Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Matrix Operations in Multi-Party Computation

Rafael Harth

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dr. Pascal Reisert
Commenced:	October 23, 2019
Completed:	June 22, 2020

Abstract

This document presents two novel techniques for Multi-Party Computation based on secret sharing where the objects of computation are square matrices over a finite field. Matrix Triples can be consumed during the evaluation phase to obtain the product of two matrices, and they have quadratic space complexity, which makes them more efficient for this purpose than ordinary multiplication triples. Matrix Power Tuples are costly to produce during the preprocessing phase but allow the parties to take a matrix to a power with just a single round of communication during the evaluation phase. The latter technique also has a considerably simpler and cheaper variant that can be applied in cases where multiplication is commutative.

We provide a bottom-up explanation of the ideas behind Matrix Power Tuples, as well as a formal specification of both techniques in the form of protocols and ideal functionalities. We use the latter to provide a security proof in the iUC model. We also provide an introduction into the most widely used techniques for Multi-Party Computation in the modern literature.

Contents

1 Introduction

The goal of Multi-Party Computation is for a set of parties $\{P_1, ..., P_n\}$, where P_i holds input x_i, to jointly compute a function $f : (x_1, ..., x_n) \mapsto (y_1, ..., y_n)$ in such that the only thing P_i learns is y_i. This has a wide range of applications, such as allowing a company to train machine learning algorithms on sensitive data while keeping the data secret.

Presently, there are two fundamentally different approaches to realize Multi-Party Computation, one based on garbling arithmetic circuits and one based on secret sharing. In this document, we focus on the latter approach.

Within the modern literature, it is common practice to model the function f as an arithmetic circuit and to assume that the objects feeding into this circuit are elements from a finite field. However, there are practical applications where the objects of computation are matrices rather than field elements. In such cases, the standard approach is to apply existing techniques on an element basis. We demonstrate in this document that it is possible to improve performance by using techniques tailored to work for computations with matrices.

1.1 Document Overview and Structure

The remaining document is structured as follows:

- Chapter 2 aims to explain the set of techniques our improvements build upon in an easily understandable way. To do this, we describe a self-contained Multi-Party Computation scheme that assumes the objects of computation are field elements. We aim to be reasonably formal, but some details are omitted.
- Chapter 3 presents two original techniques that assume the objects of computation are matrices.
- Chapter 4 contains formal descriptions of the protocols implementing these techniques, as well as their ideal functionalities.
- Chapter 5 explains the idea of Universal Composition and provides a proof that our techniques are secure in the iUC model [5].
- The appendix contains a proof for a negative result about one of our proposed techniques. Essentially, it argues that a particular approach for further improvement is likely impossible.

1.2 Notation

The following notation will be used all throughout the document:

- The symbol $\mathbb{N}$ denotes the set of natural numbers, including 0 (i.e.: $\mathbb{N} = \{0, 1, 2, ...\}$).
- Given $k \in \mathbb{N}$, the symbol $[k]$ denotes the set $\{1, ..., k\}$. Note that $[0] = \emptyset$.
- The symbol $\mathbb{F}$ denotes a finite field. Unless specified otherwise, we implicitly assume that we are working in $\mathbb{F}$ or in the set $M_d(\mathbb{F})$ of $d \times d$ matrices over $\mathbb{F}$. I.e., since all finite fields have p^k elements for some p prime and $k \in \mathbb{N}$, we assume addition and subtraction is always done modulo some p^k. We usually use lower-case letters for elements in $\mathbb{F}$ and upper-case letters for elements in $M_d(\mathbb{F})$.
- If x is a variable appearing in a protocol,
  - the symbol $\langle x \rangle_i$ will be used to reference the i-th share of x.
  - the symbol $\langle \overline{x} \rangle_i$ will be used to reference the i-th share of $x\gamma$, where γ is the MAC-key.
  - the symbol $\mathbf{x}$ will denote a ciphertext of x.

    Note that all of the above are conventions, not operators. Formally, $x, \langle x \rangle_i, \langle \overline{x} \rangle_i, \mathbf{x}$ are four different symbols that need to be defined before they are used.
  - if $\langle x \rangle_i$ and $\langle \overline{x} \rangle_i$ are defined, then $\langle\!\langle x \rangle\!\rangle_i$ is defined implicitly to be the pair $(\langle x \rangle_i, \langle \overline{x} \rangle_i)$.
- We refer to the function that is to be evaluated as f and to the parties participating as $P_1, ..., P_n$.
- We write $x \xleftarrow{u} M$ to denote that x is sampled uniformly out of the finite set M.
- We use the brackets $\{\ \}$ (different from $\{\ \}$) to denote ownership to different parties. Thus, $\{t_i\}_{i \in M}$ indicates that t_i is input by or output to to each P_i with $i \in M$. For example, $\{\langle x \rangle_i\}_{i \in [n]}$ denotes the set of shares $\{\langle x \rangle_i\}_{i \in [n]}$ plus the information that, for each $i \in [n]$, the share $\langle x \rangle_i$ is input by or output to P_i.

Additional notation will be introduced early into some chapters.

Finally, note that we use the term "protocol" to refer to a single algorithm that corresponds to an instance in the Universal Composition model (and has its own ideal functionality), and the term "scheme" or "Multi-Party Computation scheme" to refer to the entire set of algorithms that realizes secure computation of f.

2 Current Methods

In this chapter, we present a Multi-Party Computation scheme that is meant to summarize the essential techniques in the literature. It is essentially a hybrid of the paper by Damgård et al. [9] (commonly referred to as "SPDZ"[1]) and its follow-up [8].

Note that this chapter is meant to be an intuitive entry into the field, not a formal part of the scheme we present in chapter 4. In particular, we assume we are working with elements from a finite field $\mathbb{F}$ and do not give special consideration to matrices.

Additional Notation for this chapter

To structure our presentation, we will divide the functionality which our scheme consists of into ***modules***, which loosely correspond to protocols but have a different name to emphasize that we are not being fully formal and do not provide ideal functionalities or a security proof. The key property of modules is that they have clearly defined input/output behavior so that one can treat them as black boxes if needed.

If each party P_i holds a share $\langle x \rangle_i$, we say that "the parties open x" to mean that each P_i broadcasts $\langle x \rangle_i$, and, once this is done, each P_i sets $x := \sum \langle x \rangle_i$. We will only use this phrase if the condition does, in fact, hold. Similarly, we say that "the parties open x to P_j" to mean that each party P_i sends $\langle x \rangle_i$ to P_j only, and P_j sets $x := \sum \langle x \rangle_i$.

2.1 A High-level Overview

In Multi-Party Computation based on secret sharing, the main idea is that each party P_j creates an additive secret sharing $\{\langle x_j \rangle_i\}_{i \in [n]}$ of her input x_j, i.e., $\sum_{i=1}^{n} \langle x_j \rangle_i = x_j$, and sends each other party P_i the share $\langle x_j \rangle_i$. When this is done, each P_i holds $\{\langle x_j \rangle_i\}_{j \in [n]}$, i.e., one share of each input value, so that she can evaluate f on these shares to obtain additive shares $\{\langle y_j \rangle_i\}_{j \in [n]}$ of the output values. In the end, the parties open y_i to P_i for all $i \in [n]$, so that each P_i learns the value of her output y_i but learns nothing else.

If secret-sharing were inherently homomorphic, i.e., if the equations $\langle s_j \rangle_i + \langle s_k \rangle_i = \langle s_j + s_k \rangle_i$ and $\langle s_j \rangle_i \langle s_k \rangle_i = \langle s_j s_k \rangle_i$ both held, this might be the end of the story. However, for additive secret sharing, the latter equation generally fails: if $s = \langle s \rangle_0 + \langle s \rangle_1$ and $t = \langle t \rangle_0 + \langle t \rangle_1$, then $st = \langle s \rangle_0 \langle t \rangle_0 + \langle s \rangle_0 \langle t \rangle_1 + \langle s \rangle_1 \langle t \rangle_0 + \langle s \rangle_1 \langle t \rangle_1$, which shows that $\langle s \rangle_0 \langle t \rangle_0$ and $\langle s \rangle_1 \langle t \rangle_1$ are unlikely to be correct shares of the secret st.

[1] The abbreviation is based on last names of the authors (Nigel Smart, Valerio Pastro, Ivan Damgård, and Sarah Zakarias).

The state of the art technique to remedy this problem has been dubbed *multiplication triples*. For each multiplication that occurs in the arithmetic circuit of f, one assumes there is a triple (a, b, c) of numbers such that $ab = c$ and each P_i holds additive shares $(\langle a\rangle_i, \langle b\rangle_i, \langle c\rangle_i)$, but no party knows the full value of a or b or c. Now, assume each P_i holds $\langle s\rangle_i$ and $\langle t\rangle_i$, and the goal is for each P_i to obtain $\langle st\rangle_i$. Instead of multiplying her shares locally, each P_i computes $\langle s - a\rangle_i := \langle s\rangle_i - \langle a\rangle_i$ and $\langle t - b\rangle_i := \langle t\rangle_i - \langle b\rangle_i$, then the parties open the values $s - a$ and $t - b$. At this point, each P_i computes $\langle st\rangle_i := (s - a)\langle t\rangle_i + (t - b)\langle a\rangle_i + \langle c\rangle_i = s\langle t\rangle_i - a\langle t\rangle_i + t\langle a\rangle_i - b\langle a\rangle_i + \langle c\rangle_i$. Now, we have $\sum_{i=1}^{n}\langle st\rangle_i = s\sum\langle t\rangle_i - a\sum\langle t\rangle_i + t\sum\langle a\rangle_i - b\sum\langle a\rangle_i + \sum\langle c\rangle_i = st - at + ta - ba + c = st$.[2] This shows that the above definition of $\langle st\rangle_i$ "works" (unlike the definition $\langle st\rangle_i := \langle s\rangle_i\langle t\rangle_i$). The cost of applying this technique is a full round of communication to open $s - a$ and $t - b$.

If we consider ***mpc*** the module implementing the entire Multi-Party Computation scheme, it now factors into two smaller modules. First, ***preprocessing*** is called, which takes as input a number $\alpha_m \in \mathbb{N}$ (that should equal the number of multiplication gates in the arithmetic circuit of f) and outputs α_m many multiplication triples. This is commonly called the "preprocessing phase" or "offline phase" (even though it requires plenty of communication). Then, ***evaluate_circuit*** is called, which takes as input the α_m multiplication triples along with the inputs $\{\langle x_j\rangle_i\}_{j\in[n]}$ for each P_i, and it outputs $\{\langle y_j\rangle_i\}_{j\in[n]}$ to each P_i. To achieve this, every time a multiplication is to be performed, a fresh multiplication triple is utilized (and thereby "consumed") in the way we described above. This second phase is commonly called the "evaluation phase" or "online phase."

We will put the functionality that performs multiplication into its own module, which we simply dub ***multiply***. To be precise, this module takes $\{\langle s\rangle_i, \langle t\rangle_i\}_{i\in[n]}$ and $\{(\langle a\rangle_i, \langle b\rangle_i, \langle c\rangle_i)\}_{i\in[n]}$ as input, and it outputs $\{\langle st\rangle_i\}_{i\in[n]}$.

Before we look into these three modules, we first introduce an additional technique.

2.2 Adding Message Authentication Codes

One weakness of the scheme up to this point is that it can at best achieve what is called *passive security*: if every party follows the protocol perfectly, then f will be evaluated correctly, and no information will be leaked (even if several parties combine their knowledge after execution has finished). However, any party can choose to deviate from protocol and thereby affect correctness. As an example, recall that the output for party P_j is obtained by each P_i with $i \neq j$ sending $\langle y_j\rangle_i$ to P_j and P_j computing $y_j := \sum_{j=1}^{n}\langle y_j\rangle_i$. If any one P_i is willing to lie, not only can she introduce an error into the value of y_j, but she can even control the exact size of the error by sending $\langle y_j\rangle_i + \epsilon$ rather than $\langle y_j\rangle_i$. If all other parties are behaving honestly, this will lead to P_j computing $y_j + \epsilon$ rather than y_j.

There are a number of distinct stronger levels of security that a scheme can set out to achieve. We aim for what is called *actively secure with abort*, which guarantees that any meaningful deviation from protocol will be detected. It will still be possible for any party to ruin the computation, but only to the end of aborting the scheme entirely.

[2] A convenient fact at play here is that only multiplying two *shares* is problematic – multiplying a share with a publicly known number is straight-forward as $s\langle t\rangle_i$ is a valid share of st.

To achieve active security with abort, the primary idea is to add a *Message Authentication Code*, henceforth called a MAC, to each variable that appears within the scheme. Given such a variable x, each P_i will hold a share $\langle \overline{x} \rangle_i$ of $x \cdot \gamma$ in addition to her share $\langle x \rangle_i$ of x, where $\gamma \in \mathbb{F}$. We call the number γ the *MAC key*, which is the same for each variable, and we call the number γx the *MAC of* x, which is different for each variable.[3] We assume that each P_i holds one share $\langle \gamma \rangle_i$ of the MAC key.

Before each addition or multiplication, the parties should hold shares of both the input values and their MACs. Then, the operation should be performed in such a way that the MACs are updated along with the input values, making it so the parties end up with shares of both the output values and their MACs. Once f has been evaluated fully, a module ***validate*** (a new submodule of ***evaluate_circuit***) is called, which checks that the MACs are consistent for each value that appears as the output of an algebraic operation – where consistency for a variable x is defined as the equation $\sum_{i=1}^{n} \langle \overline{x} \rangle_i = \gamma(\sum_{i=1}^{n} \langle x \rangle_i)$. With this mechanism in place, if we can prevent all parties from lying in such a way that the MACs of their values remain consistent, we ensure that any meaningful deviation from protocol will be detected. This assurance will be possible since no party knows γ.

We now demonstrate how operations can be performed while preserving consistency of the MACs. Thus, we assume all parties hold shares $\langle\!\langle s \rangle\!\rangle_i$ and $\langle\!\langle t \rangle\!\rangle_i$, and we have to show that, if these shares are consistent going in, then the resulting shares $\{\langle\!\langle s+t \rangle\!\rangle_i\}_{i \in [n]}$ or $\{\langle\!\langle st \rangle\!\rangle_i\}_{i \in [n]}$ will be consistent as well. For addition, we define $\langle\!\langle s+t \rangle\!\rangle_i := (\langle s \rangle_i + \langle t \rangle_i, \langle \overline{s} \rangle_i + \langle \overline{t} \rangle_i)$, which will do the trick as $\sum_{i=1}^{n} [\langle \overline{s} \rangle_i + \langle \overline{t} \rangle_i] = \sum_{i=1}^{n} \langle \overline{s} \rangle_i + \sum_{i=1}^{n} \langle \overline{t} \rangle_i = \gamma s + \gamma t = \gamma(\sum_{i=1}^{n} \langle s \rangle_i + \sum_{i=1}^{n} \langle t \rangle_i)$.

Of course, multiplication still requires consuming a multiplication triple. Furthermore, we now require that this triple comes with MACs of its own. That is, we require that ***preprocess*** output triples $\{(\langle\!\langle a \rangle\!\rangle_i, \langle\!\langle b \rangle\!\rangle_i, \langle\!\langle c \rangle\!\rangle_i)\}_{i \in [n]}$, rather than just $\{(\langle a \rangle_i, \langle b \rangle_i, \langle c \rangle_i)\}_{i \in [n]}$. We say that such an extended triple is consistent all three sets of shares are consistent, and we say that it is *correct* iff $(\sum_{i=1}^{n} \langle a \rangle_i)(\sum_{i=1}^{n} \langle b \rangle_i) = (\sum_{i=1}^{n} \langle c \rangle_i)$. We say that it is incorrect or inconsistent if it is not correct or consistent.

If a triple is incorrect, this will likely lead to a mistake in the output of a call to ***multiply***, and consequently in the evaluation of f. If a triple is inconsistent, this will likely lead to the module ***validate*** detecting this error. Consequently, our goal will be to ensure that every incorrect triple must also be inconsistent.[4]

We now prescribe that $\langle \overline{st} \rangle_i := (s-a)\langle \overline{t} \rangle_i + (t-b)\langle \overline{a} \rangle_i + \langle \overline{c} \rangle_i$. A look at the calculation written out in section 2.1 shows that $\sum_{i=1}^{n} \langle \overline{st} \rangle_i = \gamma st = \gamma(\sum_{i=1}^{n} \langle s \rangle_i)(\sum_{i=1}^{n} \langle t \rangle_i)$ (since the equation is identical except that γ appears exactly once in each summand).

[3] To motivate this terminology, one can consider $\overline{x}$ to be the element – or "code" – that authenticates the value x, hence a literal "message authentication code," whereas the MAC key γ is merely a static key which is used to create such codes.

[4] In particular, triples that are both incorrect and inconsistent are acceptable even though they will ruin correctness. Note that ensuring that triples are correct would not, in fact, ensure that computations they are used for will be carried out correctly. When it is time for P_i to use the share $\langle a \rangle_i$ in a computation, she can always choose to use a different value than what was output by ***preprocess*** and thereby ruin correctness. On the other hand, we can ensure that following protocol honestly is the only way for any party to maintain consistency of her shares.

2.3 Generating Matrix Triples

We now turn to the implementation of ***preprocess***. Since $s - a$ and $t - b$ are made public during a call to ***multiply***, any party who learns the values of a or b can use them to compute s or t, respectively (which are supposed to remain secret). Thus, one cannot have any one P_i construct the triples by herself. Likewise, one cannot use secret sharing to obtain these triples, since secret sharing is not homomorphic under multiplication – in fact, this is precisely the problem we're trying to solve. The solution is to use a Somewhat Homomorphic Encryption scheme that supports arithmetic circuits of multiplicative depth zero and one. In particular, we assume we have a scheme with public key $\mathfrak{p}$, private key $\mathfrak{s}$, encryption function $\mathfrak{E}$, decryption function $\mathfrak{D}$, and the following two properties:

- There exist operations for homomorphic addition $\oplus$ and multiplication $\otimes$ on ciphertexts, and decryption is possible for any ciphertext of the form

$$((\mathbf{x}_1 \oplus \cdots \oplus \mathbf{x}_r) \otimes (\mathbf{y}_1 \oplus \cdots \oplus \mathbf{y}_m)) \oplus (\mathbf{z}_1 \oplus \cdots \oplus \mathbf{z}_n)$$

  where the $\mathbf{x}_i$ and $\mathbf{y}_i$ and $\mathbf{z}_i$ are fresh encryptions of plaintexts.[5]

- There is a module ***dist_decrypt_shares*** that takes as input a ciphertext $\mathbf{x}$, the public key $\mathfrak{p}$, and shares $\langle\mathfrak{s}\rangle_i, \langle\mathfrak{s}^2\rangle_i$ from each P_i. Then, if $\mathbf{x}$ is of the form described above, it outputs a share $\langle x\rangle_i$ to each P_i such that $\sum_{i=1}^n \langle x\rangle_i = \mathfrak{D}(\mathbf{x}, \mathfrak{s})$, as well as a ciphertext $\mathbf{x}'$ with the properties that $\exists x \in \mathbb{F}$ such that $x = \mathfrak{D}(\mathbf{x}', \mathfrak{s}) = \mathfrak{D}(\mathbf{x}, \mathfrak{s})$ and $\mathbf{x}' = \mathfrak{E}(x, \mathfrak{p})$. In words, $\mathbf{x}'$ is a ciphertext that decrypts to the same value as $\mathbf{x}$ but is the encryption of a plaintext rather than the result of a multiplication (which means it can be multiplied with another ciphertext in the future). Note that the prefix "dist" indicates that decryption is *distributed*, i.e., implemented based on shares $\{\langle\mathfrak{s}\rangle_i\}_{i\in[n]}$ of the secret key rather than on the secret key itself, the postfix "shares" indicates that the output consists of shares of the plaintext rather than the plaintext itself.

Given access to such a scheme, we can give a preliminary specification the ***preprocess*** module as follows:

1. ***gen_shares*** is called, which outputs $(\mathfrak{p}, \langle\mathfrak{s}\rangle_i, \langle\mathfrak{s}^2\rangle_i)$ to each P_i
2. Each P_i chooses a random number $\langle\gamma\rangle_i$, encrypts it, broadcasts the ciphertext $\boldsymbol{\gamma}_i$, and sets $\boldsymbol{\gamma} := \bigoplus_{i=1}^n \boldsymbol{\gamma}_i$. (Now, each P_i holds a share of the MAC-key and its ciphertext.)
3. The following is repeated α_m times:
   - 3.1. Each P_i chooses random numbers $\langle a\rangle_i$ and $\langle b\rangle_i$, computes $\mathbf{a}_i := \mathfrak{E}(\langle a\rangle_i, \mathfrak{p})$ and $\mathbf{b}_i := \mathfrak{E}(\langle b\rangle_i, \mathfrak{p})$, broadcasts $\mathbf{a}_i$ and $\mathbf{b}_i$, and sets $\mathbf{a} := \sum_{i=1}^n \mathbf{a}_i$ and $\mathbf{b} := \sum_{i=1}^n \mathbf{b}_i$
   - 3.2. Each P_i computes $\mathbf{c} := \mathbf{a} \otimes \mathbf{b}$
   - 3.3. The parties call ***dist_decrypt_shares*** on $\mathbf{c}$ to obtain shares $\langle c\rangle_i$ and a fresh ciphertext $\mathbf{c}'$ with the property that $\mathfrak{D}(\mathbf{c}', \mathfrak{s}) = \mathfrak{D}(\mathbf{c}, \mathfrak{s})$.

[5] In any concrete implementation, there is also a limit to the number of additions that are supported on ciphertexts. As we will see in section 2.5, both addition and multiplication cause a certain quantity called the "noise" to grow, and decryption succeeds as long as the noise isn't too large. However, addition makes it grow linearly and multiplication quadratically.

3.4. Each P_i sets $\overline{\mathbf{a}} := \boldsymbol{\gamma} \otimes \mathbf{a}$ and $\overline{\mathbf{b}} := \boldsymbol{\gamma} \otimes \mathbf{b}$ and $\overline{\mathbf{c}} := \boldsymbol{\gamma} \otimes \mathbf{c}$.

3.5. The parties subsequently call ***dist_decrypt_shares*** on $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$ and $\overline{\mathbf{c}}$ to obtain shares $\langle \overline{a} \rangle_i$ and $\langle \overline{b} \rangle_i$ and $\langle \overline{c} \rangle_i$ (the fresh ciphertexts can be ignored).

And the output is $\{\{(\langle\!\langle a^{(k)} \rangle\!\rangle_i, \langle\!\langle b^{(k)} \rangle\!\rangle_i, \langle\!\langle c^{(k)} \rangle\!\rangle_i)\}_{k \in [\alpha_m]}, \boldsymbol{\gamma}, \langle \gamma \rangle_i\}_{i \in [n]}$.

2.4 Zero Knowledge Proofs and the Sacrificing Step

At this point, although ***preprocess*** sets things up for an evaluation phase that is actively secure with abort, it is not actively secure with abort itself. In the following, we will explain two distinct ways in which parties can deviate from protocol to affect correctness and present solutions that shut the door to these attacks.

The first problem arises during distributed decryption. For reasons that will become apparent in the upcoming section, dishonest parties can always choose to insert an additive error into a ciphertext that is to be decrypted, so that ***dist_decrypt_shares***, if called on $\mathfrak{E}(x, \mathfrak{p})$, returns shares $\{\langle x \rangle_i\}_{i \in [n]}$ and ciphertext $\mathbf{c}'$ such that $\sum_{i=1}^{n} \langle x \rangle_i = x + \delta$ and $\mathfrak{D}(\mathbf{c}') = x + \delta$ for some additive error $\delta \in \mathbb{F}$.

The approach from the previous section includes four calls to ***dist_decrypt_shares***. In the last three (on $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$ and $\overline{\mathbf{c}}$), adding an error would lead to inconsistent triples, which is an acceptable outcome. However, adding an error during the first call on $\mathbf{c}$ would yield a triple that is incorrect but consistent. The problem is that the MAC for c is computed using the fresh ciphertext returned by ***dist_decrypt_shares***, which means it will be consistent regardless of whether the decryption was correct.

The solution is a technique that has been dubbed the *sacrificing step*. The idea is that we take two triples, combine them in a particular way, and then open this combination. If the combination is zero, the first triple is correct with overwhelming probability.[6] If any check fails, the entire scheme aborts. Since the second triple is discarded, the sacrificing step does not leak any information about the first triple. The cost is that we need to generate twice as many triples as we end up using.

We now describe the ***sacrifice*** module, which takes as input triples $\{(\langle\!\langle a \rangle\!\rangle_i, \langle\!\langle b \rangle\!\rangle_i, \langle\!\langle c \rangle\!\rangle_i)\}_{i \in [n]}$ and $\{(\langle\!\langle a' \rangle\!\rangle_i, \langle\!\langle b' \rangle\!\rangle_i, \langle\!\langle c' \rangle\!\rangle_i)\}_{i \in [n]}$. First, each P_i samples and broadcasts a random element $\langle t \rangle_i \in \mathbb{F}_p$. Then, the parties open t and subsequently $ta - a'$ and $b - b'$. Next, each P_i computes and broadcasts $\langle L \rangle_i := t\langle c \rangle_i - \langle c' \rangle_i - \langle b \rangle_i (ta - a') - \langle a' \rangle_i (b - b')$. Finally, the parties open L and abort the protocol iff $L \neq 0$. Note that

$$\begin{aligned}
\sum_{i=1}^{n} \langle L \rangle_i &= \sum_{i=1}^{n} t\langle c \rangle_i - \langle c' \rangle_i - \langle b \rangle_i (ta - a') - \langle a' \rangle_i (b - b') \\
&= tc - c' - b(ta - a') - a'(b - b') \\
&= tc - c' - tab + a'b - a'b + a'b' \\
&= t(c - ab) + t(a'b' - c')
\end{aligned}$$

[6] The reverse statement is not true; dishonest parties can choose to make the sacrificing step fail even if both triples are correct.

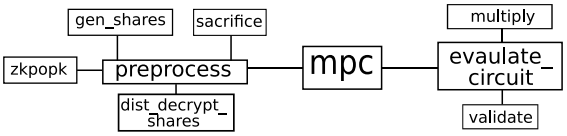

Figure 2.1: A diagram illustrating the structure of the scheme presented in chapter 2

If both triples are correct, then $c = ab$ and $c' = a'b'$ and thus $L = 0$. However, if the first triple is incorrect, then $c \neq ab$ and $L = 0 \iff t = \frac{c'-a'b'}{c-ab}$, which has probability $1/|\mathbb{F}|$ because t is uniformly distributed, and the scheme aborts with overwhelming probability.[7]

The second problem is that parties may make their ciphertexts dependent on those of others. As an example, consider the step where each P_i is instructed to sample a value $a_i \xleftarrow{u} \mathbb{F}_p$, encrypt it, and broadcast the ciphertext $\mathbf{a}_i$. We have previously assumed that the resulting sum $\bigoplus_{i=1}^{n} \mathbf{a}_i$ decrypts to a value that is uniformly distributed over $\mathbb{F}$, but in fact, this is only true if at least one of the ciphertexts $\mathbf{a}_i$ decrypts to value that is uniform over $\mathbb{F}$ and independent from all other ciphertexts. Naively, one might assume this is a given since the broadcast values are all ciphertexts which are supposed to hide the identity of their corresponding plaintexts. However, this is a case where the homomorphic properties of our encryption scheme work to our disadvantage. Concretely, suppose P_n is the last party to broadcast her share and therefore has access to the values $\mathbf{a}_1, ..., \mathbf{a}_{n-1}$. She can now choose any value $x \in \mathbb{F}$, compute $\mathbf{x} = \mathfrak{E}(\mathfrak{p}, x)$ and broadcast the ciphertext $\mathbf{a}_n := \mathbf{x} \oplus \left((-1) \cdot \bigoplus_{i=1}^{n-1} \mathbf{a}_i\right)$ (note that the encryption scheme allows multiplication of ciphertexts with constants such as -1). In that case, we have $\mathbf{a} = \bigoplus_{i=1}^{n} \mathbf{a}_i = \mathbf{x}$ and thus $\mathfrak{D}(\mathfrak{s}, \mathbf{a}) = \mathfrak{D}(\mathfrak{s}, \mathbf{x}) = x$. It follows that P_n has complete control over the value $a = \sum_{i=1}^{n} \langle a \rangle_i$ that ends up as part of the triple.

The common solution to this problem is to require each P_i to provide a *Zero Knowledge Proof of Plaintext Knowledge* for each ciphertext she broadcasts. If the ciphertext is $\mathbf{x}_i$, such a proof shows that she knows a plaintext $x \in \mathbb{F}$ such that $\mathbf{x}_i$ is a possible result of $\mathfrak{E}(x, \mathfrak{p})$, but it does not reveal the value of x. In the above example, P_n does not know $\mathfrak{D}(\mathbf{x} \oplus (-1) \cdot \bigoplus_{i=1}^{n-1} \mathbf{a}_i, \mathfrak{s})$ and is therefore unable to provide such a proof. We dub the module that handles this proof ***zkpopk***, but won't go into detail about how it is realized.

Figure 2.1 provides an overview of how the different modules work together.

2.5 The Somewhat Homomorphic Encryption Scheme

The encryption scheme used in the SPDZ family of protocols is based on the BGV scheme [1].[8] Its original formulation includes a pair of parameters (n, d) that greatly affect the structure of the resulting scheme. Of particular interest are the cases $d = 1, n > 1$ and $d > 1, n = 1$. (Note that these parameters have no relation to the n and d from our scheme.) In the following, we present a simplified version of the second case.

[7] In the scheme we present later in this document, the parties also compute a MAC for L (so that the parties cannot cheat while L it is opened and thereby correct the errors caused by wrong triples, which is otherwise possible) and the formula is altered somewhat to work for "one-and-a-half" triples $(\{\langle\!\langle a \rangle\!\rangle_i, \langle\!\langle b \rangle\!\rangle_i, \langle\!\langle c \rangle\!\rangle_i\}_{i \in [n]})$ and $(\{\langle\!\langle a' \rangle\!\rangle_i, \langle\!\langle c' \rangle\!\rangle_i\}_{i \in [n]})$ where $ab = c$ and $a'b = c'$.

[8] As with SPDZ, the abbreviation "BGV" is not from the original paper but has been given to it based on the names of its authors: Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan.

Our plaintext space will be $(\mathbb{F}_{q^k}, +, \cdot)$, the unique field with q^k elements.[9] We write $\mathcal{X}$ for $\mathbb{F}_{q^k}$ and think of the elements $x \in \mathcal{X}$ as vectors with elements in $\mathbb{F}_q$. If a plaintext is encrypted, it will become a pair, i.e., an element of the space $\mathcal{X}^2$. Note that $\mathcal{X}^2$ is not a field, so addition and multiplication need to be defined explicitly. Finally, we fix some prime number p (think of $p = |\mathbb{F}|$) smaller than q and demand for all plaintexts x that $\|x\|_\infty < p$. Note that this section is an exception to our rule that we carry out computations in the field $\mathbb{F}$.

We now define key generation, and on that basis, encryption and decryption. Frequently, terms will have summands of the form ap for some a, whose purpose is to mask various elements so that they cannot be extracted other than in the intended way. They will not influence the decrypted value because terms will always be taken modulo p before they are decrypted (this is why we demanded that that every coordinate of plaintexts must be smaller than p). The important terms will be those that are not multiples of p.

To generate a (private, public) key pair, one first samples a vector $\mathfrak{s} \xleftarrow{G} \mathcal{X}$, which will be the private key, where the symbol $\xleftarrow{G}$ means that an element is sampled according to a discrete Gaussian distribution. Then, one samples an error term $\epsilon \xleftarrow{G} \mathcal{X}$ and a vector $a \xleftarrow{u} \mathcal{X}$ and sets $b = a\mathfrak{s} + p\epsilon$. The pair (a, b) is the public key. A property that will be crucial is that $b \equiv a \cdot \mathfrak{s} \mod p$.

Next, we define the probabilistic encryption algorithm $\mathfrak{E} : \mathcal{X} \times \mathfrak{P} \to \mathcal{X}^2$, where $\mathfrak{P}$ denotes the public key space, and the deterministic decryption algorithm $\mathfrak{D} : \mathcal{X}^2 \times \mathfrak{S} \to \mathcal{X}$, where $\mathfrak{S}$ denotes the private key space. For encryption, on input $(x, (a, b))$, sample $(u, v, w) \xleftarrow{G} \mathcal{X}^3$, set $\mathbf{c}_0 := bv + pw + x$ and $\mathbf{c}_1 := av + pu$ and return $(\mathbf{c}_0, \mathbf{c}_1)$. For decryption, on input $((\mathbf{c}_0, \mathbf{c}_1), \mathfrak{s})$, return $\mathbf{c}_0 - \mathfrak{s}\mathbf{c}_1 \mod p$.

It now appears easy to check that, for any plaintext x and any valid key pair $(\mathfrak{s}, (a, b))$, one has $\mathfrak{D}\big(\mathfrak{E}(x, (a, b)), \mathfrak{s}\big) = x$. However, recall that computation takes place in the finite field $\mathcal{X}$. To illustrate why this can be a problem, suppose $p = 5$ and consider the term $(4 + 3 \cdot p) \mod 5$ in the field $\mathbb{Z}_{17}$. One might expect that it yields 4 since adding a multiple of p does not change the result – but, in fact, $4 + 3 \cdot 5$ equals 2 in $\mathbb{Z}_{17}$. On the other hand, if we were working in $\mathbb{Z}_{23}$ instead, the same problem would not arise.

In our case, numbers are taken modulo q. Thus, decryption is only guaranteed to succeed if the term $\mathbf{c}_0 - \mathfrak{s}\mathbf{c}_1$ is unaffected by the implicit modulo operation in $\mathcal{X}$. In the literature, one usually refers to this term as the *noise* of the ciphertext (even though it also contains the plaintext message) and says that the noise must stay sufficiently small for decryption to succeed.

We now define our homomorphic operations. First, we define $\oplus : \mathcal{X}^2 \times \mathcal{X}^2 \to \mathcal{X}^2$ to be component-wise addition. Then, we define $\otimes : \mathcal{X}^2 \times \mathcal{X}^2 \to \mathcal{X}^3$ to be our multiplication, and finally $\oplus : \mathcal{X}^3 \times \mathcal{X}^3 \to \mathcal{X}^3$ to be component-wise addition once again. Thus, encrypting an element in $\mathcal{X}$ yields an element in $\mathcal{X}^2$, and multiplying two such elements yields an element in $\mathcal{X}^3$ that cannot be multiplied again.

We define $\otimes$ by the rule

[9] To construct this field, one uses the fact that, if f is an irreducible polynomial over $\mathbb{Z}_q$ of degree k, the quotient ring $\mathbb{Z}_q[X]/f$ is precisely the field with q^k elements (see [1] (p7f)). However, it is possible to understand the encryption scheme without considering these details.

$$(\mathbf{c}_0, \mathbf{c}_1) \otimes (\mathbf{c}'_0, \mathbf{c}'_1) := (\mathbf{c}_0\mathbf{c}'_0, \mathbf{c}_0\mathbf{c}'_1 + \mathbf{c}'_0\mathbf{c}_1, -\mathbf{c}_1\mathbf{c}'_1)$$

for all $(\mathbf{c}_0, \mathbf{c}_1), (\mathbf{c}'_0, \mathbf{c}'_1) \in \mathcal{X}^2$. Recall from section 2.3 that we require both $\oplus$ and $\otimes$ to be homomorphic, i.e., if $E' := \mathfrak{E}(\cdot, \mathfrak{p})$ and $D' := \mathfrak{D}(\cdot, \mathfrak{s})$, we require that $D'(E'(x)+E'(y)) = x+y$ and $D'(E'(x)E'(y)) = xy$. This is straight-forward for addition since the sum $E'(x) \oplus E'(y)$ can be computed as

$$(bv+pw+x, av+pu) \oplus (bv'+pw'+y, av'+pu') = (b(v+v')+(x+y)+p(\cdots), a(v+v')+p(\cdots))$$

and it is easy to see that applying the decryption rule yields $x + y$, provided the noise is small enough. On the other hand (if we ignore all terms that are multiplies of p from the beginning), the product $E'(x) \otimes E'(y) = (bv + x, av) \otimes (bv' + y, av')$ will be

$$(b^2vv' + bvy + bv'x + xy, abvv' + av'x + abvv' + avy, -a^2vv').^{10}$$

We still need to define decryption for ciphertexts in $\mathcal{X}^3$. If one were to apply the same decryption rule as before and ignore the third coordinate, one would obtain $xy - b^2vv'$. (Turn one a into a b for each summand of the second coordinate, then subtract the second from the first). Thus, for $D : \mathcal{X}^3 \times \mathfrak{S} \to \mathcal{X}$, we prescribe the rule

$$D((\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2), \mathfrak{s}) := \mathbf{c}_0 - \mathfrak{s}\mathbf{c}_1 - \mathfrak{s}^2\mathbf{c}_2 \mod p.$$

Our second requirement for the Somewhat Homomorphic Encryption scheme was the existence of the module ***dist_decrypt_shares***. However, before we can provide an implementation, we first need a simpler module ***dist_decrypt*** that returns the plaintext itself rather than shares of the plaintext. Given a ciphertext $(\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$[11] and shares $\{\langle\mathfrak{s}\rangle_i, \langle\mathfrak{s}^2\rangle_i\}_{i\in[n]}$, party P_1 sets $x_1 := \mathbf{c}_0 - \mathbf{c}_1\langle\mathfrak{s}\rangle_1 - \mathbf{c}_2\langle\mathfrak{s}^2\rangle_1$ and each P_i with $i > 1$ sets $x_i := -\mathbf{c}_1\langle\mathfrak{s}\rangle_i - \mathbf{c}_2\langle\mathfrak{s}^2\rangle_i$. Then, each P_i publishes the value $x_i + r_ip$, where r_i is a random element.[12] Now, each party can compute the plaintext as $\sum_{i=1}^n x_i + r_ip \mod p$.

Given access to this module, ***dist_decrypt_shares*** can now be implemented as follows: on input $\{(\mathbf{x}, \langle\mathfrak{s}\rangle_i, \langle\mathfrak{s}^2\rangle_i)\}_{i\in[n]}$,

1. Each P_i samples $m_i \xleftarrow{u} \mathbb{F}$, encrypts it, broadcasts $\mathbf{m}_i$ and sets $\mathbf{m} := \bigoplus_{i=1}^n \mathbf{m}_i$
2. The parties call ***dist_decrypt*** on the ciphertext $\mathbf{x} \oplus \mathbf{m}$ to obtain the plaintext $x + m$
3. Each P_i sets $\mathbf{x}' := E(x + m, \mathfrak{p}) \ominus \mathbf{m}$ and $\langle x\rangle_i := \frac{1}{n}(x + m) - m_i$

And the output is $\{\langle x\rangle_i, \mathbf{x}'\}_{i\in[n]}$.

[10] By writing this and the former equation out fully, one can also see that $\otimes$ makes the noise grow quadratically and $\oplus$ linearly.

[11] If the ciphertext is in $\mathcal{X}^2$, one simply sets $\mathbf{c}_2 := 0$

[12] The purpose of the term r_ip is to hide information about the shares of the secret key. Since we do not provide a proof that reduces security of the cryptosystem to the Ring Learning with Errors problem, we do not go into any more detail.

3 Contributions

This chapter contains descriptions of the original contributions proposed in this document. It assumes the reader is familiar with the general ideas of Multi-Party Computation as presented in chapter 2. Our contributions come in the form of algorithms for the following two problems that can arise during the evaluation phase:

- Computing the product of two matrices
- Taking a matrix to a power

For each of the two problems, we

1. define the problem more precisely
2. briefly discuss what we consider the "naive approach" to solve this problem[13]
3. describe our proposed alternative algorithm

Note that the purpose of this chapter is to communicate the key ideas in an easily understood way. Conversely, formal descriptions can be found in chapter 4.

Figure 3.1 compares the performance of our algorithms with the naive approach. To do this, we look at three different aspects: rounds of communication in the evaluation phase ($\rightarrow$ roc), runtime in the evaluation phase ($\rightarrow$ rnt), and amount of random data required ($\rightarrow$ s/e) measured in the number of shared elements in $\mathbb{F}$. For example, a full multiplication triple $\{(\langle\!\langle a \rangle\!\rangle_i, \langle\!\langle b \rangle\!\rangle_i, \langle\!\langle c \rangle\!\rangle_i)\}_{i \in [n]}$ consists of six shared elements.

Perf.	Naive Approach			Proposed Algorithm		
	roc	rnt	s/e	roc	rnt	s/e
cmp. ST	1	$O(d^3)$	$6d^3$	1	$O(d^3)$	$6d^2$
cmp. S^m	$\leq 2\mathrm{ld}(m)$	$O(\mathrm{ld}(m)d^3)$	$\leq 12\mathrm{ld}(m)d^3$	1	$O(d(2d)^m)$	$O(d(2d)^m)$

Figure 3.1: Comparing the performance of our proposed techniques to the naive approach.

Note that exponentiation comes in two flavors. The former (presented in section 3.2.2) is more efficient and far simpler, but assumes that multiplication is commutative. The table considers the latter, which assumes elements are matrices with elements in $\mathbb{F}$. The latter is also what we formalize in chapter 4.

Justifications for the values presented in the table can be found in the upcoming sections.

[13] Note that the "naive approach" is not entirely naive but contains the most straight-forward optimizations, namely using only a single round of communication for matrix multiplication and using "galloping exponentiation" to compute the power of a matrix.

3.1 Matrix Triples

This section is about the problem of multiplying two matrices during the evaluation phase. Formally, we assume the parties hold shares $\{\langle\langle S\rangle\rangle_i\}_{i\in[n]}$ and $\{\langle\langle T\rangle\rangle_i\}_{i\in[n]}$ where $S, T \in M_d(\mathbb{F})$ and want to obtain shares $\{\langle\langle ST\rangle\rangle_i\}_{i\in[n]}$.

3.1.1 Naive Approach

The naive approach to compute a matrix product ST is to consume a fresh multiplication triple for each multiplication (of two numbers) that occurs during the computation of ST. Given two $d \times d$ matrices S and T, the coordinate (k, ℓ) of the product ST has the form

$$\sum_{j=1}^{d} s_{k,j} t_{j,\ell}.$$

Thus, d multiplications are required for each of the d^2 coordinates, leading to a total of d^3 multiplication triples ($6d^3$ shared objects) and a runtime in $O(d^3)$. Since no coordinate depends on the result of other coordinates, they can all be processed in parallel, making it so only a single round of communication is required.

3.1.2 Protocol Description

As the name indicates, we will use *matrix triples*. Since the equation proving that multiplication triples "work" only uses ring operations and $M_d(\mathbb{F})$ is a ring, the implementation is straightforward. Thus, a matrix triple has the form $\{(\langle\langle A\rangle\rangle_i, \langle\langle B\rangle\rangle_i, \langle\langle C\rangle\rangle_i)\}_{i\in[n]}$ and it is consumed by opening $S - A$ and $B - T$ and setting

$$\langle ST\rangle_i := (S - A)\langle T\rangle_i + \langle A\rangle_i (T - B) + \langle C\rangle_i$$
$$\langle \overline{ST}\rangle_i := (S - A)\langle \overline{T}\rangle_i + \langle \overline{A}\rangle_i (T - B) + \langle \overline{C}\rangle_i$$

We highlight the following two aspects:

- Since matrix multiplication is non-commutative, the precise order in the above equations is important. This is in contrast to the formula for field elements, where $\langle st\rangle_i := (s - a)\langle t\rangle_i + (t - b)\langle a\rangle_i + \langle c\rangle_i$ would also suffice.
- While the MAC for the matrices S and T is a matrix, the MAC key γ is still a number. Thus, for a matrix $X \in M_d(\mathbb{F})$, we have $\overline{X} = \gamma \cdot X$, where $\gamma \in \mathbb{F}$.[14]

It is clear that only one round of communication is required. The asymptotic runtime is that of matrix multiplication and thus $O(d^3)$, and the only shared elements are the $\{(\langle\langle A\rangle\rangle_i, \langle\langle B\rangle\rangle_i, \langle\langle C\rangle\rangle_i)\}_{i\in[n]}$ that consist of $6d^2$ elements in $\mathbb{F}$.

[14] Making the MAC key into a matrix would introduce some problems with the non-commutative nature of matrix multiplication, and while these problems may be solvable, it's not clear why that approach is worth pursuing given that having it be a number is cheaper to begin with.

3.1.3 A negative result on MACs

One might reasonably wonder whether it is necessary to have the MAC of a matrix be itself a matrix – wouldn't a smaller object do the job? Unfortunately, the answer is most likely negative. Notice that, the way MACs are used in SPDZ-based protocols, they need to be both additive and multiplicative. In other words, the MAC space needs to be a ring. If we call that space $\mathcal{R}$, then the function ϕ computing the MAC for a matrix needs to be of the form

$$\phi : M_d(\mathbb{F}) \to \mathcal{R} \text{ where } \phi(S) + \phi(T) = \phi(S+T) \text{ and } \phi(S)\phi(T) = \phi(ST)\ \forall S, T \in M_d(\mathbb{F}).$$

This type of function is also called a rng-homomorphism (different from a ring homomorphism which has the additional requirement of carrying the identity element of its domain space onto the identity element of its target space). While there are known additive functions (such as the trace, or simply the sum of all entries), and at least one well-known multiplicative function (the determinant), there are no functions which are both and whose target space is smaller and still nontrivial. This assertion is formally captured in the following theorem:

Theorem 1
Let $\mathcal{R}$ be a ring, $\mathbb{F}$ a field, and $d \in \mathbb{N}$. If $\phi : M_d(\mathbb{F}) \to \mathcal{R}$ is a rng-homomorphism, then either $\phi(M_d(\mathbb{F})) \simeq M_d(\mathbb{F})$ or $\phi(M_d(\mathbb{F})) \simeq 0_{\mathcal{R}}$, where $0_{\mathcal{R}}$ is the zero ring.

While the proof of this claim is itself very simple, it requires a few paragraphs of ring theory. For this reason, we have moved it to appendix A.

3.2 Power Tuples

This section is about the problem of taking ring elements to powers. Formally, we assume the parties hold shares $\{\langle\langle s \rangle\rangle_i\}_{i\in[n]}$ and want to obtain shares $\{\langle\langle s^m \rangle\rangle_i\}_{i\in[n]}$ for some $m \in \mathbb{N}_{\geq 2}$, where $s \in \mathcal{R}$ and $\mathcal{R}$ is a ring.

The fact that matrix multiplication is not commutative complicates this technique considerably. Therefore, we present power tuples in two flavors: regular power tuples (where we assume $\mathcal{R}$ is a commutative ring) and matrix power tuples (where we assume $\mathcal{R} = M_d(\mathbb{F})$).

3.2.1 Naive approaches

The naive approach is to treat taking powers as a subset of multiplication. To this end, we assume the implementation utilizes something analogous to galloping search: multiply S^2 with S^2 in the second step, S^4 with S^4 in the third step, and so on. After $\lfloor \mathrm{ld}(m) \rfloor$ steps, this procedure will arrive at the highest power of 2 that is at most m. For example, if $m = 15$, then $S \to S^2 \to S^4 \to S^8$ requires 3 steps, which is precisely $\mathrm{ld}(15)$ rounded down to the nearest integer. This example also illustrates that, at worst, another $\lfloor \mathrm{ld}(m) \rfloor$ steps are necessary to get to the correct matrix, namely $S^8 \to S^{12} \to S^{14} \to S^{15}$.

Consequently, the worst-case number of matrix multiplications required is upper-bounded by $2\mathrm{ld}(m)$. It follows that, for all three categories we measure, the performance of this approach equals that of the naive approach for matrix multiplication multiplied by $2\mathrm{ld}(m)$.

3.2.2 Protocol Description (Regular Power Tuples)

Given $m \in \mathbb{N}_{\geq 2}$, an m-regular power tuple has the form

$$\{(\langle\langle a \rangle\rangle_i, \langle\langle a^2 \rangle\rangle_i, ..., \langle\langle a^m \rangle\rangle_i)\}_{i \in [n]}.$$

Given shares $\{\langle s \rangle_i\}_{i \in [n]}$ of an element $s \in \mathcal{R}$ as well as shares of an m-regular power tuple, the parties can compute shares of s^m by opening $(s - a)$ and setting

$$\langle s^m \rangle_i := \sum_{j=0}^{m} \binom{m}{j} (s-a)^j \cdot \langle a^{m-j} \rangle_i$$

with the convention that $\langle a^0 \rangle_i$ is simply the number $\frac{1}{n}$.[15]

For example, if $m = 2$, then an m-regular power tuple has the form of a squaring tuple as in [11], namely $\{(\langle\langle a \rangle\rangle_i, \langle\langle a^2 \rangle\rangle_i)\}_{i \in [n]}$, and a share of s^2 can be computed by setting

$$\langle s^2 \rangle_i = (s-a)^2 \langle a^0 \rangle_i + 2(s-a)\langle a \rangle_i + \langle a^2 \rangle_i$$

It is easy to check that $\sum_{i=1}^{n} \langle s^2 \rangle_i = s^2$. Similarly, for $m = 3$ we have

$$\langle s^3 \rangle_i = (s-a)^3 \langle a^0 \rangle_i + 3(s-a)^2 \langle a \rangle_i + 3(s-a)\langle a^2 \rangle_i + \langle a^3 \rangle_i$$

and so on. Correctness follows from the equations

$$\sum_{i=1}^{n} \langle s^m \rangle_i = \sum_{j=0}^{m} \binom{m}{j} (s-a)^j \cdot a^{m-j} = \big((s-a) + a\big)^m = s^m.$$

Finally, for the corresponding MACs, the general formula is

$$\langle \overline{s^m} \rangle_i := \sum_{j=0}^{m} \binom{m}{j} (s-a)^j \cdot \langle \overline{a^{m-j}} \rangle_i$$

with $\langle \overline{a^0} \rangle_i := \langle \gamma \rangle_i$. Clearly, this approach only requires a single round of communication.

[15] This is so because the first summand in the above expression, which has this term in it, doesn't have another shared component, which means it will exist n times if all parties sum their share up. Thus, the value $\frac{1}{n}$ acts as a pseudo-share. It can be considered a sharing of the multiplicatively neutral ring element. (In the case that $\mathcal{R}$ doesn't have an element x such that $\sum_{i=1}^{n} x = 1$, a possible solution is to utilize an asymmetric sharing such as $\langle a^0 \rangle_1 = 1$ and $\langle a^0 \rangle_{i>1} = 0$.)

3.2.3 Protocol Description (Matrix Power Tuples)

In this section, we generalize the approach to work if $\mathcal{R} = M_d(\mathbb{F})$.

The problem of non-commutativity

If S and A are matrices, then the equation $(S-A)^m = \sum_{j=0}^{m} \binom{m}{j} S^j A^{m-j}$ is no longer true. In particular, we now have $\sum_{j=0}^{m} \binom{m}{j} (S-A)^j \cdot A^{m-j} \neq \big((S-A)+A\big)^m$ (this was the equation we used to prove correctness under the assumption of commutativity). As an example,

$$(S-A)^2 + 2(S-A)A + A^2 = S^2 - SA - AS + A^2 + 2SA - 2A^2 + A^2 = S^2 + (SA - AS)$$

In this case, the approach is easy to fix. Recall that $\langle S^2 \rangle_i$ was defined as the term $(S-A)^2\langle A^0\rangle_i + 2(S-A)\langle A\rangle_i + \langle A^2\rangle_i$. If we simply change this definition into

$$\langle S^2 \rangle_i := (S-A)^2\langle A^0\rangle_i + \langle A\rangle_i(S-A) + (S-A)\langle A\rangle_i + \langle A^2\rangle_i$$

the equation $\sum_{i=1}^{n} \langle S^2\rangle_i = S^2$ becomes true again. This example hints at the general fix: instead of a term of the form $\binom{m}{j}(S-A)^j A^{m-j}$ where the A's are always *behind* the $(S-A)$'s, we require them to be interleaved in all $\binom{m}{j}$ possible ways.

To simplify presentation, we will write D for $S-A$ from now on. With this notation, we can write down our requirement onto the shares $\langle S^m\rangle_i$ as follows:

$$\sum_{i=1}^{n} \langle S^m\rangle_i \stackrel{!}{=} \sum_{j=0}^{m} \left[\sum_{\mathbf{x}\in\mathbf{X}_{j,m}} \prod_{i=0}^{m-1} \mathbf{x}(i) \right] \qquad \text{where} \qquad \mathbf{X}_{j,m} := \left\{ \mathbf{x} \in \{\text{'A'}, \text{'D'}\}^m \;\middle|\; \mathbf{x}_D = j \right\}$$

where $\mathbf{x} \in \{\text{'A'}, \text{'D'}\}^m$ is called a ***matrix string.*** We write $\mathbf{x}(i)$ to denote the i-th letter (with indices beginning at 0) and $\mathbf{x}_D$ to denote the number of times 'D' occurs in $\mathbf{x}$.

For example, we require that

$$\sum_{i=1}^{n} \langle S^3\rangle_i = D^3 + AD^2 + DAD + D^2A + A^2D + ADA + D^2A + A^3$$

This equation poses a significant challenge for the definition of a share $\langle S^3\rangle_i$. On first glance, it appears to require that $\langle S^3\rangle_i$ contains the term $\langle A\rangle_i D \langle A\rangle_i$. This requirement immediately breaks the technique since $\sum_{i=1}^{n} \langle A\rangle_i D\langle A\rangle_i \neq ADA$. Informally, the issue is that A appears more than once at non-consecutive places.[16]

[16] In the case of $\langle S^3\rangle_i$, the term ADA is only problematic one; all other terms will work fine if we assume the parties hold shares of A and A^2 and A^3 (which are the shares provided by a regular m-power tuple). However, as m increases, so does the number of problematic terms. For example, if $m = 4$, we require that the parties obtain shares of ADA, A^2DA, ADA^2, and AD^2A.

Note that, if each party holds shares of some matrix X and another matrix C is publicly known, each party can locally compute shares of the matrix XC. This is done by setting $\langle (XC)_{k,\ell} \rangle_i = \langle \sum_{j=1}^{d} x_{k,j} c_{j,\ell} \rangle_i := \sum_{j=1}^{d} \langle x_{k,j} \rangle_i c_{j,\ell} \quad \forall (k,\ell) \in [d]^2$. Therefore, matrix products that begin or end with D do not require special treatment: given shares of $ADDA$, the parties can locally obtain shares of $ADDAD$ since $ADDAD = (ADDA)D$. Consequently, it suffices to consider products of the form $(A^{k_0} D^{\ell_0} \cdots A^{k_{n-1}} D^{\ell_{n-1}} A^{k_n})$ where $k_i, \ell_i > 0$.

To see why this is always possible, consider what terms such a product consists of. No matter how many matrices are multiplied, the result will be a $d \times d$ matrix where each entry consists of a sum of [products of a_i's and d_i's]. Since the d_i's are publicly known, any number of them can simply be multiplied onto existing shares by each party during the evaluation phase. Consequently, if the parties hold shares of all the terms without the d_i's (i.e., of just the a_i's), they can obtain shares of the full matrix product during the evaluation phase – and since the a_i's do not depend on the matrix S (they are just random data), they can always be created during the preprocessing phase.

For an example, suppose $A, D \in M_2(\mathbb{F})$. The product ADA has the form

$$\begin{bmatrix} a_{1,1}d_{1,1}a_{1,1} + a_{1,2}d_{2,1}a_{1,1} + a_{1,1}d_{1,2}a_{2,1} + a_{1,2}d_{2,2}a_{2,1} & a_{1,1}d_{1,1}a_{1,2} + a_{1,2}d_{2,1}a_{1,2} + a_{1,1}d_{1,2}a_{2,2} + a_{1,2}d_{2,2}a_{2,2} \\ a_{2,1}d_{1,1}a_{1,1} + a_{2,2}d_{2,1}a_{1,1} + a_{2,1}d_{1,2}a_{2,1} + a_{2,2}d_{2,2}a_{2,1} & a_{2,1}d_{1,1}a_{1,2} + a_{2,2}d_{2,1}a_{1,2} + a_{2,1}d_{1,2}a_{2,2} + a_{2,2}d_{2,2}a_{2,2} \end{bmatrix}$$

Therefore, if all parties hold shares of the following object:

$$\begin{bmatrix} a_{1,1}a_{1,1} & a_{1,2}a_{1,1} & a_{1,1}a_{2,1} & a_{1,2}a_{2,1} & a_{1,1}a_{1,2} & a_{1,2}a_{1,2} & a_{1,1}a_{2,2} & a_{1,2}a_{2,2} \\ a_{2,1}a_{1,1} & a_{2,2}a_{1,1} & a_{2,1}a_{2,1} & a_{2,2}a_{2,1} & a_{2,1}a_{1,2} & a_{2,2}a_{1,2} & a_{2,1}a_{2,2} & a_{2,2}a_{2,2} \end{bmatrix}$$

they can obtain shares of the matrix above by multiplying the terms with the d_i's and locally putting them into the correct sums.

To generalize and formalize this approach, we will introduce a new type of object called a ***skeleton matrix***. For any given matrix product represented by a matrix string beginning and ending with A – for example, $\mathbf{x}$ = 'ADDAADADDDA' – we will refer to the object which allows the parties to obtain shares of $AD^2A^2DAD^3A$ during the evaluation phase as the ***skeleton matrix corresponding to*** $\mathbf{x}$, and we will say that the skeleton matrix is ***consumed*** during this process.

Defining Skeleton Matrices

The standard matrix notation where coordinates have two indices that range from 1 to d is ill-suited for our purposes, for the following reasons:

- Our formalism will require modulo operations, which is cumbersome if indices start at 1.
- The new objects will have to grow in size since sums need to be "spread out" as in the example above.
- It will be crucial to find simple patterns in how the terms of matrix products such as the above are arranged, which is difficult if they are written in regular notation.

Therefore, we adopt the following changes: skeleton matrices will only have a single index that begins at 0 and runs through the skeleton matrix column by column, i.e.:

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \qquad \begin{bmatrix} a_0 & a_d & \cdots & a_{d^2-d} \\ a_1 & a_{d+1} & \cdots & a_{d^2-(d-1)} \\ \vdots & \vdots & & \vdots \\ a_{d-1} & a_{2d-1} & \cdots & a_{d^2-1} \end{bmatrix}$$

where the left shows a 4×4 skeleton matrix and the right a general $d \times d$ skeleton matrix. To construct skeleton matrices that mirror particular matrix products, we further define an operator $\odot$ on skeleton matrices that will work like so:

$$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \odot \begin{bmatrix} d_0 & d_1 \\ d_2 & d_3 \end{bmatrix} = \begin{bmatrix} d_0a_0 & d_1a_2 & d_2a_0 & d_3a_2 \\ d_0a_1 & d_1a_3 & d_2a_1 & d_3a_3 \end{bmatrix}$$

Thus, the height of skeleton matrices stays constant, but the width increases as they are multiplied, corresponding to the "spreading out" of sums. For $d = 2$, this means that the skeleton matrix for the matrix string 'A' will start out 2×2, but they can also be sizes 4×2 and 8×2 and 16×2, and so on. On the other hand, the skeleton matrices on the right side of the product are only allowed to be 2×2. For this reason, if the operator $\odot$ is applied to several matrices in a single term, it evaluates left-to-right by default, i.e., $A \odot B \odot C := (A \odot B) \odot C$.

Formally, given any $d, m \in \mathbb{N}_+$, any $d \times d^m$ skeleton matrix $A = (a_j)_{0 \le j < d^{m+1}}$ and any $d \times d$ skeleton matrix $D = (d_j)_{0 \le j < d^2}$, the product $A \odot D =: X = (x_j)_{0 \le j < d^{m+2}}$ is defined via

$$x_j := d_{j \text{ div } d^m} \cdot a_{j \bmod d^2} \quad \forall j \in \{0, ..., d^{m+2} - 1\}$$

where div and mod are obtained through division with rest.[17] For example, if we let $X = A \odot D$ in the above example, then $d = 2$ and $m = 1$, and so $x_5 = d_{5 \text{ div } 2} a_{5 \bmod 4} = d_2 a_1$.

If A and D are 2×2 skeleton matrices, then $A \odot D \odot A = (A \odot D) \odot A$ will have this form:

$$\begin{bmatrix} a_0d_0a_0 & a_0d_1a_2 & a_1d_2a_0 & a_1d_3a_2 & a_2d_0a_0 & a_2d_1a_2 & a_3d_2a_0 & a_3d_3a_2 \\ a_0d_0a_1 & a_0d_1a_3 & a_1d_2a_1 & a_1d_3a_3 & a_2d_0a_1 & a_2d_1a_3 & a_3d_2a_1 & a_3d_3a_3 \end{bmatrix}$$

Note that we always write the elements of the right matrix of a $\odot$ product first – this is not a formal requirement as it doesn't change the values, but it is a norm we will consistently follow. Given these definitions, the pattern that will emerge in products of skeleton matrices is highlighted in figure 3.2. The existence of this pattern is what allowed the simple definition of the $\odot$ operator, and what will allow a similarly simple algorithm to consume a skeleton matrix and reconstruct the respective matrix product.

[17] Formally, given $x, y \in \mathbb{N}_+$, let $x = ya + r$ be the unique decomposition such that $r, a \in \mathbb{N}$ and $r < a$, then $x \text{ div } y := a$ and $x \bmod y := r$.

$$\begin{bmatrix} a_0d_0\mathbf{a_0} & a_0d_1\mathbf{a_2} & a_1d_2\mathbf{a_0} & a_1d_3\mathbf{a_2} & a_2d_0\mathbf{a_0} & a_2d_1\mathbf{a_2} & a_3d_2\mathbf{a_0} & a_3d_3\mathbf{a_2} \\ a_0d_0\mathbf{a_1} & a_0d_1\mathbf{a_3} & a_1d_2\mathbf{a_1} & a_1d_3\mathbf{a_3} & a_2d_0\mathbf{a_1} & a_2d_1\mathbf{a_3} & a_3d_2\mathbf{a_1} & a_3d_3\mathbf{a_3} \end{bmatrix}$$

$$\begin{bmatrix} a_0\mathbf{d_0}a_0 & a_0\mathbf{d_1}a_2 & a_1\mathbf{d_2}a_0 & a_1\mathbf{d_3}a_2 & a_2\mathbf{d_0}a_0 & a_2\mathbf{d_1}a_2 & a_3\mathbf{d_2}a_0 & a_3\mathbf{d_3}a_2 \\ a_0\mathbf{d_0}a_1 & a_0\mathbf{d_1}a_3 & a_1\mathbf{d_2}a_1 & a_1\mathbf{d_3}a_3 & a_2\mathbf{d_0}a_1 & a_2\mathbf{d_1}a_3 & a_3\mathbf{d_2}a_1 & a_3\mathbf{d_3}a_3 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{a_0}d_0a_0 & \mathbf{a_0}d_1a_2 & \mathbf{a_1}d_2a_0 & \mathbf{a_1}d_3a_2 & \mathbf{a_2}d_0a_0 & \mathbf{a_2}d_1a_2 & \mathbf{a_3}d_2a_0 & \mathbf{a_3}d_3a_2 \\ \mathbf{a_0}d_0a_1 & \mathbf{a_0}d_1a_3 & \mathbf{a_1}d_2a_1 & \mathbf{a_1}d_3a_3 & \mathbf{a_2}d_0a_1 & \mathbf{a_2}d_1a_3 & \mathbf{a_3}d_2a_1 & \mathbf{a_3}d_3a_3 \end{bmatrix}$$

Figure 3.2: An illustration of the patterns that emerge in skeleton matrices

Constructing Skeleton Matrices for Arbitrary Matrix Products

The skeleton matrix for a particular matrix product will not have any d_i's in it. However, the existence of D's in the matrix product still changes the nature of the skeleton matrix that is required to construct the product during the evaluation phase.. In the example above, we've pretended as if we have access to the matrix D and then deleted the d_i's out of the final skeleton matrix, which yields the correct result. To do this more formally, we define the family of skeleton matrices $\{\mathbf{I}_d\}_{d\in\mathbb{N}_+}$ to be the $d \times d$ skeleton matrices consisting of all 1's. I.e.:

$$\mathbf{I}_d = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

To construct the skeleton matrix corresponding to a matrix string, we assume we are given a pair $(\mathbf{x}, d)$ as input, where $\mathbf{x} \in \{A, D\}^*$ and $d \in \mathbb{N}_+$. We begin by randomizing a $d \times d$ skeleton matrix A. Then, if $\mathbf{x} = \text{`}\mathrm{A}^{k_0}\mathrm{D}^{\ell_0} \cdots \mathrm{A}^{k_{n-1}}\mathrm{D}^{\ell_{n-1}}\mathrm{A}^{k_n}\text{'}$, we compute the product

$$A^{k_0} \odot \mathbf{I}_d \odot A^{k_1} \cdots \odot A^{k_{n-1}} \odot \mathbf{I}_d \odot A^{k_n}$$

I.e., we replace every consecutive string of D's with a single $\mathbf{I}_d$, and we deal with consecutive strings of A's by regular matrix multiplication (which means A^2 is *not* $A \odot A$ but the $d \times d$ matrix obtained from squaring A the regular way). For example, if $d = 2$ and $\mathbf{x} =$ 'ADDDA', we first sample a random skeleton matrix A, i.e.:

$$\begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix}$$

Then, we compute the product $A \odot \mathbf{I}_2$, i.e.:

$$\begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \odot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_2 & a_0 & a_2 \\ a_1 & a_3 & a_1 & a_3 \end{bmatrix}$$

Finally, we compute the product $(A \odot \mathbf{I}_2) \odot A$, i.e.:

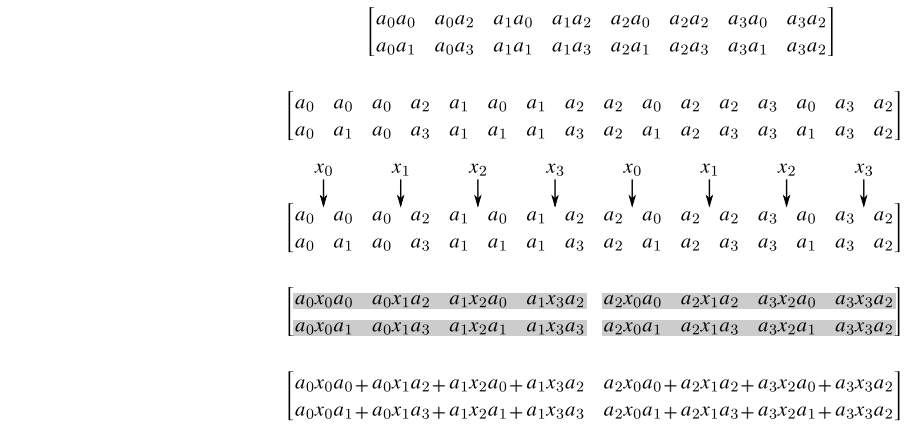

Figure 3.3: An illustration of how a skeleton matrix is "consumed" during the evaluation phase to obtain shares of the desired matrix product

$$\begin{bmatrix} a_0 & a_2 & a_0 & a_2 \\ a_1 & a_3 & a_1 & a_3 \end{bmatrix} \odot \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} = \begin{bmatrix} a_0a_0 & a_0a_2 & a_1a_0 & a_1a_2 & a_2a_0 & a_2a_2 & a_3a_0 & a_3a_2 \\ a_0a_1 & a_0a_3 & a_1a_1 & a_1a_3 & a_2a_1 & a_2a_3 & a_3a_1 & a_3a_3 \end{bmatrix}$$

The output will be the pair $(A \odot \mathbf{I}_2 \odot A, \mathbf{x})$, where $\mathbf{x}$ is needed so that the matrix AD^3A can be constructed during the evaluation phase. The reason for reducing the DDD to just $\mathbf{I}_2$ is that the matrix D^3 (or, more generally, D^n for any $n \in \mathbb{N}_+$) can simply be computed during the evaluation phase so that we can treat it as a single matrix.

Consuming Skeleton Matrices in the Evaluation Phase

Figure 3.3 illustrates how the product AD^3A is reconstructed (here $X = (x_i)_{i\in\{0,1,2,3\}} = D^3$). A general algorithm to "insert" a matrix X (such as D^3 in the preceding example) into a skeleton matrix is given in chapter 4, along with the remaining formalism.

Efficient Implementation

For any matrix power tuple, the different skeleton matrices are correlated (i.e., there are terms which appear in several matrices, such as $a_1a_1a_1$). Furthermore, the terms are highly correlated as well (e.g., $a_1a_1a_1$ and $a_1a_1a_2$ are both required and have the prefix a_1a_1). Therefore, constructing each skeleton matrix independently is suboptimal.

Instead, we will follow the following approach: given parameters $d, m \in \mathbb{N}_{\geq 2}$...

1. create a list of all terms that appear in any required skeleton matrix
2. create all such terms in a depth-first manner
3. using these terms, construct all required skeleton matrices

This approach requires a way to check whether any given term will appear in a required skeleton matrix. We describe such a method now. For this purpose, we require something a bit more formal than the notion of "term." In particular, we need something where order matters. Thus, we will deal with strings of indices, i.e., objects in $\{0, ..., d^2 - 1\}^*$. Given a term t such as $a_0a_3a_2a_1a_3$, we write s_t for the corresponding **term string**, and we assume it contains the indices in reverse order, i.e., $s_t = (3, 1, 2, 3, 0)$.

We begin by focusing on the skeleton matrix A^m only (which is $d \times d$ just as the normal matrix A^m). With the above definition at hand, we can characterize a criterion like so: given $d \in \mathbb{N}_+$, we define a function $\psi_{\text{preliminary}} : \{0, ..., d^2 - 1\}^* \to \{\top, \bot\}$ that decides whether or not a given term string appears in A^m as

$$\psi_{\text{preliminary}} : \mathbf{s} \mapsto \begin{cases} \top & ; \quad \forall i \in \{0, ..., |\mathbf{s}| - 2\} : \mathbf{s}_{i+1} \bmod d = \mathbf{s}_i \operatorname{div} d \\ \bot & ; \quad \text{otherwise} \end{cases}$$

In words, every pair of consecutive indices need to satisfy a specific condition, namely that one modulo d equals the other div d.

Because the rule is formulated for term strings, one needs to reverse it to apply it to terms (i.e., take the first index modulo d and the second div d.) If $d = 2$, this means that a_0 and a_2 need to be followed by a_0 or a_1 whereas a_1 and a_3 need to be followed by a_2 or a_3. Thus, we know that $t = a_0a_1a_2a_1a_3a_3a_3$ appears in A^7 without computing A^7 explicitly.

As an example, consider the skeleton matrix A^2 (which is identical to the regular matrix A^2 except that it is indexed with numbers $0 - 3$ rather than $(1, 1) - (2, 2)$):

$$\begin{bmatrix} a_0a_0 + a_1a_2 & a_2a_0 + a_3a_2 \\ a_0a_1 + a_1a_3 & a_2a_1 + a_3a_3 \end{bmatrix}$$

You may convince yourself that all terms that appear in this matrix follow the above rule, and all terms that follow the above rule appear in this matrix. This pattern remains true for arbitrary values of d and m.

For skeleton matrices that are not of the form A^m but include D's as well, things get somewhat more complicated. We begin with an example we've looked at previously, namely the skeleton matrix $(A \odot \mathbf{I}) \odot A$. Its corresponding string is of the form 'AD$^\ell$A' and it looks like this:

$$\begin{bmatrix} a_0a_0 & a_0a_2 & a_1a_0 & a_1a_2 & a_2a_0 & a_2a_2 & a_3a_0 & a_3a_2 \\ a_0a_1 & a_0a_3 & a_1a_1 & a_1a_3 & a_2a_1 & a_2a_3 & a_3a_1 & a_3a_3 \end{bmatrix}$$

We can see that it simply includes all combinations of two indices, regardless of the above rule. This principle always applies: if A is followed by D, any combination appears; if it is followed by another A, combinations need to satisfy the rule formulated above. For a more complex example, consider the skeleton matrix $A^2 \odot \mathbf{I} \odot A$. It corresponds to a string of the form $\mathbf{x}$:= 'AADA' [18] and it looks like this:

[18] Strictly speaking, any string of the form 'AAD$^\ell$A' with $\ell \in \mathbb{N}_+$ corresponds to this product.

$$\begin{bmatrix} & a_0a_0a_0 & a_0a_1a_2 & a_0a_2a_0 & a_0a_3a_2 & a_1a_0a_0 & a_1a_1a_2 & a_1a_2a_0 & a_1a_3a_2 & \cdots \\ & a_0a_0a_1 & a_0a_1a_3 & a_0a_2a_1 & a_0a_3a_3 & a_1a_0a_1 & a_1a_1a_3 & a_1a_2a_1 & a_1a_3a_3 & \cdots \\ \\ \cdots & a_2a_0a_0 & a_2a_1a_2 & a_2a_2a_0 & a_2a_3a_2 & a_3a_0a_0 & a_3a_1a_2 & a_3a_2a_0 & a_3a_3a_2 & \\ \cdots & a_2a_0a_1 & a_2a_1a_3 & a_2a_2a_1 & a_2a_3a_3 & a_3a_0a_1 & a_3a_1a_3 & a_3a_2a_1 & a_3a_3a_3 & \end{bmatrix}$$

where the $\cdots$ indicate that both rows are broken in two to fit on the page; the first line continues with $a_2a_0a_0$ as its 9-th entry.

This matrix also illustrates why we reverse the order for strings. Consider the term $t := a_0a_1a_2$. Since new terms are added from the left during multiplication, the a_0 corresponds to the rightmost A in the string 'AADA', the a_1 to the middle A and the a_2 to the leftmost A. There is a natural reversal, so to speak. By defining the strings of terms such that $\mathbf{s}_t = (2, 1, 0)$, we maintain the property that the first element of $\mathbf{s}_t$ corresponds to the first element of $\mathbf{x}$ – and if $\mathbf{x}'$ equals $\mathbf{x}$ with all D's deleted, then the k-th element of $\mathbf{s}_t$ corresponds to the k-th element of $\mathbf{x}'$ for all $k \in \{0, ..., |\mathbf{x}'| - 1\}$.

For strings that contain D's, the condition we have described above is necessary for every pair of indices that correspond to two consecutive A's in the matrix string. Thus, since $\mathbf{x}$ = 'AADA', the coordinate pair $(0, 1)$ of every term string needs to satisfy the condition, whereas every combination is allowed for pair $(1, 2)$.[19]

Now, term $\mathbf{s}_t$ will appear in a matrix corresponding to the matrix string $\mathbf{x}$ iff $\mathbf{x}$ has D's at all the right places. Thus, if c denotes the number of times that indices in $\mathbf{s}_t$ violate the principle, it suffices to check whether $|\mathbf{s}_t| + c$ is smaller or equal to m. If and only if this is so, it is possible to construct a matrix string $\mathbf{x}'$ whose product includes the term string $\mathbf{s}_t$ by inserting c many D's at the correct places into the string '$\mathrm{A}^{|\mathbf{s}_t|}$'. For example, given $\mathbf{s}_t = (0, 1, 2, 3)$, the principle is violated for the first two indices only ($1 \bmod 2 \neq 0 \operatorname{div} 2$), therefore we require one D inserted after the first position into the string 'AAAA'. I.e., we require the string $\mathbf{x}$:= 'ADAAA'. Since we only need to insert one D, we know that such a string exists iff $|\mathbf{s}_t| + 1 = 5 \leq m$.

With this idea in mind, we can define a general version of $\psi_{\text{preliminary}}$ like so:

$$\psi_{d,m} : \mathbf{s} \mapsto \begin{cases} \top & ; \quad |s| + E_d(s) \leq m \\ \bot & ; \quad \text{otherwise} \end{cases}$$

where E_d is the function that counts how often the rule is violated, i.e.,

$$E_d(\mathbf{s}) = |\{i \in \{0, ..., |\mathbf{s}| - 2\} : \mathbf{s}_{i+1} \bmod d \neq \mathbf{s}_i \operatorname{div} d\}|.$$

This leaves us with the problem of commutativity. For example, while the string $(3, 2, 1, 0)$ does not appear in A^4, the string $(3, 1, 0, 2)$ does. Since multiplication of field elements is commutative, we have $a_0a_1a_2a_3 = a_2a_0a_1a_3$ and so the term $a_0a_1a_2a_3$ appears in A^4 after all. However, note that re-ordering is not always possible; for example, the string $(2, 2)$ does not appear in A^2 and re-ordering it does not help at all.

[19] Conversely, if we look at terms, the second and third entry need to satisfy the condition, but the first two don't.

Absent a more sophisticated idea, one can brute-force this problem. To this end, we define an equivalence relation $\sim$ on the set of all term strings where $\mathbf{s} \sim \mathbf{s}'$ iff $\mathbf{s}'$ can be obtained from $\mathbf{s}$ by changing the order. Then, we only require one representative from each equivalence class. A possible choice is the unique representative with indices in monotonically non-decreasing order. Now, one can find all such representatives by keeping an ordered list L initialized as $()$. Then, one goes through each string $\mathbf{s} \in \{A, D\}^m$ and checks whether its representative is already in L; if it is not, one evaluates $\psi_{d,m}(\mathbf{s})$ and if it is $\top$, one adds the representative to L. This approach has a time complexity of $O^*(d^m)$ (the asterisk indicates that logarithmic factors are ignored); however, note that the resulting list depends only on d and m. Therefore, it can simply be created once and then stored in memory. This is the algorithm we will provide in chapter 4.

3.2.4 Performance

It is apparent that matrix power tuples allow the parties to obtain shares $\langle\langle S^m \rangle\rangle_i$ within a single round of communication for every $m \in \mathbb{N}_+$.

The largest skeleton matrix corresponds to a matrix term of the form 'ADAD$\cdots$ADA'. This skeleton matrix has dimensions at most $d \times d^m$. There are fewer than 2^m skeleton matrices, thus a simple upper bound on the total number of shared elements is given by $2^m \cdot d^{m+1} = d \cdot (2d)^m$.

Though we do not include this in the table, note that an improved upper-bound could be derived from figure 3.4. Let $f : \mathbb{N} \to \mathbb{N}$ be the function such that, given $m \in \mathbb{N}_+$, the image $f(m)$ equals the number of 'A' leaves in a tree of depth m, then $f(1) = f(2) = 1$ and $f(m) = f(m-1) + f(m-2)$ for $m \in \mathbb{N}_{\geq 3}$. It follows that the number of total A nodes equals the sum of the first m Fibonacci numbers, $\sum_{i=1}^{m} f(i)$, which is the same as $f(m+2) - 1$ [6]. This number is precisely the number of different skeleton matrices that need to be created since multiple concurrent D's do not lead to different skeleton matrices.

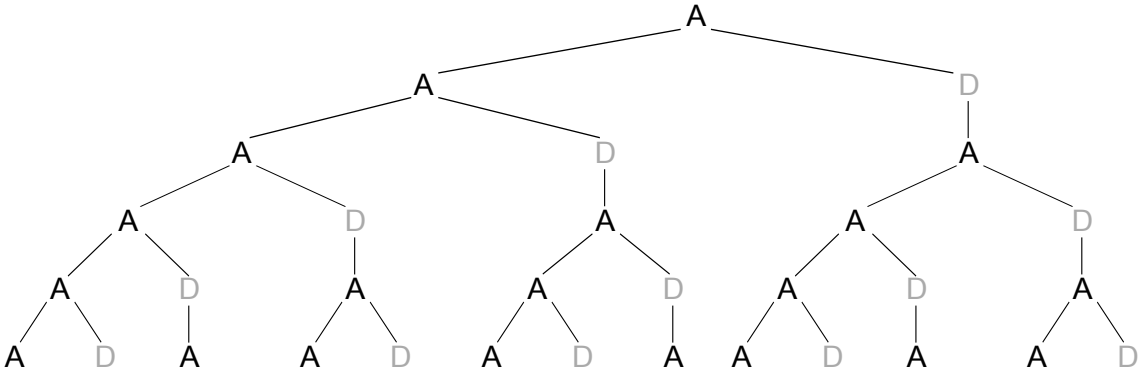

Figure 3.4: A tree where every node with an 'A' corresponds to a matrix string.

Finally, the runtime bound is the same since the algorithm A_{consume} that we present in chapter 4 consumes a skeleton matrix in time $O(d^{m+1})$.

4 The Formal MPC Scheme

In this chapter, we provide a formal description of the techniques presented in chapter 3. Since we wish to prove the resulting scheme secure in a variant of the Universal Composition model, we provide both protocols and ideal functionalities.

To specify a full Multi-Party Computation scheme, a number of additional protocols are required. For those, we do not provide our own implementations, but instead assume that one already exists in the literature – and that, if needed, it is straight-forward to modify it to work on matrices by applying the techniques component-wise. The SPDZ paper [9] provides one possible implementation for every missing protocol. This assumption of modularity is possible due to the universal composition theorem of our security model [5] (p8). An exception are the ideal functionalities $\mathcal{F}_{\text{dist\_decrypt}}$ and $\mathcal{F}_{\text{ddec\_shares}}$ that we import from [9] but provide here in our notation so that we can refer to the details during our security proof in chapter 5.

Note that this chapter includes only sparse descriptions of the protocols beyond the code itself. For a more intuitive explanation, the reader is referred to chapters 2 and 3.

4.1 Assumptions and Primitives

To simplify presentation, we assume we have access to the following constructs that we utilize as black boxes:

- A commitment scheme $\mathcal{F}_{\text{commit}}$ that is computationally hiding and binding
- A somewhat homomorphic encryption scheme with multiplicative depth 1
- A zero knowledge proof of plaintext knowledge

For implementations on all of these, we refer to [9]. The remainder of this section describes how they are utilized.

4.1.1 The commitment scheme

We will use commitments exclusively as a hidden mechanism for opening values.

The basic approach to "open" a value x is that all parties broadcast their shares $\langle x \rangle_i$ and then set $x := \sum_{i=1}^{n} \langle x \rangle_i$. The problem lies in the fact that dishonest parties can simply broadcast incorrect shares. If they know the shares of the honest parties, they can control the resulting value of x entirely; if not, they can still induce an additive error. It follows that a combination of the following two mechanisms:

- a MAC-check to catch additive errors
- a commitment scheme to prevent dishonest parties from sending shares that are dependent on those of other parties

is sufficient to rule out errors entirely since it restricts dishonest parties to additive errors while also making additive errors infeasible.

Concretely, we define the symbol O as opening a value while utilizing these two mechanisms. Given a value x, the command $O(x)$ may only be used whenever the following two conditions are fulfilled:

1. Each P_i holds $\langle x \rangle_i$
2. Each P_i holds $\langle \overline{x} \rangle_i$

and it corresponds to executing the following steps in order:

1. Each P_i uses $\mathcal{F}_{\text{commit}}$ on $\langle x \rangle_i$ to obtain commitment c_i and handle o_i
2. Each P_i broadcasts c_i (in any order)
3. Each P_i broadcasts $(\langle x \rangle_i, o_i)$ (in any order)
4. Each P_i uses $\mathcal{F}_{\text{commit}}$ to check that, for all $j \in [n]$, the handle o_j is correct for the commitment c_j
    - If any check fails, the protocol aborts
5. Each P_i sets $x := \sum_{i=1}^{n} \langle x \rangle_i$
6. Each P_i locally stores $\langle \overline{x} \rangle_i$ to use in the MAC check at the end of the scheme

Consequently, whenever the symbol O is used, one can assume that either the opened value is error-free or the MAC check at the end of the scheme will fail.

In some cases, an additive error is acceptable. Thus, we introduce the symbol O_{nomac} to open a value as above but excluding the MAC check, i.e., excluding condition #2 and step #6. If this symbol is used, we assume dishonest parties get to choose the shares they broadcast freely, but without seeing those of others.

4.1.2 The somewhat homomorphic encryption scheme

We assume we have access to a cryptosystem with the properties specified in [9] (p7-10). In particular, we assume that it supports multiplying ciphertexts once. We write $\mathfrak{E}$ for the encryption function, $\mathfrak{D}$ for the decryption function, $\mathfrak{p}$ to denote a public key, and $\mathfrak{s}$ to denote a private key. It will be possible to prove our scheme secure without going into significant depth about how encryption is implemented. Note that we will pretend the encryption scheme takes elements of $\mathbb{F}$ as input despite the fact that concrete implementations usually work on vectors (this is common practice in the literature).

4.1.3 Zero Knowledge Proofs

We assume the parties have access to a zero knowledge proof of plaintext knowledge with the properties listed in [9] (p11). Concretely, we write $\forall P_i : \mathcal{Z}(\mathbf{x}_i)$ to indicate that each party broadcasts $\mathbf{x}_i$ and subsequently performs such a proof on the ciphertext $\mathbf{x}_i$ where she acts as the prover and all other parties as the verifier. We assume that, if P_i performs a zero knowledge proof for ciphertext $\mathbf{x}_i$, then with overwhelming probability, she knows a plaintext $x_i \in \mathbb{F}$ such that $\mathbf{x}_i$ is a possible result of $\mathfrak{E}(x_i, \mathfrak{p})$. (This notion will be formalized more in chapter 5.) This restriction is important to ensure that the ciphertexts chosen by dishonest parties are independent of those chosen by honest parties.

4.1.4 Further Notation for this chapter

For a given algorithm [name], the formal protocol will be called $\Pi_{[\text{name}]}$ and the corresponding ideal functionality $\mathcal{F}_{[\text{name}]}$. Each protocol begins by specifying its input parameters and ends by specifying its output parameters. For inputs, we write $x : M$ to denote that a variable x that belongs to the set M is input. We write $\mathbf{C}$ to denote the space of all ciphertexts of elements in $\mathbb{F}$. We assume each party P_i holds the public key $\mathfrak{p}$ for encryption, shares $\langle \mathfrak{s} \rangle_i$ and $\langle \mathfrak{s}^2 \rangle_i$ of the private key, and an encryption $\boldsymbol{\gamma}$ of the MAC-key γ (and thus don't add them as input variables). Ideal functionalities always have identical inputs/outputs as their corresponding protocols.

We use $\{\langle x \rangle_i\}_{i \in I} \xleftarrow{u/c} \mathbb{F}^n$ for some set $I \subseteq [n]$ to denote that the set of shares are sampled uniformly under the condition that $\sum_{i=1}^n \langle x \rangle_i = x$. (Thus, this will only be used if x and all shares $\langle x \rangle_i$ for $i \notin I$ are already defined.) This is equivalent to sampling all but one share (say share j) uniformly, and then setting $\langle x \rangle_j := x - \sum_{i=1, i \neq j}^n \langle x \rangle_i$.[20] The $\xleftarrow{u/c}$ command will not be used in protocols, but it will play a critical role in ideal functionalities and will also appear in the code of simulators in chapter 5.

We write I_{hst} and I_{adv} to denote the set of indices of honest and dishonest (i.e., corrupt) parties, respectively. We assume these sets are static and that $I_{\text{adv}} \subsetneq [n]$. We write **[abort]** (always within conditional branches of the code) to denote that the entire scheme aborts at this point. We write := for deterministic computations and $\leftarrow$ for assignments involving randomness. We write $\oplus$, $\ominus$, $\otimes$ for homomorphic operations in the ciphertext space $\mathbf{C}$.

If an ideal functionality or protocol calls another ideal functionality $\mathcal{F}_{[\text{name}]}$, we denote this by writing $\leftarrow \mathcal{F}_{[\text{name}]}$ and putting the output of the call to the left of the $\leftarrow$ symbol. Note that, if such a call occurs within an ideal functionality, it does not define an output channel (hence why we use regular { } brackets rather than the ⦃ ⦄ brackets that denote ownership). Instead, such a call is equivalent to in-lining the respective code minus the **Output:** line in the end.

[20] To see that it doesn't matter which one is sampled last, consider some possible assignment $(v_i)_{i \in I}$ of the shares $\langle x \rangle_i$ with $x \in I$. If $\sum_{i \notin I} \langle x \rangle_i + \sum_{i \in I} v_i = x$, then $\Pr[\langle x \rangle_i = v_i \quad \forall i \in I] = \frac{1}{|\mathbb{F}|^{|I|-1}}$ (for all $i \in I$ that are not sampled last, the chance that $\langle x \rangle_i = v_i$ equals $\frac{1}{|\mathbb{F}|}$ and then the last share $\langle x \rangle_j$ must be set to v_j to satisfy the equation). Conversely, if $\sum_{i \notin I} \langle x \rangle_i + \sum_{i \in I} v_i \neq x$ then $\Pr[\langle x \rangle_i = v_i \quad \forall i \in I] = 0$ (regardless of which $\langle x \rangle_j$ is sampled last, if $\langle x \rangle_i = v_i$ for all i before it, then $\langle x \rangle_j$ cannot be v_i due to the inequality above).

Whenever values $x_{k,\ell}$ are defined for some variable x and all $(k,\ell) \in [d] \times [d]$, we implicitly refer to the matrix $(x_{k,\ell})_{1 \le k,\ell \le d}$ by the uppercase letter of the variable, in this case X. This will most commonly be done for matrices that the parties compute or randomize (typically denoted A, A', B, C, or C') but sometime also for error terms $\delta_{k,\ell}$ (in which case the matrix is Δ).

For **protocols**, the prefix in each line will indicate which parties executes it. $\forall P_i$: will mean that a command is to be executed by all parties.

For **ideal functionalities**, we use commands of the form $x \leftarrow \mathcal{S}$ or $x \rightarrow \mathcal{S}$ to denote explicit communication with the adversary/simulator (we will refer to it as the simulator), where $\leftarrow$ indicates that the simulator sends a value, and $\rightarrow$ that it receives a value. Typically, the communication consists of fewer bits than what is communicated in the protocol. For example, if the parties perform zero knowledge proofs in the protocol, the ideal functionality may merely include a line $\phi_{\text{zkcheat}} \leftarrow \mathcal{S}$. The simulator will then check whether any zero knowledge proof fails and send $\top$ if it does and $\bot$ otherwise, and the ideal functionality will check whether $\phi_{\text{zkcheat}} = \top$ and abort if the answer is affirmative.

Note that there are only two output channels of ideal functionalities: the **Output:** at the end and the explicit communication with the simulator (via $x \rightarrow \mathcal{S}$).

4.2 Ideal Functionalities and Protocols

The rest of this chapter lists all ideal functionalities and protocols that we provide in this document with sparse commentary.

4.2.1 Distributed Decryption

$\mathcal{F}_{\text{dist\_decrypt}}$

0 **Input:** $\mathbf{x} : \mathbf{C}$

1 $x := \mathfrak{D}(\mathbf{x}, \mathfrak{s})$

2 $(x, \mathbf{x}) \rightarrow \mathcal{S}$

3 $\delta \leftarrow \mathcal{S}$

4 **Output:** $\{x + \delta\}_{i \in [n]}$

Figure 4.1: The ideal functionality for distributed decryption

The ideal functionality (taken from [9] (p10)) is provided in figure 4.1. Note that we let the simulator choose an additive error rather than the decryption itself, but since the simulator knows the true result of the decryption, this is easily seen to be equivalent.

4.2.2 Distributed Decryption Returning Shares of the Plaintext

The ideal functionality is provided in figure 4.2, the protocol in figure 4.3.

Note that the protocol appears in almost identical form in [9] (p12). However, the paper only considers it a sub-protocol of the triple generation protocol and thus does not provide an ideal functionality or a security proof. For our purposes, having an ideal functionality is essential for later protocols, so we provide a proof of our own in chapter 5.

$\mathcal{F}_{\text{ddec\_shares}}$

Input: $\phi_{\text{ciphertext}} : \{\top, \bot\}, \mathbf{x} : \mathbf{C}$

$\phi_{\text{zkcheat}} \leftarrow \mathcal{S}$ # ↓ modeling failed zero knowledge proofs

`if` $\phi_{\text{zkcheat}} = \top$ `:`

  [abort]

`endif`

$\delta \leftarrow \mathcal{S}$

$x := \mathfrak{D}(\mathbf{x}, \mathfrak{s}) + \delta$

$\{\langle x \rangle_i\}_{i \in I_{\text{adv}}} \leftarrow \mathcal{S}$

$\forall i \in I_{\text{hst}} : \langle x \rangle_i \xleftarrow{u/c} \mathbb{F}$

`if` $\phi_{\text{ciphertext}} = \bot$ `:`

  Output: $\{\langle x \rangle_i\}_{i \in [n]}$

`else:`

  Output: $\{\mathfrak{E}(x, \mathfrak{p}), \langle x \rangle_i\}_{i \in [n]}$

`endif`

Figure 4.2: The ideal functionality for distributed decryption that returns shares of the decrypted value and may also return a new ciphertext with limited noise.

$\Pi_{\text{ddec\_shares}}$

0 **Input:** $\phi_{\text{ciphertext}} : \{\top, \bot\}, \mathbf{x} : \mathbf{C}$

1 $\forall P_i \ : \ \langle m \rangle_i \xleftarrow{u} \mathbb{F}$

2 $\forall P_i \ : \ \langle \mathbf{m} \rangle_i \leftarrow \mathfrak{E}(\langle m \rangle_i, \mathfrak{p})$

3 $\forall P_i \ : \ \mathcal{Z}(\langle \mathbf{m} \rangle_i)$ # broadcast $\langle \mathbf{m} \rangle_i$ and perform a zkpopk

4 $\forall P_i \ : \ \mathbf{m} := \sum_{i=1}^{n} \langle \mathbf{m} \rangle_i$

5 $\{x + m\}_{i \in [n]} \leftarrow \mathcal{F}_{\text{dist\_decrypt}}(\mathbf{x} \oplus \mathbf{m})$

6 $\forall P_i \ : \ \langle x \rangle_i := \frac{1}{n}(x + m) - \langle m \rangle_i$

7 `if` $\phi_{\text{ciphertext}} = \bot$:

8 **Output:** $\{\langle x \rangle_i\}_{i \in [n]}$

9 `else:`

10 $\forall P_i \ : \ \mathbf{x} \leftarrow \mathfrak{E}(x + m, \mathfrak{p}) \ominus \mathbf{m}$

11 **Output:** $\{\mathbf{x}, \langle x \rangle_i\}_{i \in [n]}$

12 `endif`

Figure 4.3: The protocol for distributed decryption that returns shares of the decrypted value and may also return a new ciphertext with limited noise.

4.2.3 Matrix Triple Consumption

The ideal functionality is provided in figure 4.5, the protocol in figure 4.4.

$\Pi_{\text{consume\_matrix\_triple}}$

0 **Input:** $\{\langle\langle S \rangle\rangle_i, \langle\langle T \rangle\rangle_i, (\langle\langle A \rangle\rangle_i, \langle\langle B \rangle\rangle_i, \langle\langle C \rangle\rangle_i)\}_{i \in [n]}$

1 $\mathcal{O}(S - A) \qquad \mathcal{O}(T - B)$

2 $\forall P_i \ : \ \langle ST \rangle_i := (S - A) \cdot \langle T \rangle_i + \langle A \rangle_i \cdot (T - B) + \langle C \rangle_i$

3 $\forall P_i \ : \ \langle \overline{ST} \rangle_i := (S - A) \cdot \langle \overline{T} \rangle_i + \langle \overline{A} \rangle_i \cdot (T - B) + \langle \overline{C} \rangle_i$

4 **Output:** $\{\langle\langle ST \rangle\rangle_i\}_{i \in [n]}$

Figure 4.4: The protocol for consuming a matrix triple to obtain shares of the product of two matrices

$\mathcal{F}_{\text{consume\_matrix\_triple}}$

0 **Input:** $\{\langle\langle S\rangle\rangle_i, \langle\langle T\rangle\rangle_i, (\langle\langle A\rangle\rangle_i, \langle\langle B\rangle\rangle_i, \langle\langle C\rangle\rangle_i)\}_{i\in[n]}$

1 $S := \sum_{i=1}^n \langle S\rangle_i \quad T := \sum_{i=1}^n \langle T\rangle_i \quad A := \sum_{i=1}^n \langle A\rangle_i \quad B := \sum_{i=1}^n \langle B\rangle_i$

2 $\forall i \in I_{\text{adv}} \; : \; \langle ST\rangle_i := (S-A)\cdot\langle T\rangle_i + \langle A\rangle_i \cdot (T-B) + \langle C\rangle_i$

3 $\forall i \in I_{\text{adv}} \; : \; \langle \overline{ST}\rangle_i := (S-A)\cdot\langle \overline{T}\rangle_i + \langle \overline{A}\rangle_i \cdot (T-B) + \langle \overline{C}\rangle_i$

4 $\forall i \in I_{\text{hst}} \; : \; \langle ST\rangle_i \xleftarrow{u/c} \mathbb{F}^{d^2} \qquad \langle \overline{ST}\rangle_i \xleftarrow{u/c} \mathbb{F}^{d^2}$

5 **Output:** $\{\langle\langle ST\rangle\rangle_i\}_{i\in[n]}$

Figure 4.5: The ideal functionality for consuming a matrix triple to obtain shares of the product of two matrices

Note that correctness of the triple ia a precondition. Put differently, it is only true that $\Pi_{\text{consume\_matrix\_triple}}$ emulates $\mathcal{F}_{\text{consume\_matrix\_triple}}$ if $(\sum_{i=1}^n \langle A\rangle_i)(\sum_{i=1}^n \langle B\rangle_i) = (\sum_{i=1}^n \langle C\rangle_i)$. Otherwise, the protocol would compute incorrect shares $\{\langle ST\rangle_i\}_{i\in[n]}$, whereas the ideal functionality will always compute correct shares due to the $\xleftarrow{u/c}$ command in line 4.

4.2.4 Matrix Triple Generation

The ideal functionality is provided in figure 4.6, the protocol in figure 4.7.

We briefly highlight a particular aspect of this protocol. In the sacrificing step (line 18), the parties compute and open the following value:

$$\langle Z\rangle_i := r\langle C\rangle_i - \langle C'\rangle_i - (rA - A')\langle B\rangle_i$$

Suppose the dishonest parties induce an additive error during the step where the matrix $rA - A'$ is opened, so that $\sum_{i=1}^n \langle rA - A'\rangle_i = rA - A' + \Delta^*$. Suppose further that A, A', B, C, C' are all computed correctly (i.e., the dishonest parties did not induce decryption errors). Then, the matrix Z will come out as $\sum_{i=1}^n \Delta^*\langle B\rangle_i = \Delta^* B$. Since the dishonest parties know (and even get to control the precise value of) the matrix Δ^*, this means they can learn B (simply choose $\Delta^* = I_d$). This is the reason why the ideal functionality explicitly sends the respective information to the simulator (lines 15-16). Without this, it would not be true that the protocol emulates the ideal functionality.

Fortunately, cheating in this way will cause the scheme to abort, in which case the matrix B will not be used for anything. Therefore, this leakage of information does not meaningfully compromise security.

$\mathcal{F}_{\text{g\_matrix\_triple}}$

Input: —

$\phi_{\text{zkcheat}} \leftarrow \mathcal{S}$ # ↓ modeling failed zero knowledge proofs

`if` $\phi_{\text{zkcheat}} = \top$:

[abort]

`endif`

$\forall i \in I_{\text{adv}} : (\langle A\rangle_i, \langle B\rangle_i) \leftarrow \mathcal{S}$

$\forall i \in I_{\text{hst}} : (\langle A\rangle_i, \langle B\rangle_i) \xleftarrow{u} \mathbb{F}^{2d^2}$

$A := \sum_{i=1}^{n} \langle A\rangle_i \quad B := \sum_{i=1}^{n} \langle B\rangle_i$

$\{\langle c_{k,\ell}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \mathfrak{E}((AB)_{k,\ell}, \mathfrak{p})) \quad \forall k,\ell$

$c_{k,\ell} := \sum_{i=1}^{n} \langle c_{k,\ell}\rangle_i \quad \forall k,\ell$ # Note that the $c_{k,\ell}$ may include decryption errors

$\{\langle \overline{a_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \mathfrak{E}(\gamma a_{k,\ell}, \mathfrak{p})) \quad \forall k,\ell$

$\{\langle \overline{b_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \mathfrak{E}(\gamma b_{k,\ell}, \mathfrak{p})) \quad \forall k,\ell$

$\{\langle \overline{c_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \mathfrak{E}(\gamma c_{k,\ell}, \mathfrak{p})) \quad \forall k,\ell$

$\phi_{\text{failcheck}} \leftarrow \mathcal{S}$ # ↓ sacrificing step

`if` $C \neq AB \ \vee \ \phi_{\text{failcheck}} = \top$:

$L \leftarrow \mathcal{S}$ # L is a $d \times d$ matrix

$\left(\sum_{i\in I_{\text{hst}}} L \cdot \langle B\rangle_i\right) \rightarrow \mathcal{S}$

[abort]

`endif`

Output: $\{(\langle\!\langle A\rangle\!\rangle_i, \langle\!\langle B\rangle\!\rangle_i, \langle\!\langle C\rangle\!\rangle_i)\}_{i\in[n]}$

Figure 4.6: The ideal functionality for generating a matrix triple

$\Pi_{\text{g\_matrix\_triple}}$

Input: —

$\forall P_i$: $(\langle A\rangle_i, \langle B\rangle_i, \langle A'\rangle_i) \xleftarrow{u} \mathbb{F}^{3d^2}$ # $\mathfrak{E}(M,\mathfrak{p})$ for $M \in M_d(\mathbb{F}) \triangleq$ component-wise encr.

$\forall P_i$: $\langle \mathbf{A}\rangle_i \leftarrow \mathfrak{E}(\langle A\rangle_i, \mathfrak{p})$ $\langle \mathbf{B}\rangle_i \leftarrow \mathfrak{E}(\langle B\rangle_i, \mathfrak{p})$ $\langle \mathbf{A}'\rangle_i \leftarrow \mathfrak{E}(\langle A'\rangle_i, \mathfrak{p})$

$\forall P_i$: $\mathcal{Z}(\langle \mathbf{a}_{k,\ell}\rangle_i)$ $\mathcal{Z}(\langle \mathbf{b}_{k,\ell}\rangle_i)$ $\mathcal{Z}(\langle \mathbf{a}'_{k,\ell}\rangle_i)$ $\forall k,\ell$

$\forall P_i$: $\mathbf{A} := \sum_{i=1}^n \langle \mathbf{A}\rangle_i$ $\mathbf{B} := \sum_{i=1}^n \langle \mathbf{B}\rangle_i$ $\mathbf{A}' := \sum_{i=1}^n \langle \mathbf{A}'\rangle_i$

$\forall P_i$: $\mathbf{C} := \mathbf{A} \otimes \mathbf{B}$ # i.e.: $\mathbf{c}_{k,\ell} := \bigoplus_{j=1}^n (\mathbf{a}_{k,j} \otimes \mathbf{b}_{j,\ell})$ $\forall k,\ell$

$\forall P_i$: $\mathbf{C}' := \mathbf{A}' \otimes \mathbf{B}$ # i.e.: $\mathbf{c}'_{k,\ell} := \bigoplus_{j=1}^n (\mathbf{a}'_{k,j} \otimes \mathbf{b}_{j,\ell})$ $\forall k,\ell$

$\{\langle c_{k,\ell}\rangle_i, \mathbf{c}_{k,\ell}\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\top, \mathbf{c}_{k,\ell})$ $\forall k,\ell$

$\{\langle c'_{k,\ell}\rangle_i, \mathbf{c}'_{k,\ell}\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\top, \mathbf{c}'_{k,\ell})$ $\forall k,\ell$

$\forall P_i$: $\overline{\mathbf{a}_{k,\ell}} := \mathbf{a}_{k,\ell} \otimes \boldsymbol{\gamma}$ $\overline{\mathbf{b}_{k,\ell}} := \mathbf{b}_{k,\ell} \otimes \boldsymbol{\gamma}$ $\forall k,\ell$ # $\downarrow$ MACs

$\forall P_i$: $\overline{\mathbf{c}_{k,\ell}} := \mathbf{c}_{k,\ell} \otimes \boldsymbol{\gamma}$ $\overline{\mathbf{c}'_{k,\ell}} := \mathbf{c}'_{k,\ell} \otimes \boldsymbol{\gamma}$ $\forall k,\ell$

$\{\langle \overline{a_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \overline{\mathbf{a}_{k,\ell}})$ $\forall k,\ell$

$\{\langle \overline{b_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \overline{\mathbf{b}_{k,\ell}})$ $\forall k,\ell$

$\{\langle \overline{c_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \overline{\mathbf{c}_{k,\ell}})$ $\forall k,\ell$

$\{\langle \overline{c'_{k,\ell}}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\bot, \overline{\mathbf{c}'_{k,\ell}})$ $\forall k,\ell$

$\forall P_i$: $\langle r\rangle_i \xleftarrow{u} \mathbb{F}$ # $\downarrow$ sacrificing step

$O_{\text{nomac}}(r)$

$O_{\text{nomac}}(rA - A')$

$\forall P_i$: $\langle Z\rangle_i := r\langle C\rangle_i - \langle C'\rangle_i - (rA - A')\langle B\rangle_i$

$\forall P_i$: $\langle \overline{Z}\rangle_i := r\langle \overline{C}\rangle_i - \langle \overline{C'}\rangle_i - (rA - A')\langle \overline{B}\rangle_i$

$O(Z)$

`if` $Z \neq 0_{d\times d}$:

[**abort**]

`endif`

Output: $\{(\langle\!\langle A\rangle\!\rangle_i, \langle\!\langle B\rangle\!\rangle_i, \langle\!\langle C\rangle\!\rangle_i)\}_{i\in[n]}$

Figure 4.7: The protocol for generating a matrix triple

On the implementation of Matrix Power Tuples

We now take two pages to discuss our implementation of matrix power tuples to ease understanding of the code. Whereas section 3.2.3 focused on the ideas, here we focus on implementation details and notation.

Unlike with matrix triples, generating and utilizing matrix power tuples requires performing a nontrivial amount of local and deterministic computation. To keep our protocols and ideal functionalities as short as possible, we combine this code into six algorithms that we present in section 4.3. We refer to them as $A_{[\text{name}]}$.

As with matrix triples, we divide the process of using matrix power tuples into the generation step (executed during the preprocessing phase) and the consumption step (executed during the evaluation phase).

The generation step (implemented by $\Pi_{\text{g\_matrix\_power\_tuple}}$, figures 4.12, 4.13) can itself be divided into four steps, one where all required terms are listed, one where they are generated, one where they are verified (the sacrificing step), and one where they are put into skeleton matrices.

Step 1 is straight-forward: $\Pi_{\text{g\_matrix\_power\_tuple}}$ calls $A_{\text{list\_terms}}$ with parameters m, d, ℓ to obtain all required term strings of length $\ell \in [m]$. Here, $A_{\text{list\_terms}}$ does what we have described in section 3.2.3 ($\psi_{d,m}$ is implemented by $A_{\text{check\_term}}$).

For step 2 (lines 12-22 in the protocol), it is convenient to have $A_{\text{list\_terms}}$ only list terms of one particular length. This allows $\Pi_{\text{g\_matrix\_power\_tuple}}$ to generate the required ciphertexts step by step: start with ciphertexts $\mathbf{a}_0, ..., \mathbf{a}_{d^2-1}$, then in step ℓ, generate all required ciphertexts of length ℓ by using homomorphic multiplication and the ciphertexts of length $\ell - 1$ created in the previous step. Furthermore, after every step, also save the shares of the decryptions of those ciphertexts. This requires keeping a map from the set of all term strings to the set of ciphertexts, as well as maps from the set of all term strings to $\mathbb{F}$ (that stores the shares). I.e., we require maps of the form

$$\mathbf{T} : \{0, ..., d^2 - 1\}^* \to \mathbf{C} \qquad \{\mathbf{S}_i : \{0, ..., d^2 - 1\}^* \to \mathbb{F}\}_{i \in [n]}$$

We always use the symbol $\mathbf{T}$ for the maps that store ciphertexts and $\mathbf{S}_i$ for the maps that store shares of the plaintexts for party P_i. Thus, to deal with a term string of length ℓ (say 1220), we first divide this string into its first letter an the remaining letters (i.e., 1 and 220) using the functions head and tail,[21] then use $\mathbf{T}$ to obtain the ciphertext $\mathbf{a}_2\mathbf{a}_2\mathbf{a}_0$ generated in the previous step and use homomorphic multiplication to obtain $\mathbf{a}_1 \otimes \mathbf{a}_2\mathbf{a}_2\mathbf{a}_0 = \mathbf{a}_1\mathbf{a}_2\mathbf{a}_2\mathbf{a}_0$. In the code (lines 7-9) we initialize these maps by writing $\mathbf{T} := \{t \to \mathbf{a}_t\}_{t \in \{0,...,d^2-1\}}$ and the like. Note that we require further maps $\mathbf{T}'$ and $\mathbf{S}'_i$ to generate terms used in the sacrificing step (analogous to A' and C' for matrix triples) as well as maps to store MACs for all values ($\overline{\mathbf{S}}_i$ and $\overline{\mathbf{S}}'_i$). Observe that the maps $\mathbf{T}, \mathbf{T}'$ could be thrown away once step 2 is completed.

[21] Formally, $\text{head}((x_0 \cdots x_n)) = (x_0)$ and $\text{tail}((x_0 \cdots x_n)) = (x_1 \cdots x_n)$.

For step 3 (the sacrificing step, lines 24-38 in the protocol), we simply verify the result of *every* multiplication (this is why we save shares for terms of length $< m$). After this is done, the maps $\mathbf{S}'_i$ and $\overline{\mathbf{S}}'_i$ could also be thrown away.

Step 4 corresponds to lines 40-45 in the protocol. For each relevant term string, we require a matrix with the shares of terms, another with the shares of their MACs, and we need to remember the matrix string. It will also be convenient to shorten 'DDAADAAADDAD' into '$A^2 1 A^3 2 A^1$', i.e., cut the prefix and postfix of D's, replace every remaining group of k D's with the number k and every group of k A's with A^k. Thus, our output will be a map of the form

$$\mathfrak{K}_i : \{\text{'A'}, \text{'D'}\}^m \to \mathrm{SKM}_d(\mathbb{F}) \times \mathrm{SKM}_d(\mathbb{F}) \times ([m] \cup \{A^i \mid i \in [m]\})^*$$

for each P_i. The key algorithm here is $A_{\text{create\_skeleton\_matrix}}$, which takes the parameter $d \in \mathbb{N}$, the map $\mathbf{S}_i$, and a term string in intermediate form, (i.e., where D's are combined but A's aren't, in our example 'AA1AAA2A'). It returns the shortened version of the matrix string and (a matrix with shares of) the skeleton matrix of the form described in section 3.2.3, in this case $A^2 \odot \mathbf{I}_d \odot A^3 \odot \mathbf{I}_d \odot A$. To obtain the matrix string in intermediate form, the algorithm $A_{\text{squash\_ds}}$ is called.

Note that $A_{\text{create\_skeleton\_matrix}}$ is implemented in two steps. In our example above, it first produces the object $A \odot A \odot \mathbf{I}_d \odot A \odot A \odot A \odot \mathbf{I}_d \odot A$, i.e., it ignores the fact consecutive A's can be combined. Then, it calls A_{squash}, which recursively combines them (one group of A's in each iteration) and also shortens the matrix string along the way. This turns out to be the most complex algorithm, but it is necessary since $A_{\text{create\_skeleton\_matrix}}$ works on the basis of a list of terms, so there is no easy way to treat each string of k A's as a single matrix A^k to begin with.

Finally, the consumption step (implemented by $\Pi_{\text{consume\_matrix\_power\_tuple}}$, figure 4.9) is straight-forward. Recall that $\langle S^m \rangle_i$ can be written as one large sum (see section 3.2.3) where each summand corresponds to a matrix string of length m and its corresponding skeleton matrix. Thus, each P_i begins by setting $\langle S^m \rangle_i$ to $0_{d \times d}$, then iterates through all matrix strings $\mathbf{x} \in \{\text{'A'}, \text{'D'}\}^m$, constructs the summand she needs by retrieving the skeleton matrix from $\mathfrak{K}_i(\mathbf{x})$ and calling A_{consume} on it, and adds that summand to $\langle S^m \rangle_i$. Her share of the MAC $\langle \overline{S^m} \rangle_i$ can be constructed analogously.

Note that $\mathfrak{K}_i(\mathbf{x})$ contains the skeleton matrices for the matrix string $A_{\text{squash\_ds}}(\mathbf{x})$, not for the matrix string $\mathbf{x}$. Thus, $\mathfrak{K}_i(\text{'DADDA'}) = \mathfrak{K}_i(\text{'ADDA'})$ (since $A_{\text{squash\_ds}}(\text{'DADDA'})$ and $A_{\text{squash\_ds}}(\text{'ADDA'})$ both equal 'A2A'). It follows that, even though $\mathfrak{K}_i$ only takes strings of length m as input, it still contains the skeleton matrices for matrix strings of size $< m$. In particular, it contains the skeleton matrix for matrix A (for example, under input string 'ADDDDD'). This is how the parties obtain shares of A in the protocol (line 1).

In the context of the algorithms in section 4.3, we write $\mathrm{SKM}_d(\mathbb{F})$ to denote the space of all $d \times d^j$ skeleton matrices (for all $j \in \mathbb{N}_+$). In the protocol and ideal functionality, we only have to deal with $d \times d$ matrices, but still consider them to be skeleton matrices (i.e., we assume they are indexed from 0 to $d^2 - 1$).

4.2.5 Matrix Power Tuple Consumption

$\mathcal{F}_{\text{consume\_matrix\_power\_tuple}}$

Input: $\{\langle\!\langle S \rangle\!\rangle_i, \mathfrak{K}_i\}_{i\in[n]}, d : \mathbb{N}, m : \mathbb{N}$

$\forall i \ : \ (x, \langle A\rangle_i, \langle\overline{A}\rangle_i) := \mathfrak{K}_i(\text{`AD}\cdots\text{D'})$ # $m-1$ D's

$A := \sum_{i=1}^{n} \langle A\rangle_i \quad D := \sum_{i=1}^{n} (\langle S\rangle_i - \langle A\rangle_i) \quad S := \sum_{i=1}^{n} \langle S\rangle_i$

$\forall i \in I_{\text{adv}} \ : \ \langle S^m\rangle_i := 0_{d\times d} \qquad \langle\overline{S^m}\rangle_i := 0_{d\times d}$

`for` $x \in \{\text{`A', `D'}\}^m$:

$\forall i \in I_{\text{adv}} \ : \ (K_i, \overline{K}_i, x') := \mathfrak{K}_i(x)$

$\forall i \in I_{\text{adv}} \ : \ \langle S^m\rangle_i := \langle S^m\rangle_i + A_{\text{consume}}(d, m, K_i, x', D^1, ..., D^m)$

$\forall i \in I_{\text{adv}} \ : \ \langle\overline{S^m}\rangle_i := \langle\overline{S^m}\rangle_i + A_{\text{consume}}(d, m, \overline{K}_i, x', D^1, ..., D^m)$

`endfor`

$\forall i \in I_{\text{hst}} \ : \ \langle S^m\rangle_i \xleftarrow{u/c} \mathbb{F}^{d^2} \qquad \langle\overline{S^m}\rangle_i \xleftarrow{u/c} \mathbb{F}^{d^2}$

Output: $\{\langle\!\langle S^m \rangle\!\rangle_i\}_{i\in[n]}$

Figure 4.8: The ideal functionality for consuming a matrix power tuple

$\Pi_{\text{consume\_matrix\_power\_tuple}}$

Input: $\{\langle\!\langle S \rangle\!\rangle_i, \mathfrak{K}_i\}_{i\in[n]}, d : \mathbb{N}, m : \mathbb{N}$

$\forall P_i \ : \ (x, \langle A\rangle_i, \langle\overline{A}\rangle_i) := \mathfrak{K}_i(\text{`AD}\cdots\text{D'})$ # $m-1$ D's

$\mathcal{O}(S - A)$

$\forall P_i \ : \ \langle S^m\rangle_i := 0_{d\times d} \quad \langle\overline{S^m}\rangle_i := 0_{d\times d} \quad D := S - A$

`for` $x \in \{\text{`A', `D'}\}^m$:

$\forall P_i \ : \ (K_i, \overline{K}_i, x') := \mathfrak{K}_i(x)$

$\forall P_i \ : \ \langle S^m\rangle_i := \langle S^m\rangle_i + A_{\text{consume}}(d, m, K_i, x', D^1, ..., D^m)$

$\forall P_i \ : \ \langle\overline{S^m}\rangle_i := \langle\overline{S^m}\rangle_i + A_{\text{consume}}(d, m, \overline{K}_i, x', D^1, ..., D^m)$

`endfor`

Output: $\{\langle\!\langle S^m \rangle\!\rangle_i\}_{i\in[n]}$

Figure 4.9: The protocol for consuming a matrix power tuple

4.2.6 Matrix Power Tuple Generation

$\mathcal{F}_{\text{g\_matrix\_power\_tuple}}$

Input: $d : \mathbb{N}, m : \mathbb{N}$

$\phi_{\text{zkcheat}} \leftarrow \mathcal{S}$ # ↓ modeling failed zero knowledge proofs

`if` $\phi_{\text{zkcheat}} = \top$:

  [**abort**]

`endif`

$\forall i \in I_{\text{adv}} : (\langle A \rangle_i, \langle A' \rangle_i) \leftarrow \mathcal{S}$

$\forall i \in I_{\text{hst}} : (\langle A \rangle_i, \langle A' \rangle_i) \xleftarrow{u} \mathbb{F}^{2d^2}$

$A := \sum_{i=1}^{n} \langle A \rangle_i \qquad A' := \sum_{i=1}^{n} \langle A' \rangle_i$

$\{\langle \overline{a_t} \rangle_i\}_{i \in [n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \mathfrak{E}(\gamma a_t, \mathfrak{p})) \quad \forall t \in \{0, ..., d^2 - 1\}$

$\forall i : \mathbf{S}_i := \{t \rightarrow \langle a_t \rangle_i\}_{t \in \{0,...,d^2-1\}} \qquad \overline{\mathbf{S}}_i := \{t \rightarrow \langle \overline{a_t} \rangle_i\}_{t \in \{0,...,d^2-1\}}$

`for` $\ell \leftarrow 2$ `to` m : # ↓ generate required terms

  $S_\ell := A_{\text{list\_terms}}(d, m, \ell)$

  `for` $s \in S_\ell$: # head(2301) = 2, tail(2301) = 301

    $\forall i : a_i := \mathbf{S}_i(\text{head}(s)) \qquad b_i := \mathbf{S}_i(\text{tail}(s))$

    $a := \sum_{i=1}^{n} a_i \qquad b := \sum_{i=1}^{n} b_i$

    $\{\langle c \rangle_i\}_{i \in [n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \mathfrak{E}(ab, \mathfrak{p}))$ # c might include a decr. error

    $c := \sum_{i=1}^{n} \langle c \rangle_i$

    $\{\langle \overline{c} \rangle_i\}_{i \in [n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \mathfrak{E}(\gamma c, \mathfrak{p}))$

    $\forall i : \mathbf{S}_i(s) := \langle c \rangle_i \qquad \overline{\mathbf{S}}_i(s) := \langle \overline{c} \rangle_i$

  `endfor`

`endfor`

Figure 4.10: The ideal functionality for generating a matrix power tuple (part 1/2)

$\mathcal{F}_{\text{g\_matrix\_power\_tuple}}$ (cont.)

21 `for` $\ell \leftarrow 2$ `to` m `:` # ↓ sacrificing step

22 `for` $s \in S_\ell$ `:` # same S_ℓ's as before

23 $a := \sum_{i=1}^{n} \mathbf{S}_i(\text{head}(s))$ $b := \sum_{i=1}^{n} \mathbf{S}_i(\text{tail}(s))$ $c := \sum_{i=1}^{n} \mathbf{S}_i(s)$

24 $\phi_{\text{failcheck}} \leftarrow \mathcal{S}$

25 `if` $\phi_{\text{failcheck}} = \top \ \vee \ c \neq ab$ `:`

26 $y \leftarrow \mathcal{S}$

27 $\left(\sum_{i \in I_{\text{hst}}} y \cdot \mathbf{S}_i(\text{tail}(s))\right) \rightarrow \mathcal{S}$

28 [**abort**]

29 `endif`

30 `endfor`

31 `endfor` # ↓ put terms in skeleton matrices

32 $\forall i : \mathfrak{K}_i := \{x \mapsto \bot\}_{x \in \{\text{'A'},\text{'D'}\}^m}$

33 `for` $x \in \{\text{'A'}, \text{'D'}\}^m$

34 $\forall i : (x', K_i) := A_{\text{create\_skeleton\_matrix}}(d, A_{\text{squash\_ds}}(x), \mathbf{S}_i)$

35 $\forall i : (x'', \overline{K_i}) := A_{\text{create\_skeleton\_matrix}}(d, A_{\text{squash\_ds}}(x), \overline{\mathbf{S}}_i)$

36 $\forall i : \mathfrak{K}_i(x) := (K_i, \overline{K_i}, x')$

37 `endfor`

38 **Output:** $\{\mathfrak{K}_i\}_{i \in [n]}$

Figure 4.11: The ideal functionality for generating a matrix power tuple (part 2/2)

$\Pi_{\text{g\_matrix\_power\_tuple}}$

0 **Input:** $d : \mathbb{N}, m : \mathbb{N}$

1 $\forall P_i : \langle A\rangle_i, \langle A'\rangle_i \xleftarrow{u} \mathbb{F}^{d^2}$ # $A, A' \in \mathrm{SKM}_d(\mathbb{F})$, i.e., indexed from 0 to d^2-1

2 $\forall P_i : \langle \mathbf{A}\rangle_i \leftarrow \mathfrak{E}(\langle A\rangle_i, \mathfrak{p})$ $\langle \mathbf{A}'\rangle_i \leftarrow \mathfrak{E}(\langle A'\rangle_i, \mathfrak{p})$ # component-wise encryption

3 $\forall P_i : \mathcal{Z}(\langle \mathbf{a}_t\rangle_i)$ $\mathcal{Z}(\langle \mathbf{a}'_t\rangle_i)$ $\forall t \in \{0, ..., d^2-1\}$

4 $\forall P_i : \mathbf{A} := \sum_{i=1}^{n} \langle \mathbf{A}\rangle_i$ $\mathbf{A}' := \sum_{i=1}^{n} \langle \mathbf{A}'\rangle_i$

5 $\forall P_i : \overline{\mathbf{a}_t} := \boldsymbol{\gamma} \otimes \mathbf{a}_t$ $\forall t \in \{0, ..., d^2-1\}$

6 $\{\langle \overline{a_t}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \overline{\mathbf{a}_t})$ $\forall t \in \{0, ..., d^2-1\}$

7 $\forall P_i : \mathbf{T} := \{t \to \mathbf{a}_t\}_{t\in\{0,...,d^2-1\}}$ $\mathbf{T}' := \{t \to \mathbf{a}'_t\}_{t\in\{0,...,d^2-1\}}$

8 $\forall P_i : \mathbf{S}_i := \{t \to \langle a_t\rangle_i\}_{t\in\{0,...,d^2-1\}}$ $\overline{\mathbf{S}}_i := \{t \to \langle \overline{a_t}\rangle_i\}_{t\in\{0,...,d^2-1\}}$

9 $\forall P_i : \mathbf{S}'_i := \{t \to \perp\}_{t\in\{0,...,d^2-1\}}$ $\overline{\mathbf{S}}'_i := \{t \to \perp\}_{t\in\{0,...,d^2-1\}}$

10 `for` $\ell \leftarrow 2$ `to` m : # ↓ generate required terms

11 $\forall P_i : S_\ell := A_{\text{list\_terms}}(d, m, \ell)$

12 `for` $s \in S_\ell$: # head(2301) = 2, tail(2301) = 301

13 $\forall P_i : \mathbf{a} := \mathbf{T}(\text{head}(s))$ $\mathbf{a}' := \mathbf{T}'(\text{head}(s))$ $\mathbf{b} := \mathbf{T}(\text{tail}(s))$

14 $\forall P_i : \mathbf{c} := \mathbf{a} \otimes \mathbf{b}$ $\mathbf{c}' := \mathbf{a}' \otimes \mathbf{b}$

15 $\{\langle c\rangle_i, \mathbf{c}\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\top, \mathbf{c})$

16 $\{\langle c'\rangle_i, \mathbf{c}'\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\top, \mathbf{c}')$

17 $\forall P_i : \overline{\mathbf{c}} := \mathbf{c} \otimes \boldsymbol{\gamma}$ $\overline{\mathbf{c}'} := \mathbf{c}' \otimes \boldsymbol{\gamma}$ # ↓ MACs

18 $\{\langle \overline{c}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \overline{\mathbf{c}})$

19 $\{\langle \overline{c'}\rangle_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\text{ddec\_shares}}(\perp, \overline{\mathbf{c}'})$

20 $\forall P_i : \mathbf{T}(s) := \mathbf{c}$ $\mathbf{S}_i(s) := \langle c\rangle_i$ $\overline{\mathbf{S}}_i(s) := \langle \overline{c}\rangle_i$ # ↓ store data

21 $\forall P_i : \mathbf{S}'_i(s) := \langle c'\rangle_i$ $\overline{\mathbf{S}}'_i(s) := \langle \overline{c'}\rangle_i$

22 `endfor`

23 `endfor`

Figure 4.12: The protocol for generating a matrix power tuple (part 1/2)

$\Pi_{\text{g\_matrix\_power\_tuple}}$ (cont.)

$\forall P_i : \langle r \rangle_i \xleftarrow{u} \mathbb{F}$ # ↓ sacrificing step

$\forall P_i : O_{\text{nomac}}(r)$

$\forall P_i : O_{\text{nomac}}(rA - A')$

`for` $\ell \leftarrow 2$ `to` m :

  `for` $s \in S_\ell$: # same S_ℓ's as computed earlier

    $\forall P_i : \langle b \rangle_i := \mathbf{S}(\text{tail}(s)) \quad \langle \overline{b} \rangle_i := \overline{\mathbf{S}}(\text{tail}(s))$

    $\forall P_i : \langle c \rangle_i := \mathbf{S}_i(s) \quad \langle \overline{c} \rangle_i := \overline{\mathbf{S}}_i(s) \quad \langle c' \rangle_i := \mathbf{S}'_i(s) \quad \langle \overline{c'} \rangle_i := \overline{\mathbf{S}}'(s)$

    $\forall P_i : v_{ra-a'} := (rA - A')_{\text{head}(s)}$

    $\forall P_i : \langle z \rangle_i := r\langle c \rangle_i - \langle c' \rangle_i - v_{ra-a'}\langle b \rangle_i$

    $\forall P_i : \langle \overline{z} \rangle_i := r\langle \overline{c} \rangle_i - \langle \overline{c'} \rangle_i - v_{ra-a'}\langle \overline{b} \rangle_i$

    $O(z)$

    `if` $z \neq 0$:

      [**abort**]

    `endif`

  `endfor`

`endfor`

$\forall P_i : \mathfrak{K}_i := \{x \mapsto \perp\}_{x \in \{\text{'A'},\text{'D'}\}^m}$ # ↓ put terms in skeleton matrices

`for` $x \in \{\text{'A'}, \text{'D'}\}^m$:

  $\forall P_i : (K_i, x') := A_{\text{create\_skeleton\_matrix}}(d, A_{\text{squash\_ds}}(x), \mathbf{S}_i)$

  $\forall P_i : (\overline{K}_i, x'') := A_{\text{create\_skeleton\_matrix}}(d, A_{\text{squash\_ds}}(x), \overline{\mathbf{S}}_i)$

  $\forall P_i : \mathfrak{K}_i(x) := (K_i, \overline{K}_i, x')$ # note that $x' = x''$

`endfor`

Output: $\{\mathfrak{K}_i\}_{i \in [n]}$

Figure 4.13: The protocol for generating a matrix power tuple (part 2/2)

4.3 Non-distributed Algorithms

In this section, we give algorithms for various technical problems relating to skeleton matrices. Unlike the protocols given in the previous section, these algorithms are deterministic and non-distributed and therefore trivially secure.

Algorithm 1: checks whether a given term is required for an m-matrix power tuple

$A_{\text{check\_term}}(d, m \in \mathbb{N}, \mathbf{s} \in \{0, ..., d^2 - 1\}^*) : \{\top, \bot\}$
  $c := 0$;
  for $i \in \{0, ..., |\mathbf{s}| - 2\}$ **do**
    if $\mathbf{s}_{i+1} \bmod d \neq \mathbf{s}_i \operatorname{div} d$ **then**
      $c := c + 1$;
    end
  end
return $|\mathbf{s}| + c \leq m$;

Notes on Algorithm 1:

- We assume that a boolean expression automatically resolves to $\top$ if it is true or $\bot$ if it is false

Algorithm 2: lists all required terms of length ℓ for an m-matrix power tuple

$A_{\text{list\_terms}}(d, m, \ell \in \mathbb{N}) : \mathcal{P}(\{0, ..., d^2 - 1\}^*)$
  $L := \emptyset$;
  for $\mathbf{s} \in \{0, ..., d - 1\}^\ell$ **do**
    $\mathbf{s}' := \text{sort}(\mathbf{s})$;
    if $\mathbf{s}' \notin L$ **then**
      if $A_{check\_term}(d, m, \mathbf{s})$ **then**
        $L := L \cup \{\mathbf{s}'\}$;
      end
    end
  end
return L;

Notes on Algorithm 2:

- The output only depends on the integers $d, m, \ell \in \mathbb{N}$
- The symbol $\mathcal{P}$ denotes the powerset operator
- We assume access to a function sort which simply takes a list of integers and outputs the list sorted in ascending order

Algorithm 3: removes the prefix and postfix of 'D's and replaces every remaining group of n consecutive 'D's by the number n

$A_{\text{squash\_ds}}(\mathbf{x} \in \{A, D\}^*) : (\{A\} \cup \mathbb{N})^*$
  $\mathbf{x}' := ();$
  $\ell := 0;$
  while $\mathbf{x}_\ell$ = *'D'* **do**
    $\ell := \ell + 1;$
  end
  $r := |\mathbf{x}| - 1;$
  while $\mathbf{x}_r$ = *'D'* **do**
    $r := r - 1;$
  end
  for $j \leftarrow \ell$ **to** r **do**
    if $\mathbf{x}_j$ = *'A'* **then**
      $\mathbf{x}' := \mathbf{x}' \circ$ 'A';
    else
      $n := 1;$
      while $\mathbf{x}_{j+1}$ = *'D'* **do**
        $n := n + 1;$
        $j := j + 1;$
      end
      $\mathbf{x}' := \mathbf{x}' \circ n;$
    end
  end
return $\mathbf{x}'$;

Algorithm 4: creates a skeleton matrix based on its string and the set of terms

$A_{\text{create\_skeleton\_matrix}}(d \in \mathbb{N}, \mathbf{x} \in (\{A\} \cup \mathbb{N})^*, \mathbf{S} \in \{\{0, ..., d-1\}^* \to \mathbb{F}\}) :$
  $\text{SKM}_d(\mathbb{F}) \times (\{A^i \mid i \in \mathbb{N}\} \cup \mathbb{N})^*$
  $m := |\mathbf{x}|;$
  $K := \left(0\right)_{0 \le i < d^{m+1}};$
  for $i \leftarrow 0$ **to** $d^{m+1} - 1$ **do**
    $\mathbf{s} := ();$
    for $j \leftarrow m - 1$ **to** 0 **do**
      if $\mathbf{x}_j$ = *'A'* **then**
        $\mathbf{s} := \mathbf{s} \circ ((i \text{ div } d^j) \bmod d^2);$
      end
    end
    $K_i := \mathbf{S}(\mathbf{s});$
  end
return $A_{\text{squash}}(d, |\mathbf{x}|, K, \mathbf{x});$

Notes on Algorithm 3:

- We write () to denote the empty string

Notes on Algorithm 4:

- In the code, C is a list of integers, and we write $\circ$ to add another element to the end of the list. Each C corresponds to a single term in the skeleton matrix.
- To aid understanding of what the code does, we repost figure 3.2 that illustrates how terms are partitioned in skeleton matrices

$$\begin{bmatrix} a_0d_0\boldsymbol{a_0} & a_0d_1\boldsymbol{a_2} & a_1d_2\boldsymbol{a_0} & a_1d_3\boldsymbol{a_2} & a_2d_0\boldsymbol{a_0} & a_2d_1\boldsymbol{a_2} & a_3d_2\boldsymbol{a_0} & a_3d_3\boldsymbol{a_2} \\ a_0d_0\boldsymbol{a_1} & a_0d_1\boldsymbol{a_3} & a_1d_2\boldsymbol{a_1} & a_1d_3\boldsymbol{a_3} & a_2d_0\boldsymbol{a_1} & a_2d_1\boldsymbol{a_3} & a_3d_2\boldsymbol{a_1} & a_3d_3\boldsymbol{a_3} \end{bmatrix}$$

$$\begin{bmatrix} a_0\boldsymbol{d_0}a_0 & a_0\boldsymbol{d_1}a_2 & a_1\boldsymbol{d_2}a_0 & a_1\boldsymbol{d_3}a_2 & a_2\boldsymbol{d_0}a_0 & a_2\boldsymbol{d_1}a_2 & a_3\boldsymbol{d_2}a_0 & a_3\boldsymbol{d_3}a_2 \\ a_0\boldsymbol{d_0}a_1 & a_0\boldsymbol{d_1}a_3 & a_1\boldsymbol{d_2}a_1 & a_1\boldsymbol{d_3}a_3 & a_2\boldsymbol{d_0}a_1 & a_2\boldsymbol{d_1}a_3 & a_3\boldsymbol{d_2}a_1 & a_3\boldsymbol{d_3}a_3 \end{bmatrix}$$

$$\begin{bmatrix} \boldsymbol{a_0}d_0a_0 & \boldsymbol{a_0}d_1a_2 & \boldsymbol{a_1}d_2a_0 & \boldsymbol{a_1}d_3a_2 & \boldsymbol{a_2}d_0a_0 & \boldsymbol{a_2}d_1a_2 & \boldsymbol{a_3}d_2a_0 & \boldsymbol{a_3}d_3a_2 \\ \boldsymbol{a_0}d_0a_1 & \boldsymbol{a_0}d_1a_3 & \boldsymbol{a_1}d_2a_1 & \boldsymbol{a_1}d_3a_3 & \boldsymbol{a_2}d_0a_1 & \boldsymbol{a_2}d_1a_3 & \boldsymbol{a_3}d_2a_1 & \boldsymbol{a_3}d_3a_3 \end{bmatrix}$$

Figure 3.2: An illustration of the patterns that emerge in skeleton matrices (repeated from page 24)

Algorithm 5: "consumes" a skeleton matrix and returns the resulting regular matrix

```
A_consume(d, m ∈ ℕ, K ∈ SKM_d(𝔽), x ∈ ({A^i | i ∈ [m]} ∪ [m])*, D_1, ..., D_m : SKM_d(𝔽)) :
    SKM_d(𝔽)
  ℓ := |x|;
  if ℓ = 1 then
  |  return K;
  end
  D := D_{x_{ℓ−2}};      # if x = A1A^2 2A^4 then D = D_2. Note that now D = (d_t)_{0≤t<d^2}.
  for j ← 0 to d^{ℓ+1} − 1 do
  |  r := j mod d;      # row in new skeleton matrix
  |  c := j div d^3;      # column in new skeleton matrix
  |  j_new := c · d + r;      # index in new skeleton matrix
  |  t := (j div d) mod d^2;      # index of elem in D
  |  K_{j_new} := K_{j_new} + K_j · d_t;
  end
  K := (K)_{0≤j<d^{|x|−1}};      # divide length by d^2
  x := (x)_{0≤j<ℓ−2};      # drop last two letters
 return A_consume(d, m, K, x, D_1, ..., D_k);
```

Notes on Algorithm 6:

- We assume $A_{\text{squash\_first\_group\_of\_As}}$ (which replaces the first string of m A's with the symbol A^m) is trivial and provide no implementation.

Algorithm 6: "squashes" a skeleton matrix by combining all terms that belong to the same coordinate into a single sum. Also squashes the corresponding string by replacing every group of n consecutive As with an A^n

$A_{\text{squash}}(d, m \in \mathbb{N}, K \in \text{SKM}_d(\mathbb{F}), \mathbf{x} \in (\{A\} \cup [m])^*) : \text{SKM}_d(\mathbb{F}) \times (\{A^i \mid i \in [m]\} \cup [m])^*$

```
    p := 0;
    w := 0;
    for j ← 1 to |x| − 1 do
        if x_j = 'A' ∧ x_{j−1} = 'A' then
            w := 2;
            i := j + 1;
            while x_i = 'A' ∧ i < |x| do
                w := w + 1;
                i := i + 1;
            end
            break;
        else
            p := p + 1;
        end
    end
    if w = 0 then
        return (K, x);      # in this case, there is nothing (left) to squash
    end
    s_sum := d^{p+1};       # n ≈ number of, s ≈ stepsize within
    n_sum := d^{w−1};
    s_group := s_sum · n_sum;
    n_group := |K| / s_group;
    i_new := 0;
    for i_group ← 0 to n_group − 1 do
        for i_sum ← 0 to s_sum − 1 do
            i_total := i_group · s_group + i_sum;
            K'_{i_new} := K_{i_total};
            for t ← 1 to n_sum do
                i_total := i_total + s_sum;
                K'_{i_new} := K'_{i_new} + K_{i_total};
            end
            i_new := i_new + 1;
        end
    end
return A_squash(d, m, K', A_squash_first_group_of_As(x));
```

5 Security Proof

In this chapter, we prove that the Multi-Party Computation scheme proposed in chapter 4 is secure. Section 5.1 gives an overview of the universal composition model and describes the concrete instantiation and formalism we aim to follow, whereas section 5.2 contains the security proofs themselves. It is structured in one subsection for each protocol that includes the code of the simulator and the textual proof.

Note that the notation introduced in section 4.1.4 remains relevant for this chapter.

5.1 The Security Model

We wish to prove our scheme secure in the Universal Composition (UC) model. While there have been many variations proposed in the literature, most of them build upon the same fundamental idea illustrated in figure 5.1, which we elaborate on in the following.

5.1.1 Fundamentals of UC security

The UC model assumes there are two different worlds, which we call the "real" and "ideal" world, respectively.

- In the real world, the real protocols (the $\Pi_{[\text{name}]}$) execute. Corruption of parties is modeled by a single entity, called the *adversary*, that makes all decisions for the corrupt parties and receives all information they receive.
- In the ideal world, the corresponding *ideal functionalities* (the $\mathcal{F}_{[\text{name}]}$) execute. Corruption is modeled by explicit communication of the ideal functionality with an additional entity we call the *simulator*.

Note that there will be no proof that any the ideal functionalities are secure. Instead, they should be written in such a way that they are "obviously" secure, and the security proof consists of demonstrating that a real protocol is (up to a negligible factor) at least as secure as the corresponding ideal functionality.

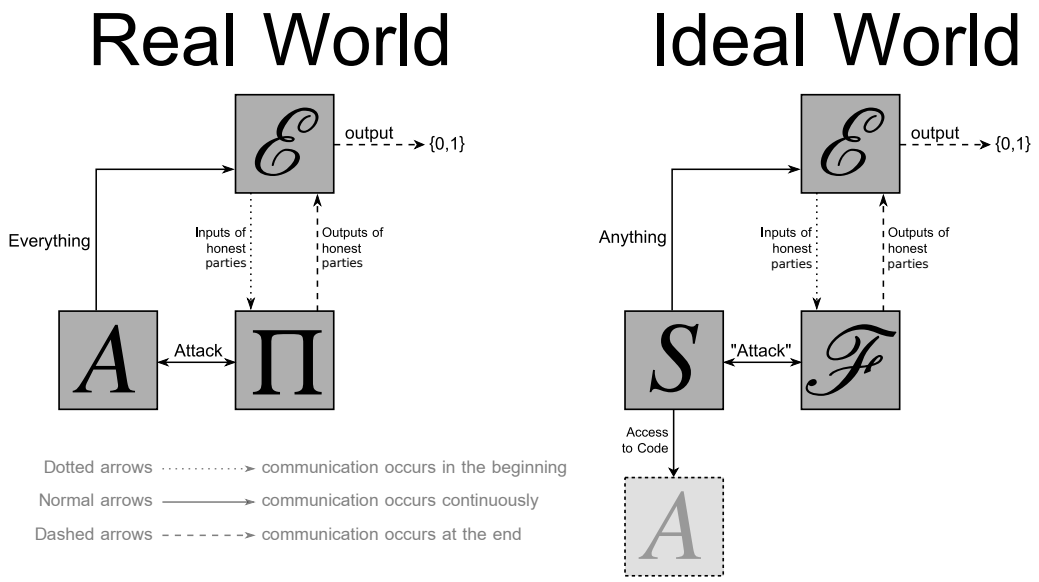

Figure 5.1: A graphical illustration of the UC security model

To model this requirement, the model utilizes a fictional entity called the *environment*, denoted $\mathcal{E}$, that exists in both worlds. $\mathcal{E}$ observes the run of the protocol without knowing which world it is in. Then, after computation has concluded, it outputs a single bit. The security proof consists of demonstrating that, regardless of how $\mathcal{E}$ chooses this bit, the probability that it outputs 1 differs by at most a negligible factor across both worlds. If this is the case, then $\mathcal{E}$ cannot distinguish between both worlds, which means that the ideal functionality cannot be meaningfully more secure than the protocol.[22]

Clearly, this approach is only meaningful if $\mathcal{E}$ receives all information that could be a factor in compromising security. In particular, in the real world, $\mathcal{E}$ receives

- All information that dishonest parties receive at any point during the execution of the protocol
- The outputs for all parties that the protocol specifies

Whereas, in the ideal world, it receives

- Whatever the simulator sends to it
- The outputs for all parties that the ideal functionality specifies

Given a pair $(\Pi, \mathcal{F})$ of a protocol and an ideal functionality, the formal definition of security demands that, for each adversary, there exists a simulator such that every environment will output 1 with similar probability across both worlds. Thus, the simulator

[22] In the literature, there is a common variant on this approach where the environment in the real world subsumes the role of the adversary. However, we shall not consider this variant in this document.

- is allowed to depend on the code of the adversary. In particular, it has access to the inputs and decisions of dishonest parties.
- can perform arbitrary computations to create objects it sends to the environment, although its goal will be to make them look exactly like the objects that dishonest parties send to the environment in the real world.
- does not have access to the inputs of honest parties.
- communicates with the ideal functionality in the ways that are explicitly stated in the ideal functionality's code.

At the heart of any UC model are the universal composition theorems that give the model its name. Such theorems argue that, if a protocol Π behaves like another protocol $\mathcal{F}$ in the way described above (from now on, we denote this claim by writing $\Pi \leq \mathcal{F}$), then this remains true regardless of how the execution environment changes. In other words, if Π^* is a protocol calling Π as a subroutine, and $\Pi^*_{\Pi \to \mathcal{F}}$ is the same protocol except that Π is substituted with $\mathcal{F}$, then $\Pi \leq \mathcal{F} \implies \Pi^* \leq \Pi^*_{\Pi \to \mathcal{F}}$.

5.1.2 Specifying a concrete approach

The UC model has first been introduced in [2]. Since then, several variants have been proposed, such as GNUC [10], IITM [13], [12] (revised version), and others [4, 7]. There is also the SUC model [3] which has been specifically designed to be simple and usable for schemes implementing Multi-Party Computation, which makes it an attractive choice for our purposes. Nonetheless, for reasons involving details of the formalism, we will adhere to the iUC model [5] ("IITM based Universal Composability") that builds on [13].

In the following, we explain what assumptions we make to further simplify this model and how they map onto the model formally. We do not provide an introduction to the iUC variant but instead refer to the paper itself.

For every protocol $\Pi_{[\text{name}]}$ defined in chapter 4, the corresponding formal *protocol system* (terminology from [5] p7) consists of $n+1$ machines $M_1, ..., M_n, S$. Machine M_i corresponds to party P_i. All of our algorithms are perfectly symmetric with regard to all parties, so we create n identical public roles $R^{(1)}_{[\text{name}]}, ..., R^{(n)}_{[\text{name}]}$, each one defined by the code of $\Pi_{[\text{name}]}$, and declare that each M_i implements the role $R^{(i)}_{[\text{name}]}$. The protocol system always implements the corruption model *static corruption*, and the algorithm **AllowCorruption** for each machine M_i is defined to be the trivial algorithm that always outputs true – except for M_n, where, instead, it is the trivial algorithm that always returns false. This is not a restriction since all n parties have identical roles.

Our biggest simplification is with regard to communication between machines. Formally, we assume each M_i has $n-1$ tapes (i,j) for all $j \in [n] \setminus \{i\}$ to communicate with each other machine.[23] Practically, however, we simply talk about parties broadcasting values without going into detail. We do not have to worry about the adversary not sending values since each party always knows exactly which values to expect at any point, so she can simply report an error and abort the scheme if a value doesn't arrive (and causing the scheme to abort is trivially possible for any party regardless).

With regard to concurrent execution, we make similar simplifying assumptions. Namely, we simply restrict ourselves to analyzing a single session only. The implementation details for this are given in [5] (p42). We do not go into any more detail.

We assume runtime bounds are trivial to establish for all proposed protocols.

Machine S will primarily handle key generation. Formally, we define its algorithm **Initialization** to do the following.

1. Generate a key pair $(\mathfrak{p}, \mathfrak{s})$ and a description $D_{\mathbb{F}}$ of a finite field $\mathbb{F}$ based on the security parameter
2. Sample $\langle \mathfrak{s} \rangle_i, \langle \mathfrak{s}^2 \rangle_i \xleftarrow{u/c} \mathbb{F}$ and $\langle \gamma \rangle_i \xleftarrow{u} \mathbb{F}$ for all $i \in [n]$, compute $\boldsymbol{\gamma} \leftarrow \mathfrak{E}(\gamma, \mathfrak{p})$ and send $(D_{\mathbb{F}}, \mathfrak{p}, \langle \mathfrak{s} \rangle_i, \langle \mathfrak{s}^2 \rangle_i, \langle \gamma \rangle_i, \boldsymbol{\gamma})$ to each M_i.[24]

This is another major simplification – the cryptographic keys are not freshly generated each time a new matrix triple is generated, and they certainly do not use different MAC keys. However, this modeling most accurately reflects our assumption that security of the entire scheme follows from security in the single-session setting that analyzes a single protocol in isolation. Note that this assumption is not uncommon in the literature.

5.1.3 Further Simplifications and Notation for this chapter

We model the simulator's access to the code of the adversary by having it use the adversary as an oracle that it can query for the objects which the dishonest parties broadcast during the execution of the real protocol. Specifically, we add commands of the form $v \leftarrow \mathcal{A}$ into the code of the simulator, which indicate that the variable v is set to the value of whatever object the dishonest parties would broadcast next. Thus, the instances of such commands need to correspond precisely to the instances of dishonest parties broadcasting values in the real protocol, even if the simulator does not use these objects.

Communication between the simulator and the ideal functionality is modeled as follows:

[23] Strictly speaking, the tape would have to be (j,i) rather than (i,j) whenever $j < i$ to ensure that Machines M_i and M_j really share a tape with the same identifier.

[24] Formally, each M_i needs an additional tape $(0,i)$, and machine S needs the n tapes $\{(0,i)\}_{i \in [n]}$.

- In the code of the ideal functionality, there are commands of the form $v \leftarrow \mathcal{S}$ and $t \rightarrow \mathcal{S}$ which indicate that the ideal functionality receives an object from the simulator and stores it in variable v/sends some value t to the simulator
- In the code of the simulator, there are commands of the form **run** $\mathcal{F}^{\rightarrow n}$, which indicate that the ideal functionality is executed up to and including line n, starting from wherever it has left off previously. If this part of the code includes a command $v \leftarrow \mathcal{S}$, we include the value for v into square brackets, i.e., we write **run** $\mathcal{F}^{\rightarrow n}[t]$ to indicate that v is set to t. Similarly, if the code contains a command $t \rightarrow \mathcal{S}$, we write a variable to the left and with a $\leftarrow$ symbol, i.e., we write $v \leftarrow$ **run** $\mathcal{F}^{\rightarrow n}$ to indicate that v is set to t.
- We write **run** $\mathcal{F}^{\rightarrow \infty}$ to indicate that the ideal functionality executes to the end. The final line in the code of the simulator will generally have this form.

To denote the simulator sending objects to the environment, we introduce the keyword "show." The objects that appear behind **show:** in the code of the simulator need to correspond precisely to the view $\mathcal{E}$ receives in the real world (up to the protocol's output).

Note that there is a subtle problem with our modeling of the simulator using $\mathcal{A}$ as an oracle. Specifically, it ignores the fact that the choices $\mathcal{A}$ makes in the real world depend on the information it has received up to this point. However, we can solve this problem by prescribing that the objects send via **show:** are sent to both $\mathcal{E}$ and $\mathcal{A}$. This suffices since $\mathcal{A}$ and $\mathcal{E}$ receive identical objects in the real world throughout execution of a protocol (the only difference is that $\mathcal{E}$ also receives the outputs of honest parties, but this happens after execution has finished).

Subroutine Calls

Suppose we are analyzing a protocol Π with corresponding ideal functionality $\mathcal{F}$, and at some point in the code of Π, it contains a subroutine call to an ideal functionality[25] $\mathcal{F}'$. In such a case, if ideal functionality $\mathcal{F}'$ contains communication with the adversary (in the form of $t \rightarrow \mathcal{S}$ or $v \leftarrow \mathcal{S}$), this communication happens during execution of the protocol in the real world. Note that, despite using the symbol $\mathcal{S}$, this adversary is the same as $\mathcal{A}$.[26] This means the simulator needs to **show:** the objects that are sent via $t \rightarrow \mathcal{S}$ (to both $\mathcal{E}$ and $\mathcal{A}$) and that it can query $\mathcal{A}$ for the objects received via $v \leftarrow \mathcal{S}$.

[25] In reality, a protocol would call another protocol rather than an ideal functionality. However, the universal composition theorems allow us to pretend as if it called an ideal functionality instead. Since this is always preferable (the entire point of an ideal functionality is that it has more desirable output channels), there will never be an instance where we assume a protocol calls another protocol.

[26] The symbol $\mathcal{S}$ is used in such cases because we generally think of ideal functionalities as running in the ideal world, but this is not true if they are called as subroutines.

If a corresponding call to $\mathcal{F}'$ occurs in the code of $\mathcal{F}$, then identical code is run in both worlds, which makes it trivial for the simulator to show the right objects to $\mathcal{E}$, namely by transferring messages unchanged: for every instance of $t \rightarrow \mathcal{S}$ in the ideal world, it receives this value and relays it to $\mathcal{E}$ and $\mathcal{A}$ to cover for the same call in the real world. However, in the cases where there is no corresponding call to $\mathcal{F}'$ in the code of $\mathcal{F}$, we need to demonstrate how the simulator can compute indistinguishable objects to show to $\mathcal{E}$.

In practice, our proofs consists of providing the code of a simulator, listing all instances where objects are sent to the environment (and to $\mathcal{A}$) in the real world, and demonstrating that the simulator sends equally distributed objects in all cases. In the case of subroutine calls, we will break away from our formal description of the code of $\mathcal{S}$ and instead resort to textual descriptions (which is the more common approach in the literature).

On opening values

Recall the assumptions we made regarding the opening of values described in chapter 4.1.1. With regard to our security proofs, we assume the following:

- When a value is opened with $O(x)$, the output will always be correct, i.e., $x = \sum_{i=1}^{n} \langle x \rangle_i$. The parties do not make any decisions.
- When a value is opened with $O_{\text{nomac}}(x)$, the dishonest parties get to choose the values they submit freely. However, since a commitment scheme is still used, they have to choose their shares before seeing those of others.

Note that this is a simplification – technically, dishonest parties are permitted to submit incorrect shares but would then be caught by the MAC-check at the end of the scheme.

5.1.4 Real vs. Meaningless ciphertexts

In section 4.1.2, we've mentioned that we assume our cryptosystem has the properties specified in [9] p7-10. We now highlight one of these properties:

We assume there are two algorithms A_{KeyGen} and $A_{\text{MeaninglessPublicKey}}$ where A_{KeyGen} is the regular key-generation algorithm outputting the pair $(\mathfrak{p}, \mathfrak{s})$ and $A_{\text{MeaninglessPublicKey}}$ puts out a key $\mathfrak{p}_0$ such that $\mathfrak{E}(x, \mathfrak{p}_0) = \mathfrak{E}(0, \mathfrak{p})$ for all plaintexts x, and $\mathfrak{p}$ and $\mathfrak{p}_0$ are computationally indistinguishable. More precisely, we assume that the advantage of any ppt algorithm U playing in the following security game is negligible as a function of η.[27]

[27] Here, the operator $[\cdot]_1$ returns the first coordinate of an ordered pair. I.e., if $b = 1$ then A becomes the algorithm that samples a pair according to A_{KeyGen} and then returns the first element of that pair (which is the proper public key).

$\mathbb{S}(U, 1^\eta)$

$b \leftarrow \{0,1\}$

if $b = 1$

$A := A_{\text{MeaninglessPublicKey}}$

else

$A := [A_{\text{KeyGen}}]_1$

$b' \leftarrow U(A, 1^\eta)$

output b'

end

5.1.5 Zero Knowledge Proofs

We assume that the parties have access to a functionality realizing a Zero Knowledge Proof of Plaintext Knowledge that is sound, complete, and honest-verifier zero-knowledge. We refer to [9] for a more detailed description of the properties we require (p10f) and an instantiation (p17ff). In this document, we make use of the following:

1. As mentioned in section 4.1.3, the parties can call $\mathcal{Z}(\mathbf{x})$ for some ciphertext $\mathbf{x}$ to broadcast $\mathbf{x}$ and then perform a zero knowledge proof for $\mathbf{x}$. The scheme aborts if any proof fails. We pretend that the proof itself does not generate any view.
2. Simulators have access to the following two functions:
   - 2.1. $\mathcal{Z}_{\text{test}}(i, \mathbf{x})$; this will return $\top$ if dishonest party i can successfully perform a zero knowledge proof for $\mathbf{x}$ and $\bot$ otherwise. For any pair $(i, \mathbf{x})$ with $i \in I_{\text{adv}}$ and $\mathbf{x} \in \mathbf{C}$, the probability that it returns $\bot$ is identical to the probability that the scheme aborts as P_i calls $\mathcal{Z}(\mathbf{x})$ in the real world.
   - 2.2. $\mathcal{Z}_{\text{extract}}(i, \mathbf{x})$; whenever $\mathcal{Z}_{\text{test}}(i, \mathbf{x})$ outputs $\top$, it holds that $x := \mathcal{Z}_{\text{extract}}(i, \mathbf{x})$ is a plaintext such that $\mathbf{x}$ is a possible result of $\mathfrak{E}(x, \mathfrak{p})$. Such a knowledge extractor exists by the properties of zero knowledge proofs, and the simulator has access to it because is allowed to depend on the precise code of the adversary.

5.2 Proofs of UC-Emulation

In this section, we provide the code of our simulators and the security proofs.

5.2.1 Distributed Decryption returning shares of the plaintext

Theorem 2
$\Pi_{\text{ddec\_shares}}$ *UC-emulates* $\mathcal{F}_{\text{ddec\_shares}}$.[28]

Proof. The ideal functionality, protocol, and simulator are provided in figures 4.2 and 4.3 and 5.2.

Note that the simulator checks whether any dishonest party fails her zero knowledge proof (lines 5-10), and if so, both the protocol and the ideal functionality abort. From now on, we assume all zero knowledge proofs succeed.

Given $\mathbf{x} \in \mathbf{C}$, let $V_R(\mathbf{x})$ and $V_I(\mathbf{x})$ denote the view that $\mathcal{E}$ receives in the real and ideal world, respectively. Note that these are random variables. It is easy to see that $V_R(\mathbf{x})$ and $V_I(\mathbf{x})$ are not identically distributed.[29] Therefore, we argue for indistinguishability by reducing the problem of distinguishing $V_R(\mathbf{x})$ and $V_I(\mathbf{x})$ to the problem of distinguishing $\mathfrak{p}$ and $\mathfrak{p}_0$ (see section 5.1.4). To do this, we provide an algorithm that takes as input a public key $\mathfrak{p}_?$, a ppt algorithm $\mathcal{A}$, and some $x \in \mathbb{F}$, and does the following:

$\mathbb{A}(\mathfrak{p}_?, \mathcal{A}, x)$
- $\langle m \rangle_i \xleftarrow{u} \mathbb{F} \quad \forall \in I_{\text{hst}}$
- $\mathbf{m}_i \leftarrow \mathfrak{E}(\langle m \rangle_i, \mathfrak{p}_?) \quad \forall \in I_{\text{hst}}$
- $\{\langle m \rangle_i\}_{i \in I_{\text{adv}}} \leftarrow \mathcal{A}(\{\mathbf{m}_i\}_{i \in I_{\text{hst}}})$
- $m := \sum_{i=1}^{n} \langle m \rangle_i$
- $\mathbf{c} \leftarrow \mathfrak{E}(x + m, \mathfrak{p}_?)$
- $\delta \leftarrow \mathcal{A}(\{\mathbf{m}_i\}_{i \in I_{\text{hst}}}, (x + m, \mathbf{c}))$
- $\langle x \rangle_i := \frac{1}{n}(x + m + \delta) - \langle m \rangle_i \quad \forall i \in [n]$

Output: $(\{\mathbf{m}_i\}_{i \in I_{\text{hst}}}, (x + m, \mathbf{c}), (x + m + \delta), \{\langle x \rangle_i\}_{i \in [n]})$

Suppose that the x input to this algorithm is the plaintext in $\mathbf{x}$. Then, it holds true that

[28] Formally, we define a protocol system $\Pi := \{M_1, ..., M_n, S\}$ and roles $\{R^{(i)}_{\text{ddec\_shares}}\}_{i \in [n]}$ and R_{setup} as described in section 5.1.2, and a second protocol system $\mathcal{F} := \{M, S'\}$ where M is a single machine with role $R_{\text{ddec\_shares}}$ whose main function is given by the code of $\mathcal{F}_{\text{ddec\_shares}}$, and S' does the same as S.

Since $\Pi_{\text{ddec\_shares}}$ makes use of $\mathcal{F}_{\text{dist\_decrypt}}$, we furthermore give each role $R^{(i)}_{\text{ddec\_shares}}$ an external tape $(n + i, 2n + 1)$. Let M' be the single machine implementing a role whose main function is given by the code of $\mathcal{F}_{\text{dist\_decrypt}}$, with external tapes $\{(2n + 1, n + i)\}_{i \in [n]}$. Then, Π and $\mathcal{F}'$ are connectable and we consider the joint protocol system $\{\Pi, \mathcal{F}'\}$. In this joint system, the role that $\mathcal{F}'$ implements is private, while all other roles remain public.

The theorem statement that $\Pi_{\text{ddec\_shares}}$ UC-emulates $\mathcal{F}_{\text{ddec\_shares}}$ now translates into the more formal statement that $\{\Pi, \mathcal{F}'\} \leq \mathcal{F}$ (as defined in [5] p8).

We implicitly assume an analogous construction for every future theorem in this chapter.

[29] Note that it is not possible to write the simulator in such a way that they are since the outputs of honest parties in the real world (the $\langle x \rangle_i = \frac{1}{n}(r + m) - \langle m \rangle_i$) are correlated with the $\langle \mathbf{m} \rangle_i$ for honest parties.

$$\mathbb{A}(\mathfrak{p}, x, \mathcal{A}) = V_R(\mathbf{x}) \qquad \text{and} \qquad \mathbb{A}(\mathfrak{p}_0, x, \mathcal{A}) = V_I(\mathbf{x})$$

where = means "identically distributed." note that no special attention need be given to $\mathcal{A}$ since it receives the same objects as $\mathcal{E}$, so if no algorithm $\mathcal{E}$ can distinguish the view, then no adversary $\mathcal{A}$ can behave differently, either. Thus, the first equation is straight-forward to verify by comparing the output of $\mathbb{A}$ to what leaks from the protocol in the real world, namely

$\mathcal{S}_{\text{ddec\_shares}}$

0 **Input:** $\mathbf{x} : \mathbf{C}$
1 $\forall i \in I_{\text{hst}}$: $\mathbf{m}_i \leftarrow \mathfrak{E}(0, \mathfrak{p})$
2 **show:** $\{\mathbf{m}_i\}_{i \in I_{\text{hst}}}$
3 $\{\langle \mathbf{m} \rangle_i\}_{i \in I_{\text{adv}}} \leftarrow \mathcal{A}$
4 `for` $i \in I_{\text{adv}}$
5 `if` $\mathcal{Z}_{\text{test}}(i, \mathbf{m}_i) = \perp$:
6 **run** $\mathcal{F}^{\rightarrow\infty}$ $[\top]$ # $\top$ for ϕ_{zkcheat}
7 `endif`
8 $m_i := \mathcal{Z}_{\text{extract}}(i, \mathbf{m}_i)$
9 `endfor`
10 **run** $\mathcal{F}^{\rightarrow 4}[\perp]$ # $\perp$ for ϕ_{zkcheat}
11 $r \xleftarrow{u} \mathbb{F}$ $\mathbf{r} \leftarrow \mathfrak{E}(0, \mathfrak{p})$
12 **show:** $(r, \mathbf{r})$
13 $\delta \leftarrow \mathcal{A}$
14 **show:** $\{r + \delta\}_{i \in I_{\text{adv}}}$
15 **run** $\mathcal{F}^{\rightarrow 5}[\{\frac{1}{n}(r + \delta) - m_i\}_{i \in I_{\text{adv}}}]$
16 **run** $\mathcal{F}^{\rightarrow\infty}$

Figure 5.2: The simulator for $\Pi_{\text{ddec\_shares}}$ and $\mathcal{F}_{\text{ddec\_shares}}$.

1. the ciphertexts $\mathbf{m}_i$ broadcast by the honest parties (line 3)
2. during the call to $\mathcal{F}_{\text{dist\_decrypt}}$ (line 5):
   2.1. the pair sent to the adversary (line 2) (the plaintext $x + m$ and its ciphertext)

2.2. the output to dishonest parties $(x + m + \delta)$

3. the shares $\langle x \rangle_i = \frac{1}{n}(x + m + \delta) - \langle m \rangle_i$ of all parties of the protocol's output (line 6)

Before we turn to the second equation, we jump ahead and conclude the argument. Let $x \in \mathbb{F}$ and let $\mathcal{E}^*$ and $\mathcal{A}^*$ be ppt algorithms. Then, we can construct a distinguisher U for the security game $\mathbb{S}$ from section 5.1.4 as follows:[30]

$U_{x,\mathcal{E}^*,\mathcal{A}^*}(A, 1^\eta)$
  $\mathfrak{p}_? \leftarrow A$
  $V \leftarrow \mathbb{A}(x, \mathcal{A}^*, \mathfrak{p}_?)$
  $b \leftarrow \mathcal{E}^*(V)$
  output b
end

It is easy to see that this distinguisher has the same advantage on the security game $\mathbb{S}$ as $\mathcal{E}$ has on distinguishing between $V_I(\mathbf{x})$ and $V_R(\mathbf{x})$. It follows that the advantage of $\mathcal{E}$ is negligible. Since this is true for any ppt algorithms, it is also true for the environment and adversary during a run of our protocol in the ideal world.

We now argue that $\mathbb{A}(\mathfrak{p}_0, \mathcal{A}, x) = V_I(\mathbf{x})$. Recall that the output of $\mathbb{A}$ is the quadruple $(\{\mathbf{m}_i\}_{i \in I_{\text{hst}}}, (x + m, \mathbf{c}), (x + m + \delta), \{\langle x \rangle_i\}_{i \in [n]})$ where all encryptions are created with the key $\mathfrak{p}_0$ and are thus distributed as fresh encryptions of 0. Conversely, the four components of $V_I(\mathbf{x})$ are the objects the simulator sends, namely

1. fresh encryptions of 0 (lines 1-2). This is trivially identically distributed (to the $\{\mathbf{m}_i\}_{i \in I_{\text{hst}}}$ from the output of $\mathbb{A}$).

   2.1. the pair $(r, \mathbf{0})$, where r is uniformly random and $\mathbf{0}$ a fresh encryption of 0 (lines 11-12). The second component is trivial; the first is identically distributed since m is uniformly distributed and therefore $x + m$ is as well.

   2.2. the value $r + \delta$ (line 13), which, in both cases, equals the first component from the pair above plus δ.

3. the protocol's output

With regard to the output, recall that we assume $n \in I_{\text{hst}}$. Consider first the set of outputs for dishonest parties in $[n - 1]$. In $\mathbb{A}$, they are computed as

[30] Note that we omit the dependence on the security parameter η for simplicity. Formally, each η defines a different view since the setup machine S takes η and can make the field $\mathbb{F}$ or the cryptosystem dependent on η.

$$\langle x\rangle_i := \tfrac{1}{n}(x + m + \delta) - \langle m\rangle_i$$

In the ideal functionality (line 7 and line 15 in the simulator), they are computed as

$$\langle x\rangle_i := \tfrac{1}{n}(r + \delta) - m_i$$

The number $\langle m\rangle_i$ in $\mathbb{A}$ equally distributed to the number m_i in the ideal functionality because both are equal to $\mathfrak{D}(\mathbf{m}_i, \mathfrak{s})$, and we have already argued that r is distributed identically to $x + m$. From this, it follows that the entire set $\{\langle x\rangle_i\}_{i\in I_{\text{adv}}\cap[n-1]}$ is identically distributed in the protocol vs. the output of $\mathbb{A}$.

In $\mathbb{A}$, the shares in $I_{\text{hst}} \cap [n-1]$ (if any) are computed in the same way as those in $I_{\text{adv}} \cap [n-1]$. In the ideal functionality, they are chosen uniformly random (line 8). This yields the same distribution since the shares in $\mathbb{A}$ contain the uniformly distributed additive component $\langle m\rangle_i$.

A crucial consideration here is whether there exists a correlation between these shares and the previous output of $\mathbb{A}$ (which would ruin the result since there clearly is no such correlation in the shares output by the ideal functionality). To see that this is not the case, simply notice that the first three components in the output of $\mathbb{A}$ merely correlate with $m = \sum_{i=1}^{n}\langle m\rangle_i$ but have no further correlation with the $\{\langle m\rangle_i\}_{i\in I_{\text{hst}}}$, and each of the shares $\{\langle x\rangle_i\}_{x\in[n-1]\cap I_{\text{hst}}}$ is independent of $\langle m\rangle_n$ and therefore independent of $\sum_{i\in I_{\text{hst}}}\langle m\rangle_i$ as well.

Finally, share $\langle x\rangle_n$ is fully determined by the previous shares in both cases. □

5.2.2 Matrix Triple Consumption

Theorem 3
$\Pi_{\text{consume\_matrix\_triple}}$ *UC-emulates* $\mathcal{F}_{\text{consume\_matrix\_triple}}$ *in the* $\mathcal{F}_{\text{commit}}$*-hybrid model.*[31]

Before we turn to the formal proof, we briefly explain why the ideal functionality (figure 4.5) has this form. Recall the formula for matrix triples:

$$\langle ST\rangle_i = (S - A)\langle T\rangle_i + \langle A\rangle_i(T - B) + \langle C\rangle_i$$

It is apparent from this formula that the shares of dishonest parties are not uniformly distributed: the shares $\langle A\rangle_i$ and $\langle C\rangle_i$ can be freely chosen by dishonest parties, and we cannot make any assumptions about the shares $\langle S\rangle_i$ or $\langle T\rangle_i$. Thus, the ideal functionality has no choice but to compute these shares the same way the protocol does. Conversely, the shares for honest parties can be shown to be uniform as we will demonstrate.

[31] This clause is added because the code of the protocol contains the $\mathcal{O}$ symbol to open values and thus implicitly uses an ideal functionality $\mathcal{F}_{\text{commit}}$ (see section 4.1.1).

Proof. The ideal functionality, protocol, and simulator are provided in figures 4.5 and 4.4 and 5.3.

We first consider the output. Since we assume values are opened correctly (see section 5.1.3), it is apparent that the shares of dishonest parties are identically distributed in the protocol and ideal functionality as they are computed in identical fashion.

$\mathcal{S}_{\text{consume\_matrix\_triple}}$

0 **Input:** $\{\langle\langle S\rangle\rangle_i, \langle\langle T\rangle\rangle_i, (\langle\langle A\rangle\rangle_i, \langle\langle B\rangle\rangle_i, \langle\langle C\rangle\rangle_i)\}_{i\in I_{\text{adv}}}$

1 $(R_i, R'_i) \xleftarrow{u} \mathbb{F}^{2d^2} \quad \forall i \in I_{\text{hst}}$

2 **show:** $\{R, R'\}_{i\in I_{\text{hst}}}$

3 **run** $\mathcal{F}^{\to\infty}$

Figure 5.3: The simulator for $\Pi_{\text{consume\_matrix\_triple}}$ and $\mathcal{F}_{\text{consume\_matrix\_triple}}$

For the shares of honest parties, note that $\langle ST\rangle_i$ includes the value $\langle C\rangle_i$ as an additive component, and the shares $\langle C\rangle_i$ are uniformly independently distributed under the condition that $C := AB$ (see figure 4.6 line 12 and figure 4.2 line 8). Therefore, for any real subset $I \subsetneq I_{\text{hst}}$, the shares $\langle ST\rangle_i$ are identically distributed in the protocol vs. ideal functionality, and then the final share is determined by the others and the value of ST (see definition of the $\xleftarrow{u/c}$ operator in section 4.1.4). An analogous argument applies for $\langle\overline{ST}\rangle_i$ and $\langle\overline{C}\rangle_i$.

This leaves the shares of honest parties that are leaked as $S - A$ and $T - B$ are opened. Note that the simulator sends uniformly independently distributed objects here (lines 1-2). Since the shares $\langle A\rangle_i$ and $\langle B\rangle_i$ for honest parties are chosen uniformly random (see figure 4.6 line 6), the sums $\langle S\rangle_i - \langle A\rangle_i$ and $\langle T\rangle_i - \langle B\rangle_i$ are uniformly random as well. Furthermore, they are not correlated with the shares $\langle ST\rangle_i$ and $\langle\overline{ST}\rangle_i$ that are output to honest parties because those do not depend on the shares $\langle A\rangle_i$ and $\langle B\rangle_i$ as we have demonstrated in the previous paragraph. □

5.2.3 Matrix Triple Generation

Theorem 4
$\Pi_{\text{g\_matrix\_triple}}$ *UC-emulates* $\mathcal{F}_{\text{g\_matrix\_triple}}$ *in the* $\mathcal{F}_{\text{commit}}$*-hybrid model.*

Proof. The ideal functionality, protocol, and simulator are provided in figures 4.6 and 4.7 and 5.4, 5.5.

$S_{\text{g\_matrix\_triple}}$

0 **Input:** —

1 $\forall i \in I_{\text{hst}} : \langle \mathbf{A} \rangle_i \leftarrow \mathfrak{E}(0_{d\times d}, \mathfrak{p}) \quad \langle \mathbf{B} \rangle_i \leftarrow \mathfrak{E}(0_{d\times d}, \mathfrak{p}) \quad \langle \mathbf{A}' \rangle_i \leftarrow \mathfrak{E}(0_{d\times d}, \mathfrak{p})$

2 **show:** $\{\langle \mathbf{A} \rangle_i, \langle \mathbf{B} \rangle_i, \langle \mathbf{A}' \rangle_i\}_{i \in I_{\text{hst}}}$

3 $\{(\langle \mathbf{A} \rangle_i, \langle \mathbf{B} \rangle_i, \langle \mathbf{A}' \rangle_i)\}_{i \in I_{\text{adv}}} \leftarrow \mathcal{A}$

4 `for` $i \in I_{\text{adv}}, (k, \ell) \in [d] \times [d]$:

5 `if` $\mathcal{Z}_{\text{test}}(i, \langle \mathbf{a}_{k,\ell} \rangle_i) = \bot \ \vee \ \mathcal{Z}_{\text{test}}(i, \langle \mathbf{b}_{k,\ell} \rangle_i) = \bot \ \vee \ \mathcal{Z}_{\text{test}}(i, \langle \mathbf{a}'_{k,\ell} \rangle_i) = \bot$:

6 **run** $\mathcal{F}^{\to\infty}[\top]$ # $\top$ for ϕ_{zkcheat}

7 `endif`

8 $\langle a_{k,\ell} \rangle_i \leftarrow \mathcal{Z}_{\text{extract}}(i, \mathbf{a}_{k,\ell})$

9 $\langle b_{k,\ell} \rangle_i \leftarrow \mathcal{Z}_{\text{extract}}(i, \mathbf{b}_{k,\ell})$

10 $\langle a'_{k,\ell} \rangle_i \leftarrow \mathcal{Z}_{\text{extract}}(i, \mathbf{a}'_{k,\ell})$

11 `endfor`

12 **run** $\mathcal{F}^{\to 4}[\bot]$ # $\bot$ for ϕ_{zkcheat}

13 **run** $\mathcal{F}^{\to 5}[\{\langle A \rangle_i, \langle B \rangle_i\}_{i \in I_{\text{adv}}}]$

Throughout lines 7-14 in the protocol, the ideal functionality $\mathcal{F}_{\text{ddec\_shares}}$ is called six times. Four four of those calls (lines 7, 11, 12, 13), corresponding calls exist in the ideal functionality (lines 8, 10, 11, 12). During those, the simulator relays messages unchanged. For the other two (lines 8, 14), the simulator calls its own copy of $\mathcal{F}_{\text{ddec\_shares}}$ with fresh encryptions of 0. It then shows the shares it has received from $\mathcal{A}$ (line 7 in figure 4.2) and uniformly sampled numbers for the shares of honest parties to the environment.

Let $\Delta = (\delta_{k,\ell})_{1\le k,\ell\le d}, \Delta' = (\delta'_{k,\ell})_{1\le k,\ell\le d}$ be the encryption errors chosen by $\mathcal{A}$ during the execution of $\mathcal{F}_{\text{ddec\_shares}}$ on the $\mathbf{c}_{k,\ell}$ and $\mathbf{c}'_{k,\ell}$ (lines 7, 8). Furthermore, let $\{\langle c \rangle_i\}_{i \in I_{\text{adv}}}$ and $\{\langle c' \rangle_i\}_{i \in I_{\text{adv}}}$ be the shares chosen by dishonest parties during those calls.

Figure 5.4: The simulator for $\prod_{\text{g\_matrix\_triple}}$ and $\mathcal{F}_{\text{g\_matrix\_triple}}$ (part 1/2)

$S_{\text{g\_matrix\_triple}}$ (cont.)

14 $\{\langle r\rangle_i\}_{i\in I_{\text{adv}}} \leftarrow \mathcal{A}$

15 $\forall i \in I_{\text{hst}} : \langle r\rangle_i \xleftarrow{u} \mathbb{F}$

16 **show:** $\{\langle r\rangle_i\}_{i\in I_{\text{hst}}}$

17 $r := \sum_{i=1}^{n} \langle r\rangle_i$

18 $\{\langle rA - A'\rangle_i\}_{i\in I_{\text{adv}}} \leftarrow \mathcal{A}$

19 $\forall i \in I_{\text{hst}} : \langle rA - A'\rangle_i \xleftarrow{u} \mathbb{F}^{d^2}$

20 **show:** $\{\langle rA - A'\rangle_i\}_{i\in I_{\text{hst}}}$

21 $\Delta^* := \sum_{i\in I_{\text{adv}}} \langle rA - A'\rangle_i - \sum_{i\in I_{\text{adv}}} r\langle A\rangle_i - \langle A'\rangle_i$ # error in opening $rA - A'$

22 `if` $\Delta' \neq 0 \vee \Delta^* \neq 0$:

23 $X \leftarrow$ **run** $\mathcal{F}^{\rightarrow 16}[\top, \Delta^*]$ # $[\phi_{\text{failcheck}}, L]$. Note that $X = \sum_{i\in I_{\text{hst}}} \langle \Delta^* B\rangle_i$.

24 $Z := -r\Delta + \Delta' + \sum_{i\in I_{\text{adv}}} \langle B\rangle_i \Delta^* + X$

25 `else:`

26 **run** $\mathcal{F}^{\rightarrow 17}[\bot]$ # $\bot$ for $\phi_{\text{failcheck}}$

27 $Z := -r\Delta$

28 `endif`

29 $v_{rA-A'} := \sum_{i=1}^{n} \langle rA - A'\rangle_i$

30 $\forall i \in I_{\text{adv}} : \langle Z\rangle_i := r\langle C\rangle_i - \langle C'\rangle_i - v_{rA-A'}\langle B\rangle_i$

31 $\forall i \in I_{\text{hst}} : \langle Z\rangle_i \xleftarrow{u/c} \mathbb{F}^{d^2}$

32 **show:** $\{\langle Z\rangle_i\}_{i\in I_{\text{hst}}}$

33 **run** $\mathcal{F}^{\rightarrow\infty}$

Figure 5.5: The simulator for $\prod_{\text{g\_matrix\_triple}}$ and $\mathcal{F}_{\text{g\_matrix\_triple}}$ (part 2/2)

First, note that, if any zero knowledge proof fails, both the ideal functionality and the protocol abort before anything but the ciphertexts corresponding to these proofs are output (lines 1-4 in the ideal functionality and 5-7 in the simulator, implicit in the protocol by section 5.1.5). Thus, we assume from now on that all zero knowledge proofs succeed. As a result, all decryptions and operations on ciphertexts yield the expected results.

In line 1, the simulator guesses that the ciphertexts $\langle \mathbf{A} \rangle_i, \langle \mathbf{B} \rangle_i, \langle \mathbf{A}' \rangle_i$ of honest parties are all fresh encryptions of 0. (Note that we write $0_{d \times d}$ for the $d \times d$ matrix of all 0's and $\mathfrak{E}(0_{d \times d}, \mathfrak{p})$ for the component-wise encryption of this matrix.) Similarly, in the instances where the simulator calls its own copy of $\mathcal{F}_{\text{ddec\_shares}}$, it uses encryptions of 0 as the input. This can be justified by the same argument we've used in theorem 2 (i.e., one can write down an algorithm that produces the view $\mathcal{E}$ receives in the ideal world given key $\mathfrak{p}_0$ and the view $\mathcal{E}$ receives in the real world given key $\mathfrak{p}$). We will not repeat the argument here.

The output of both the protocol and the ideal functionality is $\{(\langle\!\langle A \rangle\!\rangle_i, \langle\!\langle B \rangle\!\rangle_i, \langle\!\langle C \rangle\!\rangle_i)\}_{i \in [n]}$. We begin by arguing that these matrices are identically distributed in both worlds. This is trivial for $\{\langle A \rangle_i\}_{i \in [n]}$ and $\{\langle B \rangle_i\}_{i \in [n]}$ (line 1 in the protocol, 5-7 in the ideal functionality, 3-13 in the simulator). The remaining four matrices result from calls to $\mathcal{F}_{\text{ddec\_shares}}$. Here, it suffices to argue that the inputs to those calls are identically distributed in both worlds. To see this, simply note that A and B and γ are identically distributed and $C, \overline{A}, \overline{B}, \overline{C}$ are fully determined by these objects, and the respective equations ($C = AB, \overline{A} = \gamma A$, etc.) are the same for both worlds. (The adversary can introduce additive decryption errors, but those, too, are the same in both worlds.)

At many points, the simulator samples objects randomly and sends them to $\mathcal{E}$. This the case for the outputs of honest parties during the calls to $\mathcal{F}_{\text{ddec\_shares}}$ on the $\mathbf{c}'_{k,\ell}$ and $\overline{\mathbf{c}_{k,\ell}}'$, for the random element r, and for the matrix $rA - A'$. We will argue below while these objects have the same distribution as the corresponding objects from the real world in every instance, but it is also crucial to verify that they are not correlated with the protocol's output (for if they were, then this correlation would exist only in the real world, and the environment could use this to distinguish between both views). Specifically, the matrices C' and $\overline{C'}$ do not appear in the output, the random element r does not, and the sum $rA - A'$ does not correlate with A since A' does not appear in the output.

Now, the view that environment receives in the real world consists of:

1. the ciphertexts $\langle \mathbf{a}_{k,\ell} \rangle_i, \langle \mathbf{b}_{k,\ell} \rangle_i, \langle \mathbf{a}'_{k,\ell} \rangle_i$ broadcast by honest parties (line 3)
2. the values returned to dishonest parties from the calls to $\mathcal{F}_{\text{ddec\_shares}}$ (lines 7, 8, 11, 12, 13, 14) and those sent to the adversary during these calls
3. the shares of honest parties of the opened value r (line 16)
4. the shares of honest parties of the opened matrix $rA - A'$ (line 17)
5. the shares of honest parties of the opened matrix Z (line 20)

We look at the objects which the simulator sends to the environment point by point:

1. The simulator sends encryptions of 0 (line 2) which suffices as we've argued above.

2. For the calls that have analogs in the ideal functionality (lines 7, 11, 12, 13), we need only observe that the input values are identically distributed, then the relay-only approach of the simulator trivially works as identical code is executed in both worlds. For the calls that don't have analogs in the ideal functionality (lines 8, 14), the simulator calls a copy of $\mathcal{F}_{\text{ddec\_shares}}$ with a different input (namely an encryption of 0) which suffices for the reason described earlier. The ideal functionality will not produce the correct outputs, but the outputs of dishonest parties are freely chosen by the adversary anyway (see line 7 in figure 4.2) and those of honest parties never make it into the view that the environment receives.
3. The simulator simply constructs these shares in identical fashion (lines 15, 16).
4. The simulator shows uniformly sampled values (lines 19, 20). This is correct since each share $r\langle A\rangle_i - \langle A'\rangle_i$ of an honest party contains the uniformly random value $\langle A'\rangle_i$ as an additive component and is thus uniformly random itself. Again, note that $\langle A'\rangle_i$ never appears in the output.

Before we turn to the shares of the matrix Z opened for the sacrificing step (item #5), we argue that the sacrificing step "works," i.e., that, with overwhelming probability, the ideal functionality aborts on line 17 iff the protocol aborts on line 22.

Let $AB = C + \Delta$ and $A'B = C' + \Delta'$ and $\sum_{i=1}^{n}\langle rA - A'\rangle_i = rA - A' + \Delta^*$. Since we assume that the value Z is opened correctly, the protocol proceeds iff the following equation holds:

$$
\begin{aligned}
(*) \qquad & \sum_{i=1}^{n}\langle Z\rangle_i = 0 \\
\iff \quad & \sum_{i=1}^{n}\left(r\langle C\rangle_i - \langle C'\rangle_i - (rA - A' + \Delta^*)\langle B\rangle_i\right) = 0 \\
\iff \quad & rC - C' - (rA - A' + \Delta^*)B = 0 \\
\iff \quad & r(C - AB) + (A'B - C') + \Delta^* B = 0 \\
\iff \quad & -r\Delta + \Delta' + \Delta^* B = 0
\end{aligned}
$$

Assume first that $\Delta^* \neq 0$. In this case, the ideal functionality always aborts since the simulator sends $\top$ for $\phi_{\text{failcheck}}$ (lines 21-23). In the protocol, the matrix B is uniformly independently distributed with respect to $r, \Delta, \Delta', \Delta^*$, hence we find a pair (k, ℓ) such that $(\Delta^* B)_{k,\ell}$ is uniformly independently distributed. Then, the entire left hand side of $(*)$ is uniformly independently distributed at those coordinates, which implies that $(*)$ holds with negligible probability, so the protocol aborts with overwhelming probability.

Next, assume $\Delta^* = 0_{d\times d}$ and $\Delta \neq 0_{d\times d}$. Then, $AB \neq C$ and the ideal functionality always aborts due to the check in line 14. Let (k, ℓ) be such that $\delta_{k,\ell} \neq 0$, then $(*)$ for coordinates (k, ℓ) only can be written as

$$-r\delta_{k,\ell} + \delta'_{k,\ell} = 0 \iff r = \delta'_{k,\ell}/\delta_{k,\ell}$$

which has negligible probability because r is uniformly independently distributed with respect to Δ and Δ'. Thus, the protocol aborts with overwhelming probability.

Finally, if $\Delta^* = \Delta = 0_{d\times d}$, both the protocol and ideal functionality abort iff $\Delta' \neq 0_{d\times d}$.

This shows that the protocol and ideal functionality differ with respect to abortion with at most negligible probability in all cases. We now turn to the shares $\{\langle Z\rangle_i\}_{i\in I_{\text{hst}}}$ that the dishonest parties receive.

Note that $\sum_{i=1}^{n}\langle Z\rangle_i = -r\Delta + \Delta' + \Delta^* B$. Regardless of whether the value $rA - A'$ is opened correctly and $\Delta^* = 0_{d\times d}$, the simulator computes this value correctly (lines 22-28). Thus, the simulator "knows" the shares of dishonest parties as well as the sum of all shares. It remains to show that, if there is more than one honest party, the value of all those shares is distributed precisely as implied by the $\xleftarrow{u/c}$ command (see section 4.1.4) in the simulator (line 31). To see this, simply note that such an honest share is computed according to the rule $\langle Z\rangle_i := r\langle C\rangle_i - \langle C'\rangle_i - (rA - A')\langle \overline{B}\rangle_i$ and thus contains the additive component $\langle C'\rangle_i$ which is uniformly independently distributed with respect to the previous shares of Z (the entire set $\{\langle C'\rangle_i\}_{i\in I_{\text{hst}}}$ depends on the value of C', but every proper subset does not). □

5.2.4 Power Tuple Consumption

Theorem 5
$\prod_{\text{consume\_matrix\_power\_tuple}}$ *UC-emulates* $\mathcal{F}_{\text{consume\_matrix\_power\_tuple}}$ *in the* $\mathcal{F}_{\text{commit}}$*-hybrid model.*

Proof. (Code in figures 4.8, 4.9, 5.6.) Analogous to the proof of theorem 3.[32] □

$S_{\text{consume\_matrix\_power\_tuple}}$

```
0  Input: {⟪S⟫_i, ℜ_i}_{i∈I_adv}, m : ℕ
1      R_i ←u 𝔽^{d²}   ∀i ∈ I_hst
2      show: {R_i}_{i∈I_hst}
3  run F^{→∞}
```

Figure 5.6: The simulator for $\prod_{\text{consume\_matrix\_power\_tuple}}$ and $\mathcal{F}_{\text{consume\_matrix\_power\_tuple}}$

[32] However, note that the ideal functionality using the $\xleftarrow{u/c}$ command in line 11 only yields the same output as the protocol if skeleton matrices "work," i.e., if they compute correct shares of S^m. We do not prove this formally, but instead refer to the discussion in section 3.2.3.

5.2.5 Power Tuple Generation

Theorem 6
If m and d are constants,[33] *then* $\Pi_{\text{g\_matrix\_power\_tuple}}$ *UC-emulates* $\mathcal{F}_{\text{g\_matrix\_power\_tuple}}$ *in the* $\mathcal{F}_{\text{commit}}$*-hybrid model.*

Proof. The ideal functionality, protocol, and simulator are provided in figures 4.10, 4.11 and 4.12, 4.13 and 5.7, 5.8.

Since this proof is highly similar to that of theorem 4, we only go into detail on parts that are meaningfully different. In particular, discussion on zero knowledge proofs and correctness of the simulator's behavior during the calls to $\mathcal{F}_{\text{ddec\_shares}}$ (including why the inputs are identically distributed) proceeds perfectly analogously.

Note that it is easy for the simulator to follow the structure of the protocol and ideal functionality precisely, since the code guarding the two loops depends only on $A_{\text{list\_terms}}$, which is a deterministic algorithm, and the inputs are known to the simulator.

For the sacrificing steps, note that we can simply define $ab = c + \delta$ and $a'b = c' + \delta'$ and $\sum_{i=1}^{n}(rA - A')_{\text{head}(s)} = rA_{\text{head}(s)} - A'_{\text{head}(s)} + \delta^*$ for each iteration and then argue analogously to the proof of theorem 4 (some steps simplify since we deal with numbers rather than matrices). The only added difficulty is that b is not uniformly distributed (this was something we've used in the proof of theorem 4) since it is the product of a number of the a_t's, and the product of several uniformly distributed field elements is not uniformly distributed itself (the probability that it is zero increases). However, since we consider d and m to be constants, the probability that any b is zero is negligible, and we can rescue the argument since the product of any number of uniformly independently distributed elements in $\mathbb{F}^*$ is uniformly independently distributed in $\mathbb{F}^*$ itself.[34]

Finally, note that r is opened after all operations on ciphertexts are concluded, making it uniformly independently distributed with respect to the δ and δ' (this is the reason why one cannot simply open r in the beginning and then check every computation immediately). Note also that we need not consider how errors in the result of c affect further steps because, if there is an error in that c, it will be caught (and the scheme will abort) before the sacrificing step for future multiplications is executed. Lastly, the probability for a false abortion is 0 therefore does not increase across multiple sacrificing steps. □

[33] Note that this requirement also alleviates runtime concerns.

[34] To see this, let $x \in \mathbb{F}^*$ have an arbitrary distribution, and let $y \in \mathbb{F}^*$ be uniformly independently distributed. Then, given $z \in \mathbb{F}^*$, we have $\Pr[xy = z] = \Pr[y = x^{-1}z] = 1/|\mathbb{F}^*|$.

$S_{\text{g\_matrix\_power\_tuple}}$

0 **Input:** $d : \mathbb{N}, m : \mathbb{N}$

1 $\forall i \in I_{\text{hst}}$: $\langle \mathbf{A} \rangle_i \leftarrow \mathfrak{E}(0_{d \times d}, \mathfrak{p})$ $\langle \mathbf{A}' \rangle_i \leftarrow \mathfrak{E}(0_{d \times d}, \mathfrak{p})$

2 **show:** $\{\langle \mathbf{A} \rangle_i, \langle \mathbf{A}' \rangle_i\}_{i \in I_{\text{hst}}}$

3 $\{\langle \mathbf{A} \rangle_i, \langle \mathbf{A}' \rangle_i\}_{i \in I_{\text{adv}}} \leftarrow \mathcal{A}$

4 `for` $i \in I_{\text{adv}}, t \in \{0, ..., d^2 - 1\}$:

5 `if` $\mathcal{Z}_{\text{test}}(i, \mathbf{a}_t) = \bot \ \vee \ \mathcal{Z}_{\text{test}}(i, \mathbf{a}'_t) = \bot$:

6 **run** $\mathcal{F}^{\to\infty}$ $[\top]$ # $\top$ for ϕ_{zkcheat}

7 `endif`

8 $\langle a_t \rangle_i \leftarrow \mathcal{Z}_{\text{extract}}(i, \mathbf{a}_t)$ $\langle a'_t \rangle_i \leftarrow \mathcal{Z}_{\text{extract}}(i, \mathbf{a}'_t)$

9 `endfor`

10 **run** $\mathcal{F}^{\to 4}[\bot]$ # $\bot$ for ϕ_{zkcheat}

11 **run** $\mathcal{F}^{\to 5}[\{\langle A \rangle_i, \langle A' \rangle_i\}_{i \in I_{\text{adv}}}]$

In line 6 and throughout the loop in lines 10-23, the protocol contains five calls to $\mathcal{F}_{\text{ddec\_shares}}$. Four three of those calls (lines 6, 15, 18), corresponding calls exist in the ideal functionality (lines 8, 15, 17). During those, the simulator relays messages unchanged. For the other two (lines 16, 19), the simulator calls its own copy of $\mathcal{F}_{\text{ddec\_shares}}$ with a fresh encryption of 0. It then shows $\mathcal{E}$ the shares it has received from $\mathcal{A}$ (line 7 in figure 4.2) and uniformly sampled numbers for the shares of honest parties.

Let $D : \{0, ..., d^2 - 1\}^* \to \mathbb{F}^2$ be the map containing the additive encryption errors chosen by $\mathcal{A}$. Specifically, let it be such that, if $D(s) = (\delta, \delta')$, then δ and δ' are the additive errors chosen by $\mathcal{A}$ during the encryptions of $\mathbf{c}$ and $\mathbf{c}'$, respectively (lines 15, 16) during the iteration of the loop that corresponds to s. Furthermore, let $C : \{0, ..., d^2 - 1\}^* \to (\mathbb{F}^{|I_{\text{adv}}|})^2$ be such that, if $C(s) = (\{c\}_{i \in I_{\text{adv}}}, \{c'\}_{i \in I_{\text{adv}}})$, then the $\{c\}_{i \in I_{\text{adv}}}$ and $\{c'\}_{i \in I_{\text{adv}}}$ are the shares chosen by dishonest parties for these decryptions (see figure 4.2 line 7)

Figure 5.7: The simulator for $\Pi_{\text{g\_matrix\_power\_tuple}}$ and $\mathcal{F}_{\text{g\_matrix\_power\_tuple}}$ (part 1/2)

$S_{\text{g\_matrix\_power\_tuple}}$ (cont.)

12 $\{\langle r\rangle_i\}_{i\in I_{\text{adv}}} \leftarrow \mathcal{A}$

13 $\forall i \in I_{\text{hst}} : \langle r\rangle_i \xleftarrow{u} \mathbb{F}$

14 **show:** $\{\langle r\rangle_i\}_{i\in I_{\text{hst}}}$ $\forall i \in I_{\text{hst}}$

15 $\{\langle rA - A'\rangle_i\}_{i\in I_{\text{adv}}} \leftarrow \mathcal{A}$

16 $\forall i \in I_{\text{hst}} : \langle rA - A'\rangle_i \xleftarrow{u} \mathbb{F}^{d^2}$

17 **show:** $\{\langle rA - A'\rangle_i\}_{i\in I_{\text{hst}}}$

18 $\Delta^* := \sum_{i\in I_{\text{adv}}}\langle rA - A'\rangle_i - \sum_{i\in I_{\text{adv}}} r\langle A\rangle_i - \langle A'\rangle_i$ # error in opening $rA - A'$

19 `for` $\ell \leftarrow 2$ `to` k :

20 `for` $s \in S_\ell$: # $S_\ell = A_{\text{list\_terms}}(d, m, \ell)$ (the simulator computed this earlier)

21 $(\delta, \delta') := D(s)$ $j := \text{head}(s)$ # $\delta^*_j \triangleq$ add. err. for coordinate of $rA - A'$

22 `if` $\delta' \neq 0 \vee \delta^*_j \neq 0$:

23 $x \leftarrow$ **run** $\mathcal{F}^{\rightarrow 26}[\top, \delta^*_j]$ # $\phi_{\text{failcheck}}, y$

24 $z := -r\delta + \delta' + \sum_{i\in I_{\text{adv}}}\langle b\rangle_i\delta^*_j + x$

25 `else:`

26 **run** $\mathcal{F}^{\rightarrow 24}[\bot]$ # $\bot$ for $\phi_{\text{failcheck}}$

27 $z := -r\delta$

28 `endif` # $\downarrow$ recover values computed in previous loop

29 $(\{\langle c\rangle_i\}_{i\in I_{\text{adv}}}, \{\langle c'\rangle_i\}_{i\in I_{\text{adv}}}) := C(s)$ $(\{\langle b\rangle_i\}_{i\in I_{\text{adv}}}, \_) := C(\text{tail}(s))$

30 $v_{ra-a'} := (\sum_{i=1}^{n}\langle rA - A'\rangle_i)_j$

31 $\forall i \in I_{\text{adv}} : \langle z\rangle_i := r\langle c\rangle_i - \langle c'\rangle_i - v_{ra-a'}\langle b\rangle_i$

32 $\forall i \in I_{\text{hst}} : \langle z\rangle_i \xleftarrow{u/c} \mathbb{F}$

33 **show:** $\{\langle z\rangle_i\}_{i\in I_{\text{hst}}}$

34 **run** $\mathcal{F}^{\rightarrow 29}$

35 `endfor`

36 `endfor`

37 **run** $\mathcal{F}^{\rightarrow\infty}$

Figure 5.8: The simulator for $\prod_{\text{g\_matrix\_power\_tuple}}$ and $\mathcal{F}_{\text{g\_matrix\_power\_tuple}}$ (part 2/2)

Bibliography

[BGV11] Z. Brakerski, C. Gentry, V. Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *Electronic Colloquium on Computational Complexity (ECCC)* 18 (Jan. 2011), p. 111. DOI: 10.1145/2090236.2090262 (cit. on pp. 14, 15).

[Can01] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: vol. 2000. Nov. 2001, pp. 136–145. ISBN: 0-7695-1116-3. DOI: 10.1109/SFCS.2001.959888 (cit. on p. 51).

[CCL15] R. Canetti, A. Cohen, Y. Lindell. "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation". In: Aug. 2015, pp. 3–22. ISBN: 978-3-662-47999-5. DOI: 10.1007/978-3-662-48000-7_1 (cit. on p. 51).

[CDPW07] R. Canetti, Y. Dodis, R. Pass, S. Walfish. "Universally Composable Security with Global Setup." In: Jan. 2007, pp. 61–85 (cit. on p. 51).

[CKKR19] J. Camenisch, S. Krenn, R. Küsters, D. Rausch. "iUC: Flexible Universal Composability Made Simple". In: Nov. 2019, pp. 191–221. ISBN: 978-3-030-34617-1. DOI: 10.1007/978-3-030-34618-8_7 (cit. on pp. 7, 29, 51, 52, 56).

[Con19] Contributors of ProofWiki. *Sum of Sequence of Fibonacci Numbers*. 2019. URL: https://proofwiki.org/w/index.php?title=Sum_of_Sequence_of_Fibonacci_Numbers&oldid=390491 (cit. on p. 28).

[CR03] R. Canetti, T. Rabin. "Universal Composition with Joint State". In: Aug. 2003, pp. 265–281. DOI: 10.1007/978-3-540-45146-4_16 (cit. on p. 51).

[DKL+13] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N. Smart. "Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits". In: Sept. 2013, pp. 1–18. DOI: 10.1007/978-3-642-40203-6_1 (cit. on p. 9).

[DPSZ11] I. Damgård, V. Pastro, N. Smart, S. Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *IACR Cryptology ePrint Archive* 2011 (Jan. 2011), p. 535. DOI: 10.1007/978-3-642-32009-5_38 (cit. on pp. 9, 29–33, 54, 55).

[HS11] D. Hofheinz, V. Shoup. “GNUC: A New Universal Composability Framework”. In: *IACR Cryptology ePrint Archive* 2011 (Jan. 2011), p. 303. DOI: 10.1007/s00145-013-9160-y (cit. on p. 51).

[KPR18] M. Keller, V. Pastro, D. Rotaru. “Overdrive: Making SPDZ Great Again”. In: Jan. 2018, pp. 158–189. ISBN: 978-3-319-78371-0. DOI: 10.1007/978-3-319-78372-7_6 (cit. on p. 20).

[KTR20] R. Küsters, M. Tuengerthal, D. Rausch. “The IITM Model: A Simple and Expressive Model for Universal Composability”. In: *Journal of Cryptology* (June 2020). DOI: 10.1007/s00145-020-09352-1 (cit. on p. 51).

[Küs06] R. Küsters. “Simulation-Based Security with Inexhaustible Interactive Turing Machines.” In: *IACR Cryptology ePrint Archive* 2006 (Jan. 2006), p. 151. DOI: 10.1109/CSFW.2006.30 (cit. on p. 51).

[Wik20] Wikipedia contributors. *Division ring — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-May-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Division_ring&oldid=948614821 (cit. on p. 72).

A Proof of the Negative Result for MACs

Recall the formal claim we wish to prove:

Restatement of Theorem 1
Let $\mathcal{R}$ be a ring, $\mathbb{F}$ a field, and $d \in \mathbb{N}$. If $\phi : M_d(\mathbb{F}) \to \mathcal{R}$ is a rng-homomorphism, then either $\phi(M_d(\mathbb{F})) \simeq M_d(\mathbb{F})$ or $\phi(M_d(\mathbb{F})) \simeq 0_{\mathcal{R}}$, where $0_{\mathcal{R}}$ is the zero ring.

We now introduce the ring theory we will need.

Given a ring $\mathcal{R}$ and a subset $A \subseteq \mathcal{R}$, we define $A\mathcal{R} := \{a \cdot r \mid a \in A \wedge r \in \mathcal{R}\}$ and $\mathcal{R}A := \{r \cdot a \mid a \in A \wedge r \in \mathcal{R}\}$. We say that a subset $I \subseteq \mathcal{R}$ is a left ideal iff $I\mathcal{R} \subseteq I$, a right ideal iff $\mathcal{R}I \subseteq I$, and a two-sided ideal iff it is both a left ideal and a right ideal.

We next show that, given any ring $\mathcal{S}$ and any ring homomorphism $\varphi : \mathcal{R} \to \mathcal{S}$, the set $\ker(\varphi) \subseteq \mathcal{R}$ is a two-sided ideal. Let $x \in \ker(\varphi)\mathcal{R}$, then we can write x as zr for $z \in \ker(\varphi)$ and $r \in \mathcal{R}$. Thus, we have $\varphi(x) = \varphi(zr) = \varphi(z)\varphi(r) = 0\varphi(r) = 0$, and hence $\varphi(x) \in \ker(\varphi)$, as needed. The proof for an element $x \in \mathcal{R}\ker(\varphi)$ proceeds analogously.

Given some two-sided ideal $I \subseteq \mathcal{R}$, we write

$$\mathcal{R}/I$$

for the ring based on the set $\{r + I \mid r \in \mathcal{R}\}$ with operations $+$ and $\cdot$ defined by the rules $(r + I) + (r' + I) := (r + r') + I$ and $(r + I) \cdot (r' + I) := (rr') + I$. This kind of ring is called a *quotient ring*. One can show that these operations are well-defined (even though the r and r' are representatives out of an equivalence class).

As an example, consider $\mathcal{R} = \mathbb{Z}$ and $I = 2\mathbb{Z} = \{2z \mid z \in \mathbb{Z}\}$. It is easy to check that I is a two-sided ideal (since $\mathbb{Z}$ is commutative, it suffices to check that I is a left-sided ideal). Then the two elements of $\mathbb{Z}/I$ are $0 + \mathbb{Z}$ and $1 + \mathbb{Z}$; it is easy to check that there are no others, since $n + \mathbb{Z} = (n + 2) \quad \forall n \in \mathbb{N}$. The resulting quotient ring is isomorphic to $\mathbb{Z}_2$, the ring of integers with addition and multiplication modulo 2. In fact, some mathematicians consider $\mathbb{Z}/2\mathbb{Z}$ to be the definition of, rather than just isomorphic to, the ring $\mathbb{Z}_2$.

Now, for an arbitrary ring homomorphism $\varphi : \mathcal{R} \to \mathcal{S}$, we have that

$$\mathcal{R} \Big/ \ker(\varphi) \simeq \varphi(\mathcal{R})$$

This fact can be considered a generalization of the result that φ is injective iff $\ker(\varphi) = \{0\}$. The ring isomorphism Ψ which proves it is given by

$$\Psi : \mathcal{R} \Big/ \ker(\varphi) \to \varphi(\mathcal{R}) \qquad\qquad \Psi : (r + \ker(\varphi)) \mapsto \varphi(r)$$

Now Ψ is surjective because, given $s \in \varphi(\mathcal{R})$, we find $r \in \mathcal{R}$ such that $\varphi(r) = s$, which implies that $\Psi(r + I) = s$. Furthermore, Ψ is injective since

$$\begin{aligned}\Psi(r + \ker(\varphi)) = \Psi(r' + \ker(\varphi)) &\implies \varphi(r) = \varphi(r') \implies \varphi(r) - \varphi(r') = 0 \\ &\implies \varphi(r - r') = 0 \implies r - r' \in \ker(\varphi)\end{aligned}$$

and therefore $\ker(\varphi) = (r - r') + \ker(\varphi)$, from which it follows that

$$r' + \ker(\varphi) = r' + [(r - r') + \ker(\varphi)] = r + \ker(\varphi).$$

Finally, we will use the fact that $M_2(\mathbb{F})$ is a *simple ring*, i.e. its only two-sided ideals are itself and 0_R [14]. With this bit of theory established, we turn to proving theorem 1.

Proof. Let $\phi : M_d(\mathbb{F}) \to \mathcal{R}$ be a rng homomorphism. We first show that ϕ becomes a ring homomorphism if we restrict its co-domain to its image. So let $S := \phi(M_d(\mathbb{F}))$. Note that $S \subseteq \mathcal{R}$ is a subring. Given any $y \in S$, we find a $X \in M_d(\mathbb{F})$ such that $\phi(X) = y$, and we have

$$\phi(I_d)y = \phi(I_d)\phi(X) = \phi(I_d X) = \phi(X) = y = \phi(X I_d) = \phi(X)\phi(I_d) = y\phi(I_d)$$

which shows that $\phi(I_d)$ is the multiplicatively neutral element in S, which implies that $\phi : M_d(\mathbb{F}) \to S$ is indeed a ring homomorphism.

Now $\ker(\phi)$ is a two-sided ideal in $M_d(\mathbb{F})$. Since $M_d(\mathbb{F})$ is simple, this implies that $\ker(\phi) \simeq M_d(\mathbb{F})$ or $\ker(\phi) \simeq 0_{\mathcal{R}}$. In the former case,

$$\phi(M_d(\mathbb{F})) \simeq M_d(\mathbb{F}) \Big/ \ker(\varphi) \simeq M_d(\mathbb{F}) \Big/ M_d(\mathbb{F}) \simeq 0_{\mathcal{R}}$$

and in the latter case,

$$\phi(M_d(\mathbb{F})) \simeq M_d(\mathbb{F}) \Big/ \ker(\varphi) \simeq M_d(\mathbb{F}) \Big/ 0_{\mathcal{R}} \simeq M_d(\mathbb{F}). \qquad \square$$

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature