^k})^s$ and n we proceed as follows. There are two interesting cases; one where p is fixed (i.e. $p = 2$) and one where we only care that p is larger than some bound (i.e. $p > 2^{32}$, or $p > 2^{64}$). The latter case of $p > 2^{64}$ is more interesting as numbers of such size can be used more readily in applications since we can simulate integer arithmetic without overflow with such numbers. In addition, using such a value of p means we do not need to repeat our ZKPoKs, or replicate the MACs so as to get a cheating probability of less than 2^{-40}.

Our method in all cases is to first fix p, n, sec and δ, and then search using the above inequalities for (rough) values of q and N which satisfy the inequalities above. We then search

for exact values of p and N which satisfy our functional requirements on p (i.e. fixed or greater than some bound) plus N larger than the bound above, such that N is the degree of $F(X) = \Phi_m(X)$ and $F(X)$ splits into at least s factors of degree divisible by k over $\mathbb{F}_p$.

Given this precise value for N, we then return to the above inequalities to find exact values of q and r. In all our examples below we pick $n = 3$, $\mathsf{sec} = 40$, and $\delta = 1.0052$.

Example 1: We first look at $p > 2^{32}$. Our first (approximate) search reveals we need $N > 14300$, $q \approx 2^{430}$ and $r = 3.2$ (assuming $C_m \leq 2$). We then try to find an optimal value of N; this is done by taking increasing primes $p > 2^{32}$ and factoring $p-1$. The factors of $p-1$ correspond to values of m such that $\Phi_m(X)$ factors into $\phi(m)$ factors modulo p. So we want to find a p such that $p-1$ is divisible by an m, so that $N = \phi(m) > 14300$. A quick search reveals candidates of

$$(p, N, m) = (2^{32} + 32043,\ 14656,\ 14657).$$

Picking m in this way will maximise the value of $s = n$, and hence allow us to perform more operations in parallel. In addition since m is prime we know, by Lemma 3.5, that $C_m \approx 1.2732$, thus justifying our assumption in deriving the bounds of $C_m \leq 2$.

Selecting m to be the prime 14657 in addition allows us to evaluate $s = p - 1 = 14656$ runs of the triple production algorithm in parallel. The message expansion factor, given we require $N \cdot \log_2(q)$ bits to represent N elements in $\mathbb{F}_p$, is given by

$$\frac{N \cdot \log_2(q)}{N \cdot \log_2(p)} = \frac{\log_2(q)}{\log_2(p)} = \frac{430}{32} \approx 13.437.$$

Example 2: Performing the same analysis for a $p > 2^{64}$, our first naive search of parameters reveal we need an $n \approx 16700$ and $q \approx 2^{500}$. We then search for specific parameters and find $p = 2^{64} + 4867$ is pretty near to optimum, which results in a prime value of m of 16729. We find the expansion factor is given by

$$\frac{\log_2(q)}{\log_2(p)} = \frac{500}{64} \approx 7.81.$$

Example 3: We now look at the case $p = 2$ and $k = 8$, i.e. we are looking for parameters which would allow us to compute AES circuits in parallel; or more generally circuits over $\mathbb{F}_{2^8}$. Our first approximate search reveals that we need $N > 12300$, $q \approx 2^{370}$ and $r = 3.2$. So we now need to determine a value m such that

$$N = \phi(m) > 12100 \text{ and } 2^d \equiv 1 \pmod{m} \text{ and } d \equiv 0 \pmod{8}.$$

A quick search reveals candidates of

$$(m, N) = (17425, 12800)$$

since $\Phi_{17425}(X)$ factors into $s = 320$ factors of degree $d = 40$ modulo 2. Thus using this value of m we are able to work with $s = 320$ elements of $\mathbb{F}_{2^8}$ in parallel. The message expansion factor, given we require $N \cdot \log_2(q)$ bits to represent 320 elements in $\mathbb{F}_{2^8}$, is given by

$$\frac{N \cdot \log_2(q)}{8 \cdot s} = \frac{d \cdot 370}{8} = 1850.0$$

For this value of m we find $C_{17425} \approx 9.414$.

We present the following run-times we have achieved. We time the operations for encrypting and decrypting clean ciphertexts, the time to homomorphically compute $(c_x \boxtimes c_y) \boxplus c_z$, plus the time to decrypt the said result. The times are given in seconds, and in brackets we present the amortized time per finite field element. All timings were performed on an Intel Core-2 6420 running at 2.13 GHz.

Example	Enc Time (s)	Dec (Clean) Time (s)	$(c_x \boxtimes c_y) \boxplus c_z$ Time (s)	$\mathsf{Dec}_{\mathsf{sk}}((c_x \boxtimes c_y) \boxplus c_z)$ Time (s)
1	0.72 {0.00005}	0.35 {0.00002}	1.43 {0.0001}	0.72 {0.00005}
2	3.13 {0.00019}	1.54 {0.00009}	6.27 {0.0004}	3.15 {0.00018}
3	1.26 {0.00394}	0.60 {0.00188}	2.46 {0.0077}	1.23 {0.00384}

3.2.5. Estimating Equivalent Symmetric Security Level: The above examples were computed using the root Hermite factor of $\delta = 1.005$. Mapping this "hardness" parameter for the underlying lattice problem to a specific symmetric security level (i.e. 80-bit security, or 128-bit security) is a bit of a "black art" at present.

In [CN11] the authors derive an estimate for the block size needed to obtain a given root Hermite factor, assuming an efficient BKZ lattice reduction algorithm is used. They then provide estimates of the run-time needed for a specific enumeration using this block size. As an example of their analysis: they estimate that a block size of 286 is needed to obtain a root Hermite factor of $\delta = 1.005$. Then they estimate that the run-time needed to perform the enumeration in a projected lattice of such dimension (the key sub-procedure of the BKZ algorithm) takes time roughly between 2^{80} and 2^{175} operations. Thus a value of $\delta = 1.005$ can be considered secure; however, their estimates are not precise enough to produce parameters associated with a given symmetric security level.

In [LP11] the authors take a different approach and simply extrapolate run-times for the NTL implementation of BKZ. By looking at various LWE instances, they derive the following equation linking the expected run-time of a distinguishing attack and the root Hermite factor

$$\log_2 T = \frac{1.8}{\log_2 \delta} - 110.$$

The problem with this approach is that NTL's implementation of BKZ is very old, and hence is not state-of-the-art; on the other hand we are able to derive a direct linkage between δ and $\log_2 T$. Using this equation we find the following equivalences:

$\log_2 T$	80	100	128	196	256
δ	1.0066	1.0059	1.0052	1.0041	1.0034

Using these estimates for δ we re-run the above analysis to find approximate values for N and q in our three example applications, again assuming $n = 3$ and $\mathsf{sec} = 40$.

	$\mathbb{F}_p : p > 2^{32}$		$\mathbb{F}_p : p > 2^{64}$		$\mathbb{F}_{2^8}$	
	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$
$\delta = 1.0066$	11300	430	12900	490	9500	360
$\delta = 1.0059$	12600	430	14700	500	10900	370
$\delta = 1.0052$	14300	430	16700	500	12300	370
$\delta = 1.0041$	18600	440	21100	500	15600	370
$\delta = 1.0034$	22400	440	25500	500	18800	370

As can be seen the security parameter has only marginal impact on $\log_2 q$, and results in a doubling of the size of N as we increase from a security level of 80 bits to 256 bits. As a comparison if, for security level 128 bits, i.e. $\delta = 1.0052$, we increase the value of sec from 40 to 80 we find the following parameter sizes:

	$\mathbb{F}_p : p > 2^{32}$		$\mathbb{F}_p : p > 2^{64}$		$\mathbb{F}_{2^8}$	
	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$
$\delta = 1.0052$	18700	560	21000	630	16700	500

3.3. Running the Online Phase with Small Fields

In this section we face the scenario where one wants to run an online phase with error probability $2^{-\mathsf{sec}}$, but $\log p^k$ is much smaller than sec.

When we consider how to solve this problem, we will at first ignore Step 1 in the **Multiply** stage of the online protocol, where one triple is "sacrificed" to check another, as this step could

be done as part of the preprocessing. Nevertheless, we do not want to ignore the fact that this step will have a large error probability $1/p^k$. We could solve this by sacrificing $D = \lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ triples instead of one, but we can do much better, and this is described below in Section "A small sacrifice" below.

Going back to the online phase, we can compensate for the fact that $\log p^k$ is much smaller than sec by setting up the preprocessing so it can work over an extension field K of $\mathbb{F}_{p^k}$ of degree $D = \lceil \frac{\mathsf{sec}}{\log p^k} \rceil$, i.e. an element in K is represented by $\lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ elements from $\mathbb{F}_{p^k}$. All MAC keys and MACs will be generated in K whereas all values to be computed on will still be in $\mathbb{F}_{p^k}$. The preprocessing can ensure this because the ZK proof can already force a prover to choose plaintexts that decode to elements in a subfield of K.

Then the error probabilities in the proof of the online phase that were $1/p^k$ before will now be $1/|K| \leq 2^{-\mathsf{sec}}$. The computational complexity of the online phase now has an overhead of $D \log D \log \log D$ and the overhead for storage and communication is just D.

It is also possible to get error probability $2^{-\mathsf{sec}}$ while having the preprocessing work only over $\mathbb{F}_{p^k}$. Here the overhead will be larger, namely $D^2 \log D \log \log D$, but this may be the best option when D is not very large. The idea is to authenticate by doing D MACs in parallel over $\mathbb{F}_{p^k}$ for every authenticated value, using D independent keys.

3.3.1. A Small Sacrifice. In this section we describe an advanced method to check the multiplicative relation on triples $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, where $a, b, c \in \mathbb{F}_{p^k}$. The aim is to decrease the (amortized) number of triples to sacrifice per check. Our approach resembles a technique introduced by Ben-Sasson et al. in [**BSFO12**] and also another by Cramer et al. in [**CDP12**].

The first step in our construction is to consider a batch of $t+1$ triples $\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle$ for $i = 1, \ldots, t+1$ at once. There are two main ideas in the construction: the first one is to interpolate the values and get polynomials $A, B, C \in \mathbb{F}_{p^k}[X]$ such that $A(i) = a_i$, $B(i) = b_i$, $C(i) = c_i$; if the triples were correctly generated, one would expect $A(x)B(x) = C(x)$ for all x. The second idea is to think of A, B, C as polynomials over a field extension K of $\mathbb{F}_{p^k}$, so that one can check the expected multiplicative relation evaluating A, B, C at a random element $z \in K$; the probability that the check passes even if some of the triples did not satisfy the relation is inversely proportional to the size of K. We now present the full construction.

- Let $\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle$, $i = 1, \ldots, t+1$, be a batch of triples to check.
- Think of the values $a_1, \ldots, a_{t+1}$ (resp. $b_1, \ldots, b_{t+1}$) as $t+1$ evaluations over $\mathbb{F}_{p^k}$ of a unique polynomial $A \in \mathbb{F}_{p^k}[X]$ (resp. $B \in \mathbb{F}_{p^k}[X]$) of degree t. Concretely, define the polynomial A (resp. B) such that $A(i) = a_i$ (resp. $B(i) = b_i$). Since the coefficients of A (resp. B) can be computed as a linear combination of the a_i's (resp. b_i's), the players can compute representations of such coefficients by local computation.
- Players can compute $\langle a_{t+2} \rangle, \ldots, \langle a_{2t+1} \rangle$ such that $A(i) = a_i$, again by local computation, since evaluating a polynomial is a linear operation.
- Players can engage in the multiplication step of the online phase with input $\langle a_i \rangle, \langle b_i \rangle$, and get $\langle c_i \rangle$ (hopefully $c_i = a_i b_i$) for $i = t+2, \ldots, 2t+1$. Notice that players call the multiplication step t times here, so they sacrifice t triples.
- Using only linear computation players can now compute representations of coefficients of the unique polynomial $C \in \mathbb{F}_{p^k}[X]$ of degree $2t$ such that $C(i) = c_i$ for $i = 1, \ldots, 2t+1$.
- Let K be a field extension of $\mathbb{F}_{p^k}$ of degree D. It is possible to think of A, B, C as polynomials over K, by embedding the coefficients via the natural map $\mathbb{F}_{p^k} \longrightarrow K$. Players now evaluate representations for $A(z)B(z)$, and $C(z)$, where z is a public random element in K, and check if $A(z)B(z) = C(z)$ by outputting $A(z)B(z) - C(z)$ and checking if the result is zero. This check can be repeated a number of times in order to lower the error probability. If the check passed for all repetitions, players consider the original triples to be valid; otherwise, they discard the triples and start again with fresh triples.

Notice that in order to compute $A(z)B(z)$ and $C(z)$, players need to compute at most D^2 multiplications over $\mathbb{F}_{p^k}$, since $A(z)B(z)$ can be computed by multiplying a $D \times D$ matrix (dependent on $A(z)$) with the vector $B(z)$ (over K, multiplication by a fixed element is an endomorphism of K as a $\mathbb{F}_{p^k}$-vector space). Notice also that we may use the original method (presented in the online phase) of sacrificing more than one triple per multiplication to get any desired error probability for the multiplications over $\mathbb{F}_{p^k}$. We analyse below the error probability we must require.

For the analysis of the construction, observe that if the multiplicative relation was satisfied by all the original triples, the polynomials AB and C are equal, so the final test passes. If the triples do not satisfy the relation, then the polynomials AB and C are different, but since they are both of degree at most $2t$, they can agree in at most $2t$ points. Therefore, if z is a root of $AB - C$, then the test passes, and uniform elements in K are roots of $AB - C$ with probability at most $2t/|K|$. If z is not a root of $AB - C$, the test passes only if the multiplication $A(z)B(z)$ happens to give the correct result; hence, provided that we make sure this happens with probability at most $2t/|K|$ (by sacrificing enough triples in the process), the error probability of the construction is bounded by $2t/|K|$ for a single run of the test. In order to get negligible error probability we repeat this phase enough times.

An important fact to notice is that in this construction we need $2t+1 \leq \mathbb{F}_{p^k}$, since otherwise there are not enough elements to evaluate the polynomials. This restriction can be circumvented by applying the above construction with $\mathbb{F}_{p^k}$ replaced by an extension $\mathbb{F}_{p^{k'}}$ with the required property.

Asymptotically, we see that as we increase the number $t + 1$ of triples checked, we always need to sacrifice t triples, and in addition we need to check the multiplication(s) in K. If we assume that we want to hit the desired error probability with just one iteration of the test, we have $2^{-\mathsf{sec}} = 2t/|K|$ from which we get $\log |K| = \mathsf{sec} + \log 2t$. The degree of the extension to K is $\log |K| / \log p^k$, and the number of basic secure multiplications we need is at most the square of this number, which is $(\mathsf{sec} + \log 2t)^2/(\log p^k)^2$. For each of these, we need error essentially $2^{-\mathsf{sec}}$, so the number of triples we need, say m, satisfies $2^{-\mathsf{sec}} = (1/p^k)^m$, so we get $m = \mathsf{sec}/\log p^k$. This in total grows only poly-logarithmically with t, so we conclude that for a given desired error probability, the number of triples we need to sacrifice to check $t + 1$ triples is $O(t + \mathit{polylog}(t))$.

3.3.2. Comparing the Two Approaches: A Concrete Example. Here we compare the above two approaches for checking triples. Suppose $p = 2$ and $k = 8$, so $\mathbb{F}_{p^k} = \mathbb{F}_{2^8}$. Suppose there are also $t + 1 = 128$ triples to check with security level of 2^{-80}.

Using the latter approach, with $K = \mathbb{F}_{2^{16}}$, we need to sacrifice $t = 127$ triples to generate $\langle c_{t+2} \rangle, \ldots, \langle c_{2t+1} \rangle$; moreover, we need to perform 4 secure multiplications to check if $A(z)B(z) = C(z)$, since K is a vector space of dimension 2 over $\mathbb{F}_{2^8}$. In order for the multiplications to be secure enough, we need them to be correct up to error probability $(2 \cdot 127)/2^{16} \approx 2^{-8}$ for the entire multiplication $A(z)B(z)$. This will be the case if for each of the 4 small multiplications we use 3 triples: one to do the actual multiplication and two to check the first one. This gives a total error probability of at most $4 \cdot 2^{-16} \leq 2^{-8}$. Since one run of the test leads to an error probability of $\approx 2^{-8}$, we need 10 runs to decrease the error probability to 2^{-80}. Therefore, the total number of triples to sacrifice is $128 + 4 \cdot 3 \cdot 10 = 248$, while with the original approach the number of triples to sacrifice would have been $128 \cdot 10 = 1280$.

CHAPTER 4

SPDZ2 – Overview

4.1. Practical Challenges

For many decades multiparty computation was a predominantly theoretic endeavour in cryptography, but in recent years interest has arisen on the practical side. This has resulted in various improvements on implementations which are becoming more applicable to practical situations. A key aspect of this transformation from theory to practice is the adaptation of theoretical protocols by applying implementation techniques so as to significantly improve performance, whilst not sacrificing the level of security required by real world applications. We follow this modern and practically oriented trend.

Early applied work on MPC focused on the case of protocols secure against passive adversaries, both in the case of two-party protocols based on Yao circuits [**MNPS04**] and that of many-party protocols, based on secret sharing techniques [**BLW08**, **DGKN09**, **SIM**]. Only in recent years has work shifted to achieve active security [**KSS12**, **LPS08**, **PSSW09**], which appears to come at vastly increased cost when dealing with more than two players. On the other hand, in real applications, active security may be more stringent than one would actually require. In [**AL07**, **AL10**] Aumann and Lindell introduced the notion of covert security; in this security model an adversary who deviates from the protocol is detected with high (but not necessarily overwhelming) probability, say 90%, which still translates into an incentive for the adversary to behave honestly. In contrast, active security achieves the same effect, but the adversary can only succeed in cheating with negligible probability. There is a strong case to be made (see [**AL07**, **AL10**]) that covert security is a "good enough" security level for practical application; thus in this work we focus on covert security, but we also provide solutions with active security.

4.2. Our Approach

As our starting point we take the SPDZ protocol of [**DPSZ12**]. SPDZ is secure against active static adversaries in the standard model, and tolerates corruption of $n-1$ of the n parties. The SPDZ protocol follows the preprocessing model: in an offline phase some shared randomness is generated, but neither the function to be computed nor the inputs need be known; in an online phase the actual secure computation is performed. One of the main advantages of the SPDZ protocol is that the performance of the online phase scales linearly with the number of players, and the basic operations are almost as cheap as those used in the passively secure protocols based on Shamir secret sharing. Thus, it offers the possibility of being both more flexible and more secure than Shamir-based protocols, while still maintaining low computational cost.

In [**DKL+12**] an implementation report is given on an adaptation of the SPDZ protocol in the random oracle model, including performance figures for the offline and online phases for both an actively secure variant and a covertly secure variant. The implementation is over a finite field of characteristic two, since the focus is on providing a benchmark for evaluation of the AES circuit (a common benchmark application in MPC [**PSSW09**, **DK10**]).

4.3. Introduction to SPDZ2

We present a number of contributions which extend even further the ability of the SPDZ protocol to deal with the type of application that is likely to be seen in practice. We support these improvements with details of an actual implementation.

Our contributions come in two flavours. First, we present a number of improvements and extensions to the basic underlying SPDZ protocol. These protocol improvements are supported with associated security models and proofs. Second, we focus on the implementation layer, and bring in standard techniques from applied cryptography adapted for MPC.

In more detail, our protocol enhancements, in descending order of importance, are as follows:

(1) In the online phase of the original SPDZ protocol the parties are required to reveal their shares of a global MAC key in order to verify that the computation has been performed correctly. This is a major problem in practical applications since it means that unrevealed secret-shared data cannot be re-used in later applications. Our protocol adopts a method to accomplish the same task without needing to open the underlying MAC key. This means players can now go on computing on any secret-shared data they have, so general reactive computation can be performed, rather than just secure function evaluation. A further advantage of this technique is that some of the verification (the so-called "sacrificing" step) can be moved into the offline phase, providing additional performance improvements in the online phase.

(2) In the original SPDZ protocol [DKL+12, DPSZ12] the authors assume a "magic" key generation phase for the production of the distributed Somewhat Homomorphic Encryption (SHE) scheme public/private keys required by the offline phase. The authors claim this can be accomplished using standard generic MPC techniques, which are expensive. We present a key generation protocol for the BGV [BGV12] SHE scheme, which is secure against covert adversaries. In addition, we generate a "full" BGV key which supports the modulus switching and key switching used in [GHS12b]. This new sub-protocol may be of independent interest in other applications which require distributed decryption in an SHE/FHE scheme.

(3) In [DKL+12] the modification to covert security was essentially ad-hoc, and resulted in a very weak form of covert security. In addition no security proofs or model were given to justify the claimed security. We present a completely different approach to achieving covert security, and provide an extensive security model and full proofs for the modified offline phase (and the key generation protocol mentioned above).

(4) We introduce a new approach to obtain full active security in the offline phase. In [DPSZ12], active security was obtained by the use of specially designed ZKPoKs. We present a different technique, based on a method used in [NNOB12], which has running time similar to the ZKPoK approach in [DPSZ12], but it allows much stronger guarantees on the ciphertexts produced by corrupt players: the gap between the size of the "noise" that honest players put into ciphertexts and what we can force corrupt players to use was exponential in the security parameter in [DPSZ12], and is essentially linear in our solution. This yields smaller parameters for the underlying cryptosystem and makes the protocol more efficient.

It is important to understand that by combining these contributions in different ways, one can obtain two general MPC protocols: first, since our new online phase still achieves full active security, it can be combined with our new approach to active security in the offline phase. This results in a protocol that is "syntactically similar" to the one from [DPSZ12]: it has full active security assuming access to a functionality for key generation. However, it has better performance and enhanced functionality, compared to [DPSZ12], in that it can securely compute reactive functionalities. Second, we can combine our covertly secure protocols for key generation and the offline phase with the online phase to get a protocol that has covert security throughout (and in this case it does not assume key generation).

Our covert solutions all make use of the same technique to move from passive to covert security, while avoiding the computational cost of performing zero-knowledge proofs. In [DKL+12] covert security is obtained by only checking a fraction of the resulting proofs, which results in a weak notion of covert security (the probability of a cheater being detected cannot be made too large). We adopt a different approach, akin to the cut-and-choose paradigm. We require parties

to commit to random seeds for a number of runs of a given sub-protocol, then all the runs are executed in parallel, finally all bar one of the runs are "opened" by the players revealing their random seeds. If all opened runs are shown to have been performed correctly then the players assume that the single un-opened run is also correctly executed.

Note that since these checks take place in the offline phase where the inputs are not yet available, we obtain the strongest flavour of covert security defined in [AL07], where the adversary learns nothing new if he decides to try to cheat and is caught.

A pleasing side-effect of the replacement of zero-knowledge proofs with our custom mechanism to obtain covert security is that the offline phase can be run in much smaller "batches". In [DKL+12, DPSZ12] the need to amortise the cost of the expensive zero-knowledge proofs meant that the players, on each iteration of the offline protocol, executed a large computation, which produced a large number of multiplication triples [Bea91] (in the millions). With our new technique there is no need to amortise executions as much, and so short runs of the offline phase can be executed if so desired, producing only a few thousand triples per run.

Our second flavour of improvements, at the implementation layer, is more mundane.

(1) We focus on the more practical application scenario of developing MPC where the base arithmetic domain is a finite field of characteristic $p > 2$. The reader should think $p \approx 2^{32}, 2^{64}, 2^{128}$ and the type of operations envisaged in [CS10, DFK+06] etc. For such applications we can move a lot of computation into the SPDZ offline phase, and we present the necessary modifications to do so.
(2) Parameters for the underlying BGV scheme are chosen using the analysis used in [GHS12b] rather than the approach used in [DPSZ12]. In addition we pick specific parameters which enable us to optimise for our application to SPDZ with the choices of p above.
(3) We assume the random oracle model throughout: this simplifies a number of the sub-procedures in [DPSZ12], especially related to aspects of the protocol which require commitments.
(4) The underlying arithmetic is implemented using Montgomery arithmetic [Mon85], in contrast to earlier work, which generally used standard libraries (such as NTL) to provide these operations.
(5) The removal of the need to use libraries such as NTL means the entire protocol can be implemented in a multi-threaded manner. Thus, it can make use of the multiple cores on modern microprocessors.

4.4. SPDZ Overview and the Room Left for Improvements

We now present the main components of the SPDZ protocol; in this section, unless otherwise specified, we are simply recapping on prior work.

For the notation and the basic properties, we refer to Section 2.3.1. We also point out that from now on we specialise the computation on $\mathbb{F}_p$ rather than on $\mathbb{F}_{p^k}$ (in contrast to the previous chapters).

During the protocol, various values which are $\langle\cdot\rangle$-shared are "partially opened", i.e. the associated values a_i are revealed, but not the associated shares of the MAC. Note that linear operations (addition and scalar multiplication) can be performed on the $\langle\cdot\rangle$-sharings with no interaction required. Computing multiplications, however, is not straightforward.

The goal of the offline phase is to produce a set of "multiplication triples", which allow players to compute products. These are a list of sets of three $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ such that $c = a \cdot b$. Here, we extend the offline phase to also produce "square pairs", i.e. a list of pairs of $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle\}$ such that $b = a^2$, and "shared bits", i.e. a list of single shares $\langle a\rangle$ such that $a \in \{0, 1\}$.

In the online phase these lists are consumed as MPC operations are performed. The reason for the introduction of square pairs is that squares can then be computed more efficiently, as follows. To square the sharing $\langle x\rangle$, players take a square pair $\{\langle a\rangle, \langle b\rangle\}$ and partially open

$\langle x \rangle - \langle a \rangle$ to obtain ε. They then compute the sharing of $z = x^2$ from $\langle z \rangle \leftarrow \langle b \rangle + 2 \cdot \varepsilon \cdot \langle x \rangle - \varepsilon^2$. The "shared bits" are useful in computing high level operation such as comparison, bit-decomposition, fixed and floating point operations as in [**ABZS13**, **CS10**, **DFK+06**].

The offline phase produces the triples in the following way, by using a Somewhat Homomorphic Encryption (SHE) scheme, which encrypts messages in $\mathbb{F}_p$, supports distributed decryption, and allows computation of circuits of multiplicative depth one on encrypted data. To generate a multiplication triple, each player P_i generates encryptions of random values a_i and b_i (their shares of a and b). Using the multiplicative property of the SHE scheme an encryption of $c = (a_1 + \cdots + a_n) \cdot (b_1 + \cdots + b_n)$ is produced. The players then use the distributed decryption protocol to obtain sharings of c. The shares of the MACs on a, b and c needed to complete the $\langle \cdot \rangle$-sharing are produced in the same manner. Similar operations are performed to produce square pairs and shared bits. Clearly the above (vague) outline needs to be fleshed out to ensure the required covert security level. Moreover, in practice players generate many triples/pairs/shared-bits at once using the SIMD nature of the BGV SHE scheme.

CHAPTER 5

SPDZ2 – Preprocessing and Online

5.1. BGV

We now present an overview of the BGV scheme as required by the offline phase. This is only a sketch: the reader is referred to [**BGV12**, **GHS12a**, **GHS12b**] for more details, as our goal here is to present enough detail to explain the key generation protocol.

5.1.1. Preliminaries.

Underlying Algebra: Let $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/\Phi_m(X)$, for the m-th cyclotomic polynomial $\Phi_m(X)$, where m is a parameter to be determined later (see Section 6.1).

Note that q may not necessarily be prime. Let $R = \mathbb{Z}[X]/\Phi_m(X)$. Set message space to be R_p for a prime p of approximately 32, 64 or 128-bits in length, and set the ciphertext space to be either $R_{q_0}^2$ or $R_{q_1}^2$, for one of two moduli q_0 and q_1. Specifically, we choose $R = \mathbb{Z}[X]/(X^{m/2}+1)$ for m a power of two, and $p = 1 \pmod m$. By picking m and p this way the message space R_p offers $m/2$-fold SIMD parallelism, i.e. $R_p \cong \mathbb{F}_p^{m/2}$. In addition, this implies that the ring constant c_m from [**DPSZ12**, **GHS12b**] is equal to one.

We wish to generate a public key for a leveled BGV scheme for which n players each hold a share, which is itself a "standard" BGV secret key. Dealing with circuits of multiplicative depth at most one, there is only need for two levels in the moduli chain: $q_0 = p_0$ and $q_1 = p_0 \cdot p_1$. The modulus p_1 also plays the role of P in [**GHS12b**] for the $\mathsf{SwitchKey}$ operation. The value p_1 must be chosen so that $p_1 \equiv 1 \pmod p$.

Random Values: Each player is assumed to have a secure entropy source. In practice we take this to be `/dev/urandom`, which is a non-blocking entropy source found on Unix-like operating systems. This is not a "true" entropy source, being non-blocking, but provides a practical balance between entropy production and performance for our purposes. In what follows we model this source via a procedure $s \leftarrow \mathsf{Seed}()$, which generates a new seed from this source of entropy. Calling this function sets the players global variable cnt to zero. Then, every time a player generates a new random value in a protocol, the value is constructed by calling $\mathsf{PRF}_s(\mathsf{cnt})$, for some pseudo-random function PRF, and then incrementing cnt. In practice, we use AES under the key s with message cnt to implement PRF.

The point of this method for generating random values is that these values can then be verified to have been generated honestly by revealing s in the future, recomputing all the randomness used by a player, and verifying that his output is consistent with s.

From the basic PRF, we define the following "induced" pseudo-random number generators, which generate elements according to the following distributions but seeded by the seed s:

- $\mathcal{HWT}_s(h, n)$: This generates a vector of length n with elements chosen uniformly from $\{-1, 0, 1\}$ subject to the condition that the number of non-zero elements is equal to h.
- $\mathcal{ZO}_s(0.5, n)$: This generates a vector of length n with elements chosen from $\{-1, 0, 1\}$ such that each entry is equal to r with probability p_r, where $p_{-1} = 1/4$, $p_0 = 1/2$, and $p_1 = 1/4$.
- $\mathcal{DG}_s(\sigma^2, n)$: This generates a vector of length n with elements chosen according to the discrete Gaussian distribution with variance σ^2.
- $\mathcal{RC}_s(0.5, \sigma^2, n)$: This generates a triple of elements (v, e_0, e_1) where v is sampled from $\mathcal{ZO}_s(0.5, n)$ and e_0 and e_1 are sampled from $\mathcal{DG}_s(\sigma^2, n)$.

- $\mathcal{U}_s(q, n)$: This generates a vector of length n with elements generated uniformly modulo q.

Any random values which *do not* depend on a seed should be assumed to be drawn using a secure entropy source (again, `/dev/urandom` in practice). We drop the subscript s from the notation, whenever the seed is not important.

Broadcast: When broadcasting data we assume two different models. In the online phase during partial opening players use the method described in [DPSZ12]: they send their data to a nominated player who then broadcasts the reconstructed value back to the remaining players. For other applications of broadcast, we assume each party broadcasts their values to all other parties directly. In all instances, players maintain a running hash of all values sent and received in a broadcast (with a suitable modification for the variant used for partial opening). At the end of a protocol run, these running hashes are compared pair-wise. This final comparison ensures that as long as there are at least two honest parties, the adversary must be consistent in what he sends to the honest parties.

Commitments: In Figure 5.1 we present an ideal functionality $\mathcal{F}_{\text{Commit}}$ for commitment, which is used in all of our protocols.

The Ideal Functionality $\mathcal{F}_{\text{Commit}}$

Commit: On input $(\mathsf{Commit}, v, i, \tau_v)$ by P_i, or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, the ideal functionality stores (v, i, τ_v) in a list and outputs (i, τ_v) to all players and the adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i, or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all players and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all players.

FIGURE 5.1. The Ideal Functionality for Commitments.

Our protocols are UC secure, which is possible despite the presence of dishonest majority because of the random oracle. In particular, a hash function $\mathcal{H}_1$ is modelled as a random oracle and a commitment scheme implements the functionality $\mathcal{F}_{\text{Commit}}$ as follows: the commit function $\mathsf{Commit}(m)$ generates a random value r and computes $c \leftarrow \mathcal{H}_1(m\|r)$. It returns the pair (c, o) where o is the opening information $m\|r$. When the commitment c is opened, the committer outputs the value o and the receiver runs $\mathsf{Open}(c, o)$ which checks whether $c = \mathcal{H}_1(o)$ and if so returns m.

The Protocol Π_{Commit}

Commit:

(1) In order to commit to v, P_i sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.

(2) P_i then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

(1) In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player P_i broadcasts $(o = v||r, i, \tau_v)$.

(2) All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

FIGURE 5.2. The Protocol for Commitments.

LEMMA 5.1. *In the random oracle model, the protocol Π_{Commit} implements $\mathcal{F}_{\text{Commit}}$ with computational security against any static, active adversary corrupting at most $n - 1$ parties.*

Proof. We describe a simulator such that the environment cannot distinguish whether it is playing with the real protocol or the functionality composed with the simulator. Note that the simulator replies to queries to the random oracle $\mathcal{H}$ made by the adversary.

To simulate a Commit call, if the committer P_i is honest, the simulator selects a random value c and gives (c, i, τ_v) to the adversary. If, on the other hand, the committer is corrupt, then it either queries $\mathcal{H}$ to get c, or it does not query it. Therefore, on receiving (c^*, i, τ_v) from the adversary, the simulator has v (if $\mathcal{H}$ was queried) and sets $v^* \leftarrow v$. If $\mathcal{H}$ was not queried, the simulator sets a dummy input v^* and sets the internal flag $\mathsf{Abort}_{i,\tau_v}$ to true. It then sends $(\mathsf{Commit}, v^*, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$.

An Open call is simulated as follows. If the committer is honest, the simulator gets (v, i, τ_v) when P_i inputs $(\mathsf{Open}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. The simulator selects random r and sets $o \leftarrow v||r$. It can now hand (o, i, τ_v) to the adversary. If the random oracle is queried on o, the simulator sends c as response. If the committer is corrupt, the simulator gets (i, τ_v) from the adversary, checks whether $\mathsf{Abort}_{i,\tau_v}$ is true, and, if so, sends $(\mathsf{NoOpen}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. Otherwise, the simulator sends $(\mathsf{Open}, i, \tau_i)$ to $\mathcal{F}_{\text{COMMIT}}$.

The adversary will notice that queries to $\mathcal{H}$ are simulated only if o has been queried before resulting in different c, but as r is random this happens only with negligible probability (assuming that the size of the output domain of $\mathcal{H}$ is large enough). Also, in a simulated run, if the adversary does not query $\mathcal{H}$ when committing, the run will be aborted. The probability that in a real run players do not abort is equivalent to the probability that the adversary correctly guesses the output of $\mathcal{H}$, which happens with negligible probability.

□

5.1.2. Key Generation. The key generation algorithm generates a public/private key pair such that the public key is given by $\mathsf{pk} = (a, b)$, where a is generated from $\mathcal{U}(q_1, \phi(m))$ (i.e. a is uniform in R_{q_1}), and $b = a \cdot \mathsf{sk} + p \cdot \varepsilon$ where ε is a "small" error term, and sk is the secret key such that $\mathsf{sk} = \mathsf{sk}_1 + \cdots + \mathsf{sk}_n$, where player P_i holds the share sk_i. Recall that since m is a power of 2, then $\phi(m) = m/2$.

The public key is also augmented to an extended public key $\mathfrak{epk}$ by addition of a "quasi-encryption" of the message $-p_1 \cdot \mathsf{sk}^2$, i.e. $\mathfrak{epk}$ contains a pair $(a_{\mathsf{sk},\mathsf{sk}^2}, b_{\mathsf{sk},\mathsf{sk}^2})$ such that $b_{\mathsf{sk},\mathsf{sk}^2} = a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} + p \cdot \varepsilon_{\mathsf{sk},\mathsf{sk}^2} - p_1 \cdot \mathsf{sk}^2$, where $a_{\mathsf{sk},\mathsf{sk}^2} \leftarrow \mathcal{U}(q_1, \phi(m))$ and $\varepsilon_{\mathsf{sk},\mathsf{sk}^2}$ is a "small" error term. The precise distributions of all these values are determined when we treat the details of the key generation protocol.

5.1.3. Encryption and Decryption. $\underline{\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})}$: To encrypt an element $m \in R_p$, using the modulus q_1, choose one "small polynomial" (with $0, \pm 1$ coefficients) and two Gaussian polynomials (with variance σ^2), via $(v, e_0, e_1) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. Then, set $c_0 = b \cdot v + p \cdot e_0 + m$, $c_1 = a \cdot v + p \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1, 1)$.
$\underline{\mathsf{SwitchModulus}((c_0, c_1), \ell)}$: The operation $\mathsf{SwitchModulus}(\mathfrak{c})$ takes the ciphertext $\mathfrak{c} = ((c_0, c_1), \ell)$ defined modulo q_ℓ and produces a ciphertext $\mathfrak{c}' = ((c_0', c_1'), \ell - 1)$ defined modulo $q_{\ell-1}$, such that $[c_0 - \mathsf{sk} \cdot c_1]_{q_\ell} \equiv [c_0' - \mathsf{sk} \cdot c_1']_{q_{\ell-1}} \pmod{p}$. This is done by setting $c_i' = \mathsf{Scale}(c_i, q_\ell, q_{\ell-1})$ where Scale is the function defined in [GHS12b]. Note that we need the more complex function given in Appendix E of the full version of [GHS12b] if working in dCRT representation, as we need to fix the scaling modulo p (as opposed to modulo two, which was done in the main body of [GHS12b]). Since the scheme has two levels, this function can only be called when $\ell = 1$.
$\underline{\mathsf{Dec}_{\mathsf{sk}}(\mathfrak{c})}$: Note that this operation is never actually performed, since the shared secret key sk is unknown. Decryption of a ciphertext $\mathfrak{c} = (c_0, c_1, \ell)$ at level ℓ is performed by setting $m' = [c_0 - \mathsf{sk} \cdot c_1]_{q_\ell}$, then converting m' to coefficient representation and outputting $m' \bmod p$.
$\underline{\mathsf{DistDec}_{\mathsf{sk}_i}(\mathfrak{c})}$: Decryption is performed by a simplified version of the distributed decryption procedure described in [DPSZ12], since the final ciphertexts consist of only two elements as opposed to three as in [DPSZ12]. For an input ciphertext $\mathfrak{c} = (c_0, c_1, \ell)$, P_1 computes $\mathbf{v}_1 = c_0 - \mathsf{sk}_1 \cdot c_1$ and P_i $i \neq 1$ computes $\mathbf{v}_i = -\mathsf{sk}_i \cdot c_1$. Each party P_i then sets $\mathbf{t}_i = \mathbf{v}_i + p \cdot \mathbf{r}_i$ for a random element $\mathbf{r}_i \in R$ with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ and the values $\mathbf{t}_i$ are broadcast; the precise value B is determined in Section 6.1. Then the message is recovered as $\mathbf{t}_1 + \cdots + \mathbf{t}_n \pmod{p}$.

5.1.4. Operations on Encrypted Data. Homomorphic addition follows trivially from the methods of [BGV12, GHS12b]. The main remaining task is to deal with multiplication. We first define a SwitchKey operation.
SwitchKey(d_0, d_1, d_2): This procedure takes as input an extended ciphertext $\mathfrak{c} = (d_0, d_1, d_2)$ defined modulo q_1; this is a ciphertext which is decrypted via the equation

$$[d_0 - \mathsf{sk} \cdot d_1 - \mathsf{sk}^2 \cdot d_2]_{q_1}.$$

The SwitchKey operation also takes the key-switching data $\mathfrak{enc} = (b_{\mathsf{sk},\mathsf{sk}^2}, a_{\mathsf{sk},\mathsf{sk}^2})$ above and produces a standard two element ciphertext which encrypts the same message but modulo q_0.

- $c'_0 \leftarrow p_1 \cdot d_0 + b_{\mathsf{sk},\mathsf{sk}^2} \cdot d_2 \pmod{q_1}$,
  $c'_1 \leftarrow p_1 \cdot d_1 + a_{\mathsf{sk},\mathsf{sk}^2} \cdot d_2 \pmod{q_1}$.
- $c''_0 \leftarrow \mathsf{Scale}(c'_0, q_1, q_0)$,
  $c''_1 \leftarrow \mathsf{Scale}(c'_1, q_1, q_0)$.
- Output $((c''_0, c''_1), 0)$.

Notice the following equality modulo q_1:

$$\begin{aligned} c'_0 - \mathsf{sk} \cdot c'_1 &= (p_1 \cdot d_0) + d_2 \cdot b_{\mathsf{sk},\mathsf{sk}^2} - \mathsf{sk} \cdot \left((p \cdot d_1) - d_2 \cdot a_{\mathsf{sk},\mathsf{sk}^2}\right) \\ &= p_1 \cdot (d_0 - \mathsf{sk} \cdot d_1 - \mathsf{sk}^2 d_2) - p \cdot d_2 \cdot \varepsilon_{\mathsf{sk},\mathsf{sk}^2}, \end{aligned}$$

The requirement on $p_1 \equiv 1 \pmod{p}$ is from the above equation as this should produce the same value as $d_0 - \mathsf{sk} \cdot d_1 - \mathsf{sk}^2 d_2 \mod q_1$ via reduction modulo p.
Mult$(\mathfrak{c}, \mathfrak{c}')$: We only need to execute multiplication on two ciphertexts at level one, thus $\mathfrak{c} = ((c_0, c_1), 1)$ and $\mathfrak{c}' = ((c'_0, c'_1), 1)$. The output is a ciphertext $\mathfrak{c}''$ at level zero, obtained via the following steps:

- $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c})$,
  $\mathfrak{c}' \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$.
- $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c'_0, c_1 \cdot c'_0 + c_0 \cdot c'_1, -c_1 \cdot c'_1)$.
- $\mathfrak{c}'' \leftarrow \mathsf{SwitchKey}(d_0, d_1, d_2)$.

5.2. Distributed Key Generation Protocol for BGV

As remarked in the introduction of this chapter, [DPSZ12] assumed a "magic" set-up which produces not only a distributed sharing of the main BGV secret key, but also a distributed sharing of the square of the secret key. That was assumed to be done via some other MPC protocol. The effect of requiring a sharing of the square of the secret key was that even if there was no need to perform KeySwitching, ciphertexts were 50% bigger than one would otherwise expect. Here we take a very different approach: we augment the public key with the KeySwitching data from [GHS12b] and provide an explicit covertly secure key generation protocol.

Our protocol is covertly secure in the sense that the probability that an adversary can deviate without being detected is bounded by $1/c$, for a positive integer c. The idea behind achieving covert security is as follows: Each player runs c instances of the basic protocol in parallel, each with different random seeds, then at the end of the main protocol all bar a random one of the basic protocol runs are opened, along with the respective random seeds. All parties check that the opened runs were performed honestly, and if any party finds an inconsistency, the protocol aborts. If no problem is detected, the parties assume that the single unopened run is correct. Thus, (intuitively) the adversary can cheat with probability at most $1/c$.

5.2.1. Overview. We start by discussing the generation of the main public key pk_j in execution j where $j \in \{1, \ldots, c\}$. To start with, the players generate a uniformly random value $a_j \in R_{q_1}$. They then each execute the standard BGV key generation procedure, with respect

The ideal functionality $\mathcal{F}_{\text{KeyGen}}$

Generation:

(1) The functionality waits for seeds $\{s_i\}_{i\in A}$ from the adversary, and samples a seed s_i for every honest player P_i.
(2) It computes $a_i \leftarrow \mathcal{U}_{s_i}(q_1, \phi(m))$.
(3) It computes $\mathsf{sk}_i \leftarrow \mathcal{HWT}_{s_i}(h, \phi(m))$, $\varepsilon_i \leftarrow \mathcal{DG}_{s_i}(\sigma^2, \phi(m))$ and $b_i \leftarrow [a \cdot \mathsf{sk}_i + p \cdot \varepsilon_i]_{q_1}$.
(4) It computes $b \leftarrow b_1 + \cdots + b_n$.
(5) It computes $\mathfrak{enc}'_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(-p_1 \cdot \mathsf{sk}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$.
(6) It computes $\mathfrak{enc}' \leftarrow \mathfrak{enc}'_1 + \cdots + \mathfrak{enc}'_n$.
(7) It computes $\mathfrak{zero}_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(0, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$, and $\mathfrak{enc}_i \leftarrow (\mathsf{sk}_i \cdot \mathfrak{enc}') + \mathfrak{zero}_i$.
(8) It leaks $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ to the adversary and waits for either Proceed, Cheat, or Abort.

Proceed: The functionality sends sk_i, $\mathsf{pk} = (a, b)$, $\mathfrak{epk} = (\mathsf{pk}, \mathfrak{enc})$ to P_i.

Cheat: On input Cheat, with probability $1 - 1/c$ the functionality leaks NoSuccess and goes to "Abort"; otherwise:

(1) It leaks the seeds of the honest parties and sends Success to the adversary.
(2) It repeats the following loop:
- It waits for the adversary to input values a_i^*, (resp. (sk_i^*, b_i^*), $\mathfrak{enc}_i'^*$, $\mathfrak{enc}_i^*$) for $i \in A$.
- It overwrites a_i, (resp. (sk_i, b_i), $\mathfrak{enc}'_i$, $\mathfrak{enc}_i$) for $i \in A$.
- It recomputes a (resp. b, $\mathfrak{enc}'$, $\mathfrak{enc}$) accordingly.

(3) It waits for Proceed or Abort.

Abort:

(1) The functionality leaks the seeds of the honest parties if it has not already done so.
(2) It then waits for a set $S \subseteq A$, sends it to the honest players, and aborts.

FIGURE 5.3. The ideal functionality for key generation.

to the global element a_j. Each player P_i chooses a low-weight secret key and then generates an LWE instance corresponding to that secret key. Following [GHS12b]:

$$\mathsf{sk}_{i,j} \leftarrow \mathcal{HWT}_s(h, \phi(m)) \text{ and } \varepsilon_{i,j} \leftarrow \mathcal{DG}_s(\sigma^2, \phi(m)).$$

Then P_i sets his secret key as $\mathsf{sk}_{i,j}$ and his "local" public key as $(a_j, b_{i,j})$ where $b_{i,j} = [a_j \cdot \mathsf{sk}_{i,j} + p \cdot \varepsilon_{i,j}]_{q_1}$.

Note, by a hybrid argument, obtaining n ring-LWE instances for n different secret keys but the same value of a_j is secure if obtaining one ring-LWE instance is secure[1]. Also note that a key modulo q_1 can be also treated as a key modulo q_0, since q_0 divides q_1 and $\mathsf{sk}_{i,j}$ has coefficients in $\{-1, 0, 1\}$.

The global public and private key are then set to be $\mathsf{pk}_j = (a_j, b_j)$ and $\mathsf{sk}_j = \mathsf{sk}_{1,j} + \cdots + \mathsf{sk}_{n,j}$, where $b_j = [b_{1,j} + \cdots + b_{n,j}]_{q_1}$. This is essentially another BGV key pair, since if $\varepsilon_j = \varepsilon_{1,j} + \cdots + \varepsilon_{n,j}$ then

$$b_j = \sum_{i=1}^{n} (a_j \cdot \mathsf{sk}_{i,j} + p \cdot \varepsilon_{i,j}) = a_j \cdot \mathsf{sk}_j + p \cdot \varepsilon_j,$$

but generated with different distributions for sk_j and ε_j, from those of the individual key pairs above.

The above key generation is then augmented to enable the construction of the KeySwitching data. Given a public key pk_j and a share of the secret key $\mathsf{sk}_{i,j}$ our method for producing the extended public key is to produce in turn the following (see Figure 5.4 for the details of how we create these elements in our protocol):

- $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}_j}(-p_1 \cdot \mathsf{sk}_{i,j})$
- $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.
- $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}_j}(\mathbf{0})$

[1] In the LWE literature this is called "amortization".

- $\mathfrak{enc}_{i,j} \leftarrow (\mathsf{sk}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j} \in R^2_{q_1}$.
- $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$.
- $\mathfrak{epk}_j \leftarrow (\mathsf{pk}_j, \mathfrak{enc}_j)$.

Note that $\mathfrak{enc}'_{i,j}$ is not a valid encryption of $-p_1 \cdot \mathsf{sk}_{i,j}$, since $-p_1 \cdot \mathsf{sk}_{i,j}$ does not lie in the message space of the encryption scheme. However, because of the dependence on the secret key shares here, we need to assume a form of circular security: the precise assumption needed is stated in Subsection 5.2.3. The encryption of zero, $\mathfrak{zero}_{i,j}$, is added on by each player to re-randomise the ciphertext, preventing an adversary from recovering $\mathsf{sk}_{i,j}$ from $\mathfrak{enc}_{i,j}/\mathfrak{enc}'_j$. We call the resulting $\mathfrak{epk}_j$ the *extended public key*. In [GHS12b] the KeySwitching data $\mathfrak{enc}_j$ is computed directly from sk_j^2; however, in our case parties need the above round-about method since sk_j^2 is not available to them.

Finally, players open all bar one of the c executions and check they have been executed correctly. If all checks pass then the final extended public key $\mathfrak{epk}$ is output and the players keep hold of their associated secret key share sk_i. See Figure 5.4 for full details of the protocol.

The protocol Π_{KeyGen}

Initialise:

(1) P_i samples uniform $e_i \leftarrow \{1, \ldots, c\}$ and asks $\mathcal{F}_{\text{Commit}}$ to broadcast the handle $\tau_i^e \leftarrow \mathsf{Commit}(e_i)$ for a commitment to e_i.

(2) P_i samples a seed $s_{i,j}$ and asks $\mathcal{F}_{\text{Commit}}$ to broadcast $\tau_{i,j}^s \leftarrow \mathsf{Commit}(s_{i,j})$.

(3) P_i computes and broadcasts $a_{i,j} \leftarrow \mathcal{U}_{s_{i,j}}(q_1, \phi(m))$.

Stage 1:

(4) All the players compute $a_j \leftarrow a_{1,j} + \cdots + a_{n,j}$.

(5) P_i computes $\mathsf{sk}_{i,j} \leftarrow \mathcal{HWT}_{s_{i,j}}(h, \phi(m))$ and $\varepsilon_{i,j} \leftarrow \mathcal{DG}_{s_{i,j}}(\sigma^2, \phi(m))$, broadcasts $b_{i,j} \leftarrow [a_j \cdot \mathsf{sk}_{i,j} + p \cdot \varepsilon_{i,j}]_{q_1}$.

Stage 2:

(6) All the players compute $b_j \leftarrow b_{1,j} + \cdots + b_{n,j}$ and set $\mathsf{pk}_j \leftarrow (a_j, b_j)$..

(7) P_i computes and broadcasts $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}_j}(-p_1 \cdot \mathsf{sk}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.

Stage 3:

(8) All the players compute $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.

(9) P_i computes $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}_j}(0, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.

(10) P_i computes and broadcasts $\mathfrak{enc}_{i,j} \leftarrow (\mathsf{sk}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j}$.

Output:

(11) All the players compute $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$ and set $\mathfrak{epk}_j \leftarrow (\mathsf{pk}_j, \mathfrak{enc}_j)$.

(12) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_i^e)$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.

(13) All players compute the challenge $\mathsf{chall} \leftarrow 1 + ((\sum_{i=1}^n e_i) \bmod c)$.

(14) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for $j \neq \mathsf{chall}$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.

(15) All players obtain the values committed, compute all the derived values and check that they are correct.

(16) If any of the checks fail, the players output the numbers of the respective players, and the protocol aborts. Otherwise, P_i sets

- $\mathsf{sk}_i \leftarrow \mathsf{sk}_{i,\mathsf{chall}}$,
- $\mathsf{pk} \leftarrow (a_{\mathsf{chall}}, b_{\mathsf{chall}})$, $\mathfrak{epk} \leftarrow (\mathsf{pk}, \mathfrak{enc}_{\mathsf{chall}})$.

FIGURE 5.4. The protocol for key generation.

5.2.2. UC Security Proof.

THEOREM 5.2. *In the $\mathcal{F}_{\text{Commit}}$-hybrid model, the protocol Π_{KeyGen} implements $\mathcal{F}_{\text{KeyGen}}$ with computational security against any static adversary corrupting at most $n-1$ parties.*

Proof. We build a simulator $\mathcal{S}_{\text{KeyGen}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{KeyGen}}$, such that the environment cannot distinguish whether it is playing with the protocol Π_{KeyGen} and $\mathcal{F}_{\text{Commit}}$, or the simulator and $\mathcal{F}_{\text{KeyGen}}$. The simulator is given in Figure 5.7.

The procedure CheatSwitch

Pass: Checks were passed for all threads, or there was a successful cheat in earlier stages and checks now passed for at least all threads except for chall.

- The simulator continues.

Cheat: There was no cheat in previous stages and all checks have now passed except for those in a single thread j.

- The simulator adds (j, P_i) to a list L, for all players P_i making the check not pass.
- The simulator sends Cheat to $\mathcal{F}_{\text{KeyGen}}$, and gets and stores every honest seeds $s_{i,\mathsf{chall}}$.
- If the functionality sends NoSuccess:
  - If $j = \mathsf{chall}$, it resamples a different $\mathsf{chall} \leftarrow \{1, \dots, c\} \setminus \{\mathsf{chall}\}$.

  It continues the simulation according to the protocol.
- If the functionality sends Success: the simulator sets $\mathsf{chall} = j$ and continues according to the protocol.

Abort: In more than one thread checks did not pass (counting also checks in previous stages).

- The simulator adds (j, P_i) to a list L, for all branches j and players P_i making the check not pass.
- The simulator sends Abort to $\mathcal{F}_{\text{KeyGen}}$ and it continues the rest of the simulation according to the protocol.

FIGURE 5.5. The cheat switch.

The simulator $\mathcal{S}_{\text{KeyGen}}$ – Initialisation Stage.

Initialise:

- In Step 1 the simulator obtains e_i by every corrupt P_i, and broadcasts τ_i^e as $\mathcal{F}_{\text{Commit}}$ would do. It samples chall uniformly in $\{1, \dots, c\}$, and it broadcasts a handle τ_i^e for every honest P_i.
- In Step 2 the simulator sees the random values $s_{i,j}$ for $i \in A$.
  It inputs $\{s_{i,\mathsf{chall}}\}_{i \in A}$ to $\mathcal{F}_{\text{KeyGen}}$, therefore obtaining a full transcript of the thread corresponding to chall.
  For the threads $j \neq \mathsf{chall}$, for honest P_i, the simulator samples $s_{i,j}$ honestly and broadcasts a handle $\tau_{i,j}^s$ for every honest P_i for every thread.
- In Step 3, the simulator computes $a_{i,j}$ honestly for $i \notin A$ and $j \neq \mathsf{chall}$, while it defines $a_{i,\mathsf{chall}}$ for $i \notin A$ as the values a_i obtained from the transcript given by $\mathcal{F}_{\text{KeyGen}}$.
  It then broadcasts $a_{i,j}$ for honest P_i and waits for broadcasts $a_{i,j}$ by the corrupt players, and it checks $a_{i,j} = \mathcal{U}_{s_{i,j}}(q, \phi(m))$ for all dishonest P_i. For this check the simulator enters CheatSwitch. If there was a successful cheat on the thread pointed by chall, the simulator inputs $a_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KeyGen}}$ for $i \in A$.

FIGURE 5.6. The simulator for the key generation functionality.

We now proceed with the analysis of the simulation. Let A denote the set of players controlled by the adversary. In steps 1 and 2 the simulator sends random handles to the adversary, as would happen in the real protocol. In steps 3–11 the simulation is perfect for all the threads where the simulator knows the seeds of the honest players, since those are generated as in the protocol. In the case that there is no cheating or abortion, the simulation is also perfect for the thread where the simulator does not know the seeds of the honest players, since the simulator forwards honest values provided by the functionality. In case of cheating at the thread determined by chall, the simulator gets the seeds also for the remaining thread and will replace the honestly precomputed intermediate values $a_{i,\mathsf{chall}}$, $\mathsf{sk}_{i,\mathsf{chall}}$, $b_{i,\mathsf{chall}}$, $\mathfrak{enc}'_{i,\mathsf{chall}}$, $\mathfrak{enc}_{i,\mathsf{chall}}$ with new values compatible with the deviation of the adversary – the honest values

The simulator $\mathcal{S}_{\text{KeyGen}}$

Stage 1:

- In Step 4 the simulator acts as in the protocol.
- In Step 5 for all the honest seeds that are known by the simulator, the simulator computes $b_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $b_{i,\mathsf{chall}}$ for $i \notin A$ as the values b_i obtained from the transcript given by $\mathcal{F}_{\text{KeyGen}}$.
  It then broadcasts $b_{i,j}$ for honest P_i and waits for broadcasts $b_{i,j}$ by the corrupt players. It then checks $b_{i,j} = [a_{\mathsf{chall}} \cdot \mathcal{HWT}_{s_{i,\mathsf{chall}}}(h, \phi(m)) + p \cdot \mathcal{RC}_{s_{i,\mathsf{chall}}}(\sigma^2, \phi(m))]_{q_1}$ for all dishonest P_i. For this check the simulator enters CheatSwitch. If there was a successful cheat on the thread pointed by chall, the simulator inputs $(\mathsf{sk}_{i,\mathsf{chall}}, b_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KeyGen}}$ for $i \in A$.

Stage 2:

- In Step 6 the simulator acts as in the protocol.
- In Step 7 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}'_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}'_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}'_i$ obtained from the transcript given by $\mathcal{F}_{\text{KeyGen}}$.
  It then broadcasts $\mathfrak{enc}'_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}'_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}'_{i,j} = \mathsf{Enc}_{\mathsf{pk}}(-p_1 \cdot \mathsf{sk}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$ for all dishonest P_i. For this check the simulator enters CheatSwitch. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}'_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KeyGen}}$ for $i \in A$.

Stage 3:

- In Step 8, 9 the simulator acts as in the protocol.
- In Step 10 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}_i$ obtained from the transcript given by $\mathcal{F}_{\text{KeyGen}}$.
  It then broadcasts $\mathfrak{enc}_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}_{i,j} = (\mathsf{sk}_{i,j} \cdot \mathfrak{enc}'_{i,j}) + \mathfrak{zero}_{i,j}$ for all dishonest P_i. For this check the simulator enters CheatSwitch. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KeyGen}}$ for $i \in A$.

FIGURE 5.7. The simulator for the key generation functionality.

computed after a cheat reflect the adversarial behaviour of the real protocol, so a simulated run is again indistinguishable from a real run of the protocol.

Steps 12, 14 are statistically indistinguishable from a protocol run, since the simulator plays also the role of $\mathcal{F}_{\text{Commit}}$.

Step 15 needs more work: we need to ensure that the success probability in a simulated run is the same as the one in a real run of the protocol. If the adversary does not deviate, the protocol succeeds. The same applies for a simulated run, since the simulator goes through "Pass" at every stage. More in detail, the simulator samples and computes all the values at the non-challenge threads as in a honest run of the protocol, while values in the challenge thread are correctly evaluated and sent to the honest players by $\mathcal{F}_{\text{KeyGen}}$. If the adversary cheats only on one thread, in a real execution of the protocol the adversary succeeds in the protocol with probability $1/c$; the same holds in a simulated run, since the simulator goes through Cheat in CheatSwitch once and with probability $1/c$ the functionality leaks Success, and the simulation will not abort. If the adversary deviates on more than one branch (considering all stages), both the real protocol and the simulation will abort at step 15.

Finally, if the protocol aborts due to failure at the stage of opening commitments, both the functionality and the players output the number of corrupted players who failed to open their commitments. If the protocol aborts at step 15, the output is the number of players who deviated in threads other than chall in both the functionality and the protocol. □

The simulator $\mathcal{S}_{\text{KeyGen}}$ – Output Stage

Output:

- Step 11 is performed according to the protocol.
- The simulator samples e_i for $i \notin A$ uniformly such that $1 + ((\sum_{i=1}^{n} e_i) \bmod c) = \mathsf{chall}$.
- Step 12 is performed according to the protocol, but the simulator opens τ_i^e revealing the values e_i for all honest P_i, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KeyGen}}$ and inputs the set of all players that failed upon opening.
- Step 13 is performed according to the protocol.
- Step 14 is performed according to the protocol, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KeyGen}}$ and inputs the set of all players that failed upon opening.
- Step 15 is performed according to the protocol, and the simulator defines

$$S = \{i \in A \mid (j, P_i) \in L;\ j \in \{1, \ldots, c\};\ j \neq \mathsf{chall}\},$$

  i.e. the set of corrupt players who cheated at any thread different from chall.
  - If $S \neq \varnothing$ (i.e. cheats at a thread which is going to be opened), the simulator sends Abort to $\mathcal{F}_{\text{KeyGen}}$ and inputs S.
  - If $S = \varnothing$ (i.e. successful or no cheats), the simulator sends $\mathsf{Proceed}$ to $\mathcal{F}_{\text{KeyGen}}$.
- Step 16 is performed according to the protocol.

FIGURE 5.8. The simulator for the key generation functionality.

Recall that $\mathcal{F}_{\text{Commit}}$ is a standard functionality for commitment. $\mathcal{F}_{\text{KeyGen}}$ simply generates a key pair with a distribution matching what we sketched above, and then sends the values $a_i, b_i, \mathfrak{enc}_i', \mathfrak{enc}_i$ for every i to all parties, and sends shares of the secret key to the honest players.

5.2.3. Semantic Security Proof. Here we prove the semantic security of the cryptosystem resulting from an execution of $\mathcal{F}_{\text{KeyGen}}$, based on the ring-LWE problem and a form of KDM security for quadratic functions. The ring-LWE assumption we use takes an extra parameter h, as our scheme chooses binary secret keys with low Hamming weight for better efficiency and parameter sizes, but note that the results here also apply to secrets drawn from other distributions.

DEFINITION 5.3 (Decisional Ring Learning With Errors Assumption). The single sample decisional ring-LWE assumption $\mathsf{RLWE}_{q,\sigma^2,h}$ states that

$$(a, a \cdot s + e) \overset{c}{\approx} (a, u)$$

where $s \leftarrow \mathcal{HWT}(h, \phi(m))$, $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ and a, u are uniform over R_q.

The KDM security assumption, below, can be viewed as a distributed extension to the usual key switching assumption for FHE schemes. In this case we need 'encryptions' of quadratic functions of *additive shares* of the secret key to remain secure. Note that whilst it is easy to show KDM security for *linear* functions of the secret [**BV11**], it is not known how to extend this to the functions required here without increasing the length of ciphertexts.

DEFINITION 5.4 (KDM Security Assumption). If $\mathsf{sk}_i \leftarrow \mathcal{HWT}(h)$, $\mathsf{sk} = \sum_{i=0}^{n-1} \mathsf{sk}_i$ and f is any degree 2 polynomial then

$$(a, a \cdot \mathsf{sk} + p \cdot e + f(\mathsf{sk}_0, \ldots, \mathsf{sk}_{n-1})) \overset{c}{\approx} (a, a \cdot \mathsf{sk} + p \cdot e)$$

where $a, u \leftarrow \mathcal{U}(q, \phi(m))$, $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$.

The following lemma states that distinguishing any number of 'amortized' ring-LWE samples with different, independent, secret keys but common first component a from uniform is as hard as distinguishing just one ring-LWE sample from uniform. An analogous statement was proven for the (standard) LWE setting with $n = 3$ in [**PVW08**]; here we need a version with ring-LWE for any n.

LEMMA 5.5 (Adapted from [PVW08, Lemma 7.6]). *Suppose* $a, u_i \leftarrow \mathcal{U}(q, \phi(m))$, $\mathsf{sk}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ *and* $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ *for* $i = 0, \ldots, n-1$, $n \in \mathbb{N}$. *Then*

$$\{(a, a \cdot \mathsf{sk}_i + e_i)\}_i \overset{c}{\approx} \{(a, u_i)\}_i$$

under the single sample ring-LWE assumption $\mathsf{RLWE}_{q,\sigma^2,h}$.

Proof. Suppose an adversary $\mathcal{A}$ can distinguish between the above distributions with non-negligible probability. We construct an adversary $\mathcal{B}$ that solves the RLWE problem. Given a challenge (a, b) from the RLWE oracle, $\mathcal{B}$ sets $b_0 = b$ and $b_i = a \cdot \mathsf{sk}_i + e_i$ for $i = 1, \ldots, n-1$, where $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m)), \mathsf{sk}_i \leftarrow \mathcal{HWT}(h, \phi(m))$. $\mathcal{B}$ sends all pairs (a, b_i) to $\mathcal{A}$ and returns the output of $\mathcal{A}$ in response to the challenge.

Since the values (a, b_i) for $i = 1, \ldots, n-1$ are all valid amortized ring-LWE samples, the only difference between the view of $\mathcal{A}$ and that of a real set of inputs is b_0, and so the advantage of $\mathcal{B}$ in solving $\mathsf{RLWE}_{q,\sigma^2,h}$ is exactly that of $\mathcal{A}$ in solving the amortized ring-LWE problem with n samples. □

THEOREM 5.6. *If the functionality* $\mathcal{F}_{\text{KEYGEN}}$ *is used to produce a public key* $\mathfrak{epk}$ *and secret keys* sk_i *for* $i = 0, \ldots, n-1$ *then the resulting cryptosystem is semantically secure based on the hardness of* $\mathsf{RLWE}_{q_1,\sigma^2,h}$ *and the KDM security assumption.*

Proof.

Suppose there is an adversary $\mathcal{A}$ that can interact with $\mathcal{F}_{\text{KEYGEN}}$ and distinguish the public key $(\mathsf{pk}, \mathfrak{epk})$ from uniform. We construct an algorithm $\mathcal{B}$ that distinguishes amortized ring-LWE samples from uniform. By Lemma 5.5 this is at least as hard as breaking single sample ring-LWE. If the public key is pseudorandom, then semantic security of encryption easily follows, as ciphertexts are just ring-LWE samples. Note that we only consider a non-cheating adversary – if $\mathcal{A}$ cheats then it can trivially break the scheme with (non-negligible) probability $1/c$.

The challenger gives $\mathcal{B}$ the values $a_c, b_{c,0}, \ldots, b_{c,n-1}$. $\mathcal{B}$ must now simulate an execution of $\mathcal{F}_{\text{KEYGEN}}$ with $\mathcal{A}$ to determine whether the challenge is uniform or of the form $(a_c, a_c \cdot \mathsf{sk}_i + e_i)$ for $\mathsf{sk}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ and $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$.

To start with, the simulator receives the adversary's seeds s_i for every corrupt player $i \in A$. It simulates the values $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ (for all i) that are leaked to the adversary in $\mathcal{F}_{\text{KEYGEN}}$: for corrupt players it simply computes these values according to $\mathcal{F}_{\text{KEYGEN}}$ using the adversary's seeds, while for the honest players' values, it uses the challenge $a_c, b_{c,0}, \ldots, b_{c,n-1}$. First it scales the challenge by p, so that it takes the form $(a_c, a_c \cdot \mathsf{sk}_i + p \cdot e_i)$, if they are genuine RLWE samples. Since p is coprime to q the new distribution is the same as the one of the original challenge.

Now $\mathcal{B}$ calculates uniform consistent shares $a_{c,i}$, for every honest player P_i, of a_c, and sends $\mathcal{A}$ the pairs $a_{c,i}, b_{c,i}$. If the challenge values are amortized ring-LWE samples, then these are consistent with the pairs (a_i, b_i) computed by $\mathcal{F}_{\text{KEYGEN}}$, since a_i is uniform and $b_i = a_i \cdot \mathsf{sk}_i + e_i$.

Next, $\mathcal{B}$ must provide $\mathcal{A}$ with simulations of players' contributions to the key-switching data $\mathfrak{enc}'_i, \mathfrak{enc}_i$ for all honest players P_i. For both of these sets of values, $\mathcal{B}$ simply re-randomises the pair (a_c, b_c) and sends this to $\mathcal{A}$. This can be done by, for example, computing an encryption of zero under the public key (a_c, b_c) (where $b_c = \sum_i b_{c,i}$). Notice that $\mathfrak{enc}'_i$ is just an encryption of $-p_1 \cdot \mathsf{sk}_i$ under the public key (a, b), and so by the KDM security assumption it is (perfectly) indistinguishable from a re-randomised version of (a, b). For $\mathfrak{enc}_i$, recall that $\mathcal{F}_{\text{KEYGEN}}$ computes $\mathfrak{enc}_i = \mathsf{sk}_i \cdot \mathfrak{enc}' + \mathfrak{zero}_i$. Now writing $\mathfrak{zero}_i = (a \cdot v_i + p \cdot e_{0,i}, b \cdot v_i + p \cdot e_{1,i})$ and $\mathfrak{enc}' = (a \cdot v + p \cdot e_0, b \cdot v + p \cdot e_1 - p_1 \cdot \mathsf{sk})$, we see that

$$\begin{aligned}
\mathfrak{enc}_i &= (a \cdot v \cdot \mathsf{sk}_i + a \cdot v_i + p \cdot (e_0 \cdot \mathsf{sk}_i + e_{0,i}), b \cdot v \cdot \mathsf{sk}_i + b \cdot v_i + p \cdot (e_1 \cdot \mathsf{sk}_i + e_{1,i}) - p_1 \cdot \mathsf{sk} \cdot \mathsf{sk}_i) \\
&= \left(\underbrace{a \cdot (v \cdot \mathsf{sk}_i + v_i)}_{a'_i} + p \cdot \underbrace{(e_0 \cdot \mathsf{sk}_i + e_{0,i})}_{e'_{0,i}}, \underbrace{a \cdot (v \cdot \mathsf{sk}_i + v_i)}_{a'_i} \cdot \mathsf{sk} + p \cdot \underbrace{(e_1 \cdot \mathsf{sk}_i + e_{1,i} + e)}_{e'_{1,i}} - p_1 \cdot \mathsf{sk} \cdot \mathsf{sk}_i \right) \\
&= \left(a'_i + p \cdot e'_{0,i}, a'_i \cdot \mathsf{sk} + p \cdot e'_{1,i} - p_1 \cdot \mathsf{sk} \cdot \mathsf{sk}_i \right).
\end{aligned}$$

Notice that the first component of $\mathfrak{enc}_i$ corresponds to the second half of a ring-LWE sample $(a, a \cdot (v \cdot \mathsf{sk}_i + v_i) + p \cdot e_{0,i})$ with secret $v \cdot \mathsf{sk}_i + v_i$. The second component of $\mathfrak{enc}_i$ corresponds to a ring-LWE sample with secret sk and first half a'_i, with an added quadratic function of the key $-p_1 \cdot \mathsf{sk} \cdot \mathsf{sk}_i$. By the KDM security assumption, this is indistinguishable from a genuine ring-LWE sample, so $\mathfrak{enc}_i$ can also be perfectly simulated by re-randomising (a_c, b_c).

To finish the simulated execution of $\mathcal{F}_{\text{KeyGen}}$, $\mathcal{B}$ sends $\mathcal{A}$ shares of the secret key for all P_i where $i \in A$ (i.e. all dishonest players), by sampling randomness using the seeds that were provided to $\mathcal{B}$ at the beginning. $\mathcal{B}$ then waits for $\mathcal{A}$ to give an answer and returns this in response to the challenger. Notice that throughout the simulation, all values passed to $\mathcal{A}$ were ring-LWE samples derived from the challenge $(a_c, b_{0,c}, \ldots, b_{n-1,c})$. We showed that if the challenge is an amortized ring-LWE sample then $\mathcal{A}$'s input is indistinguishable from the output of $\mathcal{F}_{\text{KeyGen}}$, whereas if the challenge is uniform then so is $\mathcal{A}$'s input. Therefore if $\mathcal{A}$ is successful in distinguishing the resulting public key from uniform then $\mathcal{A}$ must have solved the ring-LWE problem.

□

5.3. EncCommit

The sub-protocol $\Pi_{\text{EncCommit}}$ replaces the Π_{ZKPoPK} protocol from [DPSZ12]. We introduce the functionality $\mathcal{F}_{\text{SHE}}$ and later describe the details of the sub-protocol $\Pi_{\text{EncCommit}}$, which implements $\mathcal{F}_{\text{SHE}}$ given access to $\mathcal{F}_{\text{Commit}}$ and $\mathcal{F}_{\text{KeyGen}}$.

The ideal functionality $\mathcal{F}_{\text{SHE}}$

Usage: The functionality is split into a one-run stage which computes the key material, and a stage which can be accessed several times and is designed to replace the zero-knowledge protocols in [DPSZ12].

KeyGen: On input KeyGen the functionality acts as a copy of $\mathcal{F}_{\text{KeyGen}}$.
Notice that all the variables used during this call are available for later use.

EncCommit: On input EncCommit the functionality does the following.

Initialise: Denote by A the set of indices of corrupt players. On input Start by all players, sample, at random, seeds $\{s_i\}_{i \notin A}$ and wait for corrupted seeds $\{s_i\}_{i \in A}$ from the adversary.

Computation:

(1) It sets $\mathbf{m}_i \leftarrow \mathsf{PRF}_{s_i}$ subject to condition cond.
(2) It sets $\mathfrak{c}_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for each player P_i.
(3) It gives $\{\mathfrak{c}_i\}_{i \notin A}$ to the adversary, and waits for signal Deliver, Cheat or Abort.

Delivery: The functionality sends $\mathbf{m}_i$, $\{\mathfrak{c}_j\}_{j \leq n}$ to player P_i.

Cheat: The functionality gives $\{s_i\}_{i \notin A}$ to the adversary, then it decides to do one of the following things:

- With probability $1/c$ it sends Success to the adversary, waits for $\{\mathbf{m}_i, \mathfrak{c}_i\}_{i \in A}$, and outputs $\mathbf{m}_i, \{\mathfrak{c}_j\}_{i \leq n}$ to player P_i.
- Otherwise it sends NoSuccess to the adversary, and goes to abort.

Abort: The functionality waits for the adversary to input $S \subseteq A$, and outputs S to all players.

FIGURE 5.9. The ideal functionality for key generation and $\Pi_{\text{EncCommit}}$.

In this section we consider a covertly secure, rather than actively secure, variant; this means that players controlled by a malicious adversary succeed in deviating from the protocol with a probability bounded by $1/c$. In our experiments we pick $c = 5$, 10 and 20. In Section 6.3 we present an actively secure variant of this protocol.

The sub-protocol assumes that players have agreed on the key material for the encryption scheme, i.e. $\Pi_{\text{EncCommit}}$ runs in the $\mathcal{F}_{\text{KeyGen}}$-hybrid model. The protocol ensures that a party outputs a validly created ciphertext containing an encryption of some random message m, where the message m is drawn from a distribution satisfying condition cond. This is done by

committing to seeds and using the cut-and-choose technique, similarly to the key generation protocol. The condition cond could either be "uniformly generated from R_p", or "uniformly generated from $\mathbb{F}_p$" (i.e. a "diagonal" element in the SIMD representation).

Protocol $\Pi_{\text{EncCommit}}$

Usage: The specific distribution of the message is defined by the input parameter cond. The output is a single message $\mathbf{m}_i$ private to each player, and a public ciphertext $\mathfrak{c}_i$ from player P_i. The protocol runs in two phases: a commitment phase, and an opening phase.

KeyGen: The players execute Π_{KeyGen} to obtain sk_i, pk, and $\mathfrak{cpk}$.

Commitment Phase:

(1) P_i samples a uniform $e_i \leftarrow \{1, \dots, c\}$, and queries $\mathsf{Commit}(e_i)$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle τ_i^e.

(2) For $j = 1, \dots, c$:

  (a) P_i samples $s_{i,j}$ and queries $\mathsf{Commit}(s_{i,j})$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts $\tau_{i,j}^s$.

  (b) P_i generates $\mathbf{m}_{i,j}$ according to cond using $\mathsf{PRF}_{s_{i,j}}$.

  (c) P_i computes and broadcasts $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_{i,j})$ using $\mathsf{PRF}_{s_{i,j}}$ to generate the randomness.

(3) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_i^e)$. All players get e_i. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.

(4) All players compute $\mathsf{chall} \leftarrow 1 + ((\sum_{i=1}^n e_i) \bmod c)$.

Opening Phase:

(5) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for all $j \neq \mathsf{chall}$ so that all players obtain the value $s_{i,j}$ for $j \neq \mathsf{chall}$. If any opening fails, the players output the numbers of the respective players, and the protocol aborts.

(6) For all $j \neq \mathsf{chall}$ and all $i' \leq n$, the players check whether $\mathfrak{c}_{i',j}$ was generated correctly using $s_{i',j}$. If not, they output the numbers of the respective players i', and the protocol aborts.

(7) Otherwise, every player P_i stores $\{\mathfrak{c}_{i',\mathsf{chall}}\}_{i' \leq n}$ and $m_{i,\mathsf{chall}}$.

FIGURE 5.10. Protocol that allows ciphertexts to be used as commitments for plaintexts.

THEOREM 5.7. *In the $(\mathcal{F}_{\text{Commit}}, \mathcal{F}_{\text{KeyGen}})$-hybrid model, the protocol $\Pi_{\text{EncCommit}}$ implements $\mathcal{F}_{\text{SHE}}$ with computational security against any static adversary corrupting at most $n-1$ parties.*

Proof. We construct a simulator $\mathcal{S}_{\text{SHE}}$ (see Figure 5.11) working on top of $\mathcal{F}_{\text{SHE}}$ such that the environment cannot distinguish whether it is playing with the real protocol $\Pi_{\text{EncCommit}}$ and $\mathcal{F}_{\text{KeyGen}}$ or with $\mathcal{F}_{\text{SHE}}$ and $\mathcal{S}_{\text{SHE}}$.

Calls to $\mathcal{F}_{\text{KeyGen}}$ are simulated as in $\mathcal{S}_{\text{KeyGen}}$. We now focus on the commitment phase.

Let A be the set of indices of corrupted players. The simulator starts assuming that the adversary will behave honestly. It samples a uniform $j_0 \leftarrow \{1, \dots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$. If the adversary does not deviate, then round j_0 will remain unopened: otherwise the simulator has to adjust this. Round j is simulated as follows. First, the simulator gets corrupted seeds $s_{i,j}$ for $i \in A$ when the adversary commits to them in step 2a. It returns random handles $\tau_{i,j}^s$ on behalf of each honest player P_i.

If $j \neq j_0$, the simulator engages with the adversary in a normal run of steps 2b to 2c using seeds $s_{i,j}$ for honest player P_i. Since the simulator knows the corrupt seeds of the current round j, it can check whether the adversary behaved honestly. If not, then the simulator stores index j in the cheating list.

If $j = j_0$, the simulator checks again whether the adversary computed the right encryptions $\{\mathfrak{c}_i\}_{i \in A}$. If it did not, the simulator stores j_0 in the cheating list. Then, the simulator calls $\mathsf{EncCommit}$ to $\mathcal{F}_{\text{SHE}}$ on seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back $\{\mathfrak{c}_i\}_{i \notin A}$, which are the values computed by the functionality. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ and passes them onto the adversary in step 2c.

Once the last round is finished, the simulator checks the cheating list. There are three possibilities:

The simulator $\mathcal{S}_{\text{SHE}}$

KeyGen: $\mathcal{S}_{\text{SHE}}$ acts as $\mathcal{S}_{\text{KeyGen}}$, except that it calls $\mathcal{F}_{\text{SHE}}$ on query KeyGen, when $\mathcal{S}_{\text{KeyGen}}$ would have called $\mathcal{F}_{\text{KeyGen}}$.

Commitment Phase:

- The simulator chooses random $j_0 \leftarrow \{1, \ldots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$.
- Acting like the $\mathcal{F}_{\text{Commit}}$ functionality, in response to query in step 1 and 2a, for $j = 1, \ldots, c$ the simulator samples $s_{i,j}$ according to the protocol for $i \notin A$ and returns random handles $\{\tau_i^e\}_{i \leq n}, \{\tau_{i,j}^s\}_{i \leq n}$.
- For $j = 1, \ldots, c$, the simulator does the following:
  - If $j \neq j_0$, it performs steps 2b and 2c according to protocol using honest seeds $s_{i,j}$ for each $i \notin A$.
  - If $j = j_0$, it calls $\mathcal{F}_{\text{SHE}}$ on query EncCommit on corrupted seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back honest encryptions $\{\mathfrak{c}_i\}_{i \notin A}$. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ for each $i \notin A$.
- In step 2c, the simulator receives encryptions $\mathfrak{c}_{i,j}^*$ for each $i \in A$ and $j \in \{1, \ldots, c\}$. It generates $\mathbf{m}_{i,j}$ subject to cond, and $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_{i,j})$, and checks if $\mathfrak{c}_{i,j} = \mathfrak{c}_{i,j}^*$. If the equality does not hold, it stores j in a (cheating) list.
- The simulator reads the cheating list. There are three possibilities:
  - The list is empty. The simulator sets chall $\leftarrow j_0$.
  - The list contains only one index j_1. The simulator sends Cheat to $\mathcal{F}_{\text{SHE}}$ and gets $\{s_i\}_{i \notin A}$ back. It then sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$.
    - If the functionality returns Success, the simulator sets chall $\leftarrow j_1$.
    - If the functionality returns NoSuccess, the simulator samples chall $\leftarrow \{1, \ldots, c\} \setminus \{j_1\}$.
  - The list contains at least two indices. The simulator sends Abort to $\mathcal{F}_{\text{SHE}}$, gets $\{s_i\}_{i \notin A}$ and sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$, and chall $\leftarrow j_0$.
- For all honest P_i the simulator sets e_i uniformly in $1, \ldots, c$ with the constraint $1 + ((\sum_{i=1}^n e_i) \bmod c) =$ chall.
- In step 3, the simulator opens the handle τ_i^e to the freshly defined value e_i, for all honest P_i. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- Step 4 is performed according to the protocol.

Opening Phase:

- In step 5, the simulator opens the handle $\tau_{i,j}^s$ to $s_{i,j}$ for all honest players $i \notin A$ and $j \neq$ chall. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- If the cheating list is empty, the simulator sends Deliver to $\mathcal{F}_{\text{SHE}}$.
- If the functionality returned Success earlier, the simulator inputs $\{\mathbf{m}_{i,\text{chall}}, \mathfrak{c}_{i,\text{chall}}^*\}_{i \in A}$ to the functionality.
- If the functionality returned NoSuccess, or if the cheating list contains at least two indices, the simulator inputs to the functionality the number of players $i \in A$ whose $\mathfrak{c}_{i,j}^*$ were computed incorrectly for some $j \neq$ chall.

FIGURE 5.11. The simulator for $\mathcal{F}_{\text{SHE}}$.

- The list is empty. I.e. the adversary behaved honestly. The simulator sets chall $\leftarrow j_0$, and sends Deliver to the functionality if all commitments are successfully opened. The output of $\mathcal{F}_{\text{SHE}}$ and what the adversary has already seen will be consistent since $\mathcal{F}_{\text{SHE}}$ was called in round j_0 with the right seeds $\{s_{i,j_0}\}_{i \in A}$.
- The list contains only one index j_1. In this case the simulator sends Cheat and gets in return seeds $\{s_i\}_{i \notin A}$ used by the functionality. It sets $s_{i,j_0} \leftarrow s_i$ for each honest player P_i. It then waits for the answer.
  - If the functionality returns Success, the simulator has to make the adversary believe that round j_1 will remain unopened. It sets chall $\leftarrow j_1$. If all commitments

are successfully opened, it sends $\{\mathbf{m}_{i,j_1}, \mathfrak{c}_{i,j_1}\}_{i\in A}$ to the functionality in order to make consistent the players' outputs and what the adversary has already seen.
  - If it returns NoSuccess, the simulator has to make the adversary believe that round j_1 will be opened. Therefore it samples $\mathsf{chall} \leftarrow \{1,\dots,c\} \setminus \{j_1\}$.
- The list contains at least two indices j_1, j_2. In this case the real protocol would result in abort, so the simulator sends Abort to the functionality and sets $\mathsf{chall} \leftarrow j_0$.

Later the simulator generates the value e_i for each honest player such that

$$1 + \left(\left(\sum_{i=1}^{n} e_i\right) \bmod c\right) = \mathsf{chall}.$$

This ensures that once the challenge is computed, it will specify a round in the same fashion as in a true protocol run. Moreover, opening τ_i^e to (any) e_i does not give any hint to the adversary as ti whether it is playing in a real run of the protocol or in a simulated one.

In the opening phase, the simulator gives $\{e_i\}_{i\notin A}$ and honest share $\{s_{i,j}\}_{i\notin A, j\neq\mathsf{chall}}$ to the adversary, and if there was an unsuccessful cheating attempt, then it also sends Abort on behalf of each honest player.

It is clear, from the construction of $\mathcal{F}_{\text{SHE}}$, that all the messages generated by the simulator are indistinguishable from those of a real run of the protocol. The simulator does the same computations, except in round j_0 where the computation is done by the functionality, and the values are then passed onto the simulator, which forwards them to the adversary.

Finally, if the protocol aborts due to failure at opening commitments, both the functionality and the players output the numbers of corrupted players who failed to open their commitments. If the protocol aborts at step 6, the output is the numbers of players who deviated in threads other than chall in both the functionality and the protocol. □

$\mathcal{F}_{\text{SHE}}$ offers the same functionality as $\mathcal{F}_{\text{KeyGen}}$ but can in addition generate correctly formed ciphertexts where the plaintext satisfies a condition cond as explained above, and where the plaintext is known to a particular player (even if he is corrupt). Of course, by using the actively secure version of $\Pi_{\text{EncCommit}}$ from 6.3, one would get a version of $\mathcal{F}_{\text{SHE}}$ where the adversary is not allowed to attempt cheating.

5.4. MAC Checking

In this section we present our method for checking MACs of partially opened values without revealing the underlying MAC key. This procedure is mainly useful in the online phase; however, we shall also use it to verify the output of the offline phase.

We assume some value a has been $\langle\cdot\rangle$-shared and partially opened, which means that players have revealed shares of the a but not of the associated MAC value γ, which is still additively shared. Since there is no guarantee that a is correct, we need to check that $\gamma = \alpha a$ where α is the global MAC key that is also additively shared. In [DPSZ12], this was done by having players commit to the shares of the MAC, then open α, and check everything in the clear. This implies that other shared values become useless because the MAC key becomes public, and the adversary can manipulate them as he desires.

So, the target is to avoid opening α. Observe that since a is public, the value $\gamma - \alpha a$ is a linear function of shared values γ, α, so players can compute shares in this value locally and can then check if it is 0 without revealing information about α. As in [DPSZ12], players can optimise the cost of this by checking many MACs in one go, by taking a random linear combination of a and γ-values and checking only the results of this. The full protocol is given in Figure 5.12; it is not intended to implement any functionality – it is just a procedure that can be called in both the offline and online phases.

Protocol MACCheck

Usage: Each player has input α_i and $(\gamma(a_j)_i)$ for $j = 1, \ldots, t$. All players have a public set of opened values $\{a_1, \ldots, a_t\}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

MACCheck$(\{a_1, \ldots, a_t\})$**:**

(1) P_i samples a seed s_i and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Commit}(s_i)$.
(2) P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^s)$ and all players obtain s_j for $1 \le j \le n$.
(3) Players set $s \leftarrow s_1 + \ldots + s_n$.
(4) Players sample a random vector $\mathbf{r} = \mathcal{U}_s(p, t)$.
Note that all players obtain the same vector as they have agreed on the seed s.
(5) Each player computes the public value $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$.
(6) P_i computes $\gamma_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$.
(7) P_i asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^\sigma \leftarrow \mathsf{Commit}(\sigma_i)$.
(8) P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^\sigma)$, and all players obtain σ_j for $1 \le j \le n$.
(9) If $\sigma_1 + \cdots + \sigma_n \neq 0$, the players output $\varnothing$ and abort.

FIGURE 5.12. Method to check MACs on partially opened values.

LEMMA 5.8. *The protocol* MACCheck *is* correct*, i.e. it accepts if all the values a_j and the corresponding MACs are correctly computed. Moreover, it is* sound*, i.e. it rejects except with probability $2/p$ in case at least one value or MAC is not correctly computed.*

Proof.

We here inspect the correctness and the soundness error of the MACCheck protocol. In order to understand the probability of an adversary being able to cheat, we design the following security game.

(1) The challenger generates the secret key $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ and MACs $\gamma(a_j)_i \leftarrow \alpha \cdot a_j$ and sends messages $a_1, \ldots, a_t$ to the adversary.
(2) The adversary sends back messages $a'_1, \ldots, a'_t$.
(3) The challenger generates random values $r_1, \ldots, r_t \leftarrow \mathbb{F}_p$ and sends them to the adversary.
(4) The adversary provides an error Δ.
(5) Set $a \leftarrow \sum_{j=0}^{t} r_j a'_j$, $\gamma_i \leftarrow \sum_{j=0}^{t} r_j \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$. Now, the challenger checks that $\sigma_1 + \cdots + \sigma_n = \Delta$

The adversary wins the game if there is an i for which $a'_i \neq a_i$ and the final check goes through.

The second step in the game where the adversary sends the a'_i's models the fact that corrupted players can choose to lie about their shares of values opened during the protocol execution. Δ models the fact that the adversary is allowed to introduce errors on the macs.

Now, we check the probability of winning the game if the r_i's are randomly chosen. If the check goes through, the following equalities hold:

$$\begin{aligned}
\Delta &= \sum_{i=1}^{n} \sigma_i = \sum_{i=1}^{n} (\gamma_i - \alpha_i \cdot a) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i - \alpha_i \cdot \sum_{j=1}^{t} r_j \cdot a'_j \right) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left(r_j \cdot \gamma(a_j)_i - \alpha_i \cdot r_j \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} \left(r_j \cdot \sum_{i=1}^{n} \left(\gamma(a_j)_i - \alpha_i \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} r_j \cdot \left(\alpha \cdot a_j - \alpha \cdot a'_j \right) \\
&= \alpha \cdot \sum_{j=1}^{t} r_j \cdot \left(a_j - a'_j \right)
\end{aligned}$$

So, the following equality holds:

$$\alpha \cdot \sum_{j=0}^{t} r_j (a'_j - a_j) = \Delta. \tag{5.1}$$

First, if $\sum_{j=0}^{t} r_j(a'_j - a_j) \neq 0$, then $\alpha = \Delta / \sum_{j=0}^{t} r_j(a'_j - a_j)$. This implies that being able to pass the check is equivalent to guessing α. However, since the adversary has no information about α, this happens with probability $1/|\mathbb{F}_p|$. What is left is to argue that $\sum_{j=0}^{t} r_j(a'_j - a_j) = 0$ also happens with very low probability. This can be seen as follows: define $\mu_j := (a'_j - a_j)$ and $\mu := (\mu_1, \ldots, \mu_t), r := (r_1, \ldots, r_t)$. Now, $f_\mu(r) := r \cdot \mu = \sum_{j=0}^{t} r_j \mu_j$ defines a linear mapping, which is not trivial, since at least one $\mu_j \neq 0$. Therefore, $\dim(\ker(f_\mu)) = t - 1$ (from the rank-nullity theorem). Also, since r is random and the adversary does not know r when choosing the a'_i's, the probability of $r \in \ker(f_\mu)$ is $|\mathbb{F}_p^{t-1}|/|\mathbb{F}_p^t| = 1/|\mathbb{F}_p|$. It follows that the total probability of winning the game is at most $2/|\mathbb{F}_p|$.

For correctness, notice that Equation 5.1 holds with probability one if $a'_j = a_j$ and $\Delta = 0$ (honest prover). □

5.5. Offline Protocol

The offline phase produces preprocessed data for the online phase (where the secure computation is performed). To ensure security against active adversaries the MAC values of any partially opened value need to be verified. We use the above method, which overcomes some limitations of the corresponding method from [DPSZ12].

The offline phase itself runs two distinct sub-phases, described below. To start with, a BGV key has been distributed according to the key generation procedure described earlier, as well as the shares of a secret MAC key and an encryption $\mathfrak{c}_\alpha$ of the MAC key as above. Let the output

The functionality $\mathcal{F}_{\text{PREP}}$

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is component-wise.

Initialise: On input (Start, p) from honest players and the adversary, the functionality sets the internal flag BreakDown to false and then it does the following:

(1) For $i \in A$, the functionality accepts shares α_i from the adversary, and it samples uniform α_i for each $i \notin A$. Then the functionality sets $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$.
(2) The functionality waits for signal Abort, Proceed or Cheat from the adversary.
(3) If it receives Proceed, the functionality outputs α_i to P_i.
(4) Otherwise, and if the functionality did not abort in Cheat, it outputs the adversary's contribution Δ_i to P_i.

Computation: On input DataGen from all honest players and the adversary, and only if the functionality received Proceed (or BreakDown is true), it executes the data generation procedures specified in Figure 5.14.

Macro $\mathsf{Angle}(\mathbf{v}_1, \ldots, \mathbf{v}_n, \Delta_\gamma, k)$**:** The following is run by the functionality at several points to create representations $\langle \cdot \rangle$.

(1) The functionality gets $\{\gamma_i\}_{i \in A}$ from the adversary.
(2) Let $\mathbf{v} = \mathbf{v}_1 + \cdots + \mathbf{v}_n$, set $\gamma(\mathbf{v}) \leftarrow \alpha \cdot \mathbf{v} + \Delta_\gamma$.
(3) It samples $\gamma_i(\mathbf{v}) \leftarrow (\mathbb{F}_p)^k$ for $i \notin A$, subject to $\gamma(\mathbf{v}) = \sum_i^n \gamma(\mathbf{v})_i$.
(4) It returns $(\gamma(\mathbf{v})_1, \ldots, \gamma(\mathbf{v})_n)$.

Cheat: The functionality chooses to do either one of the following:

- It sends, with probability $1/c$, Success to the adversary and sets the internal flag BreakDown to true.
- Otherwise, it sends NoSuccess to the adversary and players, and goes to "Abort".

Abort: The functionality waits for $S \subseteq A$ from the adversary and then outputs S to all players.

FIGURE 5.13. MAC generation and covert procedures to generate auxiliary data.

of the offline phase consist of at least n_I input tuples, n_m multiplication triples, n_s squaring tuples and n_b shared bits.

In the first sub-phase, called the tuple-production sub-phase, players over-produce the various multiplication and squaring tuples, plus the shared bits. These are then "sacrificed" in the tuple-checking phase so as to create at least n_m multiplication triples, n_s squaring tuples and n_b shared bits. In particular, in the tuple-production phase players produce (at least) $2 \cdot n_m$ multiplication tuples, $2 \cdot n_s + n_b$ squaring tuples, and n_b shared bits. Tuple-production is performed by following the protocol in Figure 5.16 and Figure 5.17.

The functionality $\mathcal{F}_{\text{PREP}}$ (continued)

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is component-wise.

Input Production: On input $\mathsf{DataType} = (\mathsf{InputPrep}, n_I)$,

(1) The functionality chooses random values $I = \{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \notin A\}$.

(2) It accepts from the adversary corrupted values $\{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \in A\}$, corrupted shares $\{\mathbf{r}_k^{(i)} \in (\mathbb{F}_p)^{n_I} \mid k \in A, i \leq n\}$, and offset for data and MACs $\{\Delta_r^{(i)}, \Delta_\gamma^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \leq n\}$. Then it does the following:

(a) Sample honest shares $\{\mathbf{r}_k^{(i)} \mid k \notin A, i \leq n\}$ subject to $\mathbf{r}^{(i)} + \Delta_r^{(i)} = \sum_{k=1}^n \mathbf{r}_k^{(i)}$.

(b) Run macro $\mathsf{Angle}(\mathbf{r}_1^{(i)}, \ldots, \mathbf{r}_n^{(i)}, \Delta_\gamma^{(i)}, n_I)$, for $i \leq n$.

(c) Output $\{\mathbf{r}^{(i)}, (\mathbf{r}_i^{(j)}, \gamma_i(\mathbf{r}^{(j)}))_{j \leq n}\}$ to P_i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to P_i.

Multiplication Triples: On input $\mathsf{DataType} = (\mathsf{Triples}, n_m)$,

(1) The functionality chooses $2 \cdot n_m$ honest shares $I = \{(\mathbf{a}_i, \mathbf{b}_i) \in (\mathbb{F}_p)^{2 \cdot n_m} \mid i \notin A\}$.

(2) It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) \in (\mathbb{F}_p)^{3 \cdot n_m} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{3 \cdot n_m}\}$ from the adversary. It performs the following:

(a) Set $\mathbf{c} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{b}_1 + \cdots + \mathbf{b}_n)$.

(b) Compute a set of honest shares $\{\mathbf{c}_i \mid i \notin A\}$ subject to $\mathbf{c} = \sum_{i=1}^n \mathbf{c}_i$.

(c) Run the macros:
$\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_m)$,
$\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_m)$,
$\mathsf{Angle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, \Delta_\gamma^{(c)}, n_m)$.

(d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{b}_i, \gamma_i(\mathbf{b})), (\mathbf{c}_i, \gamma_i(\mathbf{c}))\}$ to P_i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to P_i.

Squaring Tuples: On input $\mathsf{DataType} = (\mathsf{Squares}, n_s)$,

(1) The functionality chooses $N = n_s$ honest shares $I = \{\mathbf{a}_i \in (\mathbb{F}_p)^{n_s} \mid i \notin A\}$.

(2) It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{s}_i) \in (\mathbb{F}_p)^{2 \cdot n_s} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(s)}) \in (\mathbb{F}_p)^{2 \cdot n_s}\}$ from the adversary. It does the following:

(a) Set $\mathbf{s} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{a}_1 + \cdots + \mathbf{a}_n)$.

(b) Compute a a set of honest shares $\{\mathbf{s}_i \mid i \notin A\}$ subject to $\mathbf{s} = \sum_{i=1}^n \mathbf{s}_i$.

(c) Run the macro $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_s)$ and $\mathsf{Angle}(\mathbf{s}_1, \ldots, \mathbf{s}_n, \Delta_\gamma^{(s)}, n_s)$.

(d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{s}_i, \gamma_i(\mathbf{s}))\}$ to P_i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to P_i.

Shared Bits: On input $\mathsf{DataType} = (\mathsf{Bits}, n_b)$,

(1) It gets shares $\{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \in A\}$ and MAC offsets $\{\Delta_\gamma^{(b)} \in (\mathbb{F}_p)^{n_b}\}$ from the adversary and does the following:

(a) Uniformly sample n_b honest shares $I = \{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \notin A\}$ subject to the condition $\sum_i \mathbf{b}_i \in \{0, 1\}^{n_b}$.

(b) Run the macro $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_b)$.

(c) Output $(\mathbf{b}_i, \gamma_i(\mathbf{b}))$ to P_i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to P_i.

FIGURE 5.14. Operations to generate auxiliary data for the online phase.

Protocol Reshare

Usage: Input: $\mathfrak{c}_{\mathbf{m}}$, where $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$ is a public ciphertext and a parameter *enc*, where $enc = \mathsf{NewCiphertext}$ or $enc = \mathsf{NoNewCiphertext}$.

Output: a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; if $enc = \mathsf{NewCiphertext}$, a ciphertext $\mathfrak{c}'_{\mathbf{m}}$. The idea is that $\mathfrak{c}_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $\mathfrak{c}'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $\mathfrak{c}_{\mathbf{m}}$ and $\mathfrak{c}'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $\mathfrak{c}'_{\mathbf{m}}$.

$\mathsf{Reshare}(\mathfrak{c}_{\mathbf{m}}, enc)$**:**

(1) The players run $\mathcal{F}_{\mathrm{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{f}_i$ and all players obtain $\mathfrak{c}_{\mathbf{f}_i}$, an encryption of $\mathbf{f}_i$.
(2) The players compute $\mathfrak{c}_{\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{f}_1} + \cdots + \mathfrak{c}_{\mathbf{f}_n}$, and $\mathfrak{c}_{\mathbf{m}+\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{m}} + \mathfrak{c}_{\mathbf{f}}$. Let $\mathbf{f} = \mathbf{f}_1 + \cdots + \mathbf{f}_n$ (notice that no party can compute $\mathbf{f}$).
(3) The players invoke DistDec to decrypt $\mathfrak{c}_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m} + \mathbf{f}$.
(4) P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i $(i \neq 1)$ sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
(5) If $enc = \mathsf{NewCiphertext}$, all players set $\mathfrak{c}'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f}) - \mathfrak{c}_{\mathbf{f}_1} - \cdots - \mathfrak{c}_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f})$.

FIGURE 5.15. The protocol for sharing $\mathbf{m} \in R_p$ on input $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$.

Procedure DataGen

Input Production: This produces at least $n_I \cdot n$ shared values $r_{i,j}$ for $1 \leq i \leq n_I$ and $1 \leq j \leq n$ such that P_j holds the actual value $r_{i,j}$ and all other players hold a sharing of this value only.

(1) For $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, \lceil 2 \cdot n_I/m \rceil\}$:
  (a) P_j generates $\mathbf{r} \in R_p$.
  (b) P_j computes $\mathfrak{c} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{r})$ and broadcasts the ciphertext to all players.
  (c) The players execute $\mathsf{Reshare}(\mathfrak{c}, \mathsf{NoNewCiphertext})$ so that P_i gets share $\mathbf{r}_i$ of $\mathbf{r}$.
  (d) The players compute $\mathfrak{c}_{\gamma(\mathbf{r})} \leftarrow \mathfrak{c}_{\mathbf{r}} \cdot \mathfrak{c}_{\alpha}$.
  (e) The players execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{r})}, \mathsf{NoNewCiphertext})$. to obtain shares $\gamma(\mathbf{r})_i$.
  (f) P_i decomposes the plaintext elements $\mathbf{r}_i$ and $\gamma(\mathbf{r})_i$ into their $m/2$ slot values via the FFT and locally stores the resulting data.
  (g) P_j does the same with $\mathbf{r}$ to obtain the values $r_{(k-1)\cdot m/2+i,j}$ for $i = 1, \ldots, m/2$.

Triples: This produces at least $2 \cdot n_m$ $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

(1) For $k \in \{1, \ldots, \lceil 4 \cdot n_m/m \rceil\}$:
  (a) The players run $\mathcal{F}_{\mathrm{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
  (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
  (c) The players run $\mathcal{F}_{\mathrm{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{b}_i$ and all players obtain $\mathfrak{c}_{\mathbf{b}_i}$ an encryption of $\mathbf{b}_i$.
  (d) The players compute $\mathfrak{c}_{\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{b}_1} + \cdots + \mathfrak{c}_{\mathbf{b}_n}$ We define $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$, although no party can compute $\mathbf{b}$.
  (e) The players compute $\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{b}}$.
  (f) The players execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}}, \mathsf{NewCiphertext})$ so that P_i obtains the share $\mathbf{c}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{c}}$ encrypting the plaintext $\mathbf{c} = \mathbf{c}_1 + \cdots + \mathbf{c}_n$.
  (g) The players compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$, $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{c})} \leftarrow \mathfrak{c}_{\mathbf{c}} \cdot \mathfrak{c}_{\alpha}$.
  (h) The players execute:
  $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$,
  $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$, and
  $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{c})}, \mathsf{NoNewCiphertext})$
  to obtain shares $\gamma(\mathbf{a})_i$, $\gamma(\mathbf{b})_i$ and $\gamma(\mathbf{c})_i$.
  (i) P_i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ multiplication triples.

FIGURE 5.16. Production of tuples and shared bits.

Procedure DataGen

Squares: This produces at least $(2 \cdot n_s + n_b)$ $\langle \cdot \rangle$-shared values (a_j, b_j) such that $b_j = a_j \cdot a_j$.

(1) For $k \in \{1, \ldots, \lceil 2 \cdot (2 \cdot n_s + n_b)/m \rceil\}$:

(a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.

(b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.

(c) The players compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.

(d) The players execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}^2}, \mathsf{NewCiphertext})$ so that P_i obtains the share $\mathbf{b}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{b}}$ encrypting the plaintext $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$.

(e) The players compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$.

(f) The players execute:
$\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$, and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$
to obtain shares $\gamma(\mathbf{a})_i$ and $\gamma(\mathbf{b})_i$.

(g) P_i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ squaring tuples.

Bits: This produces at least n_b $\langle \cdot \rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

(1) For $k \in \{1, \ldots, \lceil 2 \cdot n_b/m \rceil + 1\}$:[a]

(a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.

(b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.

(c) The players compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.

(d) The players invoke protocol $\mathsf{DistDec}$ to decrypt $\mathfrak{c}_{\mathbf{a}^2}$ and thereby obtain $\mathbf{s} = \mathbf{a}^2$.

(e) If any slot position in $\mathbf{s}$ is equal to zero then set it to one. .

(f) A fixed square root $\mathbf{t}$ of $\mathbf{s}$ is taken, say the one for which each slot position is odd when represented in $\{1, \ldots, p-1\}$.

(g) Compute $\mathfrak{c}_{\mathbf{v}} \leftarrow \mathbf{t}^{-1} \cdot \mathfrak{c}_{\mathbf{a}}$, an encryption of $\mathbf{v} = \mathbf{t}^{-1} \cdot \mathbf{a}$, a message for which each slot position contains $\{-1, 1\}$, bar the one which we replaced in step (1e).

(h) The players compute $\mathfrak{c}_{\gamma(\mathbf{v})} \leftarrow \mathfrak{c}_{\mathbf{v}} \cdot \mathfrak{c}_{\alpha}$.

(i) The players execute:
$\mathsf{Reshare}(\mathfrak{c}_{\mathbf{v}}, \mathsf{NoNewCiphertext})$, and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{v})}, \mathsf{NoNewCiphertext})$
to obtain shares $\mathbf{v}_i$ and $\gamma(\mathbf{v})_i$.

(j) P_i decomposes the various plaintext elements into their slot values via the FFT, bar the ones replaced in step (1e) to obtain $\langle v_j \rangle$ for $j = 1, \ldots, B$ where $B \approx m \cdot (p-1)/(2 \cdot p)$.

(k) Set $\langle b_j \rangle \leftarrow (1/2) \cdot (\langle v_j \rangle + 1)$ and output $\langle b_j \rangle$.

[a] Notice that in the production of shared bits the number of rounds is one more than would be expected at first glance: this is because some entries of the input vector may be equal to zero, which would render those entries unusable for the procedure. This event happens with probability $1/p$, so the expected number of bits produced per iteration is $m \cdot (p-1)/(2 \cdot p)$, rather than $m/2$ (as in the case that no entries are zero). Therefore, in order to produce at least n_b elements, we add an extra round to the procedure.

FIGURE 5.17. Production of tuples and shared bits (continued).

The tuple production protocol can be run repeatedly, alongside the tuple-checking sub-phase and the online phase.

The second sub-phase of the offline phase is to check whether the resulting material from the preceding phase has been produced correctly. This check is needed, because the distributed decryption procedure needed to produce the tuples and the MACs could allow the adversary to introduce errors. We solve this problem via a sacrificing technique, as in [DPSZ12], and adapt it to the case of squaring tuples and bit-sharings. Moreover, this sacrificing is performed in the offline phase as opposed to in the online phase (as in [DPSZ12]); and the resulting partially opened values are checked in the offline phase (again as opposed to the online phase). This is made possible by our protocol MACCheck which allows the verification of MACs without revealing the MAC key α. The tuple-checking protocol is presented in Figure 5.18.

Procedure DataCheck

Usage: Note that all players have previously agreed on two common random values t_m, t_{sb}.

Checking Multiplication Triples: This produces at least n_m checked $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

(1) For $k \in \{1, \ldots, n_m\}$:
   (a) Take two unused multiplication tuples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the list determined earlier.
   (b) Partially open $t_m \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ and $\langle b\rangle - \langle g\rangle$ to obtain σ.
   (c) Partially open $t_m \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$ to obtain τ.
   (d) If $\tau \neq 0$, output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ as a valid multiplication triple.

Checking Squaring Tuples: This produces at least n_s checked $\langle\cdot\rangle$-shared values (a_j, b_j) such that $b_j = a_j^2$.

(1) For $k \in \{1, \ldots, n_s\}$:
   (a) Take two unused squaring tuples $(\langle a\rangle, \langle b\rangle), (\langle f\rangle, \langle h\rangle)$ from the previous list.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Partially open $t_{sb}^2 \cdot \langle b\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle)$ as a valid squaring tuple.

Checking Shared Bits: This produces at least n_b checked $\langle\cdot\rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

(1) For $k \in \{1, \ldots, n_b\}$:
   (a) Take an unused squaring tuple $(\langle f\rangle, \langle h\rangle)$ and an unused bit sharing $\langle a\rangle$ from the lists determined earlier.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Partially open $t_{sb}^2 \cdot \langle a\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $\langle a\rangle$ as a valid bit sharing.

FIGURE 5.18. Checking the output of the data production procedure.

We show that the resulting protocol Π_{PREP}, detailed in Figure 5.19, securely implements the functionality $\mathcal{F}_{\text{PREP}}$, which models the offline phase. The functionality $\mathcal{F}_{\text{PREP}}$ outputs some desired number of multiplication triples, squaring tuples and shared bits.

Protocol Π_{PREP}

Usage: Note that DataGeneration can be run in four distinct threads, and DataCheck in two threads with one thread executing the Square and Shared Bit checking at the same time. Each thread executes its own check for correct broadcasting using Section 5.1.1.

Initialise: This produces the keys for encryption and MACs. On input (Start, p) from all the players:

(1) The players call $\mathcal{F}_{\text{SHE}}$ on query KeyGen so player i obtains $(\mathsf{sk}_i, \mathsf{pk}, \mathfrak{enc})$.
(2) The players call $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(\mathbb{F}_p)$ so player j obtains a share α_j of the MAC key, and all players get $\mathfrak{c}_i$, and encryption of α_i, for $1 \leq i \leq n$.
(3) All players set $\mathfrak{c}_\alpha \leftarrow \mathfrak{c}_1 + \cdots + \mathfrak{c}_n$.

Data Generation: On input $(\mathsf{DataGen}, n_I, n_m, n_s, n_b)$, the players execute the following sub-procedures of DataGen from Figure 5.16 and Figure 5.17:

(1) $\mathsf{InputProduction}(n_I)$
(2) $\mathsf{Triples}(n_m)$
(3) $\mathsf{Squares}(n_s)$
(4) $\mathsf{Bits}(n_b)$

Data Check: On input DataCheck, the players do the following:

(1) Generate two random values t_m, t_{sb} by running the steps below twice:
  (a) P_i samples random $t_i \leftarrow \mathbb{F}_p$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^t \leftarrow \mathsf{Commit}(t_i)$.
  (b) P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^t)$ and all players obtain t_j for $1 \leq j \leq n$.
  (c) Every player sets $t \leftarrow t_1 + \cdots + t_n$. If $t = 0$, then repeat the previous steps.
(2) Execute $\mathsf{DataCheck}(t_m, t_{sb})$.

Finalise: For the set of partially opened values run protocol MACCheck from Figure 5.12.

Abort: If $\mathcal{F}_{\text{SHE}}$ outputs a set S of corrupted players at any time, all players output S, and the protocol aborts.

FIGURE 5.19. The preprocessing phase.

THEOREM 5.9. *In the $(\mathcal{F}_{\text{SHE}}, \mathcal{F}_{\text{COMMIT}})$-hybrid model, the protocol Π_{PREP} implements $\mathcal{F}_{\text{PREP}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

The security flavour of Π_{PREP} follows the one of EncCommit, i.e. the covert (resp. active) version of EncCommit yields to covert (resp. active) security for Π_{PREP}.

Proof. We construct a simulator $\mathcal{S}_{\text{PREP}}$ (given in Figure 5.20 and Figure 5.21) such that no polynomial-time environment can distinguish, with significant probability, a view obtained by running Π_{PREP} from a view obtained by running $\mathcal{S}_{\text{PREP}} \diamond \mathcal{F}_{\text{PREP}}$. The environment's view is the collection of all intermediate messages that corrupted players send and receive, plus the inputs and outputs of all players.

In a nutshell, the simulator will run a copy of Π_{PREP} with the adversary, in which it will act on behalf of honest players. Keys for the underlying cryptosystem and MACs are generated by simulating queries KeyGen and EncCommit to $\mathcal{F}_{\text{SHE}}$ respectively. Note that due to the distributed decryption, data for the (online) input preparation stage might be incorrectly secret shared, and any type of data might be incorrectly MAC'd. Since the simulator knows α and sk, it can compute offsets on the secret sharing and MACs and pass them to $\mathcal{F}_{\text{PREP}}$.

First, we explain how the cheat mechanism is handled in the simulation. In the execution of Π_{PREP}, the environment may send Cheat either in the initial query KeyGen or in any later query EncCommit to $\mathcal{F}_{\text{SHE}}$. Thus, the success probability depends on the number of cheat attempts. The simulator ensures two things:

(1) Whenever the environment sends the *first* Cheat to what it thinks is $\mathcal{F}_{\text{SHE}}$, the call is forwarded to $\mathcal{F}_{\text{PREP}}$, which decides whether or not the attempt is successful.

The simulator $\mathcal{S}_{\text{PREP}}$

Initialise:

- The simulator first sends (Start, p) to $\mathcal{F}_{\text{PREP}}$ and then interacts with the adversary acting as $\mathcal{F}_{\text{SHE}}$ on query KeyGen to generate the encryption public key $(\mathsf{pk}, \mathfrak{enc})$ and a complete set of shares $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_n\}$ of the secret key. If the adversary sends Cheat to $\mathcal{F}_{\text{SHE}}$, the simulator forwards it to $\mathcal{F}_{\text{PREP}}$. If the cheat is successful, the simulator sets the flag BreakDown to true; otherwise, it is set to false.
- The generation of the MAC key α is performed as in the protocol, but calling SEncCommit($\mathbb{F}_p$) instead of $\mathcal{F}_{\text{SHE}}$ on query EncCommit. The simulator stores $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ for later use.
- Lastly, it gives α_i to $\mathcal{F}_{\text{PREP}}$ for $i \in A$ if BreakDown is false, and for $i \leq n$ otherwise.
- If the simulation of $\mathcal{F}_{\text{SHE}}$ aborts on KeyGen or EncCommit, it goes to "Abort".

Command = DataGen: On input (n_I, n_m, n_s, n_b), the simulator sets:

- $\mathcal{T}_{Input} \leftarrow$ SimDataGen(InputPrep, n_I),
- $\mathcal{T}_{Triples} \leftarrow$ SimDataGen(Triples, n_m),
- $\mathcal{T}_{Squares} \leftarrow$ SimDataGen(Squares, n_s),
- $\mathcal{T}_{Bits} \leftarrow$ SimDataGen(Bits, n_b),

where SimDataGen is specified in Figure 5.21. These calls also return a decision bit. If it is set to Abort, the simulator goes to "Abort".

Command = DataCheck:

- Step 1 is executed as in the protocol, but calling to SEncCommit(R_p). The simulator goes to "Abort" if SEncCommit says so.
- The simulator performs steps (a)-(d) of sub-procedures Triples, Squares, Bits of DataCheck. In each iteration k, it gets to know the value σ_k. If any of these values are non-zero, the simulator sends Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$. Otherwise, the algebraic relation between generated data holds with probability $1 - 1/p$.

Finalise: At this point, the functionality is waiting for instruction Proceed or Abort, or a complete break down occurred, and the functionality is waiting for command DataGen upon which it will output values from the adversary.

(1) The simulator engages with the adversary in a normal run of MACCheck on behalf of each honest P_i. Note that to generate honest σ_i the simulator uses shares α_i. If $\sigma_1 + \cdots + \sigma_n \neq 0$, it sends Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$.

(2) Otherwise, it sends Success to the adversary, and sends to $\mathcal{F}_{\text{PREP}}$ the following:
   - If BreakDown is false, it sends $\mathcal{T}_{Input}$, $\mathcal{T}_{Triples}$, $\mathcal{T}_{Squares}$, $\mathcal{T}_{Bits}$.
   - If BreakDown is true, it sends all the data (corresponding to honest and corrupt players) generated in the execution of SimDataGen.

Abort: If the simulated $\mathcal{F}_{\text{SHE}}$ aborts outputting a set S of corrupt players, the simulator sends Abort and S to $\mathcal{F}_{\text{PREP}}$.

FIGURE 5.20. The simulator $\mathcal{S}_{\text{PREP}}$ for the preprocessing phase.

(2) If the cheat was successful, the simulator recreates the success probability that a real interaction would have. This is needed as otherwise the environment would be able to distinguish a real interaction. The inner procedure SEncCommit is designed for this purpose.

We now show indistinguishability. There is one main difference between a simulated run and a real execution of Π_{PREP}: in a simulated run, honest shares used in the interaction are randomly sampled by the simulator. These shares correspond to the MAC key, and shares of generated data together with the shares of their MACs. At the end of the day, $\mathcal{F}_{\text{PREP}}$ will output data using its own honest shares of α, and its own honest shares of data and MACs.

The view of the environment can be divided into four chunks. Namely, messages interchanged in DataGen, DataCheck, or MACCheck, and players' output of $\mathcal{F}_{\text{PREP}}$. Clearly, indistinguishability of simulated and real views of the DataGen chunk comes from the semantic security of the underlying cryptosystem. For the DataCheck chunk, note that all opened values are a combination of output data and sacrificed data. The latter does not form part of the final

The simulator $\mathcal{S}_{\text{PREP}}$

SimDataGen(DataType): This procedure prepares the data to be inputted to $\mathcal{F}_{\text{PREP}}$.

DataType = InputPrep**:**

- The simulator engages in a normal run of steps (a)-(g) calling to SReshare instead of Reshare. If, at any point, some of the calls returned Abort, the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{Input} \leftarrow \varnothing$.
- Otherwise, all the rounds were successful. The simulator sets Decision $\leftarrow$ Continue. Note that in step (c) (after unpacking all the rounds), the simulator gets players' shares and MAC shares $\{\hat{\mathbf{r}_k}^{(i)}, \gamma_k^{(i)} \in (\mathbb{F}_p)^{2 \cdot n_I} \mid i, k \leq n\}$. Then, $\hat{\mathbf{r}}^{(i)} = \sum_k^n \hat{\mathbf{r}}_k^{(i)}$ is (presumably) the input of P_i. The simulator has the secret key, so it can get the real input $\mathbf{r}^{(i)}$ from the broadcast ciphertexts (if P_i is corrupt), or from what he generated (if P_i is honest). It computes offsets $\Delta_r^{(i)} \leftarrow \hat{\mathbf{r}}^{(i)} - \mathbf{r}^{(i)}$ and $\Delta_\gamma^{(i)} \leftarrow \sum_k^n \gamma_k^{(i)} - \alpha \cdot \mathbf{r}^{(i)}$.
  There are two possibilities:
  - Flag BreakDown is set to false. This means no cheat has occurred, so the simulator prepares corrupt inputs, corrupt shares and MAC shares, and offsets. That is, it sets $\mathcal{T}_{Input} \leftarrow \{\mathbf{r}^{(k)}, \hat{\mathbf{r}}_k^{(i)}, \Delta_r^{(i)}, \Delta_\gamma^{(i)}, \gamma_k^{(i)} \mid k \in A, i \leq n\}$
  - Flag BreakDown is set to true. Then, there was at least one successful cheat, and the functionality is waiting for adversary's contributions. The simulator sets $\mathcal{T}_{Input}$ to be the output of each player.

DataType = Triples, Squares, Bits**:** The simulator engages in a normal run of the subprocedure specified by DataType, but calling to SEncCommit(R_p) and SReshare$(\mathfrak{c}_{\mathbf{m}})$ instead of $\mathcal{F}_{\text{EncCommit}}$ and Reshare$(\mathfrak{c}_{\mathbf{m}})$. If any of the above macros returned Abort, the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{DataType} \leftarrow \varnothing$. In any other case the simulator sets Decision $\leftarrow$ Continue, handles the BreakDown flag as above, and does:

Triples**:** Set

$\mathcal{T}_{Triples} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \gamma(\mathbf{c})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{9 \cdot (2 \cdot n_m)} \mid i \leq n\}$. The shares are unpacked in step (i): corrupt shares are given by the adversary, and honest shares are sampled uniformly. MAC shares are produced after executing SReshare to simulate step (h), and the offsets are computed as explained earlier.

Squares**:** Set $\mathcal{T}_{Squares} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{6 \cdot (2 \cdot n_s + n_b)} \mid i \leq n\}$ Shares, MAC shares and offsets are obtained as explained above.

Bits**:** Set $\mathcal{T}_{Bits} \leftarrow \{(\mathbf{b}_i, \gamma_i, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{3 \cdot (2 \cdot n_b')} \mid i \leq n\}$. A number $n_b' \geq n_b$ of binary shares and MACs has been computed, The exact amount n_b' is round-dependent and it is expected to be approximately $(n_b + m/2) \cdot (p-1)/p$.

Output: The simulator returns (Decision, $\mathcal{T}_{DataType}$).

FIGURE 5.21. Internal procedures of the simulator $\mathcal{S}_{\text{PREP}}$.

output, and therefore by no means can the environment reconstruct the set of opened values using its view, as it does not know honest shares of the sacrificed data. In other words, openings are randomised via sacrificing from the environment's point of view, so the best it can do is to guess sacrificed honest shares, which has success probability $1/|\mathbb{F}_p|$ for each guess. For the MACCheck chunk, we refer to the fact that the soundness error of MACCheck is $2/p$, as shown in Lemma 5.8. Both these probabilities are negligible if p is exponential in the security parameter. Lastly, there is consistency between the output of $\mathcal{F}_{\text{PREP}}$ and what the environment sees in corrupted transcripts, since the offsets (those quantities denoted by Δ) are simply the difference between deviated and correctly computed data, and therefore are independent of what data refers to.

If the protocol aborts in DataCheck or MACCheck, the players output $\varnothing$, and so does $\mathcal{F}_{\text{PREP}}$ upon the instruction of the simulator. This corresponds to the fact that those protocols do not reveal the identity of any corrupted party.

The simulator $\mathcal{S}_{\text{PREP}}$

Macro SEncCommit(cond): This macro simulates a call to $\mathcal{F}_{\text{SHE}}$ on query EncCommit.

- The simulator receives corrupt seeds s_i from the adversary, when it thinks is interacting with $\mathcal{F}_{\text{SHE}}$, and computes $\mathbf{m}_i$ and $\mathfrak{c}_{\mathbf{m}_i}$ for $i \in A$ which are given to the adversary. Then, the simulator generates uniform $\mathbf{m}_i$ and $\mathfrak{c}_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_i)$ for $i \notin A$, and gives $\mathfrak{c}_i$ to the adversary. It waits for response Proceed, Cheat or Abort.
- If the adversary gives Proceed, the simulator sets Decision $\leftarrow$ Continue, and if the adversary gives Abort, it sets Decision $\leftarrow$ Abort and sends Abort to $\mathcal{F}_{\text{PREP}}$.
- If the adversary gives $(\text{Cheat}, \{\mathbf{m}_i^*, \mathfrak{c}_i^*\}_{i\in A})$, it sets $\mathbf{m}_i \leftarrow \mathbf{m}_i^*$, $\mathfrak{c}_i \leftarrow \mathfrak{c}_i^*$ for $i \in A$, and does the following:
  (1) If BreakDown is false, it sends Cheat to $\mathcal{F}_{\text{PREP}}$ and sets BreakDown to true. There are two possibilities:
    (a) The functionality returns Success: the simulator sets Decision $\leftarrow$ Continue.
    (b) The functionality returns NoSuccess: the simulator sets Decision $\leftarrow$ Abort.
  (2) If BreakDown is set to true, with probability $1/c$ it sets Decision $\leftarrow$ Continue; otherwise Decision $\leftarrow$ Abort.
- It returns $(\text{Decision}, \mathbf{m}_1, \ldots, \mathbf{m}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n)$.

Macro SReshare($\mathfrak{c}_{\mathbf{m}}$):

- The simulator sets $(\mathbf{f}_1, \ldots, \mathbf{f}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n) \leftarrow \text{SEncCommit}(R_p)$, $\mathbf{f} \leftarrow \sum_i \mathbf{f}_i$, and Decision $\leftarrow$ Abort if SEncCommit says so.
- Otherwise, it sets Decision $\leftarrow$ Continue and runs steps 2-5 of Reshare. Note that in step 3 the simulator might get an invalid value $(\mathbf{m}+\mathbf{f})^*$. It sets $\mathbf{m}_1 \leftarrow (\mathbf{m}+\mathbf{f})^* - \mathbf{f}_1$ and $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
- The simulator returns $(\text{Decision}, \mathbf{m}_1, \ldots, \mathbf{m}_n)$.

FIGURE 5.22. Internal procedures of the simulator $\mathcal{S}_{\text{PREP}}$ – Macros.

It remains to show what happens in case Cheat or Abort is sent by the environment. If the cheat is not successful, the players' output is a single message S for a set S of corrupted players in both real and simulated interaction. On the other hand, if the cheat went through, the functionality $\mathcal{F}_{\text{PREP}}$ breaks down, and the simulator can decide what MAC key is used and what data is outputted to every player, so it just gives to $\mathcal{F}_{\text{PREP}}$ what has been generated during the interaction. If the environment sends Abort and a set S of corrupted players, this is simply passed to $\mathcal{F}_{\text{PREP}}$, which forwards it to the players.

□

5.6. Online Phase

In this section we describe a protocol Π_{ONLINE} that implements $\mathcal{F}_{\text{ONLINE}}$ which performs the secure computation of the desired function, decomposed as a circuit over $\mathbb{F}_p$.

The online protocol makes use of the preprocessed data coming from $\mathcal{F}_{\text{PREP}}$ in order to input, add, multiply or square values. Our protocol is similar to the one described in [DPSZ12]; however, it brings a series of improvements (the "sacrificing" can be moved to the preprocessing phase, the online phase contains special procedures for squaring, etc., and the MAC-checking method allows the computation of reactive functionalities). The method for checking the MACs is simply the MACCheck protocol on all partially opened values; note that this method has a lower soundness error than that proposed in [DPSZ12], since the linear combination of partially opened values is truly random in our case, while in [DPSZ12] it has lower entropy.

The following theorem shows that the protocol Π_{ONLINE}, given in Figure 5.24, securely implements the functionality $\mathcal{F}_{\text{ONLINE}}$, which models the online phase.

THEOREM 5.10. *In the $\mathcal{F}_{\text{PREP}}$-hybrid model, the protocol Π_{ONLINE} implements $\mathcal{F}_{\text{ONLINE}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

Proof.

Functionality $\mathcal{F}_{\text{ONLINE}}$

Initialise: On input $(init, p, k)$ from all parties, the functionality stores $(domain, p, k)$ and waits for an input from the environment. Depending on this, the functionality does the following:

- Proceed: It sets BreakDown to false and continues.
- Cheat: With probability $1/c$, it sets BreakDown to true, outputs Success to the environment and continues. Otherwise, it outputs NoSuccess and proceeds as in Abort.
- Abort: It waits for the environment to input a set S of corrupt players, outputs it to the players, and aborts.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$. If BreakDown is true, it also outputs x to the environment.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$.

Square: On input $(square, varid_1, varid_2)$ from all parties (if $varid_1$ is present in memory and $varid_2$ is not), the functionality retrieves $(varid_1, x)$, and stores $(varid_2, x^2)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment.

- If BreakDown is false, the functionality waits for an input from the environment. If this input is Deliver then y is output to all players. Otherwise $\varnothing$ is output to all players.
- If BreakDown is true, the functionality waits for y^* from the environment and outputs it to all players.

FIGURE 5.23. The ideal functionality for MPC.

We construct a simulator $\mathcal{S}_{\text{ONLINE}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{ONLINE}}$, such that the adversary cannot distinguish whether it is playing with the protocol Π_{ONLINE} and $\mathcal{F}_{\text{PREP}}$, or the simulator and $\mathcal{F}_{\text{ONLINE}}$.

We now proceed with the analysis of the simulation, by first arguing that all the steps before the output are perfectly simulated and finally showing that the simulated output is statistically close to that of the protocol.

During initialisation, the simulator merely acts as $\mathcal{F}_{\text{PREP}}$ would, with the difference that the decision about the success of a cheating attempt is made by $\mathcal{F}_{\text{ONLINE}}$. If the cheating was successful, $\mathcal{F}_{\text{ONLINE}}$ will output all honest inputs, and the simulator can determine all outputs. Therefore, the simulation will precisely agree with the protocol. For the rest of the proof, we will assume that there was no cheating attempt.

In the input stage the values broadcast by the honest players are uniform in the protocol as well as in the simulation. Addition does not involve communication, while multiplication and squaring involve partial openings: in the protocol a partial opening reveals uniform values, and the same happens also in a simulated run. Moreover, MACs carry the same distribution in both the protocol and the simulation.

In the output stage of both the real and simulated run if the output y is delivered, the environment sees y and the honest players' shares, which are uniform and compatible with y and its MAC. Moreover, in a simulated run the output y is a correct evaluation of the function on the inputs provided by the players in the input phase. In order to conclude, we need to make sure that the same applies to the real protocol with overwhelming probability. As shown in Lemma 5.8, the adversary was able to cheat in one MACCheck call with probability $2/p$. Thus, the overall cheating probability is negligible since p is assumed to be exponential in the security parameter. This concludes the proof.

Protocol Π_{ONLINE}

Initialise: The parties call $\mathcal{F}_{\text{PREP}}$ to get the shares α_i of the MAC key, a number of multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, squares $(\langle a\rangle, \langle b\rangle)$, bits $\langle b\rangle$, and mask values $(r_i, \langle r_i\rangle)$ as needed for the circuit being evaluated. If $\mathcal{F}_{\text{PREP}}$ aborts outputting a set S of corrupt players, the players output S and abort. Then the operations specified below are performed according to the circuit.

Input: To share his input x_i, P_i takes an available mask value $(r_i, \langle r_i\rangle)$ and does the following:

(1) Broadcast $\varepsilon \leftarrow x_i - r_i$.

(2) The players compute $\langle x_i\rangle \leftarrow \langle r_i\rangle + \varepsilon$.

Add: On input $(\langle x\rangle, \langle y\rangle)$, the players locally compute $\langle x + y\rangle \leftarrow \langle x\rangle + \langle y\rangle$.

Multiply: On input $(\langle x\rangle, \langle y\rangle)$, the players do the following:

(1) In case the preprocessing outputs non-verified triples and random values $[\![t]\!]$:

(a) Take two unused multiplication tuples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$.

(b) Open a random value $[\![t]\!]$.

(c) Partially open $t \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ and $\langle b\rangle - \langle g\rangle$ to obtain σ.

(d) Evaluate and partially open $\langle \tau\rangle \leftarrow t \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$.

(e) If $\tau \neq 0$ then output $\varnothing$ and abort.

(f) Output $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ as a valid multiplication triple.

(2) Take a valid multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and open $\langle x\rangle - \langle a\rangle, \langle y\rangle - \langle b\rangle$ to get ε, ρ respectively.

(3) Locally each player sets $\langle z\rangle \leftarrow \langle c\rangle + \varepsilon \cdot \langle b\rangle + \rho \cdot \langle a\rangle + \varepsilon \cdot \rho$

Square: On input $\langle x\rangle$ the players do the following:

(1) Take a square pair $(\langle a\rangle, \langle b\rangle)$ and partially open $\langle x\rangle - \langle a\rangle$ so all players get ε.

(2) All players locally compute $\langle z\rangle \leftarrow \langle b\rangle + 2 \cdot \varepsilon \cdot \langle x\rangle - \varepsilon^2$.

Output: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle y\rangle$ has not yet been opened.

(1) The players call the MACCheck protocol on input all opened values so far. If it fails, they output $\varnothing$ and abort. $\varnothing$ represents the fact that the corrupt players remain undetected in this case.

(2) The players open $\langle y\rangle$ and call MACCheck on input y to verify its MAC. If the check fails, they output $\varnothing$ and abort; otherwise they accept y as a valid output.

FIGURE 5.24. Operations for secure function evaluation.

□

The observant reader will be wondering where the shared bits produced in the offline phase are used. These will be used in "higher level" versions of the online phase (i.e. versions which do not just evaluate an arithmetic circuit) which implement different types of operations presented in [**CS10**, **DFK+06**].

Simulator $\mathcal{S}_{\text{ONLINE}}$

Initialise: The simulation of the initialisation procedure is performed running a local copy of $\mathcal{F}_{\text{PREP}}$. Notice that all the data given to the adversary is known by the simulator.

If the environment inputs Proceed, Cheat, or Abort to the copy of $\mathcal{F}_{\text{PREP}}$, the simulator does so to $\mathcal{F}_{\text{ONLINE}}$ and forwards the output of $\mathcal{F}_{\text{ONLINE}}$ to the environment. If the output is Success, the simulator sets BreakDown to true and uses the environment's inputs as preprocessed data. If $\mathcal{F}_{\text{ONLINE}}$ outputs NoSuccess of the input was Abort, the simulator waits for input S from the environment, forwards it to $\mathcal{F}_{\text{ONLINE}}$, and aborts.

Input:

- If BreakDown is false, honest input is performed according to the protocol, with a dummy input, for example zero.
- If BreakDown is true, $\mathcal{F}_{\text{ONLINE}}$ outputs the inputs of honest players, which then can be used in the simulation.

For inputs given by a corrupt player P_i, the simulator waits for P_i to broadcast the (possibly incorrect) value ε', computes $x_i' \leftarrow r_i + \varepsilon'$ and uses x_i' as input to $\mathcal{F}_{\text{ONLINE}}$.

Add/Multiply/Square: These procedures are performed according to the protocol. The simulator also calls the respective procedure to $\mathcal{F}_{\text{ONLINE}}$.

Output: $\mathcal{F}_{\text{ONLINE}}$ outputs y to the simulator.

- If BreakDown is false, the simulator now has to provide the honest players' shares of such a value; it already computed an output value y', using the dummy inputs for the honest players, so it can select a random honest player and modify its share adding $y - y'$ and modify the MAC adding $\alpha(y - y')$, which is possible for the simulator, since it knows α. After that, the simulator is ready to open y according to the protocol. If y passes the check, the simulator sends Deliver to $\mathcal{F}_{\text{ONLINE}}$.
- If BreakDown is true, the simulator inputs the result of the simulation to $\mathcal{F}_{\text{ONLINE}}$.

FIGURE 5.25. Simulator for the online phase.

CHAPTER 6

SPDZ2 – Addenda

6.1. Parameters of the BGV Scheme

In this section we present an analysis of the parameters needed by the BGV to ensure that the distributed decryption procedure can decrypt the ciphertexts produced in the offline phase and that the scheme is "secure". Unlike [DPSZ12], which presents a worst case analysis, we use the expected case analysis used in [GHS12b].

6.1.1. Expected Values of Norms. Given an element $a \in R$ (represented as a polynomial) let $\|a\|_p$ be the standard p-norm of the coefficient vector (usually for $p = 1, 2$ or ∞). Let $\|a\|_p^{\mathsf{can}}$ be the p-norm of the same element when mapped into the canonical embedding i.e.

$$\|a\|_p^{\mathsf{can}} = \|\kappa(a)\|_p$$

where $\kappa(a) : R \longrightarrow \mathbb{C}^{\phi(m)}$ is the canonical embedding. The two key relationships are that

$$\|a\|_\infty \leq c_m \cdot \|a\|_\infty^{\mathsf{can}} \text{ and } \|a\|_\infty^{\mathsf{can}} \leq \|a\|_1,$$

for some constant c_m depending on m. Since m is a power of two then we have $c_m = 1$.

The *canonical embedding norm reduced modulo* q of an element $a \in R$ is the smallest canonical embedding norm of any a' which is congruent to a modulo q. It is denoted as

$$|a|_q^{can} = \min\{\ \|a'\|_\infty^{\mathsf{can}} :\ a' \in R,\ a' \equiv a \pmod q\ \}.$$

We also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q.

Following [GHS12b][Appendix A.5] we examine the variances of the different distributions used in our protocol. Let ζ_m denote any complex primitive m-th root of unity. Sampling $a \in R$ from $\mathcal{HWT}(h, \phi(m))$ and looking at $a(\zeta_m)$ produces a random variable with variance h; sampling from $\mathcal{ZO}(0.5, \phi(m))$ with variance $\phi(m)/2$; sampling from $\mathcal{DG}(\sigma^2, \phi(m))$ with variance $\sigma^2 \cdot \phi(m)$, and sampling from $\mathcal{U}(q, \phi(m))$ with variance $q^2 \cdot \phi(m)/12$. By the law of large numbers we can use $6 \cdot \sqrt{V}$, where V is the above variance, as a high probability bound on the size of $a(\zeta_m)$, and this provides a bound on the canonical embedding norm of a.

If we take a product of two, three, or four such elements with variances $V_1, V_2, \ldots, V_4$ we use $16 \cdot \sqrt{V_1 \cdot V_2}$, $9.6 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3}$ and $7.3 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3 \cdot V_4}$ as the resulting bounds since

$$\mathsf{erfc}(4)^2 \approx \mathsf{erfc}(3.1)^3 \approx \mathsf{erfc}(2.7)^4 \approx 2^{-50}.$$

6.1.2. Key Generation. We first need to establish the rough distributions (i.e. variances) of the resulting keys arising from our key generation procedure. For our purposes we are only interested in the variance of the associated distributions in the canonical embedding.

$$\begin{aligned}
\mathsf{Var}(\kappa(\mathsf{sk}_j)) &= n \cdot \mathsf{Var}(\kappa(\mathsf{sk}_{i,j})) = n \cdot h, \\
\mathsf{Var}(\kappa(a_j)) &= q_1^2 \cdot \phi(m)/12, \\
\mathsf{Var}(\kappa(\varepsilon_j)) &= n \cdot \mathsf{Var}(\kappa(\varepsilon_{i,j})) = n \cdot \sigma^2 \cdot \phi(m).
\end{aligned}$$

We also need to analyse the distributions of the randomness needed to produce $\mathfrak{enc}_j$. Here we assume that all parties follow the protocol and we are only interested in the output final extended public key: thus we write (dropping the j to avoid overloading the reader)

$$\mathfrak{enc} = (b_{\mathsf{sk},\mathsf{sk}^2}, a_{\mathsf{sk},\mathsf{sk}^2})$$

where

$$b_{\mathsf{sk},\mathsf{sk}^2} = a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} + p \cdot e_{\mathsf{sk},\mathsf{sk}^2} - p_1 \cdot \mathsf{sk}^2.$$

We also have

$$\begin{aligned}\mathfrak{enc}'_i &= (b \cdot v_i + p \cdot e_{0,i} - p_1 \cdot \mathsf{sk}_i, a \cdot v_i + p \cdot e_{1,i}) \\ \mathfrak{zero}_i &= (b \cdot v'_i + p \cdot e'_{0,i}, a \cdot v'_i + p \cdot e'_{1,i})\end{aligned}$$

where $(v_i, e_{0,i}, e_{1,i}) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$ and $(v'_i, e'_{0,i}, e'_{1,i}) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. Therefore

$$a_{\mathsf{sk},\mathsf{sk}^2} = \sum_{i=1}^{n} \mathsf{sk}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} (a \cdot v'_i + p \cdot e'_{1,i}),$$

and

$$\begin{aligned}
b_{\mathsf{sk},\mathsf{sk}^2} &= \sum_{i=1}^{n} \mathsf{sk}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathsf{sk}_j \right) + \sum_{i=1}^{n} (b \cdot v'_i + p \cdot e'_{0,i}) \\
&= a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} - \mathsf{sk} \cdot \sum_{i=1}^{n} \mathsf{sk}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} \mathsf{sk}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathsf{sk}_j \right) \\
&\quad + \sum_{i=1}^{n} \left((a \cdot \mathsf{sk} + p \cdot \varepsilon) \cdot v'_i + p \cdot e'_{0,i} \right) - \mathsf{sk} \cdot \sum_{i=1}^{n} (a \cdot v'_i + p \cdot e'_{1,i}) \\
&= a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} + \sum_{i=1}^{n} \left(\sum_{j=1}^{n} b \cdot v_j \cdot \mathsf{sk}_i + p \cdot e_{0,j} \cdot \mathsf{sk}_i - p_1 \cdot \mathsf{sk}_i \cdot \mathsf{sk}_j - \mathsf{sk} \cdot \mathsf{sk}_i \cdot a \cdot v_j - \mathsf{sk} \cdot \mathsf{sk}_i \cdot p \cdot e_{1,j} \right) \\
&\quad + p \cdot \sum_{i=1}^{n} \left(\varepsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathsf{sk} \right) \\
&= a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} + p \cdot \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\varepsilon \cdot v_j \cdot \mathsf{sk}_i + e_{0,j} \cdot \mathsf{sk}_i - \mathsf{sk} \cdot \mathsf{sk}_i \cdot e_{1,j}) + \varepsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathsf{sk} \right) - p_1 \cdot \mathsf{sk}^2 \\
&= a_{\mathsf{sk},\mathsf{sk}^2} \cdot \mathsf{sk} + p \cdot e_{\mathsf{sk},\mathsf{sk}^2} - p_1 \cdot \mathsf{sk}^2
\end{aligned}$$

where

$$e_{\mathsf{sk},\mathsf{sk}^2} = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\varepsilon \cdot v_j \cdot \mathsf{sk}_i + e_{0,j} \cdot \mathsf{sk}_i - \mathsf{sk} \cdot \mathsf{sk}_i \cdot e_{1,j}) + \varepsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathsf{sk} \right). \tag{6.1}$$

Thus the values $\mathfrak{enc}$ are indeed genuine "quasi-encryptions" of $-p_1 \cdot \mathsf{sk}^2$ with respect to the secret key sk and the modulus q_1. Equation 6.1 will be used later to establish the properties of the output of the $\mathsf{SwitchKey}$ procedure.

6.1.3. BGV Procedures. The "noise" of a given ciphertext $\mathfrak{c} = (c_0, c_1, \ell)$ is an upper bound on the value

$$\|c_0 - \mathsf{sk} \cdot c_1\|_\infty^{\mathsf{can}}.$$

$\underline{\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})}$: Given a fresh ciphertext $(c_0, c_1, 1)$, a bound (with high probability) on the output noise is

$$\begin{aligned}
\|c_0 - \mathsf{sk} \cdot c_1\|_\infty &\le \|c_0 - \mathsf{sk} \cdot c_1\|_\infty^{\mathsf{can}} \\
&= \|((a \cdot \mathsf{sk} + p \cdot \varepsilon) \cdot v + p \cdot e_0 + \mathbf{m} - (a \cdot v + p \cdot e_1) \cdot \mathsf{sk}\|_\infty^{\mathsf{can}} \\
&= \|\mathbf{m} + p \cdot (\varepsilon \cdot v + e_0 - e_1 \cdot \mathsf{sk})\|_\infty^{\mathsf{can}} \\
&\le \|\mathbf{m}\|_\infty^{\mathsf{can}} + p \cdot \left(\|\varepsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathsf{sk}\|_\infty^{\mathsf{can}} \right) \\
&\le \phi(m) \cdot p/2 + p \cdot \sigma \cdot \left(16 \cdot \phi(m) \cdot \sqrt{n/2} + 6 \cdot \sqrt{\phi(m)} + 16 \cdot \sqrt{n \cdot h \cdot \phi(m)} \right) = B_{\mathsf{clean}}.
\end{aligned}$$

Note this value of B_{clean} is different from that in [GHS12b] due to the different distributions resulting from the distributed key generation.

$\underline{\mathsf{SwitchModulus}((c_0, c_1), \ell)}$: If the input ciphertext has noise ν then the output ciphertext has noise ν' where

$$\nu' = \frac{\nu}{p_\ell} + B_{\mathsf{scale}}.$$

The value B_{scale} is an upper bound on the quantity $\|\tau_0 + \tau_1 \cdot \mathsf{sk}\|_\infty^{\mathsf{can}}$, where $\kappa(\tau_i)$ is drawn from a distribution which is close to a complex Gaussian with variance $\phi(m) \cdot p^2/12$. We can therefore take the upper bound to be (with high probability)

$$\begin{aligned} B_{\mathsf{scale}} &= 6 \cdot p \cdot \sqrt{\phi(m)/12} + 16 \cdot p \cdot \sqrt{n \cdot \phi(m) \cdot h/12}, \\ &= p \cdot \sqrt{3 \cdot \phi(m)} \cdot \left(1 + 8 \cdot \sqrt{n \cdot h}/3\right). \end{aligned}$$

Again, note the dependence on n (compared to [GHS12b]) as the secret key sk is selected from a distribution with variance $n \cdot h$, and not just h. Also note the dependence on p due to the plaintext space being defined mod p as opposed to mod 2 in [GHS12b].
$\underline{\mathsf{Dec}_{\mathsf{sk}}(\mathfrak{c})}$: As explained in [DPSZ12, GHS12b] this procedure works when the noise ν associated with a ciphertext satisfies $\nu = c_m \cdot \nu < q_\ell/2$.
$\underline{\mathsf{DistDec}_{\mathsf{sk}_i}(\mathfrak{c})}$: The value B is an upper bound on the noise ν associated with a ciphertext we decrypt in our protocols. To ensure valid distributed decryption we require

$$2 \cdot (1 + 2^{\mathsf{sec}}) \cdot B < q_\ell.$$

Given a value of B, we obtain a lower bound on p_0 by the above inequality. The addition of a random term with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ in the distributed decryption procedure ensures that the individual *coefficients* of the sum $\mathbf{t}_1 + \cdots + \mathbf{t}_n$ are statistically indistinguishable from random, with probability $2^{-\mathsf{sec}}$. This does not imply that the adversary has this probability of distinguishing the simulated execution in [DPSZ12] from the real execution, since each run consists of the exchange of $\phi(m)$ coefficients, and the protocol is executed many times over the execution of the whole protocol. However, we feel that setting concentrating solely on the statistical indistinguishability of the coefficients is valid in a practical context.
$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$: In order to estimate the size of the output noise term we need first to estimate the size of the term

$$\|p \cdot d_2 \cdot \varepsilon_{\mathsf{sk},\mathsf{sk}^2}\|_\infty^{\mathsf{can}}.$$

Using Equation 6.1 we find that

$$\begin{aligned} \|p \cdot d_2 \cdot e_{\mathsf{sk},\mathsf{sk}^2})\|_\infty^{\mathsf{can}}/q_0 \leq p \cdot \sqrt{\frac{\phi(m)}{12}} \cdot \Big[& n^2 \cdot \sigma \cdot \Big(7.3 \cdot \sqrt{n \cdot h \cdot \phi(m)^2/2} + 9.6 \cdot \sqrt{h \cdot \phi(m)} \\ & + 7.3 \cdot h \cdot \sqrt{n \cdot \phi(m)}\Big) \\ & + n \cdot \Big(9.6 \cdot \sigma \cdot \sqrt{n \cdot \phi(m)^2/2} + 16 \cdot \sigma \cdot \sqrt{\phi(m)} \\ & + 7.6 \cdot \sigma \cdot \sqrt{\phi(m) \cdot n \cdot h}\Big)\Big] \\ \leq p \cdot \phi(m) \cdot \sigma \cdot \Big[& n^{2.5} \cdot (1.49 \cdot \sqrt{h \cdot \phi(m)} + 2.11 \cdot h) + 2.77 \cdot n^2 \cdot \sqrt{h} \\ & + n^{1.5} \cdot (1.96 \cdot \sqrt{\phi(m)} + 2.77 \cdot \sqrt{h}) + 4.62 \cdot n\Big] \\ = B_{\mathsf{KS}}. \end{aligned}$$

Then if the input to $\mathsf{SwitchKey}$ has noise bounded by ν, the output noise value is bounded by

$$\nu + \frac{B_{\mathsf{KS}} \cdot q_0}{p_1} + B_{\mathsf{scale}}.$$

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: Combining all of the above, if we take two ciphertexts of level one with input noise bounded by ν and ν', the output noise level from multiplication is bounded by

$$\nu'' = \left(\frac{\nu}{p_1} + B_{\mathsf{scale}}\right) \cdot \left(\frac{\nu'}{p_1} + B_{\mathsf{scale}}\right) + \frac{B_{\mathsf{KS}}}{p_1} + B_{\mathsf{scale}}.$$

6.1.4. Application to the Offline Phase. In all of our protocols we evaluate the following circuit: we first add n ciphertexts together and perform a multiplication, giving a ciphertext with respect to modulus p_0 with noise

$$U_1 = \left(\frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}} \right)^2 + \frac{B_{\mathsf{KS}} \cdot p_0}{p_2} + B_{\mathsf{scale}}.$$

We then add on another n ciphertexts, which are added at level one and then reduced to level zero. We therefore obtain a final upper bound on the noise for our adversarially generated ciphertexts of

$$U_2 = U_1 + \frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}}.$$

To ensure valid (distributed) decryption, we require

$$2 \cdot U_2 \cdot (1 + 2^{\mathsf{sec}}) < p_0,$$

i.e. we take $B = U_2$ in our distributed decryption protocol.

This ensures valid decryption in our offline phase; however, we still need to select the parameters to ensure security. Following the analysis in [GHS12b] of the BGV scheme we set, for 128-bit security,

$$\phi(m) \geq 33.1 \cdot \log \left(\frac{q_1}{\sigma} \right).$$

Combining the various inequalities together, we can fix the $\sigma = 3.2$, $\mathsf{sec} = 40$ and $h = 64$; several choices of p, n yield the estimates in tables 6.1, 6.2 and 6.3 – these are the parameters choices which we use to generate the primes and rings in our implementation.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	8192	130	104	234	89
3	8192	132	104	236	90
4	8192	132	104	236	91
5	8192	132	106	238	90
6	8192	132	106	238	91
7	8192	132	108	240	91
8	8192	132	108	240	91
9	8192	132	110	242	91
10	8192	132	110	242	91
20	8192	134	110	244	93
50	8192	136	114	250	94
100	8192	136	116	252	95

TABLE 6.1. Parameters for $p \approx 2^{32}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	16384	196	136	332	154
3	16384	196	138	334	154
4	16384	196	140	336	155
5	16384	196	142	338	155
6	16384	198	140	338	156
7	16384	198	140	338	156
8	16384	198	140	338	157
9	16384	198	142	340	156
10	16384	198	142	340	156
20	16384	198	146	344	157
50	16384	200	148	348	158
100	16384	202	150	352	160

TABLE 6.2. Parameters for $p \approx 2^{64}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	32768	324	202	526	283
3	32768	326	202	528	285
4	32768	326	204	530	284
5	32768	326	204	530	285
6	32768	326	206	532	284
7	32768	326	206	532	285
8	32768	326	208	534	285
9	32768	326	208	534	285
10	32768	326	208	534	285
20	32768	328	210	538	286
50	32768	330	212	542	289
100	32768	330	216	546	288

TABLE 6.3. Parameters for $p \approx 2^{128}$.

6.2. Experimental Results

6.2.1. KeyGen and Offline Protocols. To present performance numbers for our key generation and new variant of the offline phase for SPDZ we first need to define secure parameter sizes for the underlying BGV scheme (and in particular how it is used in our protocols). This is done in Section 6.1, by the method of Appendices A.4, A.5 and B of [GHS12b], for various choices of n (the number of players) and p (the field size).

We implemented the preceding protocols in C++ on top of the MPIR library for multi-precision arithmetic. Modular arithmetic was implemented with bespoke code using Montgomery arithmetic [Mon85] and calls to the underlying `mpn_` functions in MPIR. The offline phase was implemented in a multi-threaded manner, with four cores producing initial multiplication triples, square pairs, shared bits and input preparation mask values. Then two cores performed the sacrificing for the multiplication triples, square pairs and shared bits.

In Table 6.4 we present execution times (in wall time measured in seconds) for key generation and for an offline phase which produces 100000 each of the multiplication tuples, square pairs, shared bits and 1000 input sharings. We also present the average time to produce a multiplication triple for an offline phase running on one core and producing 100000 multiplication triples only. The run-times are given for various values of n, p and c, and all timings were obtained on 2.80 GHz Intel Core i7 machines with 4 GB RAM, running on a local network.

n	$p \approx$	c	Run Times KeyGen	Run Times Offline	Time per Triple (sec)
2	2^{32}	5	2.4	156	0.00140
2	2^{32}	10	5.1	277	0.00256
2	2^{32}	20	10.4	512	0.00483
2	2^{64}	5	5.9	202	0.00194
2	2^{64}	10	12.5	377	0.00333
2	2^{64}	20	25.6	682	0.00634
2	2^{128}	5	16.2	307	0.00271
2	2^{128}	10	33.6	561	0.00489
2	2^{128}	20	74.5	1114	0.00937
3	2^{32}	5	3.0	292	0.00204
3	2^{32}	10	6.4	413	0.00380
3	2^{32}	20	13.3	790	0.00731
3	2^{64}	5	7.7	292	0.00267
3	2^{64}	10	16.3	568	0.00497
3	2^{64}	20	33.7	1108	0.01004
3	2^{128}	5	21.0	462	0.00402
3	2^{128}	10	44.4	889	0.00759
3	2^{128}	20	99.4	2030	0.01487

TABLE 6.4. Execution Times For Key Gen and Offline Phase (Covert Security)

In the case of covert security the authors of [DKL+12] report figures of 0.002 seconds per (un-checked) 64-bit multiplication triple for both two and three players; however, the probability of cheating being detected was lower bounded by 1/2 for two players, and 1/4 for three players, in contrast to our probabilities of 4/5, 9/10 and 19/20. Since the triples in [DKL+12] were unchecked we need to scale their run-times by a factor of two, yielding to an estimate of 0.004 seconds per multiplication triple. Thus for covert security we see that our protocol for checked tuples is superior in terms of error probabilities for a comparable run-time.

For the active security variant we aimed for a cheating probability of 2^{-40}, so as to be able to compare with run times previously obtained in [DKL+12], which used the method from [DPSZ12]. Again we performed two experiments: one where four cores produced 100000 multiplication triples, squaring pairs and shared bits, plus 1000 input sharings, and one where one core produced just 100000 multiplication triples (in order to estimate the average cost for a triple). The results are in Table 6.5.

$p \approx$	$n = 2$ Offline	$n = 2$ Time per Triple	$n = 3$ Offline	$n = 3$ Time per Triple
2^{32}	2366	0.01955	3668	0.02868
2^{64}	3751	0.02749	5495	0.04107
2^{128}	6302	0.04252	10063	0.06317

TABLE 6.5. Execution Times For Offline Phase (Active Security)

For comparison: for a prime of 64 bits, the authors of [DKL+12] report on an implementation which takes 0.006 seconds to produce an (un-checked) multiplication triple for the case of two parties and active security, and 0.008 seconds per multiplication triple for the case of three parties and active security. Since we produce checked triples, the cost per triple for the results in [DKL+12] need to be (at least) doubled, to give totals of 0.012 and 0.016 seconds respectively.

Thus, in this test, our new active protocol has running time about twice that of the previous active protocol from [DPSZ12] based on ZKPoKs. From analysis of the protocols, we do expect that the new method will be faster, but only if we produce the output in large enough batches. Due to memory constraints we were so far unable to do this, but we can extrapolate from these results: in the test we generated 12 ciphertexts in one go, and if we were able to increase this by a factor of about 10, then we would get results better than those of [DPSZ12, DKL+12], all other things being equal. More information can be found in Section 6.3.

6.2.2. Online. For the new online phase we have developed a purpose-built bytecode interpreter, which reads and executes pre-generated sequences of instructions in a multi-threaded manner. Our runtime supports parallelism on two different levels: independent rounds of communication can be merged together to reduce network overhead, and multiple threads can be executed at once to allow for optimal usage of modern multi-core processors.

Each bytecode instruction is either some local computation (e.g. addition of secret shared values) or an 'open' instruction, which initiates the protocol to reveal a shared value. The data from independent open instructions can be merged together to save on communication costs. Each player may run up to four different bytecode files in parallel in distinct threads, with each such thread having access to some shared memory resource. The advantage of this approach is that bytecode files can be pre-compiled and optimized, and then quickly loaded at runtime – the online phase runtime is itself oblivious to the nature of the programs being run.

In Table 6.6 we present timings (again in elapsed wall time for a player) for multiplying two secret shared values. Results are given for three different varieties of multiplication, reflecting the possibilities available: purely sequential multiplications, parallel multiplications with communication merged into one round (50 per round), and parallel multiplications running in 4 independent threads (50 per round, per thread). The experiments were carried out on the same machines as were used for the offline phase, running over a local network with a ping of around 0.27ms. For comparison, the original implementation of the online phase in [DPSZ12] gave an amortized rate of 20000 multiplications per second over a 64-bit prime field, with three players.

		Multiplications/sec		
		Sequential	50 in Parallel	
n	$p \approx$	Single Thread	Single Thread	Four Threads
2	2^{32}	7500	134000	398000
2	2^{64}	7500	130000	395000
2	2^{128}	7500	120000	358000
3	2^{32}	4700	100000	292000
3	2^{64}	4700	98000	287000
3	2^{128}	4600	90000	260000

TABLE 6.6. Online Times

6.3. Active Security

The following is a sketch of a method for an actively secure version of $\Pi_{\text{EncCommit}}$. More specifically, we assume players have access to an ideal functionality $\mathcal{F}^A_{\text{KeyGen}}$ which generates the key material like $\mathcal{F}_{\text{KeyGen}}$, but it models active security rather than covert security. More concretely, this just means that there is no "cheat option" that the adversary can choose. The purpose of this section is therefore to describe a protocol $\Pi^A_{\text{EncCommit}}$ which securely implements

an ideal functionality $\mathcal{F}_{\text{SHE}}^{A}$ in the $\mathcal{F}_{\text{KeyGen}}^{A}$-hybrid model, where $\mathcal{F}_{\text{SHE}}^{A}$ behaves as $\mathcal{F}_{\text{SHE}}$, but, again, models active security.

The protocol is inspired by the protocol from [NO09] where a particularly efficient variant of the cut-and-choose approach was developed.

Let P_i be the player producing ciphertexts to be verified by the other players. The protocol is parametrized by two natural numbers T, b where b divides T. We set $t = T/b$. The protocol produces as output t ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$.

Each such ciphertext is generated according to the algorithm described earlier, and is therefore created from the public key and four polynomials m, v, e_0 and e_1. To make the notation easier to deal with below, we rename these as f_1, f_2, f_3, f_4. We can then observe that there exist ρ_l, for $l = 1, \ldots, 4$ such that $\|f_l\|_\infty \leq \rho_l$ except with negligible probability. Concretely, we can use $\rho_1 = p/2$, $\rho_2 = 1$ and $\rho_3 = \rho_4 = \rho$ where ρ can be determined by a tail-bound on the Gaussian distribution used for generating f_3, f_4.

Each player P_i also creates a set of random *reference ciphertexts* $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ that are used to verify that $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ are well-formed and that P_i knows what they contain. Each $\mathfrak{d}_j$ is created from 4 polynomials $g_1, \ldots, g_4$ in the same way as above, but the polynomials are created with a different distribution. Namely, they are random subject to $\|g_i\|_\infty \leq 4 \cdot \delta \cdot \rho_i \cdot T \cdot \phi(m)$, where $\delta > 1$ is some constant.

The protocol proceeds as follows:

(1) P_i is given some number of attempts to prove that his ciphertexts are correctly formed. The protocol is parametrised by a number M which is the maximal number of allowed attempts. We start by setting a counter $v = 1$.

(2) P_i broadcasts the ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and the reference ciphertexts $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ containing plaintexts. These ciphertexts should be generated from seeds $s_0, \ldots, s_{2T-1}$ that are first sent through the random oracle whose output is used to generate the plaintext and randomness for the encryptions.

(3) A random index subset of size T is chosen, and P_i must broadcast s_i for $i \in T$. Players check that each opened s_i indeed induces the ciphertext $\mathfrak{d}_i$, and abort if this is not the case.

(4) A random permutation π on T items is generated and the unopened ciphertexts are permuted according to π. We renumber the permuted ciphertexts and call them $\mathfrak{d}_0, \ldots, \mathfrak{d}_{T-1}$.

(5) Now, for each $\mathfrak{c}_i$, the subset of ciphertexts $\{\mathfrak{d}_{bi+j} |\ j = 0, \ldots, b-1\}$ is used to demonstrate that $\mathfrak{c}_i$ is correctly formed. This is called the block of ciphertexts assigned to $\mathfrak{c}_i$. We do as follows:
   (a) For each i, j do the following: let $f_1, \ldots, f_4$ and $g_1, \ldots, g_4$ be the polynomials used to form $\mathfrak{c}_i$, respectively $\mathfrak{d}_{bi+j}$. Define $z_l = f_l + g_l$, for $l = 1, \ldots, 4$.
   (b) Player P_i checks that $\|z_l\|_\infty \leq 4 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m) - \rho_l$. If this is the case, he broadcasts z_l, for $l = 1, \ldots, 4$. Otherwise he broadcasts $\perp$.
   (c) In the former case players check that $\|z_l\|_\infty$ is in range for $l = 1, \ldots, 4$ and that the z_l's induce the ciphertext $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$.
   (d) At the end, players verify that for each $\mathfrak{c}_i$, P_i has correctly opened $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ for all ciphertexts in the block assigned to $\mathfrak{c}_i$.
   (e) If all checks go through, output $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and exit. Else, if $v < M$, increment v and go to step 2. Finally, if $v = M$, the prover has failed to convince us M times, so abort the protocol.

It is possible to adapt the protocol for proving that the plaintexts in $\mathfrak{c}_i$ satisfy certain special properties. For instance, assume we want to ensure that the plaintext polynomial f_1 is a constant polynomial, i.e., only the degree-0 coefficient is non-zero. We do this by generating the reference ciphertexts such that for each $\mathfrak{d}_i$, the polynomial g_1 is also a constant polynomial. When opening we check that the plaintext polynomial is always constant. The proof of security is trivially adapted for this case.

Some intuition for why this works: after half the reference ciphertexts are opened, we know that except with exponentially small probability, almost all the unopened ciphertexts are well formed. A simulator will be able to extract randomness and plaintext for all the well formed ones. When we split the unopened $\mathfrak{d}_j$'s randomly in blocks of b ciphertexts, it is therefore very unlikely that some block contains only bad ciphertexts. It can be shown that the probability that this happens is at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$ [NO09].

Assume P_i is corrupt: now, if he survives one iteration of the test, and no block was completely bad, it follows that for every $\mathfrak{c}_i$, he has opened opened at least one $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ where $\mathfrak{d}_{bi+j}$ was well formed. The simulator can therefore extract a way to open $\mathfrak{c}_i$ since $\mathfrak{c}_i = (\mathfrak{c}_i + \mathfrak{d}_{bi+j}) - \mathfrak{d}_{bi+j}$. It will be able to compute polynomials f_l for $\mathfrak{c}_i$ with $\|f_l\|_\infty \leq 8 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m)$. Therefore, if some $\mathfrak{c}_i$ is not of this form, the prover can survive one iteration of the test with probability at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$. To survive the entire protocol, the prover needs to win in at least one of the M iterations, and this happens with probability at most $M \cdot t^{1-b} \cdot (e \cdot \ln(2))^{-b}$, by the union bound.

Assume P_i is honest: then when he decides whether to open a given ciphertext, the probability that a single coefficient is in range is $\frac{1}{4 \cdot \delta \cdot \phi(m) \cdot T}$. There are $4 \cdot \phi(m)$ coefficients in a single ciphertext and up to T ciphertexts to open, so by a union bound, P_i will not need to send $\perp$ at all, except with probability $1/\delta$. The probability that an honest prover fails to complete the protocol is hence $(1/\delta)^M$. We therefore see that the completeness error vanishes exponentially with increasing M, and in the soundness probability, we only lose $\log M$ bits of security.

It is easy to see that for each opening done by an honest prover, the polynomials z_l will have coefficients that are uniformly distributed in the expected range, so the protocol can be simulated.

Finally, note that in a normal run of the protocol, only one iteration is required, except with probability $1/\delta$. So in practice, what counts for the efficiency is the time we spend on one iteration.

In our experiments we implemented the above protocol with the following parameter choices: $\delta = 256$, $M = 5$, $t = 12$ and $b = 16$. This guaranteed a cheating probability of 2^{-40}, as well as the probability of an honest prover failing of 2^{-40}. In addition, the choice of $t = 12$ was to ensure that each run of the protocol created enough ciphertexts to be run in two executions of the main loop of the multiplication triple production protocol. By increasing t and decreasing b one can improve the amortized complexity of the protocol while keeping the error probabilities the same. This comes at the cost of increased memory usage, primarily because decreasing b by, say, a half means that t needs to be replaced by essentially t^2. On our test machines $t = 12$ seemed to provide the best compromise.

Part 2

Zero-Knowledge Protocols

CHAPTER 7

Zero-Knowledge Protocols for Multiplicative Relations

7.1. Introduction

The notions of commitment schemes and zero-knowledge proofs (ZK proofs) are among the most fundamental in the theory and practice of cryptographic protocols. Intuitively, a commitment scheme provides a way for a prover to put a value x in a locked box and commit to x by giving this box $[x]$ to a verifier. Later the prover can choose to open the box by giving away the key to the box.

In a zero-knowledge protocol, a prover wants to convince a verifier that some statement is true, such that the verifier learns nothing except the validity of the assertion. Typically, the prover claims that an input string u is in a language L, and after the interaction, the verifier accepts or rejects. We assume the reader is familiar with the basic theory of zero-knowledge protocols and just recall the most important notions informally: the protocol is an interactive zero-knowledge proof system for L if it is *complete*, i.e. if $u \in L$, then the verifier accepts – and *sound*, i.e. if $u \notin L$ then no matter what the prover does, the verifier accepts with at most probability ε, where ε is called the soundness error of the protocol. Finally, zero-knowledge means that given only that $u \in L$, conversations between the honest prover and an arbitrary poly-time verifier can be efficiently simulated and are indistinguishable from real conversations.

In this chapter we concentrate on commitments to elements in a finite field K, or to integers and we assume that commitments are also homomorphic, i.e. both commitments and randomness are chosen from (finite) groups, and $[x] \cdot [y] = [x+y]$ (we will describe this property more in detail in Sections 7.2.1 and 7.2.2). For $K = \mathbb{F}_q$ for a prime q, such commitments can, for instance, be constructed from any q-invertible group homomorphism [**CD98**] that exists, if factoring or discrete log are hard problems. One can show using known – but perhaps less well known – techniques that homomorphic commitments with unconditional hiding *and* binding can be built assuming preprocessing; e.g. the committer gets random field elements and information theoretic MACs and the receiver gets corresponding keys. More details on this shall be given later (see Section 7.2.1). Finally, homomorphic commitments to integers based on factoring were proposed in [**FO97**, **DF02**].

In typical applications of these commitment schemes, the prover needs to convince the verifier that the values he commits to satisfy a certain algebraic relation. A general way to state this is that the prover commits to $x_1, \ldots, x_v$, and the verifier wants to know that $D(x_1, \ldots, x_v) = 0$ for an algebraic circuit D defined over K or over the integers. If D uses only linear operations, the verifier can himself compute a commitment to $D(x_1, \ldots, x_v)$ (using the homomorphic properties of the commitment scheme) and the prover opens this linear combination to reveal 0. However, if D uses multiplications, we need a zero-knowledge protocol where the prover convinces the verifier that three committed values x, y, z satisfy $xy = z$.

In [**CDD+99**], such a multiplication protocol was proposed for homomorphic commitments over any finite field K. The soundness error for that protocol is $1/|K|$, which is not necessarily negligible – for example when K is a field with small size (constant or logarithmic in the security parameter). The only known way to have a smaller error is to repeat the protocol. This solution leads to a protocol with communication complexity $\Theta(\kappa l)$ for soundness error 2^{-l} and where commitments have size κ bits.

Likewise, a multiplication protocol for integer commitments was proposed in [**FO97**, **DF02**]. This protocol has essentially optimal communication complexity $\Theta(\kappa + l + k)$, where k is the size

in bits of the prover's secret integers, but it requires an extra assumption, namely the strong RSA assumption. The best known complexity for protocols based on commitment schemes requiring only factoring is $\Theta((\kappa + k)l)$.

An approach to improving this state of affairs was proposed in [CD09], where it was suggested to take advantage of the fact that many applications require the prover to make many ZK proofs of similar statements. The idea is to make the amortized complexity per proof be small by combining all the proofs into one protocol. In our case, this would mean that the prover commits to x_i, y_i, z_i for $i = 1, \ldots, l$ and wants to convince the verifier that $x_i y_i = z_i$ for all i. The technique from [CD09] yields a protocol with amortized complexity $\Theta(\kappa + l)$ but, unfortunately, requires that all x_i's are equal (or all y_i's are equal), and in most applications, this condition is not satisfied.

7.1.1. Our Contribution. In this chapter, we construct a new zero-knowledge protocol that works for arbitrary x_i, y_i, z_i, and uses black-box access to any homomorphic commitment scheme. If we instantiate the commitments by a standard unconditionally binding and computationally hiding scheme, the amortized complexity is $O(\frac{u}{l} \cdot \kappa)$ bits for error probability 2^{-u}. In particular, for $l = u$, we get $O(\kappa)$. Therefore, when the committed values are from a field of small constant size, we improve the complexity of previous solutions by a factor of l. We also propose (based on standard techniques) a way to implement unconditionally secure homomorphic commitments assuming preprocessing. Using this implementation, the amortized complexity is $O(\frac{u}{l} \cdot u)$. In particular, for both types of commitments and any fixed error probability, the amortized overhead vanishes as we increase l.

We generalise our approach to obtain a protocol that verifies l instances of an algebraic circuit D over K with v inputs, in the following sense: given committed values $x_{i,j}$ and z_i, with $i = 1, \ldots, l$ and $j = 1, \ldots, v$, the prover shows that $D(x_{i,1}, \ldots, x_{i,v}) = z_i$ for $i = 1, \ldots, l$ (the protocol easily generalises to circuits with more than one output). The amortized cost to verify one circuit with multiplicative depth δ is $O(2^\delta \kappa + v\kappa + \delta \log l)$ bits for an error probability of 2^{-l} and so does not depend on the circuit size. For circuits with small multiplicative depth (sometimes known as the classes $K\text{-}SAC^0$ or $K\text{-}SAC^1$), this approach is better than using our first protocol; in fact, the amortized communication cost can be asymptotically smaller than the number of multiplications in D.

Another interesting feature of this protocol is that prover and verifier can execute it given only black-box access to an algorithm computing the function implemented by D. This is a notable contrast from standard protocols where the parties work their way through the circuit and must therefore agree on the layout. Our protocol would, for instance, allow the verifier to outsource computation of the function to a third party. Provided that the verifier chooses the random challenge in the protocol, this would be secure if the prover is malicious and the third party is semi-honest.

Our final result is a zero-knowledge protocol using black-box access to homomorphic commitments to k-bit integers. For checking l integer multiplications and error probability 2^{-l}, the amortized complexity is $O(\kappa + k + l \log(l))$. When instantiating the commitments using a standard computationally secure scheme, this improves security of previous solutions that needed the strong RSA assumption, while we need no assumption other than what the underlying commitment scheme requires (typically factoring). We also show a new technique for implementing unconditionally secure homomorphic commitments to integers based on preprocessing. This makes the protocol be much more efficient, as only a constant number of multiplications per commitment is required.

When using information theoretically secure commitments based on preprocessing, our protocols are perfect zero-knowledge against general verifiers. When using standard computationally secure commitments, they are only honest verifier zero-knowledge, but can be made zero-knowledge in general using standard techniques.

Our technique is somewhat related to the "MPC-in-the-head" technique from [IKOS09], but with an important difference: both strategies make use of "virtual players", that is, the

prover in his head imagines n players that receive shares of his secret values and he must later reveal information to the verifier relating to these shares. The protocol from [IKOS09] has complexity linear in n, because the prover must commit to the view of each virtual player. We use a different approach, exploiting the homomorphic property of the commitment scheme to get a protocol with complexity logarithmic in n. This is the reason our amortized overhead vanishes instead of being constant, as it would be using MPC-in-the-head. On the other hand, we show that a combination of "multiparty computation in the head" and our protocol for verifying algebraic circuits can actually improve the communication complexity for some parameter values. In concurrent and independent work, Ben-Sasson et al. [BSFO12] show a multiparty protocol for honest majority that checks several multiplicative relations on secret-shared values with low amortized complexity. The technique is somewhat related in that it is based on secret sharing, but the checking works in a different way since in that setting there is no single prover who knows all values.

7.1.2. Applications. One obvious application of our protocol is to give ZK proofs for satisfiability of a Boolean circuit C: the prover commits to the bits on each wire in the circuit, opens the output as a 1 and shows that, for each AND-gate, the corresponding multiplicative relation holds for the committed bits. To explain how this compares to previous work, we define the (computational or communication) overhead of a protocol to be its (computational or communication) complexity divided by $|C|$. One can think of this as the overhead factor one has to pay to get security, compared to an insecure implementation. Now, in the ideal commitment model (i.e. assuming access to a ideal commitment functionality) [IKOS09] obtained constant communication overhead and polynomial computation overhead, as a function of the security parameter u, for error probability 2^{-u}. Later, [DIK10] showed how to make both overheads poly-logarithmic. For both protocols, the ideal commitments can be implemented by doing preprocessing, and the resulting "on-line" protocol still has the same complexity.

As mentioned, our protocol can be thought of as working in the ideal *homomorphic* commitment model where the commitment functionality can do linear operations on committed bits (but where we of course charge for the cost of these operations). In this model our protocol achieves constant computational and communication overhead.

We may then instantiate the commitments using the information theoretically secure homomorphic scheme. This incurs an extra cost for local computing, so as a result we obtain a ZK-protocol with constant communication overhead and polynomial computation overhead (essentially $O(u \log u)$). Asymptotically, the overheads match those of [IKOS09], but the involved constants are smaller in our case because we do not need the "detour" via a multiparty protocol. Finally, we pay no communication for linear operations, which seems hard to achieve in the protocol from [IKOS09].

Another application area where our result can improve the state of the art is the following: as shown in [CDN01], general multiparty computation can be based on additively homomorphic encryption schemes. Many such schemes are known, and in several cases, the plaintext space is a small field. One example is the Goldwasser-Micali (GM)-scheme [GM84], where the plaintext space is $\mathbb{F}_2$. Supplying inputs to such a protocol amounts to sending them in encrypted form to all players and proving knowledge of the corresponding plaintexts. However, in many applications one would want to check that inputs satisfy certain conditions, e.g. an auction may require that bids are numbers in a certain interval. Since ciphertexts in such an additively homomorphic scheme can be thought of as homomorphic commitments over the field, our protocol can be used by a player to prove that his input satisfies a given condition much more efficiently than by previous techniques.

A final type of application is in the area of anonymous credentials and group signatures. Such constructions are often based on zero-knowledge proofs that are made non-interactive using the Fiat-Shamir heuristic. If the proof requires showing that a committed number is in a given interval, the standard solution is to "transfer" the values to an integer commitment scheme and use the proof technique of Boudot [Bou00]. This in turn requires multiplication proofs,

so if sufficiently many proofs are to be given in parallel, one can use our technique for integer commitments. Assuming preprocessing and our information theoretically secure commitments this can be very efficient, requiring only a constant number of multiplications per commitment.

7.2. Preliminaries

7.2.1. Information Theoretic Commitments. In this section we assume a setup that allows commitments to be unconditionally secure. We use $[v]$ as shorthand for a commitment to v. Operations on commitments are supposed to be multiplicative, while values that are committed lie in an additive group. Therefore a commitment scheme is homomorphic if $[v]\cdot[v'] = [v+v']$ for all v, v' in the relevant domain (either a finite field K or the integers). Also, if $\mathbf{v} = (v_1, \ldots, v_m)$ is a vector with entries in K (or in the integers), $[\mathbf{v}]$ denotes a vector of commitments, one for each coordinate in $\mathbf{v}$. If $\mathbf{u} = (u_1, \ldots, u_m)$ is a vector of the same length as $\mathbf{v}$, then $[\mathbf{v}]^{\mathbf{u}}$ means $[\mathbf{v}]^{\mathbf{u}} = \prod_i [v_i]^{u_i}$, which is a commitment containing the inner product of $\mathbf{u}$ and $\mathbf{v}$. Moreover $[\mathbf{u}]*[\mathbf{v}]$ refers to the component-wise product.

Field Scenario:

Let K be a finite field and L be an extension of K. Although the set-up is general, we will think of K as a small constant size field. Let $a \in L$ be a private value held by the verifier. We suppose that the prover has a list A of uniform values $u_1, \ldots, u_i, \ldots \in K$ and for each u_i he also has a value $m_{u_i} = a \cdot u_i + b_{u_i}$, where b_{u_i} is uniform in L and privately held by the verifier. One can think of m_{u_i} as an information theoretic message authentication code on u_i, and of (a, b_{u_i}) as the key to open such a MAC on u_i. It is possible to achieve this situation assuming a functionality for the preprocessing phase of a multiparty computation protocol, such as in the ones in [**BDOZ11**, **DPSZ12**, **DKL+13**].

With this setup, commitments can be made as follows: in order for the prover to commit to $v \in K$, the prover sends $u - v$ to the verifier and sets $m_v = m_u = a \cdot u + b_u$, where u is the first unused value in the list A; the verifier then updates the corresponding key b_u into $b_v = b_u + a \cdot (u - v)$. A commitment to v can therefore expressed by the following data (where P denotes the prover, and V denotes the verifier);

$$[v] = \begin{cases} P: & v,\ u,\ m_v = a \cdot u + b_u \\ V: & u - v,\ a,\ b_v = b_u + a \cdot (u - v) \end{cases}$$

In order to open commitment $[v]$, the prover sends v, m_v to the verifier, who checks whether $a \cdot v + b_v$ equals m_v.

Commitments of this form are unconditionally binding: a prover committing to v can send an opening $\widetilde{v}, \widetilde{m}$ with $\widetilde{v} \neq v$ if and only if $a \cdot \widetilde{v} + b_v = \widetilde{m}$. This is equivalent to the prover being able to sample two distinct points $(P_x, P_y) = (v, m_v)$, $(Q_x, Q_y) = (\widetilde{v}, \widetilde{m})$ from the line $Y = a \cdot u + b_v$; which is equivalent to the prover having knowledge of the key (a, b_v), that is privately held by the verifier. This shows that the probability of a prover succeeding in opening to a incorrect value is bounded by the probability of guessing a random element in a line over L, which is equals $1/|L|$. Since we want to have a negligible probability of breaking the binding property of the commitment scheme, we require $|L| = 2^{\Theta(\kappa)}$, where κ is the security parameter. In particular, this means that in the (unfortunate) case where K is a field with constant size, L is an extension of K of degree linear in κ.

These commitments are also unconditionally hiding: a verifier receiving a commitment $u - v$ only knows a and $b_{u_1}, \ldots, b_{u_i}, \ldots$, which are all independent of v.

Moreover, the above commitments are homomorphic (meaning $[v] \cdot [v'] = [v + v']$), where $[v] \cdot [v']$ is defined as follows:

$$[v] \cdot [v'] := \begin{cases} P: & v + v',\ u + u',\ m_{v+v'} = m_u + m_{u'} \\ V: & (u - v) + (u' - v'),\ a,\ b_{v+v'} = b_u + b_{u'} + a \cdot (u - v + u' - v') \end{cases}$$

Integers Scenario:

We here give a construction of unconditionally secure commitments on k-bit integers. Unlike the previous construction, this construction is new, to the best of our knowledge. Let a be a

prime in the interval $[-2^\kappa, \ldots, 2^\kappa]$, that is privately held by the verifier. We assume the prover has a list A of integer values $u_1, \ldots, u_i, \ldots$ uniform in $[-2^{k+\kappa}, \ldots, 2^{k+\kappa}]$ and for each u_i he also has an integer $m_{u_i} = a \cdot u_i + b_{u_i}$, where b_{u_i} is a uniform integer in $[-2^{k+3\kappa}, \ldots, 2^{k+3\kappa}]$ privately held by the verifier.

With this setup, commitments can be made as follows: in order for the prover to commit to the integer $v \in [-2^k, \ldots, 2^k]$, the prover sends $u - v$ to the verifier and sets $m_v = m_u = a \cdot u + b_u$, where u is the first unused value in the list A; the verifier then updates the key corresponding to b_u by setting $b_v = b_u + a \cdot (u - v)$. A commitment to v can therefore expressed by the following data (where P denotes the prover, and V denotes the verifier);

$$[v] = \begin{cases} P: & v,\ u,\ m_v = a \cdot u + b_u \\ V: & u - v,\ a,\ b_v = b_u + a \cdot (u - v) \end{cases}$$

In order to open commitment $[v]$, the prover sends v, m_v to the verifier, who checks if $a \cdot v + b_v$ equals m_v.

Commitments of this form are homomorphic, unconditionally hiding (by the same arguments as above) and unconditionally binding: a prover committing to v can send an opening $\widetilde{v}, \widetilde{m}$ with $\widetilde{v} \neq v$ if and only if $a \cdot \widetilde{v} + b_v = \widetilde{m}$. Subtracting the latter equation to the relation $m_v = a \cdot v + b_v$, we obtain $\widetilde{m} - m_v = a \cdot (\widetilde{v} - v)$, so the prover must know a multiple of a of length $k + \kappa$ bits. Any $(k + \kappa)$-bit integer can be thought of as its factorization, and the prover can break the binding property if he knows a $(k+\kappa)$-bit integer where a appears in its factorization. Since a $(k+\kappa)$-bit integer contains at most $(k + \kappa)/\kappa$ prime factors of length κ and the number of κ-bit primes is $\Theta(2^\kappa/\kappa)$, the error probability of the scheme is equal to $\Theta(((k+\kappa)/\kappa) \cdot (\kappa/2^\kappa)) = \Theta((k+\kappa)/2^\kappa)$. If $k = O(\kappa)$, the error probability is $O(\kappa/2^\kappa)$.

7.2.2. Commitment Schemes Based on Computational Assumptions. We consider two kinds of commitment schemes: the first one is over a finite field K and can be viewed as a function $com_{pk} : K \times H \to G$ where H, G are finite groups and pk is a public key (this includes the examples suggested in [CD98]). The second scheme is over the integers, and $com_{pk} : \mathbb{Z} \times \mathbb{Z} \to G$.

The public key pk is generated by a PPT algorithm $\mathcal{G}$ on input a security parameter κ. To commit to a value $x \in K$ or an integer x, the prover chooses r uniformly in H (or, in case of integer commitments, in some appropriate interval) and sends $C = com_{pk}(x, r)$ to the verifier. A commitment is opened by sending x, r. We assume that the scheme is homomorphic, i.e. $com_{pk}(x, r) \cdot com_{pk}(y, s) = com_{pk}(x + y, rs)$. For simplicity, we assume throughout that K is a prime field. Then, by repeated addition, that we also have $com_{pk}(x, r)^y = com_{pk}(xy, r^y)$ for any $y \in K$. We also use $[x]$ as shorthand for a commitment to x in the following, and hence suppress the randomness from the notation.

We consider *computationally hiding* schemes: for any two values x, x' the distributions of pk, $com_{pk}(x, r)$ and pk, $com_{pk}(x', s)$ must be computationally indistinguishable, where pk is generated by $\mathcal{G}$ on input security parameter κ. Such schemes are usually *unconditionally binding*, meaning that for any pk that can be output from $\mathcal{G}$, there does not exist x, r, x', s with $x \neq x'$ such that $com_{pk}(x, r) = com_{pk}(x', s)$. For such schemes, the prover usually runs $\mathcal{G}$, sends pk to the verifier and may have to convince him that pk was correctly generated before the scheme is used.

One may also consider *unconditionally hiding and computationally binding* schemes, where pk, $com_{pk}(x, r)$ and pk, $com_{pk}(x', s)$ must be statistically indistinguishable, and where it must be infeasible to find x, r, x', s with $x \neq x'$ such that $com_{pk}(x, r) = com_{pk}(x', s)$.

7.2.3. Linear Secret Sharing Schemes. The model of linear secret sharing schemes we consider here is essentially equivalent to both the monotone span program formalism [KW93, CDM00] and the linear code based formalism [CCG+07]. However, we generalise to schemes where several values from the underlying field can be shared simultaneously. The model is designed to facilitate clear description of the protocol.

Let K be a finite field and let m be a positive integer. Consider the m-dimensional K-vector space K^m. Consider the index set $I = \{1, 2, \ldots, m\}$, and write $\mathbf{x} = (x_i)_{i \in I}$ for the coordinates of $\mathbf{x} \in K^m$. In the following we consider linear functions between finite spaces. It is useful to recall that because such functions are (additive) group homomorphisms, they are always regular; that is, each element in the image has the same number of pre-images, namely the cardinality of the kernel.

For a non-empty set $A \subseteq I$, the **restriction** to A is the K-linear function

$$\begin{aligned} \pi_A : K^m &\longrightarrow K^{|A|} \\ \mathbf{x} &\longmapsto (x_i)_{i \in A}. \end{aligned}$$

Let $C \subseteq K^m$ be a K-linear subspace which we keep fixed throughout this section. Let $A, S \in I$ be non-empty sets. We say that S **offers uniformity** if $\pi_S(C) = K^{|S|}$. Note that by regularity of π_S, if $\mathbf{c}$ is uniform in C, then $\pi_S(\mathbf{c})$ is uniform in $K^{|S|}$.

Jumping ahead, we will use the subspace C for secret sharing by choosing a random vector $\mathbf{c} \in C$ such that $\pi_S(\mathbf{c}) = \mathbf{s}$ where S is a set offering uniformity and $\mathbf{s}$ is the vector of secret values to be shared. The shares are then the coordinates of $\mathbf{c}$ that are not in S.

We say that A **determines** S if there is a function $f : K^{|A|} \to K^{|S|}$ such that, for all $\mathbf{c} \in C$, $(f \circ \pi_A)(\mathbf{c}) = \pi_S(\mathbf{c})$. Note that such an f is K-linear if it exists. Note that if $\mathbf{c}$ is uniformly chosen from C and if A determines S, then $\pi_A(\mathbf{c})$ determines $\pi_S(\mathbf{c})$ with probability 1.

We say that A and S are **mutually independent** if the K-linear function

$$\begin{aligned} \phi_{A,S} : C &\longrightarrow \pi_A(C) \times \pi_S(C) \\ \mathbf{c} &\longmapsto (\pi_A(\mathbf{c}), \pi_S(\mathbf{c})) \end{aligned}$$

is surjective. Note that $\pi_S(C) = \{\mathbf{0}\}$ is the only condition under which it occurs that both A and S are independent and A determines S. In particular, if $\mathbf{c}$ is uniformly chosen from C, then $\pi_S(C) \neq \{\mathbf{0}\}$ and if A and S are independent, then $\pi_A(\mathbf{c})$ and $\pi_S(\mathbf{c})$ are distributed independently.

Suppose S offers uniformity. Let e be a positive integer and let

$$g : K^{|S|+e} \longrightarrow C$$

be a surjective K-linear function. Define $\pi_g : K^{|S|+e} \to K^{|S|}$ as the projection to the first $|S|$ coordinates. We say that g is an S**-generator** for C if $\pi_g = \pi_S \circ g$, that is, if the first $|S|$ coordinates of $\rho \in K^{|S|+e}$ are the same as the coordinates of $g(\rho)$ designated by S. Such an S-generator always exists, by elementary linear algebra, with $|B| + \rho = \dim_K(C)$.

For any S-generator g we have that if $\mathbf{s} \in K^{|S|}$ is fixed and if $\rho_{\mathbf{s}}$ is uniformly chosen in $K^{|S|+e}$ subject to $\pi_g(\rho_{\mathbf{s}}) = \mathbf{s}$, then $g(\rho_{\mathbf{s}})$ has the uniform distribution on the subset of C consisting of those $\mathbf{c} \in C$ with $\pi_S(\mathbf{c}) = \mathbf{s}$.

We are now ready to define linear secret sharing schemes in our model: Let $S \subset I$ be non-empty and proper. Write $S^* = I \setminus S$. The tuple (C, S) is a **linear secret sharing scheme** if S offers uniformity and if S^* determines S.

If that is the case, S^* is called the **player set**, $\pi_S(C)$ is the **secret-space**, and $\pi_{S^*}(C)$ is the **share-space**. If $j \in S^*$, then $\pi_j(C)$ is called the **share-space** for the j-th player. If $l = |S|$, the scheme is said to be l**-multi-secret**. For $A \subseteq S^*$, we say that the scheme has A**-privacy** (or A is an unqualified set) if $A = \varnothing$ or if A and S are independent. There is A**-reconstruction** (or A is qualified) if A is non-empty and if A determines S. The scheme offers t**-privacy** if, for all A in the player set with $|A| = t$, there is A-privacy. The scheme offers r**-reconstruction** if, for all A in the player set with $|A| = r$, there is A-reconstruction.
Note that $0 \leq t < r \leq |S^*|$ if there is t-privacy and r-reconstruction. A **generator** for (C, S) is an S-generator for C.

Let (C, S) be a secret sharing scheme, and let g be a generator. If $\mathbf{s} \in K^{|S|}$ is the secret, shares for the players in S^* are computed as follows. Select a vector $\rho_{\mathbf{s}}$ according to the uniform

probability distribution on $K^{|S|+e}$, subject to $\pi_g(\rho_{\mathbf{s}}) = \mathbf{s}$ and compute $\mathbf{c} = g(\rho_{\mathbf{s}})$. The "full vector of shares" is the vector $\pi_{S^*}(\mathbf{c})$.

In the following, where we write $\rho_{\mathbf{s}}$, it will usually be understood that it holds that $\pi_g(\rho_{\mathbf{s}}) = \mathbf{s}$, and we say that $\rho_{\mathbf{s}}$ is consistent with the secret $\mathbf{s}$.

Multiplication Properties:

For any $\mathbf{x}, \mathbf{y} \in K^m$, their **Schur-product** (or component-wise product) is the element $(\mathbf{x} * \mathbf{y}) \in K^m$ defined as $(\mathbf{x} * \mathbf{y}) = (x_j \cdot y_j)_{j \in I}$. If $C \subset K^m$ is a K-linear subspace, then its **Schur-product transform** is the subspace $\widehat{C} \subset K^m$ defined as the K-linear subspace generated by all elements of the form $\mathbf{c} * \mathbf{c}'$, where $\mathbf{c}, \mathbf{c}' \in C$.

Note that if (C, S) is a linear secret sharing scheme, then S offers uniformity in $\widehat{C}$ as well. But in general it does not hold that S^* determines S in $\widehat{C}$. However, suppose that it does (so $(\widehat{C}, S)$ is a linear secret sharing scheme). Then (C, S) is said to offer $\widehat{r}$**-product reconstruction** if $(\widehat{C}, S)$ offers $\widehat{r}$-reconstruction.

Sweeping vectors:

Let (C, S) be a linear secret sharing scheme, let g be a generator for it and let A be an unqualified set. Since A and S are mutually independent so that $\phi_{A,S}$ is surjective, it follows that for any index $j \in S$, there exists $\mathbf{c}_{A,j} \in C$ such that $\phi_{A,S}(\mathbf{c}_{A,j}) = (\mathbf{0}, \mathbf{e}_j)$ where $\mathbf{e}_j$ is the vector with a 1 in position j and zeros elsewhere. Note that since the generator g is surjective on C we can choose $\mathbf{w}_{A,j}$ such that $g(\mathbf{w}_{A,j}) = \mathbf{c}_{A,j}$, and $\pi_g(\mathbf{w}_{A,j}) = \mathbf{e}_j$. The vector $\mathbf{w}_{A,j}$ is called a j**th sweeping vector**.

To see the purpose of these vectors, suppose we have shared a vector of $|S|$ zeros, so we have $\mathbf{c}_0 = g(\rho_{\mathbf{0}})$. It is now easy to see that the vector

$$\rho_{\mathbf{0}} + \sum_{j=1}^{|S|} x_j \mathbf{w}_{A,j}$$

is consistent with the secret $(x_1, \ldots, x_{|S|})$. Moreover, if we apply g to this vector, the player set A gets the same shares as when 0's were shared.

7.3. A Protocol for the Field Scenario

We are now ready to solve the problem mentioned in the introduction: namely, the prover holds values $\mathbf{x} = (x_1, \ldots, x_l), \mathbf{y} = (y_1, \ldots, y_l), \mathbf{z} = (z_1, \ldots, z_l)$, has sent commitments $[\mathbf{x}], [\mathbf{y}], [\mathbf{z}]$ to the verifier and now wants to convince the verifier that $x_i y_i = z_i$ for $i = 1, \ldots, l$, i.e., that $\mathbf{x} * \mathbf{y} = \mathbf{z}$.

Suppose that both the prover and the verifier agreed on using an l-multisecret linear secret sharing scheme (C, S), for d players, offering $\widehat{r}$-product reconstruction, and with privacy threshold t. Fix a generator $g : K^{l+e} \to C$. Moreover, suppose that $\widehat{g} : K^{l+\widehat{e}} \to \widehat{C}$ is a generator for $(\widehat{C}, S)$ and that a public basis for K^{l+e} (respectively for $K^{l+\widehat{e}}$) has been chosen such that the linear mapping g (resp. $\widehat{g}$) can be computed as the action of a matrix M (resp. $\widehat{M}$).

The idea of the protocol is as follows: the prover secret shares $\mathbf{x}$ and $\mathbf{y}$ using (C, S) and $\mathbf{z}$ using $(\widehat{C}, S)$, in such a way that the resulting vectors of shares $\mathbf{c}_{\mathbf{x}}, \mathbf{c}_{\mathbf{y}}$, $\widehat{\mathbf{c}}_{\mathbf{z}}$ satisfy $\mathbf{c}_{\mathbf{x}} * \mathbf{c}_{\mathbf{y}} = \widehat{\mathbf{c}}_{\mathbf{z}}$, which is possible since (C, S) offers product reconstruction. The prover commits to the randomness used in all sharings, which, by the homomorphic property, allows the verifier to compute commitments to any desired share. The verifier now chooses t coordinate positions randomly and asks the prover to open the commitments to the shares in those positions. The verifier can then check that the shares in $\mathbf{x}, \mathbf{y}$ multiply to the shares in $\mathbf{z}$. This is secure for the prover since any t shares reveal no information; and, on the other hand, if the prover's claim is false, i.e. $\mathbf{x} * \mathbf{y} \neq \mathbf{z}$, then $\mathbf{c}_{\mathbf{x}} * \mathbf{c}_{\mathbf{y}}$ and $\widehat{\mathbf{c}}_{\mathbf{z}}$ can be equal in at most $\widehat{r}$ positions, so the verifier has a good chance of finding a position that reveals the cheat. More formally, the protocol goes as follows:

Protocol Verify Multiplication

(1) The prover chooses two vectors $\mathbf{r_x}, \mathbf{r_y} \in K^e$, and sets $\rho_\mathbf{x} = (\mathbf{x}, \mathbf{r_x})$, $\rho_\mathbf{y} = (\mathbf{y}, \mathbf{r_y})$. Define $\mathbf{c_x} = M\rho_\mathbf{x}, \mathbf{c_y} = M\rho_\mathbf{y}$. Now, the prover computes $\widehat{\rho}_\mathbf{z} \in K^{l+\widehat{e}}$ such that $\widehat{\rho}_\mathbf{z}$ is consistent with secret $\mathbf{z}$ and such that $\widehat{M}\widehat{\rho}_\mathbf{z} = \mathbf{c_x} * \mathbf{c_y}$ (Note that this is possible by solving a system of linear equations, exactly because $\mathbf{x} * \mathbf{y} = \mathbf{z}$.) We then write $\widehat{\rho}_\mathbf{z} = (\mathbf{z}, \widehat{\mathbf{r}}_\mathbf{z})$ for some $\widehat{\mathbf{r}}_\mathbf{z} \in K^{\widehat{e}}$. Set $\widehat{\mathbf{c}}_\mathbf{z} = \widehat{M}\widehat{\rho}_\mathbf{z}$.

(2) The prover sends vectors of commitments $[\mathbf{r_x}], [\mathbf{r_y}], [\widehat{\mathbf{r}}_\mathbf{z}]$ to the verifier. Together with the commitments to $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$, the verifier now holds vectors of commitments $[\rho_\mathbf{x}], [\rho_\mathbf{y}], [\widehat{\rho}_\mathbf{z}]$.

(3) The verifier chooses t uniform indices $O \subset S^*$ and sends them to the prover.

(4) Let $\mathbf{m}_i$ be the ith row of M and $\widehat{\mathbf{m}}_i$ the ith row of $\widehat{M}$. For each $i \in O$, using the homomorphic property of the commitments, both prover and verifier compute commitments

$$[(\mathbf{c_x})_i] = [\rho_\mathbf{x}]^{\mathbf{m}_i}, \qquad [(\mathbf{c_y})_i] = [\rho_\mathbf{y}]^{\mathbf{m}_i}, \qquad [(\widehat{\mathbf{c}}_\mathbf{z})_i] = [\widehat{\rho}_\mathbf{x}]^{\widehat{\mathbf{m}}_i}.$$

The prover opens these commitments to the verifier.

(5) The verifier accepts if and only if the opened values satisfy $(\mathbf{c_x})_i \cdot (\mathbf{c_y})_i = (\widehat{\mathbf{c}}_\mathbf{z})_i$ for all $i \in O$.

FIGURE 7.1. Protocol to verify a multiplicative relation

THEOREM 7.1. *Assume the commitment scheme used is the one described in Section 7.2.1. Then protocol Verify Multiplication is perfect zero-knowledge, and if for some i, $x_i y_i \neq z_i$, the verifier accepts with probability at most $((\widehat{r}-1)/d)^t + 1/|L|$.*

Proof. For soundness, we suppose that the prover is dishonest (so $x_i y_i \neq z_i$ for some i) and we compute the probability that the protocol accepts. Note first that, from the prover's commitments, vectors $\mathbf{c_x}, \mathbf{c_y}, \widehat{\mathbf{c}}_\mathbf{z}$ are determined, where we know that $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ respectively appear in coordinates designated by S. Since $x_i y_i \neq z_i$ for some i, it follows that $\mathbf{c_x} * \mathbf{c_y} \neq \widehat{\mathbf{c}}_\mathbf{z}$.

Denote by $T \subset S^*$ the index set in the share space where the vectors $\mathbf{c_x} * \mathbf{c_y}$ and $\widehat{\mathbf{c}}_\mathbf{z}$ agree.

Note that the cardinality of T is at most $\widehat{r} - 1$, because $\mathbf{c_x} * \mathbf{c_y}$ and $\widehat{\mathbf{c}}_\mathbf{z}$ are consistent with different secrets. In order for the prover to be successful, one of the following must hold:

- All t entries the verifier asks the prover to unveil are in T.
  In this case the prover can behave honestly, since the choice of the verifier points to values that already satisfy the expected multiplicative relation.
- The prover can break the binding property of the commitment scheme.
  In this case the prover can open commitments to arbitrary values.

For the first case: the probability that one entry chosen by the verifier is in T is at most equal to $(\widehat{r}-1)/d$, since the choice is uniform. Repeating this argument t times implies that the event that all t entries asked by the verifier are in T happens with probability equal to $((\widehat{r}-1)/d)^t$.

If this event does not happen, there is at least an entry chosen by the verifier that lies outside T. The prover can still pass the check by opening to values that satisfy the multiplicative relation expected by the verifier, even for those entries that lie outside T. Since there is at least one entry outside T, the prover must break the binding property of the commitment scheme to succeed, and this event happens with probability $1/|L|$, as shown in Section 7.2.1. Therefore, the soundness error of the protocol is bounded by $(\widehat{r}-1)/d + 1/|L|$.

To show zero-knowledge, we build the following simulator.

- For the setup, the simulator runs a local copy of the ideal functionality for the preprocessing, obtaining values
  - $(u_1, m_{u_1}), \ldots, (u_i, m_{u_i}), \ldots \in K \times L$ and
  - $(a_{u_1}, b_{u_1}), \ldots, (a_{u_i}, b_{u_i}), \ldots \in L \times L$

  such that
  $$m_{u_i} = a_{u_i} \cdot u_i + b_{u_i}, \qquad \text{for all } i.$$
  The simulator sends the values $(a_{u_1}, b_{u_1}), \ldots, (a_{u_i}, b_{u_i}), \ldots$ to the verifier.
- The simulator runs the protocol exactly as a honest prover would until step 3.

- In step 4, in order to open each value $[(\mathbf{c_x})_i] = [\rho_{\mathbf{x}}]^{\mathbf{m}_i}$ (respectively $[(\mathbf{c_y})_i]$), the simulator first computes the corresponding key $(a, b_{(\mathbf{c_x})_i})$ (respectively $(a, b_{(\mathbf{c_y})_i})$), samples a uniform $x'_i \in K$ (respectively y'), computes $m_{x'_i} = a \cdot x'_i + b_{(\mathbf{c_x})_i}$ (respectively $m_{y'_i} = a \cdot y'_i + b_{(\mathbf{c_y})_i}$) and sends $(x'_i, m_{x'_i})$ (respectively $(y'_i, m_{y'_i})$) as the opened value. To open $[(\widehat{\mathbf{c}}_{\mathbf{z}})_i]$ the simulator computes the corresponding key $(a, b_{(\widehat{\mathbf{c}}_{\mathbf{z}})_i}$, sets $z'_i = x'_i \cdot y'_i$, computes $m_{z'_i} = a \cdot z'_i + b_{(\widehat{\mathbf{c}}_{\mathbf{z}})_i}$ and sends $(z'_i, m_{z'_i})$ as the opened value.

The simulation is clearly polynomial time. The distribution of the messages sent to the verifier is the same as in a real execution of the protocol until step 3. In step 4 the opened values have the same distribution as in the real protocol (x'_i, y'_i are uniform, and $z'_i = x'_i \cdot y'_i$), and the openings are valid, since they pairs sent satisfy the linear relation expected by the verifier. □

Representing a Sequence of Points:

We here compare two approaches to represent a sequence O of t elements drawn uniformly and independently from a set $S^* = \{1, \ldots, d\}$. This comparison helps us to find the best communication complexity achievable by our protocols, since in all of them there is one step in which such O has to be sent between players.

One method is to send the sequence of points in O: this procedure would require $\log d$ bits per point, so the total amount of bits is $t \cdot \log d$. An alternative method is to send a bit-vector of dimension d, where its ith coordinate is equal to 1 if and only if $i \in O$; the total amount of bits is then d.

In the proceedings version of our paper [CDP12], we assume that the communication between players was performed using the first method, while in the full version we take advantage of the most convenient method: when the choice of the parameters is such that $d < t \cdot \log d$ holds, we use the latter (in the protocol for verification of a single multiplication, both for finite fields and for the integers), and if not we use the former (in the protocol for verification of a circuit).

Demands upon the Secret Sharing Schemes:

Above, we have described the protocol for a fixed secret sharing scheme, but what we really want is to look at is the asymptotic behaviour as a function of l, the number of secrets we handle in one execution, and u, where we want error probability 2^{-u}. For this, we need a family of secret sharing schemes, parametrised by l and u, which makes t, d, e, $\widehat{r}$ and $\widehat{e}$ be functions of l, u.

Let's say that committing requires sending κ_c bits, while opening requires κ_o bits. For standard computationally secure commitments, it is usually the case that κ_c is $\Theta(\kappa_o)$, but this is not the case for the information theoretically secure commitments, where κ_c can be much smaller than κ_o. Using this notation, the communication complexity of the protocol is $O(\kappa_c(e + \widehat{e}) + \kappa_o t + d)$ bits.

Now, suppose we can build a family of secret sharing schemes, where $e, \widehat{e}$ are $O(u)$ and $\widehat{r}$ is $O(l + u)$, t is $\Theta(u)$ and $(\widehat{r} - 1)/d$ is $O(1)$. This allows d to be $O(l + u)$ and so we can achieve the complexities promised earlier: for standard computationally hiding commitments, we get soundness error $2^{-\alpha u}$ for some constant $\alpha > 0$ for one instance of the protocol. For the information theoretically secure commitments, we get the same if we set $|L| = 2^{\Theta(u)}$. In any case, if necessary, we can achieve 2^{-u} by repeating the protocol in parallel a constant number of times. Inserting into the above expression, and dividing by l, we get the complexity per multiplicative relation: $O(\frac{u}{l}(\kappa_c + \kappa_o))$. For standard commitments we have $\kappa_c = \kappa_o = \kappa$, and for information theoretically secure commitments we have $\kappa_c = O(1)$ and $\kappa_o = O(u)$. So this gives us the complexities promised in the Introduction.

We show in Section 7.3.1 how to construct a secret sharing scheme with the desired properties.

Application to Zero-Knowledge Proofs for Circuit Satisfiability:

An obvious application of our protocol is to give ZK proofs for Boolean circuit satisfiability: the prover commits to the bits on each wire in the circuit C, opens the output as a 1 and shows that, for each AND-gate, the corresponding multiplicative relation holds for the committed bits.

We can apply the Verify Multiplication protocol to do this. Then we have that l is $O(|C|)$. If we run the protocol with error probability 2^{-u}, our expression for the total communication complexity becomes $O(|C|\kappa_c + u(\kappa_c + \kappa_o))$, note that we have to add the cost of committing to the bits in the circuit.

Now we note that if we use the information-theoretic commitments as described above, then $\kappa_c = 1$ and κ_o is $O(u)$. Therefore the complexity is actually $O(|C|+u^2))$ bits, and when dividing by $|C|$ we get communication overhead $O(1)$, as promised in the introduction.

7.3.1. A Concrete Example. In this section we explain how to design a secret-sharing scheme meeting the demands we stated earlier. For simplicity we first show the details for the case of $u = l$.

As a stepping stone, we consider the following scheme based on Shamir's scheme. Suppose $2(t + l - 1) < d$ and $d + l \leq |K|$. Choose pairwise distinct elements $q_1, \ldots, q_l, p_1, \ldots, p_d \in K$, and define

$$C = \{(f(q_1), \ldots, f(q_l), f(p_1), \ldots, f(p_d)) \mid f \in K[X]_{\leq t+l-1}\} \subset K^{l+d},$$

where $K[X]_{\leq t+l-1}$ denotes the K-vector space of polynomials with coefficients in K and of degree at most $t + l - 1$. Let S correspond to the first l coordinates. Then, by Lagrange interpolation, it is straightforward to verify that (C, S) is an l-multi-secret K-linear secret sharing scheme of length d, with t-privacy and $(2t + 2l - 1)$-product reconstruction. So if we set $t = l$ (and hence the degrees are at most $2l - 1$), $d = 8l$, and $|K| \geq 9l$, then $2(t + l - 1) = 4l - 2 < 8l = d$, $d + l = 9l \leq |K|$, and $\widehat{r} = 2t + 2l - 1 = 4l - 1 < 4l = d/2$. In particular, $\frac{\widehat{r}-1}{d} < \frac{1}{2}$. Moreover, $e = 2l$, and $\widehat{e} = 4l - 1$. So all requirements are satisfied, except for the fact that in this approach $|K| = \Omega(\log l)$.

Before we present a scheme which works over a *constant size field*, yet asymptotically meets all requirements, we describe a simple, useful lifting technique. Suppose the finite field of interest K, i.e. the field over which our zero-knowledge problem is defined, does not readily admit the required secret sharing scheme, but that some degree-u extension L of K does. Then we may choose a K-basis of L of the form $1, x, \ldots, x^{u-1}$ for some $x \in L$. It is then easy to "lift" the commitment scheme and to obtain one that is L-homomorphic instead: simply consider the elements of L as coordinate-vectors over K, according to the basis selected above, and commit to such a vector by committing separately to each coordinate. This scheme is clearly homomorphic with respect to addition in L. Multiplication by (publicly known) scalars from L is easily seen to correspond to applying an appropriate (publicly known) K-linear form to the vector of K-homomorphic commitments. Furthermore, K is embedded into L by mapping $a \in K$ to $a + 0 \cdot x + \ldots + 0 \cdot x^{u-1}$. When committing to $a \in K$, simply commit to a in the original commitment scheme, and append $u - 1$ "default commitments to 0." This way, the protocol problem can be solved over K, with a secret sharing scheme over L. However, communication-wise, even though all further parameters may be satisfied, there are now $O(ul)$ commitments, instead of $O(l)$ as required.

For example, if the above secret sharing scheme is implemented, then since K is of constant size, the field L over which the secret sharing is defined must grow proportionally to $\log l$. Hence, the total communication is a logarithmic factor off of our target. This is resolved as follows, by using a technique that allows passing to an extension whose degree u is *constant* instead of logarithmic.

Let F be an algebraic function field over the finite field $\mathbb{F}_q$ with q elements. Write g for its genus and n for its number of rational points. Suppose $2g + 2(t + l - 1) < d$ and $d + l \leq n$. Choose pairwise distinct rational points $Q_1, \ldots, Q_l, P_1, \ldots, P_d \in F$, and define

$$C = \{(f(Q_1), \ldots, f(Q_l), f(P_1), \ldots, f(P_d)) \mid f \in \mathcal{L}(G)\} \subset \mathbb{F}_q^{l+d},$$

where G is a divisor of degree $2g + t + l - 1$ whose support does not contain any of the Q_j's nor any of the P_i's, and where $\mathcal{L}(G)$ is the Riemann-Roch space of G. As before, let S correspond

to the first l coordinates. Using a similar result as in [CC06], we prove, using the Riemann-Roch Theorem, that (C, S) is an l-multi-secret $\mathbb{F}_q$-linear secret sharing scheme of length d, with t-privacy and $(2g + 2t + 2l - 1)$-product reconstruction. Moreover, $e = g + t + l$ and $\widehat{e} \leq 3g + 2t + 2l - 1$. Asymptotically, using this result in combination with optimal towers over the *fixed* finite field $\mathbb{F}_q$ where $q \geq 49$ is a square, we get $g/n = \frac{1}{\sqrt{q}-1} < 1/6$. Hence, if we set, for example, $t = l = n/20$ and $d = 19/20n$, then there is $\Omega(l)$-privacy, $\frac{\widehat{r}-1}{d} < c < 1$ for some constant c, and $e = \Omega(l)$, $\widehat{e} = \Omega(l)$. Therefore, at most a degree 6 extension of the field of interest is required, as the maximum is attained for $K = \mathbb{F}_2$ with the extension being $\mathbb{F}_{64}$. Finally, these schemes can be implemented efficiently.

The more general case where u and l are independent parameters follows easily from the above and we leave the details to the reader. The basic reason that it works is that the number of required random field elements for a sharing $(e, \widehat{e})$ is linear in the required privacy threshold which we want to be $\Theta(u)$, and furthermore the reconstruction threshold $(\widehat{r})$ is linear in the sum of the length of the secret vector and the privacy threshold, which here is $l + \Theta(u)$.

7.3.2. Verify Multiplication Protocol for Standard Commitments.

THEOREM 7.2. *Assume the commitment scheme used is unconditionally binding and computationally hiding. Then the Verify Multiplication protocol is a computationally honest-verifier zero-knowledge interactive proof system for the language*

$$\left\{([x_i], [y_i], [z_i])_{i=1}^{l} \mid x_i y_i = z_i, \text{ for } i = 1, \ldots, l\right\}$$

with soundness error $((\widehat{r} - 1)/d)^t$.

Proof. The proof of soundness follows the same structure as that of Theorem 7.1.

To show (honest-verifier) zero-knowledge, the idea is to "execute the protocol" exactly as the honest prover would have done, but assuming that all secret values are 0. After that, we adjust the relevant values so they become consistent with the actual values of $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$.

So we first generate random vectors $\rho_{\mathbf{0}}^{\mathbf{x}} = (\mathbf{0}, \mathbf{r}_{\mathbf{0}}^{\mathbf{x}}), \rho_{\mathbf{0}}^{\mathbf{y}} = (\mathbf{0}, \mathbf{r}_{\mathbf{0}}^{\mathbf{y}})$, both consistent with sharing the all-0 vector. We compute $\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}} = (\mathbf{0}, \mathbf{r}_{\mathbf{0}}^{\mathbf{z}})$ such that $\widehat{M}\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}} = (M\rho_{\mathbf{0}}^{\mathbf{x}}) * (M\rho_{\mathbf{0}}^{\mathbf{y}})$. We then choose a random subset $A \subset S^*$ of t indices. Note that we have $(M\rho_{\mathbf{0}}^{\mathbf{x}})_i (M\rho_{\mathbf{0}}^{\mathbf{y}})_i = (\widehat{M}\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}})_i$ for $i \in A$, and these shares have the same distribution as in the real conversation, since any t shares are distributed independently of the actual secrets. We then form random vectors of commitments $[\mathbf{r}_{\mathbf{0}}^{\mathbf{x}}], [\mathbf{r}_{\mathbf{0}}^{\mathbf{y}}], [\mathbf{r}_{\mathbf{0}}^{\mathbf{z}}]$. Note that since the commitment function is a homomorphism from $K \times H$ to G, the neutral element 1_G is a commitment to $0 \in K$. Therefore we can form vectors of commitments as follows:

$$[\rho_{\mathbf{0}}^{\mathbf{x}}] = ((1_G, \ldots, 1_G), [\mathbf{r}_{\mathbf{0}}^{\mathbf{x}}]), \quad [\rho_{\mathbf{0}}^{\mathbf{y}}] = ((1_G, \ldots, 1_G), [\mathbf{r}_{\mathbf{0}}^{\mathbf{y}}]), \quad [\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}}] = ((1_G, \ldots, 1_G), [\mathbf{r}_{\mathbf{0}}^{\mathbf{z}}]).$$

As described above, we can assume existence of sweeping vectors $\mathbf{w}_{A,j}$ and $\widehat{\mathbf{w}}_{A,j}$ for the secret sharing schemes (C, S) and $(\widehat{C}, S)$, respectively, and we know that the vectors

$$\rho_{\mathbf{x}} = \rho_{\mathbf{0}}^{\mathbf{x}} + \sum_{j=1}^{l} x_j \cdot \mathbf{w}_{A,j}, \qquad \rho_{\mathbf{y}} = \rho_{\mathbf{0}}^{\mathbf{y}} + \sum_{j=1}^{l} y_j \cdot \mathbf{w}_{A,j}, \qquad \widehat{\rho}_{\mathbf{z}} = \widehat{\rho}_{\mathbf{0}}^{\mathbf{z}} + \sum_{j=1}^{l} z_j \cdot \widehat{\mathbf{w}}_{A,j}$$

are consistent with sharing $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$, respectively, but where the subset A gets the same shares as when 0's were shared. The simulator cannot compute $\rho_{\mathbf{x}}, \rho_{\mathbf{y}}, \widehat{\rho}_{\mathbf{z}}$, but it can compute commitments to them. Using the fact that the commitments $[x_j], [y_j], [z_j]$ are given and the sweeping vectors are public, it can compute, for instance, a vector of commitments $[x_i \cdot \mathbf{w}_{A,j}]$ and hence

$$[\rho_{\mathbf{x}}] = [\rho_{\mathbf{0}}^{\mathbf{x}}] \prod_{j=1}^{l} [x_j \cdot \mathbf{w}_{A,j}], \qquad [\rho_{\mathbf{y}}] = [\rho_{\mathbf{0}}^{\mathbf{y}}] \prod_{j=1}^{l} [y_j \cdot \mathbf{w}_{A,j}], \qquad [\widehat{\rho}_{\mathbf{z}}] = [\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}}] \prod_{j=1}^{l} [z_j \cdot \widehat{\mathbf{w}}_{A,j}].$$

It is easy to verify that because we used neutral elements 1_G as the first entries in $[\rho_{\mathbf{0}}^{\mathbf{x}}], [\rho_{\mathbf{0}}^{\mathbf{y}}], [\widehat{\rho}_{\mathbf{0}}^{\mathbf{z}}]$, the first l entries of $[\rho_{\mathbf{x}}], [\rho_{\mathbf{y}}], [\widehat{\rho}_{\mathbf{z}}]$ as computed above are $[\mathbf{x}], [\mathbf{y}]$ and $[\mathbf{z}]$. The simulator therefore extracts the last e commitments from $[\rho_{\mathbf{x}}]$ and $[\rho_{\mathbf{y}}]$, and the last $\widehat{e}$ commitments from $[\widehat{\rho}_{\mathbf{z}}]$, and uses these to simulate the commitments sent in Step 2.

It then outputs the index set A as simulation of step 3.

For step 4, note that the simulator may compute and open commitments to

$$[(M\rho_{\mathbf{x}})_i] = [(M\rho_{\mathbf{0}}^{\mathbf{x}})_i], \qquad [(M\rho_{\mathbf{y}})_i] = [(M\rho_{\mathbf{0}}^{\mathbf{y}})_i], \qquad [(\widehat{M}\rho_{\mathbf{0}}^{\mathbf{z}})_i],$$

for $i \in A$, where these equalities follow by the sweeping vector properties. By construction, the opened values satisfy the multiplicative property expected by the verifier.

This simulation is clearly polynomial time, and we argued that the distribution of all values that are opened are exactly as in a real conversation. The commitments $[\rho_{\mathbf{x}}]$ and $[\rho_{\mathbf{y}}]$ are also distributed correctly. Therefore, the only difference between simulation and conversation lies in the distribution of $\widehat{\rho}_{\mathbf{z}}$ hidden in $[\widehat{\rho}_{\mathbf{z}}]$ (in a real conversation, the choice of $\widehat{\rho}_{\mathbf{z}}$ ensures that the resulting $\widehat{\mathbf{c}}_{\mathbf{z}}$ satisfies $(\mathbf{c}_{\mathbf{x}})_i(\mathbf{c}_{\mathbf{y}})_i = (\widehat{\mathbf{c}}_{\mathbf{z}})_i$ for all indices i, whereas for the simulation this only holds for $i \in A$). It therefore follows from the hiding property of commitments and a standard hybrid argument that simulation is computationally indistinguishable from real conversations. □

7.3.3. Verify Multiplication Protocol for Unconditionally Hiding Commitments. We briefly sketch how to modify the protocol to work for an unconditionally hiding and computationally binding commitment scheme. The protocol would then be a proof of knowledge that the prover can open his input commitments to reveal strings $\mathbf{x}, \mathbf{y}, \mathbf{z}$ with $\mathbf{x} * \mathbf{y} = \mathbf{z}$. We need to add in Step 2 that the prover must prove that he knows how to open all the commitments $[\mathbf{x}], [\mathbf{y}], [\mathbf{z}], [\mathbf{r}_{\mathbf{x}}], [\mathbf{r}_{\mathbf{y}}], [\widehat{\mathbf{r}_{\mathbf{z}}}]$. This can be done by simply invoking the amortized efficient zero-knowledge protocol from [CD09] since the commitment function we assume is exactly of the form this protocol can handle. The overhead introduced by this is only a constant factor.

The proof of zero-knowledge is exactly the same, except that we get perfectly (statistical) zero-knowledge if the commitment scheme is perfect (statistically) hiding.

For soundness, we argue that parameters are chosen such that $((\widehat{r}-1)/d)^t$ is negligible in the security parameter, and if the prover convinces the verifier with non-negligible probability, then there exists a knowledge extractor that uses the prover to compute openings of his input commitments to strings $\mathbf{x}, \mathbf{y}, \mathbf{z}$ with $\mathbf{x} * \mathbf{y} = \mathbf{z}$ (except with negligible probability). This algorithm would first invoke the knowledge extractor for the protocol from [CD09] to get opening of all the prover's initial commitments, to strings $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{r}_{\mathbf{x}}, \mathbf{r}_{\mathbf{y}}, \widehat{\mathbf{r}_{\mathbf{z}}}$. We claim that except with negligible probability, we have $\mathbf{x} * \mathbf{y} = \mathbf{z}$.

This follows since, as we now argue, if $\mathbf{x} * \mathbf{y} \neq \mathbf{z}$ then we could break the binding property of the commitments. To see this, notice that from the openings we know of the prover's initial commitments, we can use the homomorphic property to compute openings of any commitment to a share that the prover can be asked to open in Step 4. Call these the *predetermined* openings. Note that the shares in question are consistent with secret sharing the strings $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

Now we send a random challenge to the prover, and by assumption on the prover, his reply passes the verifier's test with non-negligible probability, i.e., for all $i \in O$, the opened values $s_{x,i}, s_{y,i}, s_{z,i}$ satisfy $s_{x,i}s_{y,i} = s_{z,i}$. However, this is not the case for the predetermined openings of the same commitments: it immediately follows from the soundness proof of the previous theorem that because the predetermined openings are consistent with actually secret sharing $\mathbf{x}, \mathbf{y}, \mathbf{z}$, these openings satisfy the multiplicative relation with only negligible probability $((\widehat{r}-1)/d)^t$ (over the choice of the verifier's challenge). It follows that with non-negligible probability, there is at least one commitment to a share for which the predetermined opening is different from the opening done by the prover in response to the challenge. We have therefore broken the binding property.

7.4. A More General Approach

In this section we define linear secret sharing with a more general multiplicative property, and we use the notation from Section 7.2.3. Let D be an arithmetic circuit over K with v inputs and one output. Then for $\mathbf{c}_1, \ldots, \mathbf{c}_v \in K^m$, we define $D(\mathbf{c}_1, \ldots, \mathbf{c}_v) \in K^m$ as the vector whose jth coordinate is $D((\mathbf{c}_1)_j, \ldots, (\mathbf{c}_v)_j)$, i.e., we simply apply D to the jth coordinate of all input vectors.

If $C \subset K^m$ is a linear subspace, then C^D is defined as the K-linear subspace generated by all vectors of form $D(\mathbf{c}_1, \ldots, \mathbf{c}_v)$ where $\mathbf{c}_1, \ldots, \mathbf{c}_v \in C$. Just as for the standard multiplication property, if (C, S) is a secret sharing scheme, then S offers uniformity in C^D, but in general it does not necessarily hold that S^* determines S in C^D. If it does, however, so that (C^D, S) is a linear secret sharing scheme, then we say that (C, S) offers $(\widetilde{r}, D)$**-product reconstruction** if (C^D, S) offers $\widetilde{r}$-product reconstruction.

As a concrete example of this, consider Shamir secret sharing. Here, each $\mathbf{c}_i$ is a sequence of evaluations of a polynomial f_i at a fixed set of points. Then $D(\mathbf{c}_1, \ldots, \mathbf{c}_v)$ denotes the vector having coordinates of the form $D(f_1(j), \ldots, f_v(j))$ for j in the set of evaluation points. These coordinates can be thought as the evaluations of the polynomial $D(f_1, \ldots, f_v)$ (defined in the natural way), of which sufficiently many will determine $D(f_1, \ldots, f_v)$ uniquely.

Based on this more general notion, we can design a protocol where a prover commits to vectors $\mathbf{x}_1, \ldots, \mathbf{x}_v, \mathbf{z}$ and wants to prove that $D(\mathbf{x}_1, \ldots, \mathbf{x}_v) = \mathbf{z}$.

Similarly to our approach for the first protocol, we suppose that both the prover and the verifier agreed on using an l-multisecret linear secret sharing scheme (C, S), for d players, with $(\widetilde{r}, D)$-product reconstruction, and t-privacy. We fix a generator $g : K^{l+e} \to C$. Moreover, we suppose that $\widetilde{g} : K^{l+\widetilde{e}} \to C^D$ is a generator for (C^D, S) and that a public basis for K^{l+e} (respectively for $K^{l+\widetilde{e}}$) has been chosen such that the linear mapping g (resp. $\widetilde{g}$) can be computed as the action of a matrix M (resp. $\widetilde{M}$). The protocol goes as follows:

Protocol Verify Circuit

(1) The prover chooses v vectors $\mathbf{r}_1, \ldots, \mathbf{r}_v \in K^e$, and sets $\rho_j = (\mathbf{x}_j, \mathbf{r}_j)$ for $j = 1, \ldots, v$. Define $\mathbf{c}_j = M\rho_j$. Now, the prover computes $\widetilde{\rho}_{\mathbf{z}} \in K^{l+\widetilde{e}}$ such that $\widetilde{\rho}_{\mathbf{z}}$ is consistent with secret $\mathbf{z}$ and such that $\widetilde{M}\widetilde{\rho}_{\mathbf{z}} = D(\mathbf{x}_1, \ldots, \mathbf{x}_v)$ (Note that this is possible by solving a system of linear equations, because $D(\mathbf{x}_1, \ldots, \mathbf{x}_v) = \mathbf{z}$.) We then write $\widetilde{\rho}_{\mathbf{z}} = (\mathbf{z}, \widetilde{\mathbf{r}}_{\mathbf{z}})$ for some $\widetilde{\mathbf{r}}_{\mathbf{z}} \in K^{\widetilde{e}}$. Set $\widetilde{\mathbf{c}}_{\mathbf{z}} = \widetilde{M}\widetilde{\rho}_{\mathbf{z}}$.

(2) The prover sends vectors of commitments $[\mathbf{r}_j]$, $j = 1, \ldots, v$ and $[\widetilde{\mathbf{r}}_{\mathbf{z}}]$ to the verifier. Together with the commitments to $\mathbf{x}_j$ and $\mathbf{z}$, the verifier now holds vectors of commitments $[\rho_j]$, $j = 1, \ldots, v$, and $[\widetilde{\rho}_{\mathbf{z}}]$.

(3) The verifier chooses t uniform indices $O \subset S^*$ and sends them to the prover.

(4) Let $\mathbf{m}_i$ be the ith row of M and let $\widetilde{\mathbf{m}}_i$ be the ith row of $\widetilde{M}$. For each $i \in O$, using the homomorphic property of the commitments, both prover and verifier compute commitments

$$[(\mathbf{c}_j)_i] = [\rho_j]^{\mathbf{m}_i}, \text{ for } j = 1, \ldots, v, \qquad [(\widetilde{\mathbf{c}}_{\mathbf{z}})_i] = [\widetilde{\rho}_{\mathbf{z}}]^{\widetilde{\mathbf{m}}_i}.$$

The prover opens these commitments to the verifier.

(5) The verifier accepts if and only if the opened values satisfy $D((\mathbf{c}_1)_i, \ldots, (\mathbf{c}_v)_i) = (\widetilde{\mathbf{c}}_{\mathbf{z}})_i$ for all $i \in O$.

FIGURE 7.2. Protocol to verify a circuit

Using a similar proof as for Theorem 7.1, it is easy to show the following.

THEOREM 7.3. *Assume the commitment scheme used is the one described in section 7.2.1. Then the protocol Verify Circuit is perfect honest-verifier zero-knowledge and if $D(\mathbf{x}_1, \ldots, \mathbf{x}_v) \neq \mathbf{z}$, the verifier accepts with probability at most $((\tilde{r} - 1)/d)^t + 1/|L|$.*

The interesting question is whether we can build secret sharing schemes with this type of D-reconstruction and whether the resulting more general protocol offers advantages over the first one.

The answer to the first question is positive, and the construction was already hinted at above: we can base a scheme on Shamir secret sharing extended à la Franklin and Yung [FY92] to share blocks of l secrets. This requires polynomials of degree $e = l + t - 1$. Since each multiplication in D doubles the degree of the polynomials, the degree after applying D is $2^\delta t$ where δ is the multiplicative depth of D. This means that $\widetilde{e} = \widetilde{r} = 2^\delta t$ for this construction, and d should be a constant factor larger than $\widetilde{r}$ to get exponentially small error probability.

We assume for simplicity that the cardinality of K is larger than $d + l$, in order to have the required number of evaluation points. If this is not the case, we can pass to an extension field at cost of a logarithmic factor, as explained in the previous section. Note that the algebraic geometric approach presented in Section 7.2.3 does not give any non-constant improvement over the Shamir-based approach in the setting of D-reconstruction. However, it appears that the algebraic geometric approach can be extended to get a non-trivial improvement here as well, using more advanced techniques.

We can now compare two natural approaches to verifying that committed vectors $\mathbf{x}_1, \ldots, \mathbf{x}_v$, $\mathbf{z}$ satisfy $D(\mathbf{x}_1, \ldots, \mathbf{x}_v) = \mathbf{z}$:

The first approach is to perform the Verify Circuit protocol using the secret sharing scheme we sketched. If we go for error probability 2^{-l} and therefore choose t to be $\Theta(l)$, and representing the index set O in the most convenient way (see "Representing s Sequence of Points" in section 7.3), simple inspection of the protocol shows:

LEMMA 7.4. *Using the Verify Circuit Protocol, the amortized communication complexity to verify one instance of a circuit with multiplicative depth δ and v inputs is $O(2^\delta \kappa + v\kappa + \delta \log l)$ bits for an error probability of 2^{-l}.*

Note that, except for the cost of committing to the inputs, the communication complexity only depends on the depth of the circuit.

The second approach is to use the Verify Multiplication protocol. The prover will, for every multiplication gate T in D, commit to a vector $\mathbf{z}_T$ where $(\mathbf{z}_T)_i$ is the output from T in the instance of D where the inputs are $(\mathbf{x}_1)_i, \ldots, (\mathbf{x}_v)_i$. Now, for every multiplication gate T the verifier can compute vectors of commitments $[\mathbf{x}_T], [\mathbf{y}_T]$ to the inputs to T (since any linear operations in D "between multiplication gates" can be done by the verifier alone). We then use the Verify Multiplication protocol to check that $\mathbf{x}_T * \mathbf{y}_T = \mathbf{z}_T$. Using this protocol verifying a multiplication has communication cost $O(\kappa)$ bits, so the total cost to verify one instance of the circuit corresponds to $O(\mu\kappa + v\kappa)$ bits, where μ is the number of multiplication gates in D.

Notice that large fan-out comes at no cost in our model, and that linear operations with large fan-in are also for free. Moreover, both approaches generalize easily to circuits with several outputs. Therefore, there is no fixed relation between μ and δ, and in particular, we could consider families of circuits where δ is constant or logarithmic in the input size, but μ grows faster than 2^δ. In such a case, using the Verify Circuit protocol is better: it has the interesting property that the amortized cost of verifying a single instance of D can be asymptotically smaller than the number of multiplication gates in D.

THEOREM 7.5. *Assume the commitment scheme used is unconditionally binding and computationally hiding. Then the protocol Verify Circuit is a computationally honest-verifier zero-knowledge interactive proof system for the language*

$$\{([\mathbf{x}_1], \ldots, [\mathbf{x}_v], [\mathbf{z}]) \mid D(\mathbf{x}_1, \ldots, \mathbf{x}_v) = \mathbf{z}\}$$

with soundness error $((\tilde{r} - 1)/d)^t$.

7.4.1. Using MPC in the Head for the Verify Circuit Protocol. We now sketch a final variant of the Verify Circuit protocol that leads to a complexity that is in general incomparable to the first one, and for reasonable parameter values gives an improvement.

The idea is as follows: instead of committing to the values in $\tilde{\mathbf{r}}_\mathbf{z}$ in the usual way, the prover simply sends the required commitments to shares $[(\widetilde{\mathbf{c}}_\mathbf{z})_i]$ and use the "MPC in the head"

approach from [IKOS09] (the IKOS compiler) to prove to the verifier that the commitments contain the correct shares.

To use this approach, we first specify a multiparty protocol that creates the desired output, the IKOS compiler produces a 2 party zero-knowledge protocol proving the result is correct, assuming also a suitable commitment scheme (not necessarily that used in the basic protocol).

The multiparty protocol goes as follows: we have $a \in \Theta(l)$ players, of which a constant fraction may be actively corrupted. The first step is to generate a set of random secret shared values $r_1, \ldots, r_{\tilde{e}}$, shared among the a players using standard Shamir sharing over K. Using a simple variant of the protocol by Hirt and Berliova based on hyperinvertible matrices, this can be done in total communication complexity $O(\tilde{e}lk)$ bits where k is the size of a field element. We now set $\tilde{\mathbf{r}}_{\mathbf{z}} = (r_1, \ldots, r_{\tilde{e}})$ and we let the shares of these values be $r_{u,j},\ u = 1, \ldots, \tilde{e},\ j = 1, \ldots, a$. Let $\mathbf{n}_i$ be the last $\tilde{e}$ entries of $\mathbf{m}_i$. Then each player outputs a commitment to the inner product $[t_{i,j}] = [(r_{1,j}, \ldots, r_{\tilde{e},j}) \cdot \mathbf{n}_i]$.

Let $\lambda_1, \ldots, \lambda_a$ be the Lagrange coefficients to reconstruct the secret given correct shares. Everybody can now compute $[t_i] = \prod_j [t_{i,j}]^{\lambda_j}$.

Note that we do not yet know if the value is correct, but if all virtual players output correct commitments, then the desired commitment to $(\tilde{\mathbf{c}}_{\mathbf{z}})_i$ can be computed as a "linear combination" of the commitments $\mathbf{z}$ and $[t_i]$.

Note that the $t_{i,j}$'s are in fact Shamir shares in t_i, and they are all correct, except for a constant fraction. Therefore, it follows that t_i is correct if all $t_{i,j}$ are on a polynomial of low enough degree. We check this by computing commitments to the "syndrome" of the set of $t_{i,j}$'s: these commitments should contain all 0's. In a normal multiparty situation, we could not open these commitments, but in our case a prover is executing the protocol in his head, so we can just ask the prover to open these.

When we compile this protocol to a 2-party protocol, the idea is, as mentioned, that the prover executes the protocol in his head and commits to the views of all players. We do this with a separate commitment scheme that does not need to be homomorphic. The verifier asks the prover to open the views of a randomly chosen unqualified subset of players and checks the views for consistency. The IKOS results show that if the protocol has not worked correctly, the verifier rejects, except with probability $2^{-\Theta(a)}$. As a result, we get the commitments to the shares we wanted.

Since a and t, the number of opened shares are $\Theta(l)$, the cost of this is $O(l^2\kappa)$ bits for the commitments and $\tilde{e}l^2k$ bits for the views of virtual players.

This new protocol should be compared to the normal one where the cost is $O((\tilde{e}+l)\kappa)$. We see that if $\kappa > l^2k$ and $\tilde{e} > l^2$ then the new solution has smaller cost.

7.5. Proving Integer Multiplication

In the following, we show a protocol designed for the case where the prover's secret values are integers. We make use of a specific integer linear secret sharing scheme based on polynomials and assume that the underlying commitment scheme is computationally binding and unconditionally hiding. The idea of the protocol is similar to the one for finite fields.

Let

$$\Delta = \prod_{\substack{i,j=1,\ldots,d,\\ i\neq j}} (i-j),$$

where d is the number of players. Assume that the secrets $s_1,\ldots,s_l$ to be shared satisfy $s_i \in \{-2^k,\ldots,2^k\}$ for all i, for some k. In order to share them, sample random integers $a_1,\ldots,a_t \in \{-l2^{k+u}\Delta d!,\ldots,l2^{k+u}\Delta d!\}$ (where u is the security parameter) and use Lagrange interpolation over the rationals to find $g \in \mathbb{Q}[X]$ such that

$$g(-i) = s_i \qquad \text{and} \qquad g(-l-j) = a_j,$$

for $i = 1,\ldots,l$ and $j = 1,\ldots,t$. Since there are $t+l$ points to interpolate, g has degree (less or) equal to $t+l$. Define $f = \Delta \cdot g$. It follows that f is indeed a polynomial over the integers, since Δ is a multiple of each denominator appearing in the coefficients of g. The shares are then the values $f(1),\ldots,f(d)$. Given at least $t+l+1$ shares, one can reconstruct the secrets, simply by doing Lagrange interpolation over $\mathbb{Q}$.

Moreover, any set of at most t shares has distribution statistically independent of $s_1,\ldots,s_l$: let A be an index set designating t players. By Lagrange interpolation we can construct a polynomial $w'_{A,i}$ of degree at most $t+l$ with rational coefficients such that $w'_{A,i}(-i) = 1$ and $w'_{A,i}(j) = 0$ for $i = 1,\ldots,l$ and for $j \in A$. From the standard construction of $w'_{A,i}$, it follows that $w_{A,i} = \Delta w'_{A,i}$ has integer coefficients, and that $w_{A,i}(-l-1),\ldots,w_{A,i}(-l-t)$ are all numerically at most $\Delta d!$.

Now suppose we have shared the secret consisting of l 0's using a polynomial h. Then $f = h + \sum_{i=1}^{l} s_i w_{A,i}$ is a polynomial consistent with sharing the secrets $s_1,\ldots,s_l$, but the shares rising from f received by player set A are the same as the ones rising from h. If f is such that $f(-l-1),\ldots,f(-l-t)$ are in the correct interval we conclude that the set of shares in question will be chosen for A with the same probability whether the secrets are $0,\ldots,0$ or $s_1,\ldots,s_l$. But the evaluations $h(-l-1),\ldots,h(-l-t)$ were chosen in an interval a factor 2^u larger than the size of the the evaluations of $\sum_{i=1}^{l} s_i w_{A,i}$, and hence h and f are both legal, except with probability negligible in u. Hence the distribution of shares seen by A is, for any tuple of secrets, statistically indistinguishable from the distribution for the zeros tuple.

7.5.1. A Protocol to Prove Integer Multiplication. In this section, we give a protocol allowing the prover to show that committed vectors $\mathbf{x},\mathbf{y},\mathbf{z}$ with *integer* entries satisfy $\mathbf{x} * \mathbf{y} = \mathbf{z}$ where, as in the previous section, an honest prover will choose the committed integers from $\{-2^k,\ldots,2^k\}$. We use the secret sharing scheme from the preceding section where we set the security parameter u to be l (this is consistent with previous sections where we have used l as the parameter controlling error probabilities). We use the same notation for commitments as in previous sections, and as an example of the concrete commitment scheme based on computational assumptions, the reader may think of the factoring based scheme from [FO97, DF02]. As an example of the the unconditionally secure commitment scheme, we refer to section 7.2.1.

Before stating the actual protocol, we fix some notation. Let f be a polynomial of degree equal to m. Write $f(X) = \sum_{j=0}^{m} f_j X^j$. Define $\mathbf{f} = (f_0,\ldots,f_m)$ and $\mathbf{ev}(i) = (1,i,\ldots,i^m)$. Notice that

$$f(i) = \sum_{j=0}^{m} f_j \cdot i^j = \mathbf{f}\cdot\mathbf{ev}(i), \qquad \text{and} \qquad [f(i)] = \prod_{j=0}^{m} [f_j]^{i^j} = [\mathbf{f}]^{\mathbf{ev}(i)}.$$

The formulation on the right hand side of these equations is the one used in the protocol, which proceeds as follows. Note that the protocol differs depending on the kind of commitment scheme used (based on computational assumptions or unconditionally secure).

The prover holds values $\mathbf{x} = (x_1, \ldots, x_l), \mathbf{y} = (y_1, \ldots, y_l)$ and $\mathbf{z} = (z_1, \ldots, z_l)$, and has sent commitments $[\mathbf{x}], [\mathbf{y}]$ and $[\mathbf{z}]$ to the verifier. We assume they both agreed in using the linear secret sharing scheme described in section 7.5. In case they are using a commitment scheme based on computational assumptions, we suppose there exists an interactive zero-knowledge proof of knowledge P_C for the relation

$$C = \{(a, w) \mid a = com_{pk}(x, r), w = (x, r)\}.$$

Moreover, we assume this interactive proof of knowledge is a Σ-protocol that can prove knowledge of opening for l commitments at once, with knowledge error 2^{-l}. Conversations in such a protocol have form (a, e, z) where e is random challenge issued by the verifier. Because commitments are homomorphic, such a proof of knowledge follows immediately from the techniques described in [CD09]. In the protocol below, we execute a variant of our protocol from the previous sections in parallel with P_C. Thus the overall protocol will have the form of a Σ-protocol, which simplifies the proof of soundness.

Protocol Verify Multiplication for Integers

Note! Text in *italic font* denotes actions performed only if using a commitment scheme based on computational assumptions.

(1) The prover chooses $\mathbf{a_x}, \mathbf{a_y} \in \mathbb{Z}^t$ and uses Lagrange interpolation (over the rationals) to generate two polynomials g_x, g_y, having degree $t + l$, such that

$$g_x(-i) = x_i, \qquad g_x(-l-j) = (\mathbf{a_x})_j, \qquad g_y(-i) = y_i, \qquad g_y(-l-j) = (\mathbf{a_y})_j,$$

for $i = 1, \ldots, l$ and $j = 1, \ldots, t$. The prover now sets $\widehat{g}_z = g_x \cdot g_y$, $f_x = \Delta g_x, f_y = \Delta g_y$ and $\widehat{f}_z = \Delta^2 \widehat{g}_z$. As explained above, f_x and f_y are polynomials with integral coefficients and have degree at most $t + l$. Notice that $\widehat{f}_z$ is also a polynomial with integral coefficients, but has degree at most $2(t + l)$.

(2) The prover sends commitments $[\mathbf{f_x}]$, $[\mathbf{f_y}]$ and $[\widehat{\mathbf{f}}_\mathbf{z}]$.

(3) The verifier checks that $[\mathbf{x}], [\mathbf{y}]$ and $[\mathbf{z}]$ are consistent with f_x, f_y and $\widehat{f}_z$: namely, for all $i = 1, \ldots, l$ it computes

$$[\mathbf{f_x}]^{\mathbf{ev}(-i)}[\Delta x_i]^{-1}, \qquad [\mathbf{f_y}]^{\mathbf{ev}(-i)}[\Delta y_i]^{-1}, \qquad [\widehat{\mathbf{f}}_\mathbf{z}]^{\mathbf{ev}(-i)}[\Delta^2 z_i]^{-1},$$

and asks the prover to open these commitments to zero. If any of these openings do not agree with the commitments, the verifier quits.

(4) *The prover defines the vector* $x_C = ([\mathbf{x}], [\mathbf{y}], [\mathbf{z}], [\mathbf{f_x}], [\mathbf{f_y}], [\widehat{\mathbf{f}}_\mathbf{z}])$ *containing committed values. We think of* x_C *as a vector of instances for the protocol* P_C*. The prover computes a vector* a_C *as the first message for the protocol* P_C *with instance* x_C*. The prover sends* a_C *to the verifier.*

(5) The verifier chooses t uniform indices $O \subset \{1, \ldots, d\}$. Similarly as above, the verifier computes

$$[\mathbf{f_x}]^{\mathbf{ev}(i)} = [(\mathbf{b_x})_i], \qquad [\mathbf{f_y}]^{\mathbf{ev}(i)} = [(\mathbf{b_y})_i], \qquad [\widehat{\mathbf{f}}_\mathbf{z}]^{\mathbf{ev}(i)} = [(\widehat{\mathbf{b}}_\mathbf{z})_i],$$

for $i \in O$. *The verifier generates a vector* e_C *as a challenge on* (x_C, a_C) *according to* P_C*.* The verifier sends e_C *together with* the index set O to the prover.

(6) *The prover computes the vector* z_C *as a reply for* (x_C, a_C, e_C) *according to* P_C*.* The prover sends z_C *together with* the openings of $[(\mathbf{b_x})_i], [(\mathbf{b_y})_i]$ and $[(\widehat{\mathbf{b}}_\mathbf{z})_i]$ for $i \in O$.

(7) The verifier accepts if and only if (x_C, a_C, e_C, z_C) *is an accepted conversation for* P_C *and* the opened values satisfy $(\mathbf{b_x})_i \cdot (\mathbf{b_y})_i = (\widehat{\mathbf{b}}_\mathbf{z})_i$ for $i \in O$.

FIGURE 7.3. Protocol to verify a multiplicative relation over the integers

Using a proof similar to the one of theorem 7.1 we obtain the following result:

THEOREM 7.6. *Assume the commitment scheme used is the one described in section 7.2.1. Then protocol above is perfect zero-knowledge, and if for some i, $x_i y_i \neq z_i$, the verifier accepts with probability at most $(2(t+l)/d)^t + \kappa/2^\kappa$.*

In the theorem below, we show soundness and honest verifier zero-knowledge for the above protocol. It may seem strange at first sight that the theorem does not assume that commitments are binding. This is because we show that the protocol *unconditionally* is a proof of knowledge that either the prover knows $\mathbf{x}, \mathbf{y}, \mathbf{z}$ with the expected multiplicative relation, or he knows a commitment that he can open in two different ways. Using this result in an application, one would apply the knowledge extractor and then argue that because the commitment scheme is computationally binding, the event that the prover breaks the binding property occurs with negligible probability. Since the primary example we know of integer commitments ([**FO97**, **DF02**]) has binding based on factoring, applications of this result only need to assume factoring is hard, in contrast to earlier techniques where strong RSA was required.

THEOREM 7.7. *Assume the homomorphic commitment scheme used is unconditionally hiding. Then the above protocol is a statistical honest-verifier zero-knowledge interactive proof of knowledge for the relation*

$$\begin{aligned} M = &\{(a,w) \mid a = (pk, A_i, B_i, C_i)_{i=1}^l, w = (x_i, r_i, y_i, s_i, z_i, t_i)_{i=1}^l : \\ &com_{pk}(x_i, r_i) = A_i, com_{pk}(y_i, s_i) = B_i, com_{pk}(z_i, t_i) = C_i, z_i = x_i y_i \text{ for } i = 1, \ldots, l\} \cup \\ &\{(a,w) \mid a = (pk, A), w = (v, r, v', r') : com_{pk}(v, r) = A = com_{pk}(v', r'), v \neq v'\} \end{aligned}$$

with knowledge error $ke_M = \max\{(2(t+l)/d)^t, 2^{-l}\}$.

Proof. For soundness, for any prover P^* that makes the protocol accept with probability p we build a knowledge extractor E_M having running time $(p - ke_M)^{-1}\mathsf{poly}(u)$, where ke_M is equal to $\max\{(2(t+l)/d)^t, 2^{-l}\}$. The latter equality allows us to assume $p > (2(t+l)/d)^t$. Note that by the result from [**BG06**], we may assume that P^* is deterministic. Therefore p is simply the fraction of challenges e_C, O that P^* answers correctly.

(i) E_M runs the protocol with P^* until step 3; E_M stores each opening (v, r) in a list L.
(ii) E_M continues the protocol. It receives a_C during step 4.
(iii) E_M sends e_C, O computed according to the protocol at step 5.
(iv) During step 6 E_M receives z_C and the openings of $[(\mathbf{b_x})_i], [(\mathbf{b_y})_i]$ and $[(\widehat{\mathbf{b}_\mathbf{z}})_i]$ for $i \in O$. E_M rewinds the prover to step 5 and goes to (iii) until it sees two conversations (x_C, a_C, e_C, z_C), (x_C, a_C, e'_C, z'_C) valid for P_C and such that $e_C \neq e'_C$. At this point E_M can retrieve the witness for x_C, namely the values and the randomness used to make the commitments.
(v) E_M checks whether $\mathbf{x} * \mathbf{y} = \mathbf{z}$. If that is the case, it outputs $w = (x_i, y_i, z_i, r_i, s_i, t_i)_{i=1}^l$ as a witness for the committed values [**x**],[**y**] and [**z**] and quits.
(vi) E_M performs the check of step 3 on its own, using the retrieved values and randomness. Each result (v', r') is stored on a list L', using the same ordering as the one for L (i.e. for each possible j, the j-th entry of L and of L' correspond to the opening of the same commitment). If there exists an index j such that $L_j = (v, r)$, $L'_j = (v', r')$ and $v \neq v'$, then E_M outputs $w = (v, r, v', r')$ as a witness for $com_{pk}(v, r) = com_{pk}(v', r')$ and quits.
(vii) E_M defines T as $i \in T$ if and only if $x_i y_i = z_i$. Then, E_M rewinds the prover to step 5 and sends e_C, O according to the protocol.
(viii) During step 6, if for some index $i \notin T$ the prover outputs $(x'_i, r'_i, y'_i, s'_i, z'_i, t'_i)$ such that $x'_i y'_i = z'_i$, then E_M outputs $w = (x_i, r_i, x'_i, r'_i)$ if $x_i \neq x'_i$, $w = (y_i, s_i, y'_i, s'_i)$ if $y_i \neq y'_i$, or $w = (z_i, t_i, z'_i, t'_i)$ if $z_i \neq z'_i$ and quits. Else E_M rewinds the prover to step 5 and repeats this step.

The expected running time of this algorithm can be analyzed as follows:

- Step (i) runs in polynomial time, since E_M stores a polynomial amount of data. Notice that the check at step 3 must pass, since p is bigger than zero.
- Step (ii), (iii), (iv) run in polynomial time. The number of rewindings to pass step (iv) is bounded by $2(p-2^{-l})^{-1}$, which is under the constrains. Retrieving the commitments requires polynomial time (from the special soundness property of sigma protocols).
- Step (v) runs in polynomial time (l multiplications).
- Step (vi) runs in polynomial time, since it requires to perform a polynomial amount of linear operations and multiplications.
- Step (vii) runs in polynomial time.
- Step (viii) happens if P^* makes the test $\mathbf{x} * \mathbf{y} = \mathbf{z}$ fail. We now bound the probability p that P^* succeeds in the protocol in such a situation. In order for P^* to be successful, it had to open the values pointed in O correctly. Since $\mathbf{x} * \mathbf{y} \neq \mathbf{z}$, then $f_x f_y \neq \widehat{f}_z$. Notice that $f_x f_x$ and $\widehat{f}_z$ are both polynomials of degree $2(t+l)$ and since they are distinct, they have at most $2(t+l)$ roots in common. This implies that one way for P^* to succeed is that all the entries in O point to common roots (that is, $O \subset T$). Since the choice of O is uniform and independent from P^*'s choices, the probability that $O \subset T$ is $(2(t+l)/d)^t$. Since p is assumed to be greater than $(2(t+l)/d)^t$, it means there exists some set $O \not\subset T$ that make P^* succeed. The probability that a uniform O makes P^* succeed and $O \not\subset T$ is equal to $p - (2(t+l)/d)^t$. This implies that the expected number of rewinds in step (viii) is equal to $(p-(2(t+l)/d)^t)^{-1}$, so the total running time of the algorithm is within the constraints even if it terminates in step (viii).

To show (honest-verifier) zero-knowledge we use the same technique we exploit in the field scenario. The simulator samples two random polynomials h_x, h_y of degree $t+l$ such that $h_x(-i) = 0 = h_y(-i)$, that is h_x, h_y are both consistent with sharing the secret consisting of l zeros. It then computes $\widehat{h}_z = h_x h_y$. Let $A \subset \{1, \ldots, d\}$ be a subset of players of size t. Notice that $h_x(i)h_y(i) = \widehat{h}_z(i)$ for all $i \in A$ and that these shares have distribution statistically indistinguishable from a real conversation, since any t shares are essentially independent of the actual secrets. Using the polynomials $w_{A,i}$, $i = 1, \ldots, l$ we define

$$f_x = h_x + \sum_{i=1}^{l} x_i w_{A,i}, \qquad f_y = h_y + \sum_{i=1}^{l} y_i w_{A,i}, \qquad \widehat{f}_z = \widehat{h}_z + \sum_{i=1}^{l} \Delta z_i w_{A,i}.$$

These three polynomials are consistent with sharing $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$, and the subset A gets the same shares as when 0's were shared. The simulator cannot compute these polynomials, but it can compute commitments to the coefficients. Using the fact that the commitments $[\mathbf{x}], [\mathbf{y}]$ and $[\mathbf{z}]$ are given, and the polynomials $w_{A,i}$ are public, it can compute commitments

$$[\mathbf{f_x}] = [\mathbf{h_x}] \prod_{k=1}^{l} [x_k \cdot \mathbf{w_{A,k}}], \qquad [\mathbf{f_y}] = [\mathbf{h_y}] \prod_{k=1}^{l} [y_k \cdot \mathbf{w_{A,k}}], \qquad [\widehat{\mathbf{f}}_\mathbf{z}] = [\widehat{\mathbf{h}}_\mathbf{z}] \prod_{k=1}^{l} [z_k \cdot \Delta\mathbf{w_{A,k}}].$$

Step 2 is simulated sending commitments $[\mathbf{f_x}], [\mathbf{f_y}]$ and $[\widehat{\mathbf{f}}_\mathbf{z}]$. The verifier in step 3 checks the consistency of the received data and the check passes. Here we prove it for x_i (with a similar proof one shows the check passes for y_i, z_i, for $i = 1, \ldots, l$). The verifier can use the homomorphic properties of the commitment schemes to check whether

$$[\mathbf{f_x}]^{\mathbf{ev}(-i)} \cdot [\Delta x_i]^{-1} = [0].$$

From the construction of f_x, it follows that

$$\begin{aligned}[\mathbf{f_x}]^{\mathbf{ev}(-i)} \cdot [\Delta x_i]^{-1} &= \left([\mathbf{h_x}]^{\mathbf{ev}(-i)} \cdot \prod_{k=1}^{l} [x_k \cdot \mathbf{w_{A,k}}]^{\mathbf{ev}(-i)} \right) \cdot [\Delta x_i]^{-1} \\ &= [h_x(-i)] \cdot \prod_{k=1}^{l} [x_k \cdot w_{A,k}(-i)] \cdot [\Delta x_i]^{-1} \\ &= [0] \cdot [\Delta x_i] \cdot [\Delta x_i]^{-1} = [0].\end{aligned}$$

For step 4, the simulator can compute and open commitments

$$[(\mathbf{f_x})_i] = [(\mathbf{h_x})_i], \qquad [(\mathbf{f_y})_i] = [(\mathbf{h_y})_i], \qquad [(\widehat{\mathbf{f}}_\mathbf{z})_i] = [(\widehat{\mathbf{h}}_\mathbf{z})_i],$$

for $i \in A$. By construction, the opened values satisfy the multiplicative property expected by the verifier.

This simulation is clearly polynomial time, and we argued underway that the distribution of all values that are opened is statistically close to that of a real conversation. The commitments $[\mathbf{f_x}]$ and $[\mathbf{f_y}]$ are also distributed correctly. Therefore, the only difference between simulation and conversation lies in the distribution of $\widehat{\mathbf{f}}_\mathbf{z}$ hidden in $[\widehat{\mathbf{f}}_\mathbf{z}]$ (in a real conversation, the choice of $\widehat{\mathbf{f}}_\mathbf{z}$ ensures that the resulting $\widehat{\mathbf{b}}_\mathbf{z}$ satisfies $(\mathbf{b_x})_i(\mathbf{b_y})_i = (\widehat{\mathbf{b}}_\mathbf{z})_i$ for all indices i, whereas for the simulation this only holds for $i \in A$). Since commitments are statistically hiding, it follows that the simulation is computationally indistinguishable from a real conversation. □

On the complexity of the protocol:

We now examine the complexity of the integer multiplication protocol assuming we want a knowledge error that is exponentially small in l, as in previous sections. It is easy to see that this can be arranged if we choose the parameters t and d to be $\Theta(l)$. Recall also that we already chose the statistical security parameter of the secret sharing scheme to be $\Theta(l)$. With these parameter choices, simple inspection of the protocol and secret sharing scheme shows that the amortized complexity per multiplication proved is $O(\kappa + k)$. This also includes the cost of the proof P_C: this can be verified by a direct inspection of the technique from [CD98], for a case where a proof is given for l commitments in parallel and where the statistical security parameter of the proof is also set to l.

Bibliography

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium – NDSS 2013*. Internet Society, 2013.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[AJLA+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.

[AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *Theory of Cryptography – TCC 2007*, volume 4392 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007.

[AL10] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. 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, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[BG06] Mihir Bellare and Oded Goldreich. On probabilistic versus deterministic provers in the definition of proofs of knowledge. *Electronic Colloquium on Computational Complexity (ECCC)*, 13(136), 2006.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS*, pages 309–325. ACM, 2012.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security – ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *STOC*, pages 1–10. ACM, 1988.

[Bou00] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 431–444. Springer, 2000.

[BSFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680. Springer, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[CC06] Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 521–536. Springer, 2006.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Janos Simon, editor, *STOC*, pages 11–19. ACM, 1988.

[CCG+07] Hao Chen, Ronald Cramer, Shafi Goldwasser, Robbert de Haan, and Vinod Vaikuntanathan. Secure computation from random error correcting codes. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 291–310. Springer, 2007.

[CD98] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 1998.

[CD09] Ronald Cramer and Ivan Damgård. On the amortized complexity of zero-knowledge protocols. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2009.

[CDD+99] Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In *EUROCRYPT*, pages 311–326, 1999.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[CDN12] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing – An Information Theoretic Appoach*. www.daimi.au.dk/~ivan/MPCbook.pdf, 2012. Book Draft, Accessed: 2013-05-22.

[CDP12] Ronald Cramer, Ivan Damgård, and Valerio Pastro. On the amortized complexity of zero knowledge protocols for multiplicative relations. In Adam Smith, editor, *ICITS*, volume 7412 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 2012.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[CN11] Yuanmi Chen and Phong Q. Nguyen. Bkz 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography – FC 2010*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[DF02] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In Yuliang Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 125–142. Springer, 2002.

[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, *Theory of Cryptography – TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 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, *Public Key Cryptography – PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

[DK10] Ivan Damgård and Marcel Keller. Secure multiparty AES. In Radu Sion, editor, *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 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*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263. Springer, 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 *ESORICS*, 2013. To appear.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

[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*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[FO97] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710. ACM, 1992.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[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 V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.

[GN08] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *STOC*, pages 21–30. ACM, 2007.

[IKOS09] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium – 2012*, pages 285–300, 2012.

[KW93] Mauricio Karchmer and Avi Wigderson. On span programs. In *Structure in Complexity Theory Conference*, pages 102–111, 1993.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In Aggelos Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[LPR11] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. 2011. Manuscript.

[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, *Security and Cryptography for Networks – SCN 2008*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - Secure two-party computation system. In *USENIX Security Symposium – 2004*, pages 287–302, 2004.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.

[MR08] Daniele Micciancio and Oded Regev. Lattice-based cryptography, 2008.

[MSas11] Steven Myers, Mona Sergi, and abhi shelat. Threshold fully homomorphic encryption and secure computation. *IACR Cryptology ePrint Archive*, 2011:454, 2011.

[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, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[PM08] Markus Püschel and José M. F. Moura. Algebraic signal processing theory: Cooley-tukey type algorithms for dcts and dsts. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[PVW08] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. *Advances in Cryptology–CRYPTO 2008*, pages 554–571, 2008.

[SIM] SIMAP Project. Simap: Secure information management and processing. http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx.

[SV11] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic simd operations. *IACR Cryptology ePrint Archive*, 2011:133, 2011.

[WW10] Severin Winkler and Jürg Wullschleger. On the efficiency of classical and quantum oblivious transfer reductions. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 707–723. Springer, 2010.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.