Assorted algorithms and protocols for secure computation

Assorted Algorithms and Protocols for Secure Computation

Niels de Vreede

This research was supported by the European Union's Seventh Framework Programme for Research and Technological Development project PRACTICE: Privacy-Preserving Computation in the Cloud under grant agreement number 609611 and the European Union's Horizon 2020 Framework Programme for Research and Innovation project PRIViLEDGE: Privacy-Enhancing Cryptography in Distributed Ledgers under grant agreement number 780477.

Printed by Gildeprint Drukkerijen, Enschede, The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-5182-8

Cover: Assorted colorful pebbles

Assorted Algorithms and Protocols for Secure Computation

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de rector magnificus prof. dr. ir. F. P. T. Baaijens,
voor een commissie aangewezen door het College voor Promoties,
in het openbaar te verdedigen op vrijdag 11 december 2020 om 16:00 uur

door

Niels de Vreede

geboren te Helmond

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof. dr. J. J. Lukkien
1e promotor:	dr. ir. L. A. M. Schoenmakers
2e promotor:	dr. B. Škorić
leden:	prof. dr. R. J. F. Cramer (Universiteit Leiden)
	prof. dr. ir. W. P. A. J. Michiels
	prof. dr. P. Schwabe (Radboud Universiteit)
adviseurs:	dr. A. T. Hülsing
	dr. M. Jakobsson (ByteDance)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Contents

Chapter 1

Introduction

In this thesis we present an assortment of algorithms and protocols for use in cryptography and in particular in secure multiparty computation. These results are united by their efficiency under diverse complexity measures. We contribute to a variety of results in the area of secure computation and most of our results pertain to secure multiparty computation (MPC) in various ways.

We make use of existing MPC protocols as foundational building blocks for our own secure protocols, as well as develop novel protocols for specific problems in MPC. Therefore, we will first give a brief overview and introduction of MPC and how it pertains to this thesis in Section 1.1.

This work includes fundamental complexity studies, novel secure protocols, and advanced algorithms to deal with specific functionalities under particular types of complexity measures. In Section 1.2 we will give an overview of the challenges addressed in this thesis and the features typical of our results. We give a more structured and detailed outline of the problems and results discussed in each of the chapters of this thesis in Section 1.3.

1.1 Secure Multiparty Computation

Summarized in one sentence, secure multiparty computation (MPC) is the collective name of techniques and protocols that allow multiple, mutually distrusting parties to compute a function on their joint input data without revealing anything but the result to each other. In his seminal works, Yao formally introduces the notion of secure two-party computation and gives a protocol for general secure two-party computation [Yao82, Yao86]. The work by Goldreich,

Micali, and Wigderson generalizes this to secure computation involving any number of parties [GMW87]. The authors furthermore show that any computable function on inputs held by any number of mutually distrusting parties can be computed as long as at least half of the parties follow the prescribed protocol. These results are in the so-called cryptographic setting, assuming the existence of one-way functions and computational limitations on the adversary.

Ben-Or, Goldwasser and Wigderson give information-theoretically secure protocols, which are secure against adversaries with unlimited computational capabilities [BGW88]. They show that in this setting, security can only be achieved assuming that none of the participants deviate from the prescribed protocol and at most a minority of the participants collaborate to try to obtain private information from the other parties. Alternatively, if participants are assumed to deviate from the protocol in arbitrary ways then protocols for general computation can only be secure if fewer than one third of the participants behaves dishonestly.

The early works on MPC were mostly of theoretical interest. In recent years, however, due to both the increase in computational power, and implementation efforts in which much attention has been given to efficiently adapting theoretical protocols to real-world applications, MPC is more often also considered as practically applicable. Thus, the attention has shifted from purely theoretical problems to issues focusing on practical efficiency of protocols. This trend is reflected in this thesis, in which we focus on efficiency and practical applicability of protocols for solving specific problems within the context of secure computation.

There are many kinds of adversarial behavior that MPC protocols must withstand to be secure. Adversarial behavior is typically modelled as a single adversary which can corrupt parties participating in an MPC protocol, meaning that it can observe and act through corrupted parties. Protocols are proved secure under particular assumptions on the adversary.

Above we have already noted two kinds of such assumption. The first assumption was on the computational capabilities of the adversary. We distinguished the cryptographic setting from the information-theoretic setting. In the cryptographic setting, we assume that one-way functions exist and that the adversary is computationally bounded. In the information-theoretic setting, we permit the adversary unlimited computational capabilities and we rely on the existence of private communication channels between parties.

The second kind of assumption on adversarial behavior noted above is whether the adversary is assumed to follow the prescribed protocol and only attempts to obtain as much information as possible, which is known as the

passive, *semi-honest* or *honest-but-curious* adversary. The other kind of adversary mentioned above is the *active* or *malicious* adversary, which is allowed arbitrary deviations from the protocol.

Another kind of adversarial behavior that must be considered is whether the corruptions are *static*, i.e., fixed at the outset of the protocol, or *adaptive*, meaning that the adversary can decide to corrupt parties as the protocol progresses and depending on messages exchanged so far.

Commonly, the security of a protocol is defined against a particular kind of adversary using the simulation paradigm. Informally, for security proofs in the simulation paradigm, we aim to show that any information obtained by the adversary can be simulated only based on information the adversary is allowed to know. Somewhat more formally, to prove that a protocol securely implements a given functionality, we must give a simulator which interacts with the adversary and the functionality simulating a protocol execution on behalf of the honest, i.e., uncorrupted parties. The simulator must accomplish this without interacting with the honest parties, as the honest parties themselves only interact directly with the functionality. We must then prove that the output of the adversary and the honest parties in this setting cannot be distinguished from the output of the adversary and the honest parties running the actual protocol.

For most of this thesis, we will not give formal security proofs in the simulation paradigm, as instead we assume the existence of secure protocols in the so-called arithmetic black box model [DN03]. In the arithmetic black box model we assume the existence of secure protocols for elementary arithmetic operations. Security of protocols composed out of such elementary operations then follows from the security of the component protocols. As secure protocols for elementary arithmetic operations exist for various adversarial models, our protocols built on top of the arithmetic black box simply 'inherit' the adversarial model, i.e., the security properties, of the underlying arithmetic protocols.

The existence of secure arithmetic black box protocols does not suffice to prove the security of all protocols in this thesis. Where needed, we complement the use of secure elementary arithmetic with specific security proofs of our protocols. For a more in-depth discussion, we refer to Chapter 6, in which we define a specific security model for outsourced computation.

Throughout this thesis we rely on secret sharing to conceal sensitive data from participants to MPC protocols. Secret sharing schemes allow a secret value to be distributed among multiple parties in such a way that any individual party, or any subset of parties deemed unqualified by the distributor cannot learn any information about the secret, but any (necessarily larger) qualified set

can combine their shares to reconstruct the secret value. Various forms of secret sharing exist in both the cryptographic and information-theoretic settings. The protocols that make up our arithmetic black box are specific to a particular secret sharing scheme. In most of this thesis, we also take a generic approach, assuming the existence of secure arithmetic black box protocols, and take the same attitude regarding the underlying secret sharing scheme.

However, for concreteness, we will briefly discuss Shamir's secret sharing scheme [Sha79] and the elementary secure protocols for arithmetic on Shamir secret-shares in the style of [BGW88]. We will first assume that our data can be represented as elements of some finite field, the order of which is at least greater than the number of parties involved in the computation. We will further assume that there exist reasonable bounds on the size of input values such that the characteristic of the field is large enough so that arithmetic results computed over the field correspond to the same arithmetic result when computed over the integers. For simplicity, the order of the field may be considered prime in the following discussion.

An element of the field can be secret-shared among n parties, such that at least $k+1$ of them need to cooperate to reconstruct the secret using Shamir's scheme, by taking a random univariate polynomial of degree (at most) k such that the constant coefficient is equal to the secret. Each party is then given the evaluation of the polynomial at one particular point, where each of these points are mutually distinct. If more than k parties collaborate, they can reconstruct the secret by determining the unique polynomial that interpolates their secret shares. Conversely, k or fewer parties cannot learn any information about the secret, as the number of polynomials interpolating their set of points and any potential secret is identical for each potential secret. This makes Shamir secret sharing information-theoretically secure.

Shamir's secret sharing scheme has the property that it allows for linear manipulation without interaction between the parties. Two secrets can be added by adding the secret sharing polynomials and a secret can be multiplied by a constant by multiplying the secret sharing polynomial by this constant, i.e., the parties can simply apply linear transformations to their shares of the secret.

Multiplication of two secrets is more involved. The parties can multiply their own shares of the secret. This will result in a sharing of a polynomial that represents the product of the secrets, however, the degree of this polynomial is (at most) $2k$ and, furthermore, this polynomial is not uniformly random. The parties can resolve these two problems by simply secret sharing the product of their secret shares. This results in a secret sharing of the product polynomial. Since polynomial interpolation and evaluation (in given points) is a

linear operation, the parties can simply extrapolate the secret product from the shared polynomial. We call this operation *resharing*. This is the secure protocol of [BGW88] in a nutshell.

Because at least $2k + 1$ secret shares are needed to represent the product polynomial, this protocol requires that $n \geq 2k + 1$, therefore, this protocol can only be secure if at least half of the participants are honest.

The multiplication protocol can be generalized, because further linear operations can be carried out before resharing. Therefore, we can actually evaluate arbitrary quadratic formulas with a single resharing. More generally, for $n \geq \ell k + 1$, it is possible to evaluate ℓ-linear forms using a single resharing. We will exploit these properties in Chapters 2 and 5.

Although we have tried to keep our basis for MPC as general as possible, the protocols described in this thesis were designed with BGW-style MPC on Shamir secret shares in mind.

1.2 Overview of the Work

Most results in the thesis relate to MPC in one way or another. We make use of MPC to provide data privacy in our secure outsourced computation protocol, we develop novel and efficient protocols for securely sampling and manipulating permutations and for the secure computation of the Moore–Penrose pseudoinverse in MPC, and we study the multiplicative complexity of polynomial evaluation and its generalization and a novel multi-pivot variant of quicksort, both of which lead to algorithms that are particularly suitable for application in MPC.

The techniques and protocols used in MPC give rise to a unique model of computation in which operations do not only take computational time, but also require communication between protocol participants. The time required for communication typically exceeds the computational time requirements by some orders of magnitude. This requires us to balance the computational time required by a protocol against the communication time to obtain efficient protocols. Under various models of computation for MPC we can carry out somewhat complex operations locally in time negligible to the communication time of one elementary secure multiplication, but we can not perform arbitrary computation without considering the computation time. Furthermore, as communication is bound by both throughput and latency, we must consider both of these aspects. This gives rise to unique complexity measures for determining the efficiency of secure MPC protocols.

In this thesis, we focus on practically applicable secure protocols for a variety of problems. By practically applicable, we mean that we give particular attention to reducing the pertinent complexities of the protocols and algorithms discussed and aim to exploit the parallelism inherent in MPC, as well as leverage the unique fact that some operations can be considered free of cost and some complex operations can be considered equally costly as their elementary counterpart.

For example, in some MPC schemes, the secure evaluation of a dot product between two vectors can be performed using the same amount of communication as an elementary multiplication. We fully exploit this property in our study of the complexity of polynomial evaluation in which it leads to the surprising result that the number of elementary secure operations needed to obliviously evaluate a secret-shared polynomial in a secret-shared point scales with the square root of the degree of the polynomial. If the polynomial is publicly known, the number of elementary secure operation scales as the cube root of the degree instead. This kind of surprising complexity result is typical of MPC. Having secure dot products as elementary secure operation also leads to efficient protocols for linear algebra and we also employ this property in our efficient secure protocol for computing the Moore–Penrose pseudoinverse.

For Trinocchio, our secure outsourced computation protocol, on the other hand, we leverage the relative efficiency of linear transformations in for both privacy preserving computation and verifiable computation, and combine these two models of computation into the Trinocchio protocol.

Where possible, we have taken the effort to generalize our solutions to allow for a configurable trade-off between the number of secure operations and the number of communication rounds of our protocols. In practice this means that throughput and latency bounds can be balanced against each other to obtain the most efficient configuration of our protocols.

We have taken care to describe our results with the right degree of generality so that our results are directly applicable for implementation on top of a range of MPC protocols, while retaining the properties of underlying schemes that make our protocols efficient. Our protocols should also be very straightforward to implement, given a framework for general MPC, providing the arithmetic black box.

Although our results on asymptotically optimal hash chain reversal are not related to MPC, also these share the common theme of efficiency under a particular complexity measure in which not the total computation time must be minimized, but the computation time of many different rounds must be balanced and optimized under stringent space limitations.

1.3 Thesis Outline

We will give a brief outline of the remaining chapters of this thesis.

In Chapter 2 we study the multiplicative complexity of polynomial evaluation and its generalization to the d-linear complexity. From these results, lower bounds are derived for the secure evaluation of symmetric functions using secure arithmetic. The results can also be applied directly to efficiently implement secure functions with small domains, which is relevant, for example, in the construction of the comparison protocol of [ABSdV19] (not discussed in this thesis), for implementation of oblivious arrays or RAM computing in MPC, as discussed in various application examples throughout Chapters 2 and 3, or this can in turn be combined with the results of Chapters 3 and 4 for secure computation on lists of complex objects.

Chapter 3 is dedicated to novel protocols for securely sampling random permutations and obliviously manipulating secret-shared permutations. We describe a representation of secret-shared permutations that is tailored to our efficient protocols. We also consider the issue of securely sampling random permutations with a particular structure. We show that permutations in our secret-shared representation can be efficiently applied to obliviously permute secret-shared lists of arbitrary data. Such protocols are a useful building block for more complex protocols for secure computation and also find application in anonymous communication. As perhaps the most well-known form of random permutation is shuffling a deck of playing cards, we also briefly discuss how the results of this chapter apply to cards games in particular.

In Chapter 4 we construct a multi-pivot variant of quicksort, which is particularly suited for practically efficient application in secure computation when used in conjunction with the aforementioned protocols for oblivious permutation. We consider this result both of independent merit, and to demonstrate the relevance of the protocols from Chapter 3. We combine the results of Chapters 3 and 4 into a secure protocol for obliviously sorting a secret-shared list.

In Chapter 5, we study secure protocols for the computation of the Moore–Penrose pseudoinverse. Existing, more theoretical results, resolve this problem in a constant number of rounds. Our novel protocol sacrifices the constant round property and trades it for greater efficiency in terms of the number of secure operations. The communication complexity of our protocol is both asymptotically smaller than that of existing protocols and is more practically efficient. Furthermore, our protocol is very straightforward to implement. We also identify the conditions under which the pseudoinverse computed over a finite field corresponds to the pseudoinverse over the rational numbers, since

many MPC protocols perform computation over a finite field, but the pseudoinverse over the rational numbers is necessary for most real-world applications.

In Chapter 6 we introduce Trinocchio, our protocol for secure outsourced computation. For outsourced computation we distinguish data privacy and correctness of computation as separate security concerns and define a security model in which correctness of computation is achieved unconditionally and independent of data privacy. This is a useful security notion for settings in which parties that do not actively participate in the computation protocol are interested in the computation results, as in outsourced computation, but, e.g., also for computations in which members of the public have an interest in the result. We construct our protocol using a combination of MPC techniques and results in verifiable computation.

In Chapter 7 we study the problem of asymptotically optimal hash chain reversal (which is not related to MPC). Hash chains find practical application in authentication protocols for computationally limited devices. Algorithms for hash chain reversal are called pebblers. We give a novel framework for describing and analysing pebblers, which allows us to compose smaller pebblers into larger pebblers, i.e., to compose several algorithms for reversing short hash-chains into one algorithm for reversing a longer chain. Our framework is tailored to higher order compositions of pebblers. Existing works achieve $1/2(\log_2 n)^2$ space-time complexity, where n is the length of the hash chain, and show that this is within a factor 2 from the optimum. Using our higher order composition we obtain a fundamental result, demonstrating the existence of pebblers with space-time complexity arbitrarily close to $1/4(\log_2 n)^2$.

As indicated above, we mostly take a generic arithmetic black box approach to MPC, specifying neither the concrete protocols used to carry out arithmetic operations, nor the exact security model. In this way we 'inherit' the security model from the underlying protocols. Only in Chapter 6 do we formally specify the security model in which we prove our construction secure and even here we rely on the proven security of MPC based on Shamir secret sharing ([BGW88]). In Chapters 3, 4, and 5 we build on top of an arithmetic black box for secure computation and argue that any information revealed as part of our protocols does not leak information about the input and output. Chapter 2, instead, is only concerned with multiplicative complexity of polynomial evaluation, and, although motivated by application in MPC, we study this topic independently of MPC. Chapter 7, finally, does not concern MPC.

Chapter 2

Multiplicative Complexity of Polynomial Evaluation

Secure multiparty computation is typically carried out over some algebraic structure which features addition and multiplication as elementary operations. Even more so than for plaintext computation, secure multiplication tends to be far more computationally intensive than secure addition, to the point that we will often treat addition as a free operation. This is because typically addition is a local operation that parties can carry out independently, whereas multiplication requires a communication step. Scalar multiplication, i.e., multiplying a secret value with a known constant can be computed by repeated addition of the secret value to itself and will therefore also be treated as a free operation. In general, we thus treat linear transformations on secret data as free. To accurately represent the communication cost of securely evaluating a particular function using secure operations, we therefore count the number of non-scalar multiplications required to evaluate that function. This number is called the multiplicative complexity of the function.

In this chapter we will study the multiplicative complexity of polynomial evaluation over some finite field $\mathbb{F}$. Although this topic does not deal directly with secure computation, these results are relevant for determining the performance of secure polynomial evaluation using secure computation. Since some secure computation schemes feature elementary operations that are more general or higher order than simple multiplication, e.g., taking dot products of vectors, or, more generally, evaluating k-linear forms for some integer $k > 1$, we will also generalize our complexity results to such operations.

The results of this chapter do not rely on special protocols or properties of a secure computation scheme other than the fact that we treat linear operations as free. As such, we will focus on computational processes which consist entirely of addition and multiplication operations in a fixed structure. Such computational structures can be represented as arithmetic circuits or, equivalently, straight line programs. Because we are interested in the multiplicative complexity, rather than the arithmetic complexity, we will introduce our own definition of straight line programs that incorporates the fact that we consider taking linear combinations a free operation.

2.1 Motivation

In [Sch88] Schnorr proves that the multiplicative complexity of a Boolean circuit evaluating a function $f : \{0,1\}^n \to \{0,1\}$, i.e., the minimum number of logical-and gates required to evaluate f, is at least $\deg f - 1$, where $\deg f$ is the degree of f expressed as a polynomial in n variables. It turns out that this so-called degree lower bound does not hold for *arithmetic* circuits evaluating f. Bounds on the multiplicative complexity of Boolean circuits evaluating *symmetric* Boolean functions were studied by Boyar and Peralta [BP08]. Again, the bounds on the multiplicative complexity do not apply to arithmetic circuits. The original motivation for the work presented in this chapter was to study the multiplicative complexity of arithmetic circuits for evaluating symmetric Boolean functions.

The problem of evaluating symmetric Boolean functions $f : \{0,1\}^n \to \{0,1\}$ by arithmetic circuits over a finite field $\mathbb{F}$ with sufficiently large characteristic $q > n$ can be straightforwardly related to the problem of evaluating polynomials of degree n over $\mathbb{F}$ as follows. The value of a symmetric function is invariant under permutation of the tuple of values it is applied to. A Boolean symmetric function in n variables therefore only depends on the Hamming weight of the variables. The Hamming weight of n variables taking on values in $\{0,1\}$ can be computed by simply summing up all variables, provided that the characteristic, q, of the field in which the computation is carried out is such that $q > n$. The Hamming weight can take on all values between 0 and n, inclusive. Let $\hat{f} : \{0,\ldots,n\} \to \{0,1\}$ be the function such that $\hat{f}(H(\boldsymbol{x})) = f(\boldsymbol{x})$ for all $\boldsymbol{x}$, where H denotes the Hamming weight. Then $\hat{f}$ is a polynomial of degree at most n that can be determined simply through interpolation. The problem of evaluating $f(\boldsymbol{x})$ is now reduced to the problem of evaluating a univariate polynomial of degree at most n, which is the main topic of this chapter.

2.2 Multiplicative Complexity

The original motivation for this work was to investigate the multiplicative complexity of symmetric Boolean functions. Having resolved the question of the multiplicative complexity of univariate polynomial evaluation, our initial results, Theorems 2.6 and 2.7, led us through a literature study to the work of Paterson and Stockmeyer [PS73]. Although the results presented in this section were independently rediscovered, more general results were already reported in [PS73]. This section is included for three reasons. First, in addition to the multiplicative complexity, we also study the *depth complexity*. Second, in Section 2.3, we will generalize the notion of multiplication and the results in this section serve as a basis for the generalized results. Third, the proof of our lower bound of Theorem 2.6 is more straightforward, because it is tailored to the case of finite fields.

Definition 2.1. A multiplicative straight line program with n input values, $x_1, \ldots, x_n$, and m output values $y_1, \ldots, y_m$, is a sequence of $n+k+m$ gates $g_1, \ldots, g_{n+k+m}$, where k is the number of computation gates. Each gate is specified as a number of parameters that describe the computation it represents. For $1 \leq i \leq n$, gate g_i is the ith input gate. For $n < i \leq n+k$, the ith computation gate $g_i = (\boldsymbol{l}_i, \boldsymbol{r}_i)$, where $\boldsymbol{l}_i, \boldsymbol{r}_i \in \mathbb{F}^{i-1}$. For $i > n+k$, the ith output gate $g_i = \boldsymbol{c}_i$, where $\boldsymbol{c}_i \in \mathbb{F}^{n+k+1}$.

The number of parameters specifying gate g_i is 0 if g_i is an input gate, $2i-2$ if it is a computation gate and $n+k+1$ if it is an output gate. The total number of parameters specifying the program is $k(2n+k-1)+m(n+k+1)$.

Definition 2.2. To compute a program is to assign values to its gates. The value of a gate g is denoted $[g]$. The value of an input gate, $1 \leq i \leq n$, is simply its corresponding input value, $[g_i] = x_i$. For computation gates, $n < i \leq n+k$, the value is computed as $[g_i] = (\boldsymbol{l}_i \cdot ([g_1], \ldots, [g_{i-1}]))(\boldsymbol{r}_i \cdot ([g_1], \ldots, [g_{i-1}]))$, i.e., we take a linear combination of gate values computed so far specified by $\boldsymbol{l}_i$ and multiply by the linear combination of gate values specified by $\boldsymbol{r}_i$. Finally, output gates, $n+k < i \leq n+k+m$, are computed as $[g_i] = \boldsymbol{c}_i \cdot (1, [g_1], \ldots, [g_{n+k}])$, i.e., output gates are linear combinations of all input and computation gate values augmented with the constant 1, specified by $\boldsymbol{c}_i$.

Computation gates carry out exactly one multiplication of two linear combinations of all preceding gate values. Output gates, instead, perform a linear combination on the preceding gate values augmented with the constant 1,

but do not perform any non-scalar multiplication. Having only output gates perform such an augmented linear combination while the computation gates are restricted to ordinary linear combinations is not a limitation, as any augmented linear combination that could be carried out in computation gates can be 'pushed forward' to subsequent gates until only the output gates perform augmented linear combinations.

To clarify, suppose that we have a multiplicative straight line program as defined above, except that the computation gates are specified as augmented linear combinations. Then the ith computation gate is specified by two additional parameters, λ_i and ρ_i, compared to computation gates specified in terms of ordinary linear combinations. The value of the ith gate is then computed as

$$\begin{aligned}[g_i] &= (\lambda_i + \boldsymbol{l}_i \cdot ([g_1], \ldots, [g_{i-1}]))(\rho_i + \boldsymbol{r}_i \cdot ([g_1], \ldots, [g_{i-1}])) \\ &= [g_i'] + (\lambda_i \boldsymbol{r}_i + \lambda_i \rho_i + \rho_i \boldsymbol{l}_i) \cdot ([g_1], \ldots, [g_{i-1}]),\end{aligned}$$

where g_i' is the computation gate taking ordinary linear combinations specified by $(\boldsymbol{l}_i, \boldsymbol{r}_i)$. We can then replace g_i by g_i' and all subsequent augmented linear combinations $\boldsymbol{d}_j$ by $\boldsymbol{d}_j' = \boldsymbol{d}_j + ((\lambda_i \rho_i) \| (\lambda_i \boldsymbol{r}_i + \rho_i \boldsymbol{l}_i))$, for $j > i$, whether $\boldsymbol{d}_j$ is one of the augmented linear combinations taken by a computation gate, or the augmented linear combination taken by an output gate. Here $\|$ denotes concatenation of two vectors. This replacements yields a multiplicative straight line program computing the same function in which the ith gate is specified in terms of ordinary linear combinations. We can apply this procedure iteratively over all computation gates in order until all augmented linear combinations in computation gates have been eliminated and replaced by ordinary linear combinations.

In the following example we will show the construction of a program that evaluates a fixed polynomial.

Example 2.3. Consider the univariate polynomial in x of degree N specified by $\sum_{n=0}^{N} a_n x^n$. We will construct a straight line program that takes as input a point x and produces as output the evaluation of the polynomial at x.

The first gate will be the input gate: $[g_1] = x$. Let $\boldsymbol{e}_k^n$ denote the vector of length n containing a 1 at the kth position and 0 at all other positions. For $2 \leq n \leq N$, the nth gate is specified by $g_n = (\boldsymbol{e}_1^{n-1}, \boldsymbol{e}_{n-1}^{n-1})$. Then the values of the gates are given by $[g_n] = [g_1][g_{n-1}] = x^n$. The output gate is specified as $g_{N+1} = (a_0, \ldots, a_N)$ and has value $[g_{N+1}] = a_0 + \sum_{n=1}^{N} a_n [g_n] = \sum_{n=0}^{N} a_n x^n$, as required.

Because the specification of gates in terms of the coefficient vectors (the $\boldsymbol{l}$, $\boldsymbol{r}$ and $\boldsymbol{c}$ vectors) is rather cumbersome, we will often specify gates directly in terms of their values, as shown in the next example. For this example, we will show the construction of a program that takes both a polynomial and a point as input and evaluates the polynomial at that point.

Example 2.4 (Horner's rule)**.** Let the polynomial be specified by $\sum_{n=0}^{N} a_n x^n$ as before, however, now the coefficients are treated as additional input to the program. The first $N+2$ gates are the input gates: $[g_1] = x$ and $[g_n] = a_{N+2-n}$ for $2 \leq n \leq N+2$, i.e., the coefficients are input in reverse order. Define $[g_{N+3}] = [g_1][g_2] = a_N x$ and $[g_{N+2+n}] = [g_1]([g_{N+1+n}] + [g_{n+1}]) = \sum_{k=1}^{n} a_{N-n+k} x^k$ for $2 \leq n \leq N$. Finally, the output gate is $[g_{2N+3}] = [g_{2N+2}] + [g_{N+2}] = \sum_{n=0}^{N} a_n x^n$, as required.

This scheme is known as Horner's rule and uses exactly N additions and N multiplications for the evaluation of a degree N polynomial. It was proved optimal by Pan [Pan66] in the sense that any scheme for the evaluation of a degree N polynomial without initial conditioning of the coefficients requires at least N additions and N multiplications. The phrase "without initial conditioning of the coefficients" concretely means that any constants appearing in the scheme do not depend on the values of the coefficients.

The multiplicative complexity of a straight line program is simply the number of computation gates k. For application in secure computation, we are also interested in the round complexity, i.e., the number of communication rounds required. For circuit-based approaches, the round complexity scales linearly with the depth of the circuit. The depth of our straight line programs is defined as follows. Let g_i and $g_j = (\boldsymbol{l}_j, \boldsymbol{r}_j)$ be computation gates with $i < j$. We say that a gate g_j depends on gate g_i if $l_{j,i} \neq 0$ or $r_{j,i} \neq 0$. The depth of a computation gate is defined as one more than the maximum of the depths of all gates it depends on (or zero if it does not depend on any gate). The input gates have depth zero by definition and the depth of an output gate is the maximum of the depths of the gates it depends on. The depth of a program is the maximum of the depths of its output gates.

Example 2.5. The multiplicative complexity of the program of Example 2.3 is $N-1$, since there are simply that many computation gates. The depth of the program is also $N-1$, as the depth of the input gate g_1 is zero by definition and for $2 \leq n \leq N$, the depth of g_n is one greater than the depth of g_{n-1} because g_n depends only on g_{n-1} and g_1. Because the output gate depends on

all computation gates, the depth of the program is the maximum of the depths of the computation gates.

Example 2.3 shows how to construct a program for the evaluation for a fixed polynomial. This construction achieves linear multiplicative and depth complexities. In the remainder of this section, we shall show that this is not optimal. We shall first prove a lower bound on the multiplicative complexity.

Theorem 2.6. *For each N, there exists a polynomial of degree at most N such that a straight line program evaluating that polynomial requires at least $\sqrt{N}-1$ multiplications.*

Proof. Let q be the order of $\mathbb{F}$. Then there are exactly q^{N+1} polynomials of degree at most N. A program to evaluate a univariate polynomial has only one input and one output gate. Any such program with k computation gates has $k(k+1)+k+2=(k+1)^2+1$ parameters. Because each parameter is an element of $\mathbb{F}$, there are exactly $q^{(k+1)^2+1}$ distinct programs with one input gate, k computation gates and one output gate. Furthermore, because we can specify programs in which the output gate is independent of some computation gates, the programs with k computation gates also cover all programs with fewer gates. Therefore, if $N>(k+1)^2$ then there must exist a polynomial of degree at most N that cannot be evaluated by a circuit with at most k computation gates. This means there exists a polynomial of degree at most N, which requires at least $\sqrt{N}-1$ multiplications to evaluate. □

Note that two programs with distinct parameters are not necessarily functionally distinct.

Theorem 2.7. *For each polynomial $p(x)$ of degree $N\geq 1$ over $\mathbb{F}$, there exists a straight line program which evaluates $p(x)$ using $\lceil 2\sqrt{N}\rceil-2$ multiplications.*

Proof. We first rewrite the polynomial

$$p(x)=\sum_{i=0}^{N}a_ix^i=\sum_{i=0}^{\lceil\frac{N}{\ell}\rceil-1}\left(\sum_{j=1}^{\min(\ell,N-\ell i)}a_{\ell i+j}x^j\right)\left(x^\ell\right)^i+a_0, \tag{2.1}$$

for some $\ell\geq 1$, i.e., we consider a polynomial of degree N in x as a polynomial of degree $\lceil\frac{N}{\ell}\rceil$ in x^ℓ. The coefficients of this latter polynomial are evaluations of polynomials of degree at most ℓ at x. To efficiently evaluate $p(x)$, we therefore first compute all powers of x up to the ℓth, i.e., we compute $x,x^2,x^3,\ldots,x^\ell$.

If we compute x^{i+1} as $x \cdot x^i$, then each of these, except the first, costs a single multiplication, for a total of $\ell - 1$ multiplications. We can then evaluate the coefficients of the right hand polynomial of (2.1) for free. It is well known that the polynomial can then be evaluated by, for example, Horner's rule using $\lceil \frac{N}{\ell} \rceil - 1$ multiplications. If we now optimize for ℓ, we find that using $\ell = \lceil \sqrt{N} \rceil$ leads to a total of exactly $\lceil 2\sqrt{N} \rceil - 2$ multiplications. □

Although the number of multiplications required in Theorem 2.7 is within a factor 2 from the lower bound given in Theorem 2.6, the depth of the program is linear in the degree of the polynomial. For Theorem 2.9 we will refine the construction of Theorem 2.7 to a construction which only has logarithmic (in the degree of the polynomial) overhead compared to Theorem 2.7, but achieves logarithmic depth. This refinement is also based on (2.1), however, because this approach reduces the evaluation of a fixed polynomial to the evaluation of a lower degree polynomial for which the coefficients are no longer fixed, we will first consider the problem of evaluating an unknown polynomial.

Lemma 2.8. *For $N > 2$, there exists a straight line program that given the coefficients of a degree N polynomial p and a point x evaluates $p(x)$ over $\mathbb{F}$. This program uses $N + \lceil \log_2(N+1) \rceil - 1$ multiplications and has depth $\lceil \log_2(N+1) \rceil$.*

Proof. Let h_0 be the input gate x. For $1 \leq i < \lceil \log_2(N+1) \rceil$, let $h_i = h_{i-1}^2 = x^{2^i}$. Then gate h_i has depth i, since h_0 has depth 0 and the depth of h_i is one greater than the depth of h_{i-1}.

For $0 \leq j \leq N$, let $f_{0,j} = a_j$. For $1 \leq i \leq \lceil \log_2(N+1) \rceil$ and $0 \leq j \leq N$, where j is an integer multiple of 2^i, let

$$f_{i,j} = h_{i-1} \sum_{k=0}^{\min(i-1, \lceil \log_2(N-j-2^{i-1}+1) \rceil)} f_{k,j+2^{i-1}}.$$

A gate $f_{i,j}$ has depth i, since each gate $f_{0,j}$ has depth 0 and the depth of $f_{i,j}$ is one greater than the maximum of the depths of h_{i-1} and gates $f_{i',j+2^{i'}}$, where $i' < i$.

Finally, let $y = \sum_{k=0}^{\lceil \log_2(N+1) \rceil} f_{k,0}$.

The sequence of gates, as specified above, forms a valid straight line program. We will now show that this program computes $\sum_{i=0}^{N} a_i x^i$. We first prove that

$$f_{i,j} = \sum_{\ell=\lfloor 2^{i-1} \rfloor}^{\min(2^i-1, N-j)} a_{\ell+j} x^\ell$$

for all appropriate i and j as defined above. In the base case, $i = 0$, we have that $f_{0,j} = \sum_{\ell=0}^{0} a_{\ell+j} x^{\ell} = a_j$. Using induction we have that

$$\begin{aligned} f_{i+1,j} &= h_i \sum_{k=0}^{\min(i,\lceil \log_2(N-j-2^i+1)\rceil)} f_{k,j+2^i} \\ &= x^{2^i} \sum_{k=0}^{\min(i,\lceil \log_2(N-j-2^i+1)\rceil)} \sum_{\ell=\lfloor 2^{k-1}\rfloor}^{\min(2^k-1,N-j-2^i)} a_{\ell+j+2^i} x^{\ell} \\ &= \sum_{\ell=0}^{\min(2^i-1,N-j-2^i)} a_{\ell+j+2^i} x^{\ell+2^i} \\ &= \sum_{\ell=\lfloor 2^i\rfloor}^{\min(2^{i+1}-1,N-j)} a_{\ell+j} x^{\ell}. \end{aligned}$$

We then substitute this into

$$\begin{aligned} y &= \sum_{k=0}^{\lceil \log_2(N+1)\rceil} f_{k,0} \\ &= \sum_{k=0}^{\lceil \log_2(N+1)\rceil} \sum_{\ell=\lfloor 2^k-1\rfloor}^{\min(2^k-1,N)} a_{\ell} x^{\ell} \\ &= \sum_{\ell=0}^{N} a_{\ell} x^{\ell}. \end{aligned}$$

Computing all h gates requires $\lceil \log_2(N+1)\rceil - 1$ multiplications. The number of multiplications required to compute the gates $f_{i,j}$ for all appropriate i and j is equal to $\sum_{i=1}^{\lceil \log_2(N+1)\rceil} \lfloor \frac{N+2^{i-1}}{2^i} \rfloor = N$. Thus, the multiplicative complexity of this program is $N + \lceil \log_2(N+1)\rceil - 1$. □

Based on Lemma 2.8 we construct a program for the evaluation of a fixed polynomial that has only logarithmic (in the degree of the polynomial) depth. The overhead in multiplicative complexity, compared to Theorem 2.7 is only logarithmic in the degree of the polynomial.

Theorem 2.9. *For each polynomial $p(x)$ of degree N over $\mathbb{F}$, there exists a straight line program which evaluates $p(x)$ using at most $2\sqrt{N} + \frac{1}{2}\log_2 N$ multiplications and has depth at most $\log_2 N + 2$.*

Proof. We rewrite $p(x)$ as in (2.1). We again evaluate the lowest ℓ powers of x, but now explicitly at depth $\lceil \log_2 \ell \rceil$. Concretely, once we have evaluated the powers of x up to x^{2^i} at depth i, we can compute $x^{2^i+1}, \ldots, x^{2^{i+1}}$ at depth $i+1$ by multiplying x^{2^i} by x^j for $1 \leq j \leq \min(2^i, \ell)$. This requires $\ell - 1$ multiplications and has depth $\lceil \log_2 \ell \rceil$.

We then evaluate each $c_i = \sum_{j=1}^{\min(\ell, N-\ell i)} a_{\ell i+j} x^j$ for $0 \leq i \leq \lceil N/\ell \rceil - 1$ and we write $p(x) = \sum_{i=0}^{\lfloor N/\ell \rfloor - 1} c_i u^i + a_0$, where $u = x^\ell$, i.e., we write $p(x)$ as an unknown polynomial of degree $\lceil N/\ell \rceil - 1$, where each of the coefficients is the evaluation of a low degree polynomial $c_i = \sum_{j=1}^{\min(\ell, N-i\ell)} a_{\ell i+j} x^j$. Because the coefficients c_i are unknown, we use the technique of Lemma 2.8 to evaluate this polynomial, which requires $\lceil N/\ell \rceil + \lceil \log_2 \lceil N/\ell \rceil \rceil - 2$ multiplications at a depth of $\lceil \log_2 \lceil N/\ell \rceil \rceil$.

The total number of multiplications is then $\ell + \lceil N/\ell \rceil + \lceil \log_2 \lceil N/\ell \rceil \rceil - 3$. We find the minimum at approximately, but no more than $2\sqrt{N} + \frac{1}{2} \log_2 N$ multiplications, where ℓ is close to $\sqrt{N}$. The total depth is then at most $\log_2 N + 2$, due to rounding. □

The multiplicative complexity given by both Theorem 2.7 and Theorem 2.9 is dominated by $2\sqrt{N}$, where N is the degree of the polynomial. In both cases we construct the programs using the values of the coefficients without any processing. By preprocessing the coefficients, it is possible to reduce the number of multiplications needed by a factor $\sqrt{2}$. However, in the following, we will study the bounds on the evaluation of polynomials for programs in a more general model of computation, for which we will focus on the case of *unknown* polynomials, i.e., the case where the polynomial itself is also considered part of the input, just as in Lemma 2.8 and Example 2.4 of Horner's rule. The results of this section serve as introduction to the following sections and, because we will focus on the evaluation of unknown polynomials, we have disregarded programs which require preprocessing of the coefficients. For details on preprocessing we refer to [PS73].

2.3 Degree d Complexity

Some computation schemes feature more general primitive operations than elementary multiplication. In particular, [BGW88]-style secure computation based on Shamir secret sharing, secure computation based on replicated secret sharing or generalizations of techniques using precomputed Beaver triples allow for the evaluation of any multivariate polynomial of degree at most 2 using

a single secure operation. Depending on the configuration, it may even be possible to evaluate polynomials of higher degree in one operation. In this section, we will take the evaluation of a degree d multivariate polynomial as our primitive operation, i.e., our gates, and generalize our results on the complexity of polynomial evaluation to programs consisting of such gates.

Definition 2.10. A degree d straight line program over $\mathbb{F}$ is a multiplicative straight line program in which each computation gate represents an arbitrary degree d form in terms of all preceding gates and the constant 1. Each computation gate g_i for $n < i \leq n + k$ is specified by $\binom{d+i-1}{d}$ elements from $\mathbb{F}$, which represent the coefficients of each term in the degree d form. The depth of a degree d straight line program is defined analogously to the depth of a multiplicative straight line program. Similarly, the output gates are specified as a linear combination of all input and computation gates and the constant 1. A degree d straight line program with n input gates, m output gates and k computation gates is specified by exactly $\binom{d+n+k}{d+1} - \binom{d+n}{d+1} + m(n + k + 1)$ parameters.

Note that by including the constant 1 as input to each gate, we can effectively include terms of any degree lower than d in the form.

Example 2.11 (Dot product)**.** For two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of length n, the dot product is defined as $\sum_{i=1}^{n} a_i b_i$. This is clearly a quadratic form and can therefore be evaluated using a single degree 2 gate.

Theorem 2.12. *For each $N \geq d$, there exists a degree d straight line program that takes as input the coefficients of a degree N polynomial p and a point x and evaluates $p(x)$ over $\mathbb{F}$. This program uses fewer than $d \sum_{k=1}^{\lceil \log_d \log_d N \rceil + 1} \sqrt[d^k]{N}$ degree d gates. The depth of this program is $\log_d n + O(\log_d \log_d N)$.*

Proof. Let $C(N)$ be the minimal number of degree d gates required to evaluate a degree N polynomial and let $B(N) = d \sum_{k=1}^{\lceil \log_d \log_d N \rceil + 1} \sqrt[d^k]{N}$. Note that both C and B are monotonic. We prove our claim that $C(N) < B(N)$ for all $N \geq d$ by induction. Clearly $C(2) \leq 2 < 2\sqrt{2} = B(2)$ and $C(3) \leq 3 < 2\sqrt{2} + 2\sqrt[4]{3} = B(3)$.

For $N \geq 4$, we write the polynomial as in (2.1) and evaluate it as follows. Let $L_j = \prod_{i=1}^{j-1} \ell_i$ for some ℓ_i, where $1 \leq i \leq d$. First, we evaluate the powers x^{iL_j} for $1 \leq j < d$ and $0 \leq i \leq \ell_j$. This requires exactly $L_d - d + 1$ degree d

gates. Then, for each i, such that $0 \le i \le \lceil \frac{N}{L_d} \rceil - 1$, we evaluate

$$b_i = \sum_{\boldsymbol{k}} a_{iL_d + \sum_{j=1}^{d-1} k_j L_j} \prod_{i=1}^{d-1} x^{k_j L_j},$$

where $\boldsymbol{k}$ ranges over all length $d-1$ vectors of integers such that $1 \le k_j \le \ell_j$ for $1 \le j \le d-1$ and $\sum_{j=1}^{d-1} k_j L_j \le N - iL_d$. This requires $\lceil \frac{N}{L_d} \rceil$ degree d gates. Finally, we treat these evaluations as the coefficients of a polynomial of degree $\lceil \frac{N}{L_d} \rceil - 1$ in x^{L_d}:

$$p(x) = \sum_{i=0}^{N} a_i x^i = a_0 + \sum_{i=0}^{\lceil \frac{N}{L_d} \rceil - 1} b_i (x^{L_d})^i.$$

To evaluate the right hand polynomial requires $C(\lceil \frac{N}{\ell} \rceil - 1)$ degree d gates.

Now, assume that $C(m) < B(m)$ holds for all $2 \le m < k$. Then

$$\begin{aligned} C(N) &= \min_{\boldsymbol{\ell}} \ell - 1 + \lceil \frac{N}{L_d} \rceil + C(\lceil \frac{N}{L_d} \rceil - 1) \\ &< \min_{\boldsymbol{\ell}} \ell + \frac{N}{L_d} + B(\frac{N}{L_d}). \end{aligned}$$

If we take $\ell_i = \sqrt[d]{N}$ for all i, then $\frac{N}{L_d} = \sqrt[d]{N}$, and

$$\begin{aligned} C(N) &\le d\sqrt[d]{N} + B(\sqrt[d]{N}) \\ &< d\sqrt[d]{N} + d \sum_{k=1}^{\lceil \log_d \log_d \sqrt[d]{N} \rceil + 1} \sqrt[d^k]{\sqrt[d]{N}} \\ &= d \sum_{k=1}^{\lceil \log_d \log_d N \rceil + 1} \sqrt[d^k]{N}. \end{aligned}$$

In the final step we use the fact that $\log_d \log_d \sqrt[d]{N} = \log_d \log_d N - 1$.

Note that the powers of x are evaluated at depth $\lceil \log_d L_d \rceil = \lceil \frac{d-1}{d} \log_d N \rceil$ or lower. By the induction hypothesis, the depth of the program is equal to $\lceil \frac{d-1}{d} \log_d N \rceil + 1 + \log_d \sqrt[d]{N} + O(\log_d \log_d \sqrt[d]{N}) = \log_d n + O(\log_d \log_d N)$. □

The use of degree 2 gates is of special interest for application in multiparty computation. We mention the case $d = 2$ separately for emphasis.

Corollary 2.13. *For each $N \geq 2$, there exists a degree 2 straight line program that takes as input the coefficients of a degree N polynomial p and a point x and evaluates $p(x)$ over $\mathbb{F}$. This program uses fewer than $2\sum_{k=1}^{\lceil \log_2 \log_2 N \rceil + 1} \sqrt[2^k]{N}$ degree 2 gates. The depth of this program is $\log_2 N + O(\log \log N)$.*

Example 2.14. A useful application for efficient polynomial evaluation is array indexing. Suppose we are given N values $v_1, \ldots, v_N$ and an index $1 \leq n \leq N$ and we wish to determine v_n. This problem can be solved by computing the polynomial interpolating $\boldsymbol{v}$ and evaluating it at n. Since polynomial interpolation (on a fixed set of points) is a linear operation, which we consider free, the only cost is the evaluation of a polynomial of degree $N-1$. In Chapter 3 we will extend this example to oblivious arrays that can also be modified, i.e., written to, at an oblivious index.

For completeness, we will give the generalizations of Theorems 2.6 and 2.9 to degree d gates.

Theorem 2.15. *For each N, there exists a polynomial of degree at most N such that a straight line program evaluating that polynomial requires $O((d+1)\sqrt[d+1]{N})$ degree d gates.*

Proof. The proof is similar to that of Theorem 2.6. Let q be the order of $\mathbb{F}$. There are exactly q^{N+1} polynomials of degree at most N over $\mathbb{F}$. A degree d straight line program evaluating a univariate polynomial has 1 input, 1 output and k computation gates and is therefore specified by exactly $\binom{d+1+k}{d+1} - \binom{d+1}{d+1} + k + 2$ parameters from $\mathbb{F}$. These straight line programs are not all functionally distinct. For any program which computes only a single value, we can actually eliminate the output gate parameters entirely by absorbing the parameters for the linear combination taken in the output gate into the final computation gate and simply set the output value to the value of that gate. Such a program would compute the same function as the original program and consist of the same number of computation gates, but could be specified using only $\binom{d+1+k}{d+1} - 1$ parameters. The depth of such a program could be 1 greater than the depth of the original program, but that is of no concern for this theorem. We can therefore safely take $\binom{d+1+k}{d+1} - 1$ as the number of parameters.

If the number of distinct straight line programs consisting of k degree d computation gates is smaller than the number of polynomials of degree at most N, i.e., if $q^{\binom{d+1+k}{d+1}-1} < q^{N+1}$, then there exists a polynomial of degree at most N which cannot be evaluated by such a program. Using the well-known fact that $\binom{n}{k} < \left(\frac{en}{k}\right)^k$, where e is Euler's constant we can rewrite this

condition as $\binom{d+1+k}{d+1} < \left(\frac{e(d+1+k)}{d+1}\right)^{d+1} < N+2$ or, solving for k, as $k < \frac{d+1}{e}\sqrt[d+1]{N+2} - d - 1$. We can therefore conclude that there exists a polynomial of degree at most N which requires a degree d straight line program with $O((d+1)\sqrt[d+1]{N})$ computation gates to evaluate. □

Theorem 2.16. *For each polynomial $p(x)$ of degree N over $\mathbb{F}$, there exists a straight line program which evaluates $p(x)$ using $(d+1)\sqrt[d+1]{N} + O(d\sqrt[d(d+1)]{N})$ degree d gates and has depth $\log_d N + O(\log_d \log_d N)$.*

Proof. We rewrite the polynomial as in (2.1). We then compute the necessary powers of x that will allow us to evaluate polynomials of degree ℓ with only a single degree d gate. The powers $x^{i\lceil \ell^{j/d} \rceil}$ for $1 \le i \le \lceil \ell^{1/d} \rceil$ and $0 \le j \le d-1$ are sufficient for this purpose and can be computed using $d(\lceil \ell^{1/d} \rceil - 1) < d\sqrt[d]{\ell}$ degree d gates at depth at most $\lceil \log_d \ell \rceil$. We then evaluate the $\lceil \frac{N}{\ell} \rceil$ degree ℓ polynomials of (2.1) with one degree d gate each, which increases the depth by one.

We can evaluate the full polynomial with $O(d\sqrt[d]{\lceil \frac{N}{\ell} \rceil})$ degree d gates by applying Theorem 2.12. This adds $\log_d \lceil \frac{N}{\ell} \rceil + O(\log_d \log_d \lceil \frac{N}{\ell} \rceil)$ to the depth.

Using $\ell = N^{\frac{d}{d+1}}$, the degree d complexity is $(d+1)\sqrt[d+1]{N} + O(d\sqrt[d(d+1)]{N})$ and the depth $\log_d N + O(\log_d \log_d N)$. □

Chapter 3

Secure Shuffle

3.1 Introduction

In this chapter we will study the problem of securely shuffling a list of secret-shared values. Shuffling a list of secret-shared data obliviously is a useful primitive for various applications. One particular application is that after obliviously shuffling a list, the list can be sorted using a general sorting algorithm, rather than an oblivious one, without revealing the order of the original. Such a protocol for securely sorting lists is the subject of Chapter 4. Section 3.9 focuses on another application, namely securely shuffling playing cards for online cards games and deals with the issues of that application.

We break down the task of obliviously shuffling a given secret-shared list into two component tasks. The first is to obliviously sample a uniformly random secret-shared permutation of a publicly given size; the second is to obliviously rearrange the elements in the secret-shared input list according to a given secret-shared permutation. We will present secure protocols for each subtask, as well as a representation of secret-shared permutations which allows for efficient application and composition. For clarity, we distinguish the full *shuffle* protocol from the *rearrangement* protocol where needed.

Dividing the task of shuffling a list in this way permits us to shuffle composite objects by obliviously sampling a single permutation and applying it to multiple lists independently. Furthermore, sampling the random permutation depends only on the size of the input list and can therefore, in certain applications, be considered data-independent and suitable for precomputation.

The multiplicative complexity of our protocol for shuffling a list of length N

is $O(N \log N)$ and the round complexity is $O(\log N)$. These complexities arise from using a binary tree with N leaves as part of our data structure representing permutations. The internal nodes of the tree correspond to multiplications. The technique from [BB89] for constant round unbounded fan-in multiplication enables us to use M-ary trees, rather than binary trees. The shuffling protocol then requires $O(N \log_M N)$ fan-in M multiplications and $O(\log_M N)$ rounds. Since the complexity of performing M multiplications using the unbounded fan-in multiplication protocol is linear in M, when expressed in terms of primitive, fan-in 2 multiplications, we state the multiplicative complexity of the shuffling protocol as $O(MN \log_M N)$ in terms of ordinary multiplications.

The use of unbounded fan-in multiplication allows for a trade-off between multiplicative complexity and round complexity. Letting $M = N^{1/c}$ for some parameter c will lead to a protocol whose round complexity, $O(c)$, is independent of N, with multiplicative complexity $O(cN^{1+1/c})$. In any case, the constants in the big-O notation are fairly small and practical. However, taking values for M other than 2 may be mostly of theoretical importance, due to the overhead of unbounded fan-in multiplication.

The permutation protocol is a data-independent protocol that can be used to obliviously sample a random permutation of given length N. The protocol is based on the observation that a product is invariant under permutation of its factors. By obliviously sampling random irreducible polynomials and revealing the product, we can retrieve the set of original polynomials using polynomial factorization, but not the order they were sampled in. The correspondence between the secret-shared originals and the publicly known factors establishes a permutation. Concretely, we take monic polynomials of degree 1 for our random irreducible polynomials and perform our computations directly on the roots of the product polynomial, rather than its factors.

The rearrangement protocol applies a random permutation obtained from the permutation protocol to a list of secret-shared values. To do so, first each of the publicly known roots from the permutation protocol is assigned a secret value of the list to be reordered. This gives a set of points for which we then obtain the interpolating polynomial. As it is derived from secret-shared data, this polynomial exists in secret-shared form. To create an oblivious rearrangement of the input data we then evaluate the interpolating polynomial in each of the secret roots. We must use efficient multi-point polynomial evaluation to keep the complexity in check.

Related Work

There is a large body of literature on secret shuffles for the purpose of mix-nets, which were first introduced by Chaum [Cha81]. In this context a mixer is given a list of encryptions and tasked with producing a list of encryptions that would decrypt to the same plaintexts as the original list, but in random order. Such constructions are useful, for example, in secure voting and anonymous communication applications. Some key works in this area are [SK95, Abe99, FS01, Nef01]

Mixing protocols are typically constructed from encryption schemes that allow for random re-encryption and a zero-knowledge proof of correct permutation and re-encryption. As such, these works are not concerned with the issue of obliviously sampling and representing a secret-shared permutation, which is the topic of this chapter. Our topic has received far less attention in the literature. It should be noted, however, that the key observation behind our oblivious protocol—i.e., that multiplication of polynomials preserves its factors, but due to the commutativity of multiplication, it is invariant under permutation of the order of the multiplicands—is the same observation which enables zero-knowledge proofs of correct permutation first described in [Nef01].

Laur et al. [LWZ11] give three methods for oblivious shuffling. In the first, a sufficient number of parties secret shares a permutation matrix and the random permutation is computed as the product of these matrices. An intuitive argument against the practical use of this method is that manipulation of permutation matrices necessarily involve some steps that exhibit quadratic communication complexity (in the size of the permutation). By not representing permutations as matrices, our methods do not incur a quadratic lower bound on the communication complexity.

The second method consists of the parties jointly creating a list of secret-shared random values. The permutation which sorts the list is then obliviously computed and is uniformly random. This method features better asymptotic complexity than the first method, however, this requires an oblivious sorting protocol. As we shall see in the following chapter, we can obtain more efficient oblivious sorting by first shuffling the input list. Such sorting method can not be used to construct the shuffle itself, however.

Their third method is actually analogous to mixing in an MPC setting. The communication complexity of this method is linear in the size of the permutation, but suffers from combinatorial explosion in the number of parties, or, more accurately, is linear in the number of maximal unqualified sets implied by the access structure. Furthermore, this method leaves open the issue of effi-

ciently representing the permutation for use in further operations without, for example, making use of potentially inefficient permutation matrices.

3.2 Notation

Throughout this chapter, the symbol N will denote the size of the permutation, M will denote the fan-in of multiplication. For convenience, let $k = \lceil \log_M N \rceil$.

Tree data structures will be denoted using capitals, e.g., T. All trees will be balanced M-ary trees with N leaves. We refer to nodes (or leaves) of a tree T explicitly as $T_{i,j}$, where $0 \leq i \leq k$ and $0 \leq j < N_i = \lceil \frac{N}{M^i} \rceil$. Here, the first index refers to the height of the node and the second to the position of the node in relation to all other nodes at that height. N_i is the number of nodes at height i. Leaves have height 0 by definition and we refer to the root of a tree T as $T_{k,0}$. Trees are not necessarily full trees and, as such, the number of children is not equal for all nodes at the same height, however, because trees are balanced, the difference in number of children between any two nodes at the same height is at most one. The number of children of node $T_{i,j}$ in tree T is equal to

$$M_{i,j} = \left\lceil \frac{N_{i-1} - j}{N_i} \right\rceil . \tag{3.1}$$

For $i > 0$, the children of node $T_{i,j}$ are $T_{i-1,j+\ell N_i}$ for all $0 \leq \ell < M_{i,j}$. The parent of a node $T_{i,j}$ is $T_{i+1,j \bmod N_{i+1}}$. An example of a tree is given in Figure 3.1. The exact representation of the tree is not necessary to understand our algorithms, but is chosen so that it is possible to refer to the parent or children of a given node using only arithmetic on the indices, which aids in the analysis of our protocols.

For a polynomial p, we denote by $p[i]$ the ith coefficient of p, where $p[i] = 0$ for any i greater than the degree of p. We may therefore write $p(x)$ as $\sum_i p[i]x^i$. The derivative of p is denoted p' and $p'[i-1] = ip[i]$ for any $i > 0$.

We will denote the application of a permutation π by multiplication from the left, for example, $\pi \boldsymbol{v}$ is the vector $\boldsymbol{v}$ permuted by π. We will use the same notation for permuting the rows of a matrix, or for the composition of permutations as well. For simplicity, the permutation can be thought of as a permutation matrix in this notation. However, we aim to treat permutations as abstract objects in this chapter and do not explicitly rely on the representation of permutations as matrices. For secret-shared permutations, we will explicitly specify the representation and rather than denote application of a secret-shared

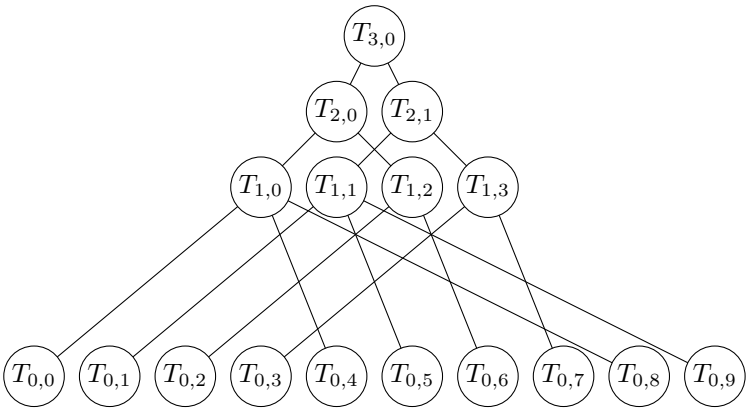


Figure 3.1: Structure of a ternary tree with 10 leaves. The leaves and nodes are presented in the same order as in the data structure. Note how the children of node $T_{i,j}$ are interleaved at distance N_i apart and that at each level of the three the number of children per node differs by at most one.

permutation as a multiplication, we will explicitly refer to the secure protocol for rearrangement or composition.

For any object x, be it a field element, a vector of field elements, a polynomial over the field, or a tree whose nodes contain other objects, we denote the secret sharing of that object as $[\![x]\!]$. For composite objects, we always consider the structure of the object publicly known and secret share the coefficients representing the object over the field. For example, we consider the size of a secret-shared vector publicly known and, for a vector of size N, the secret sharing simply consists of the N secret-shared coefficients that make up the vector.

3.3 Common Polynomial Operations

Before we delve into the details of the permutation and rearrangement protocols, we will first describe some auxiliary protocols for common operations on secret-shared polynomials. Secret-shared polynomials are represented by vectors of secret-shared coefficients. We consider the degree of a polynomial to be public information and therefore do not have to hide the length of the coefficient vector. In many cases, we will work with monic polynomials, i.e., we know

that the leading coefficient is 1, however in some cases the leading coefficient is secret. In that case it is possible that the polynomial is actually of a lower degree than is publicly known and the value of the supposed leading coefficient is 0. As we shall see, this does not cause any problems for our purposes and we will simply compute with a publicly known upper bound on the degree, without actually knowing whether this bound is exact.

The protocols in this section are based on well-known algorithms, see for example [vzGG03, Chapter 10]. Several of these well-known techniques for polynomial manipulation were first adapted to the shared coefficient setting by Mohassel and Franklin [MF06b]. We will briefly review their protocols for polynomial multiplication and polynomial division with remainder. We will also adapt algorithms for efficient polynomial interpolation and, its counterpart, multipoint evaluation to secure protocols in the shared coefficient setting.

Polynomial Multiplication

Multiplication of secret-shared polynomials is remarkably straightforward when we make use of the more general secure evaluation of quadratic forms, as opposed to ordinary secure multiplications. Because the coefficients of the product polynomial are all quadratic forms, these can be evaluated directly, and no special protocol is needed for efficient polynomial multiplication. Each player should still use an algorithm for efficient polynomial multiplication for the local computations required for the evaluation of the coefficients.

We must also look at polynomial multiplication in terms of secure ordinary multiplications for two reasons. First, for the sake of generality, we do not assume our secure computation scheme directly supports the evaluation of quadratic forms. Second, we are also interested in the more general case of multiplying many polynomials using unbounded fan-in multiplication.

We represent secret-shared polynomials as a list of secret-shared coefficients. To multiply polynomials directly in this representation is rather costly, i.e., it would take a number of multiplications quadratic in the degree of the result. Instead of direct multiplication, we can make use of the fact that any set of $N+1$ points that are distinct in their x-coordinate uniquely define a polynomial of degree at most N that interpolates those points. To multiply polynomials, we first evaluate each of these polynomials in $N+1$ points, where N is the degree of the product polynomial. Then we perform point-wise multiplication of these points. Finally, we interpolate the product polynomial.

Let N_1, N_2 up to N_M be the respective degrees of M polynomials. Then the degree of the product of these polynomials is $N = \sum_{i=1}^{M} N_i$. Using the

above technique, evaluating the product of these M polynomials would require exactly $N + 1$ fan-in M multiplications. These multiplications can be carried out in parallel in a constant number of rounds.

Polynomial evaluation and interpolation on a fixed set of x-coordinates are linear operations. Therefore, these can be carried out locally by each party, without interaction. To do so naïvely would result in $O(N^2)$ complexity of local operations. There are textbook techniques, however, which can be used to reduce the complexity of these local operations to $O(N \log N)$, see, e.g., [vzGG03, Chapter 10]. The secure multipoint evaluation and interpolation protocols described in this section are secure adaptations of such techniques. The most well known of these techniques is the Fast Fourier Transform, which relies on the existence of roots of unity. We will, however, use different techniques, to avoid this requirement.

For our permutation protocol, we can use the fact that we only work with monic polynomials, i.e., polynomials with leading coefficient equal to 1. A product of monic polynomials will itself be monic. Furthermore, the second coefficient of the product will be equal to the sum of the second coefficients of its factors. Since this is a linear operation, it can be computed locally without interaction. Knowing the two leading coefficients of the product allows us to compute it using two fewer multiplications. If we simply evaluate the constituent polynomials in two fewer points and subtract the evaluations of the known part of the polynomial, we can then interpolate the remaining coefficients.

Although the number of multiplications saved this way is asymptotically negligible, it allows us to show how our permutation protocol can be seen as a generalization of the random bit protocol from [DFK+06] in Section 3.4

Polynomial Division with Remainder

Another technique needed to efficiently carry out our protocols is to determine the remainder of polynomial division. Let f and g be two polynomials of degree n and m respectively, with $n > m$. The protocol to compute the quotient, q, of division of f by g is given in Protocol 3.1, which we will briefly describe here. For a more thorough description, see [MF06b].

Computation on polynomials in the polynomial division protocol is carried out over the ring $\mathbb{F}[X]/X^{n-m+1}$. Operationally, this corresponds to truncating all polynomials to degree (at most) $n - m$. On line 6, we take the inverse of the polynomial u. By this inverse, we mean the inverse of u in the ring $\mathbb{F}[X]/X^{n-m+1}$, which exists if the constant term u_0 is non-zero. Note that,

Protocol 3.1 PolDiv($[\![f]\!], [\![g]\!]$), computed over $\mathbb{F}[X]/X^{n-m+1}$

1: $[\![s]\!] \xleftarrow{\$} \mathbb{F}[X]/X^{n-m+1}$
2: $[\![t]\!] \leftarrow \mathrm{rev}_m([\![g]\!])[\![s]\!]$
3: $u \leftarrow \mathrm{reveal}([\![t]\!])$
4: **if** $u_0 = 0$ **then**
5: abort with failure
6: $[\![v]\!] \leftarrow u^{-1}[\![s]\!]$
7: $[\![q]\!] \leftarrow \mathrm{rev}_{n-m}(\mathrm{rev}_n([\![f]\!])[\![v]\!])$
8: **return** $[\![q]\!]$

because we will only perform polynomial division where the degree of the divisor g is exactly m, the constant term u_0 will be equal to zero only if the constant term of s is equal to zero, which happens with probability $1/|\mathbb{F}|$. If the field is sufficiently large, this will happen with negligible probability. Otherwise, the protocol will have to be repeated multiple times, either in parallel or iteratively, so that the probability of all repetitions failing would be negligible.

In the polynomial division protocol $\mathrm{rev}_n(f)$ denotes a polynomial such that $\mathrm{rev}_n(f)(x) = x^n f(1/x)$ defines the *reversal* of the polynomial. If the degree of f is at most n, $\mathrm{rev}_n(f)(x)$ represents the polynomial of degree n which has the same coefficients as f, if necessary padded with zeroes, but in reversed order. Note that computing the reversal is a local operation which does not require any interaction.

Consider the equation $f(x) = g(x)q(x) + r(x)$, where we take the degree of r as $m-1$. Substituting $1/x$ for x and multiplying by x^n allows us to rewrite it as

$$\begin{aligned}
& x^n f(1/x) = (x^m g(1/x))(x^{n-m} q(1/x)) + x^{n-m+1}(x^{m-1} r(1/x)) \\
\iff \quad & \mathrm{rev}_n(f)(x) = \mathrm{rev}_m(g)(x)\,\mathrm{rev}_{n-m}(q)(x) + x^{n-m+1}\,\mathrm{rev}_{m-1}(r)(x) \\
\implies \quad & \mathrm{rev}_n(f)(x) = \mathrm{rev}_m(g)(x)\,\mathrm{rev}_{n-m}(q)(x) \pmod{x^{n-m+1}} \\
\implies \quad & \mathrm{rev}_{n-m}(q)(x) = \mathrm{rev}_n(f)(x)\,\mathrm{rev}_m(g)^{-1}(x) \pmod{x^{n-m+1}} \\
\implies \quad & q(x) = \mathrm{rev}_{n-m}(\mathrm{rev}_n(f)(x)\,\mathrm{rev}_m(g)^{-1}(x) \bmod x^{n-m+1}),
\end{aligned}$$

where the last implication holds, because the degree of q is $n-m$. The protocol essentially evaluates the final equation to find the quotient. To evaluate this equation, it uses the technique for computing inverses from [BB89].

Protocol 3.2 uses the quotient protocol to compute the remainder after polynomial division as $r = f - qg$ and is given for completeness.

Protocol 3.2 PolRem($[\![f]\!], [\![g]\!]$)

1: $[\![q]\!] \leftarrow \mathsf{PolDiv}([\![f]\!], [\![g]\!])$
2: $[\![r]\!] \leftarrow [\![f]\!] - [\![q]\!][\![g]\!]$
3: **return** $[\![r]\!]$

Protocol 3.3 PolTree($[\![\boldsymbol{L}]\!]$)

Input: $[\![\boldsymbol{L}]\!]$, a list of N polynomials
1: **for** $j \in \{0, 1, \ldots, N-1\}$ **do**
2: $\quad [\![T_{0,j}]\!] \leftarrow [\![L_{j+1}]\!]$
3: **for** $i \leftarrow 1$ to k **do**
4: $\quad$ **for** $j \in \{0, 1, \ldots, N_i - 1\}$ **do**
5: $\quad\quad [\![T_{i,j}]\!] \leftarrow \prod_{\ell=0}^{M_{i,j}} [\![T_{i-1,j+\ell N_i}]\!]$
6: **return** $[\![T]\!]$

The Polynomial Product Tree

In this section we will show a protocol for building the polynomial product tree. This tree enables an efficient multipoint evaluation protocol, which is needed for the rearrangement protocol. Additionally, the same tree forms the core of the permutation protocol.

The leaves of a polynomial product tree contain a polynomial. Each of the nodes contains the product of the polynomials contained by its children. The root, thus, contains the product of all leaf polynomials. The protocol, displayed in Protocol 3.3 follows a typical divide and conquer strategy, but is specified in iterative form to clearly demonstrate the round and multiplicative complexity.

Recall that we denote the secret-shared list $\boldsymbol{L}$ and trees, such as T as $[\![\boldsymbol{L}]\!]$ and $[\![T]\!]$, respectively. By this we mean that the values in the list or tree are secret-shared, but the structure is publicly known. Similarly, for the secret-shared polynomials in $\boldsymbol{L}$, we mean that the coefficients are secret-shared, but the degree of the polynomials is publicly known.

Note that the only step involving secure computation is step 5. The multiplication in step 5 uses fan-in M secure polynomial multiplication. The computations taking place in the inner loop, step 4, can be parallelized.

In the following, we will only compute polynomial product trees for lists of monic non-constant polynomials. Therefore the following theorem is restricted to that particular case.

Theorem 3.1. *Let $\boldsymbol{L}$ be a vector of N monic non-constant polynomials. Let d be the sum of the degrees of the polynomials in $\boldsymbol{L}$. Protocol 3.3 on $[\![\boldsymbol{L}]\!]$ terminates after $O(\log_M N)$ rounds of communication in which $O(dM \log_M N)$ secure operations are performed.*

Proof. Step 5 is the only step involving secure computation. Computing the product of the monic polynomials involves $d' - 1$ invocations of the fan-in M multiplication protocol, where d' is the degree of the resulting polynomial.

For any level of the tree, the polynomial multiplications can be carried out in parallel. The total number of multiplications of fan-in at most M for level i is equal to d minus the number of nodes at that level of the tree. Computing a single level of the tree takes a constant number of rounds.

Let $k = \lceil \log_M N \rceil$ be the number of levels in the tree, excluding the leaves, which require no secure operations to compute. Summing the costs per level over the k levels gives us a total of kd minus the total number of internal nodes in the tree fan-in at most M multiplications. The total number of internal nodes is equal to $\sum_{\ell=1}^{k} \lceil \frac{N}{M^\ell} \rceil$. This translates to $O(Mkd)$ elementary secure operations in $O(k)$ rounds. □

For ordinary, binary multiplication, i.e., $M = 2$, each level of the tree takes exactly one round to compute and the total number of multiplications is exactly equal to $kd - \sum_{\ell=1}^{k} \lceil \frac{N}{2^\ell} \rceil \leq kd - N + 1$. Note that if all polynomials in $\boldsymbol{L}$ are of degree one, then we can take $d = N$.

Multipoint Polynomial Evaluation

In this section we will combine the protocols from the previous sections to create a protocol for multipoint polynomial evaluation. Multipoint polynomial evaluation is the problem of, given a polynomial f and a list of N points $x_1, x_2, \ldots, x_N$ to evaluate $f(x_1), f(x_2), \ldots, f(x_N)$. For our purposes we restrict our protocol to the evaluation of polynomials of degree at most $N - 1$. The protocol, shown in Protocol 3.4 is simply composed of two sub-protocols. The first sub-protocol is Protocol 3.3, PolTree, to construct the polynomial product tree with N leaves containing the polynomials $x - x_i$ for all $1 \leq i \leq N$. In the second sub-protocol, shown in Protocol 3.5, the tree is traversed from the root to the leaves, evaluating the polynomial.

The protocol is simply the well-known efficient algorithm for multipoint polynomial evaluation, as described in, for example, [vzGG03, Chapter 10], where the inputs, outputs and intermediate values are all secret-shared and the

Protocol 3.4 MultiEval($[\![f]\!], [\![\boldsymbol{x}]\!]$)

Input: • $[\![f]\!]$, a polynomial of degree at most $N-1$
• $[\![\boldsymbol{x}]\!]$, a list of N points

1: $[\![T]\!] \leftarrow \mathsf{PolTree}((x - [\![x_i]\!])_{i=1}^{N})$
2: **return** TreeEval($[\![f]\!], [\![T]\!]$)

Protocol 3.5 TreeEval($[\![f]\!], [\![T]\!]$)

Input: • $[\![f]\!]$, a polynomial of degree at most $N-1$
• $[\![T]\!]$, a polynomial product tree with N leaves

1: $[\![E_{k,0}]\!] \leftarrow [\![f]\!]$
2: **for** $i \leftarrow k-1$ **down to** 0 **do**
3: **for** $j \in \{0, 1, \ldots, N_i - 1\}$ **do**
4: $[\![E_{i,j}]\!] \leftarrow \mathsf{PolRem}([\![E_{i+1,j \bmod N_{i+1}}]\!], [\![T_{i,j}]\!])$
5: **return** $([\![E_{0,j}]\!])_{j=1}^{N}$

construction of the polynomial product tree and evaluations of remainders after polynomial division have been replaced by the corresponding secure protocols.

The correctness of the protocol follows from two observations. First, for any constant x_0, it holds that $f(x_0) \equiv f(x) \pmod{x - x_0}$. Second, for polynomials f_1, f_2, f_3, g_1 and g_2, if $f_2 \equiv f_1 \pmod{g_1 g_2}$ and $f_3 \equiv f_2 \pmod{g_1}$, then $f_3 \equiv f_1 \pmod{g_1}$.

Theorem 3.2 (Complexity)**.** *On input of size N, the round complexity of Protocol 3.5 is $O(\log_M N)$ and the multiplicative complexity is $O(MN \log_M N)$.*

Proof. The only step which requires the parties to interact is step 4. A constant round protocol for polynomial modular reduction is given in Protocol 3.2, due to [MF06b]. The complexity of Protocol 3.2 is linear in the degree of the polynomial to be reduced.

At the ith layer of the tree, for $0 \leq i < k$, we need to perform N_i polynomial modular reductions of polynomials of degree smaller than M^{i+1}. Summing over all levels of the tree, this gives a round complexity of $O(k) = O(\log_M N)$ and multiplicative complexity of less than $\sum_{i=0}^{k-1} N_i O(M^{i+1}) = kO(MN) = O(MN \log_M N)$. □

The trade-off between round complexity and multiplicative complexity of Protocol 3.5 matches that of Protocol 3.3 due to the use of the same tree

Protocol 3.6 Interpolate($[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]$)

Input: • $[\![\boldsymbol{x}]\!]$, a list of N distinct points
• $[\![\boldsymbol{y}]\!]$, a list of N points

1: $[\![T]\!] \leftarrow \mathsf{PolTree}((x - [\![x_i]\!])_{i=1}^{N})$
2: **return** InterpTree($[\![T]\!], [\![\boldsymbol{y}]\!]$)

structure.

Note that the only secure operation needed in Protocol 3.5 is secure polynomial modular reduction (step 4). Given the security of Protocol 3.2 [MF06b], we can conclude that Protocol 3.10 securely performs multipoint polynomial evaluation.

Polynomial Interpolation

Polynomial interpolation is the inverse problem to multipoint polynomial evaluation: given a list of N distinct points, $x_1, x_2, \ldots, x_N$ and another list of N not necessarily distinct points $y_1, y_2, \ldots, y_N$, find the unique polynomial f of degree less than N, such that $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$.

The protocol for polynomial interpolation is a secure adaptation of a well-known algorithm as described in, e.g., [vzGG03, Chapter 10], just as the protocol for multipoint polynomial evaluation. The protocol is shown in Protocol 3.6. Also this protocol simply consists of two sub-protocols, where the first constructs the same polynomial product and the second, shown in Protocol 3.7, performs the interpolation traversing the tree from the root to the leaves by, in fact, relying on the TreeEval protocol, and then constructing a tree with the same structure from the leaves back to the root. As we will show, asymptotically, polynomial interpolation has the same complexity as multipoint polynomial evaluation, however, because the tree has to be traversed in both directions, secure polynomial interpolation is a somewhat more costly operation.

We will give an outline of why the interpolation protocol is correct. Given $y_1, y_2, \ldots, y_N$ and distinct $x_1, x_2, \ldots x_N$, the Lagrange interpolation polynomial

$$L(x) = \sum_{i=1}^{N} y_i \prod_{\substack{j=1 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j} \tag{3.2}$$

has the desired properties of having degree less than N and $L(x_i) = y_i$ for all $1 \leq i \leq N$.

Protocol 3.7 TreeInterp($[\![T]\!], [\![\boldsymbol{y}]\!]$)

Input: • $[\![T]\!]$, a polynomial product tree with N leaves
• $[\![\boldsymbol{y}]\!]$, a list of N points

1: $[\![p]\!] \leftarrow [\![T_{k,0}]\!]$
2: **for** $i \in \{1, 2, \ldots, N\}$ **do**
3: $\quad [\![p'[i-1]]\!] \leftarrow i[\![p[i]]\!]$
4: $[\![\boldsymbol{v}]\!] \leftarrow$ TreeEval($[\![p']\!], [\![T]\!]$)
5: **for** $i \in \{1, 2, \ldots, N\}$ **do**
6: $\quad [\![v_i^{-1}]\!] \leftarrow [\![v_i]\!]^{-1}$
7: $\quad [\![C_{0,i}]\!] \leftarrow [\![y_i]\!][\![v_i^{-1}]\!]$
8: **for** $i \leftarrow 1$ **to** k **do**
9: $\quad$ **for** $j \in \{1, 2, \ldots, N_i\}$ **do**
10: $\quad\quad [\![C_{i,j}]\!] \leftarrow \sum_{\ell=1}^{M_{i,j}} [\![C_{i-1,j+\ell N_i}]\!] \prod_{\substack{q=1 \\ q \neq \ell}}^{M_{i,j}} [\![T_{i+1,q}]\!]$
11: **return** $[\![C_{k,0}]\!]$

Protocol 3.6 builds the polynomial product tree for $(x - x_i)_{i=1}^N$ and passes it to Protocol 3.7. Protocol 3.7 assigns the root of the given polynomial product tree, $\prod_{i=1}^N (x - x_i)$, to p and then computes the derivative with respect to x, which, due to the product rule for derivatives, can be expressed as

$$p'(x) = \sum_{i=1}^{N} \prod_{\substack{j=1 \\ j \neq i}}^{N} (x - x_j). \tag{3.3}$$

Using the protocol for secure multipoint polynomial evaluation, the values

$$v_i = p'(x_i) = \prod_{\substack{j=1 \\ j \neq i}}^{N} (x_i - x_j) \tag{3.4}$$

are evaluated for all $1 \leq i \leq N$. Note that these are precisely the values of the product appearing in the denominator of the Lagrange interpolation polynomial. These values are inverted, multiplied by the corresponding y_i values and

assigned to $C_{0,i}$ for all $1 \leq i \leq N$, i.e.,

$$C_{0,i} = y_i \left(\prod_{\substack{j=1 \\ j \neq i}}^{N} (x_i - x_j) \right)^{-1} . \tag{3.5}$$

What remains is to multiply these values by the polynomial $\prod_{\substack{j=1 \\ j \neq i}}^{N}(x - x_j)$ and sum the results together. This is performed in the final loop of Protocol 3.7. The intuition is that the protocol goes up the polynomial product tree and, at every level, multiplies in the product of all *other* product polynomials within a set of siblings so that eventually each $C_{0,i}$ is multiplied by the product of all $(x - x_j)$ except for $j = i$, as required.

Theorem 3.3 (Complexity)**.** *On input of size N, the round complexity of Protocol 3.7 is $O(\log_M N)$ and the multiplicative complexity is $O(MN \log_M N)$.*

Proof. The only steps requiring interaction are steps 4, 6, 7, and 10. In total, steps 6 and 7 require N secure multiplications and N secure inversions in a constant number of rounds, which is negligible with respect to the complexities of subprotocol TreeEval, step 4.

The $C_{i,j}$ are polynomials of degree less than M^i, so the computation of each $C_{i,j}$ requires at most $M_{i,j}M^i \leq M^{i+1}$ secure multiplications. The total number of secure multiplications required for step 10 is at most $\sum_{i=1}^{k} N_i M^{i+1} \leq kNM = MN \log_M N$. Due to the parallelizability of the inner loop, the round complexity is $O(k)$. □

3.4 The Permutation Protocol

The permutation protocol allows us to sample a uniformly random permutation of given size N. It is based on the observation that the product of polynomials is invariant under permutation of its factors. Protocol RandomPermutation, displayed in Protocol 3.8, consists of three stages. First, N random monic degree 1 polynomials are sampled obliviously. Then, the product of these polynomials is computed securely. Finally, the product polynomial is made public and factored by each party individually to retrieve the degree 1 polynomials, or rather, their roots, from the first stage. These roots are sorted in an arbitrary, but canonical order. If there are any collisions, the protocol fails. Otherwise, the correspondence between the secret-shared and publicly known lists of roots

Protocol 3.8 RandomPermutation(N)

1: $[\![\hat{\boldsymbol{r}}]\!] \xleftarrow{\$} \mathbb{F}^N$
2: $[\![T]\!] \leftarrow \mathsf{PolTree}((x - [\![\hat{r}_i]\!])_{i=1}^N)$
3: $f \leftarrow \text{reveal}([\![T_{k,0}]\!])$
4: find the (ordered) list of roots $\boldsymbol{r}$ of f
5: **if** $r_i = r_j$ for any $i \neq j$ **then**
6: abort with failure
7: **else**
8: **return** $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$.

establishes a permutation. These two lists are returned together with the polynomial product tree as a representation of the permutation. The inclusion of the polynomial product tree later allows for more efficient operations involving the random permutation.

Protocol RandomPermutation imposes two restrictions. First, we must be able to factor polynomials efficiently and uniquely. For this reason, we have chosen to confine ourselves to secure schemes that compute over finite fields. In particular, a scheme based on Shamir secret sharing is entirely suitable. Second, the range of values we can express must be sufficiently large to avoid collisions. We require that the order of the field in which we compute is sufficiently greater than N, such that the probability of sampling the same element more than once when sampling N elements uniformly and independently from the finite field is negligible. This establishes a hidden dependency of the communication complexity, in terms of bits, on the size of the permutation N.

The secret-shared representation of a permutation π is given as the triple $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$, where $\boldsymbol{r}$ is a vector of pairwise distinct coefficients, $\hat{\boldsymbol{r}} = \pi\boldsymbol{r}$ and T is the polynomial product tree of $(x - \hat{r}_i)_{i=1}^N$. The polynomial product tree is not necessary to represent the permutation, but is used to efficiently compute operations involving the permutation. We choose to explicitly incorporate the polynomial product tree in the representation of a permutation, so that it is computed only once, instead of every time an operation involving the permutation is performed.

Theorem 3.4. *Protocol 3.8 terminates after $O(\log_M N)$ rounds of communication in which $O(MN \log_M N)$ secure operations are performed. Upon successful termination, $\boldsymbol{r}$ will be a uniformly random vector of N distinct elements and $\hat{\boldsymbol{r}}$ will securely establish a uniformly random permutation of $\boldsymbol{r}$.*

Proof. The complexity of this protocol is dominated by step 2, which requires $O(MN \log_M N)$ secure operations in $O(\log_M N)$ rounds by Theorem 3.1.

The only other steps that require any interaction between the parties are 1 and 3 and for each of these steps the amount of interactions is equivalent to at most N elementary multiplications. This brings the total complexity of secure operations, expressed in elementary multiplications to $O(MN \log_M N)$. These steps have constant round complexity, since these operations can be carried out in parallel.

Because factorization of polynomials over finite fields is unique up to the order of the factors and multiplication by constants, it is straightforward to see that, if this protocol succeeds, then there exists a unique permutation π such that $\hat{\boldsymbol{r}} = \pi \boldsymbol{r}$. Since the values of $\boldsymbol{r}$ were sampled independently, their order is uniformly random, meaning the permutation is uniformly random.

To argue that this permutation is established securely, note that the only information revealed by this protocol is the product polynomial at the root of the multiplication tree (given the fact that revealing a secret value does not leak any information other than the value itself and that the remaining secure operations reveal no information at all, by the universal composability of our arithmetic black box). Due to commutativity of multiplication, the product polynomial is invariant under permutation of its factors, i.e., the product polynomial remains the same regardless of the order in which the constituent polynomials were multiplied. □

For this protocol to function correctly, the parties must establish an order of the recovered list of roots $\boldsymbol{r}$. This can be achieved using an additional round of communication. However, because the order of $\boldsymbol{r}$ is independent of the order of $\hat{\boldsymbol{r}}$, the parties can simply sort the list of roots into $\boldsymbol{r}$ using some canonical ordering.

The failure probability of Protocol 3.8 is dictated by the birthday paradox. A simple approximation of the probability of sampling only unique elements when sampling N elements uniformly and independently from a set of $|\mathbb{F}|$ elements is given by

$$\Pr[\text{no collisions}] = \frac{|\mathbb{F}|}{|\mathbb{F}|} \frac{|\mathbb{F}|-1}{|\mathbb{F}|} \cdots \frac{|\mathbb{F}|-N+1}{|\mathbb{F}|} > \left(1 - \frac{N}{|\mathbb{F}|}\right)^N \geq 1 - \frac{N^2}{|\mathbb{F}|}, \quad (3.6)$$

where the second inequality is Bernoulli's inequality. This shows that field order $|\mathbb{F}| \geq 2^\kappa N^2$ is sufficient to ensure that the probability of encountering any collisions is smaller than $2^{-\kappa}$. For this chapter, we will simply assume

that the order of the field over which we compute is large enough to make the probability of collisions negligible. If we do not have the freedom to choose the field over which we compute, and the order of the field is too small to make the failure probability of the protocol negligible, we can sample the random permutation by taking the random roots from an extension field. To permute a given list, as is the topic of Section 3.5, then requires to convert the input list of secret-shared data to secret-shares over the extension field and convert the secret-shared output list back to secret-shares over the base field. For MPC based on Shamir's secret sharing scheme the conversion between the base field and the extension field and vice versa is possible without interaction, if the players use the same share reconstruction coefficients, taken from the base field, for both types of secret-shares. Alternatively, the protocol can be run multiple times in parallel. Both methods lead to obvious increases in the communication complexity.

Aside from the complexity of secure operations, we should also pay some attention to the computation complexity of local operations. The local computation complexity is determined by the polynomial factorization step (step 4). The complexity of an algorithm for our problem of finding the N distinct roots of the product polynomial is $O(N(\log N)^3)$ operations in $\mathbb{F}$ [vzGG03, Chapter 14]. Asymptotically, this easily dominates the complexity of the protocol, however, due to the enormous difference between local computation and secure operations involving communication, this is no object in practice.

We will briefly illustrate how the RandomPermutation protocol can be seen as a generalization of the random bit protocol of [DFK+06], shown in Protocol 3.9. The random bit protocol starts by sampling a random element $a \in \mathbb{F}$ and then computes s^2, revealing the product. This corresponds to the first three steps of RandomPermutation, for $N = 2$, where we restrict the sampling of $\hat{\boldsymbol{r}}$ to vectors such that $\hat{\boldsymbol{r}} = (a, -a)$. For such vectors, the root of the polynomial product tree will be equal to $x^2 - a^2$ and revealing this polynomial is equivalent to revealing a^2. The protocol then aborts if $a^2 = 0$, which corresponds to $x^2 - a^2$ not having two distinct roots. If the random bit protocol does not abort, it can be seen as having sampled a random permutation of size two. The remainder of the protocol simply transforms the random permutation into a binary value.

Protocol 3.9 The random bit protocol of [DFK+06]

1: $[\![a]\!] \xleftarrow{\$} \mathbb{F}$
2: $[\![a^2]\!] \leftarrow [\![a]\!][\![a]\!]$
3: $a^2 \leftarrow \text{reveal}([\![a^2]\!])$
4: **if** $a^2 = 0$ **then**
5: abort with failure
6: **else**
7: $b \leftarrow \sqrt{a^2}$
8: $[\![c]\!] \leftarrow b^{-1}[\![a^2]\!]$
9: $[\![d]\!] \leftarrow 2^{-1}([\![c]\!] + 1)$
10: **return** $[\![d]\!]$

Protocol 3.10 $\mathsf{Rearrange}([\![\boldsymbol{v}]\!], (\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!]))$

Input: • $[\![\boldsymbol{v}]\!]$, a list of N secret-shared values
• $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$, the representation of a random permutation
1: $[\![f]\!] \leftarrow \mathsf{Interpolate}(\boldsymbol{r}, [\![\boldsymbol{v}]\!])$
2: $[\![\hat{\boldsymbol{v}}]\!] \leftarrow \mathsf{TreeEval}([\![f]\!], [\![T]\!])$
3: **return** $[\![\hat{\boldsymbol{v}}]\!]$

3.5 The Rearrangement Protocol

Given a random secret-shared permutation, as produced by Protocol 3.8, the rearrangement protocol transforms a list of secret-shared values into a shuffled list of those same values without any party obtaining information about the permutation or the values themselves. The rearrangement protocol is displayed in Protocol 3.10.

Note that the interpolation in step 1 is a linear transformation of $[\![\boldsymbol{v}]\!]$, and therefore Interpolate in this step does not refer to Protocol 3.6, but to a polynomial interpolation algorithm which can be carried out locally.

It is also possible to apply the inverse permutation to a given list by reversing the roles of $\boldsymbol{r}$ and $[\![\hat{\boldsymbol{r}}]\!]$ in the interpolation and multipoint polynomial evaluation steps. For completeness, the inverse shuffle protocol is shown in Protocol 3.11. Note that here the interpolation step is the secure protocol and the multipoint polynomial evaluation is reduced to a linear operation on secret-shared values. Because the inverse rearrangement protocol depends on secure interpolation, it requires somewhat more secure operations to complete. This is

Protocol 3.11 InverseRearrange($[\![\boldsymbol{v}]\!], (\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$)

Input: • $[\![\boldsymbol{v}]\!]$, a list of N secret-shared values
• $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$, the representation of a random permutation

1: $[\![f]\!] \leftarrow$ TreeInterp($[\![T]\!], [\![\boldsymbol{v}]\!]$)
2: $[\![\hat{\boldsymbol{v}}]\!] \leftarrow$ MultiEval($[\![f]\!], \boldsymbol{r}$)
3: **return** $[\![\hat{\boldsymbol{v}}]\!]$

the reason why we have chosen the approach of Protocol 3.10 as the "forward" rearrangement and the approach of Protocol 3.11 as the inverse.

Theorem 3.5 states that given a secret-shared permutation of size n represented as the result of Protocol 3.8 and its multiplication tree, Protocol 3.10 applies the permutation to a given secret-shared list and Protocol 3.11 applies the inverse permutation to the given secret-shared list.

Theorem 3.5 (Correctness)**.** *Given a secret-shared input list $[\![\boldsymbol{v}]\!]$ of length N and a secret-shared permutation $[\![\pi]\!]$ of size N, represented as $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$, Protocol 3.10 produces the permuted input list $[\![\hat{\boldsymbol{v}}]\!]$, such that $\hat{\boldsymbol{v}} = \pi\boldsymbol{v}$. On the same input, Protocol 3.11 produces the secret-shared, inversely permuted input list $[\![\boldsymbol{v}']\!]$, such that $\boldsymbol{v}' = \pi^{-1}\boldsymbol{v}$.*

Proof. Let W and $\hat{W}$ be the $n \times n$ Vandermonde matrices of $\boldsymbol{r}$ and $\hat{\boldsymbol{r}}$ respectively. Since $\hat{\boldsymbol{r}} = \pi\boldsymbol{r}$, it follows that $\hat{W} = \pi W$. Multipoint evaluation of a polynomial corresponds to multiplication (from the left) of the vector containing the coefficients of the polynomial by the Vandermonde matrix of the points in which the polynomial is to be evaluated. Interpolation of a polynomial is the inverse of multipoint evaluation and can be expressed as multiplication from the left by the inverse Vandermonde matrix.

The coefficients of the interpolating polynomial computed in step 1 of Protocol 3.10 can be represented as $W^{-1}[\![\boldsymbol{v}]\!]$. The evaluation thereof in the points $[\![\hat{\boldsymbol{r}}]\!]$ is represented by multiplication from the left by $[\![\hat{W}]\!]$. As such, the result corresponds to $[\![\hat{W}]\!]W^{-1}[\![\boldsymbol{v}]\!] = [\![\pi W]\!]W^{-1}[\![\boldsymbol{v}]\!] = [\![\pi\boldsymbol{v}]\!]$.

Similarly, because the roles of $\boldsymbol{r}$ and $[\![\boldsymbol{r}']\!]$ are reversed, the result of Protocol 3.11 is $W[\![\hat{W}]\!]^{-1}[\![\boldsymbol{v}]\!] = W[\![W^{-1}\pi^{-1}]\!][\![\boldsymbol{v}]\!] = [\![\pi^{-1}\boldsymbol{v}]\!]$. □

Rearranging Composite Objects

Protocol Rearrange can be used to rearrange a list of ordinary secret-shared values according to the given permutation. We briefly consider the problem of

securely rearranging secret-shared composite objects, i.e., collections of multiple secret-shared values. We will consider secret-shared objects that are represented as vectors of secret-shared values. We require furthermore that all vectors representing the list of objects to be rearranged have the same length. Let L be the length of these vectors. This means a list of N secret-shared composite objects can be represented as a secret-shared $N \times L$ matrix, of which each row represents one object.

Since protocol Rearrange rearranges the input list of secret-shared values deterministically according to the given permutation, a matrix representing a list of secret-shared composite objects can be rearranged by rearranging each column of the matrix according to the same permutation.

Each column can be rearranged independently of the other columns and these rearrangements can be performed in parallel. Therefore the round complexity of this approach is equal to the round complexity of rearranging a list of N elementary secret-shared values. The multiplicative complexity is $O(LNM \log_M N)$; a factor L greater than that of the elementary case.

For $L \gg N$ it may be beneficial to, instead of running protocol Rearrange directly on the input matrix, compute the rearrangement of the identity matrix. The rearranged identity matrix is the permutation matrix corresponding to the given permutation. The input matrix can then be rearranged through multiplication from the left with permutation matrix. Computation of the permutation matrix requires $O(N^2 M \log_M N)$ secure operations. Here we make use of efficient dot products, so that the multiplicative complexity of the matrix product is exactly NL. We disregard the complexity of local operations for the matrix product.

Oblivious Arrays

In this section we will briefly extend Example 2.14 for oblivious array indexing to show how random shuffles can be used to apply techniques similar to the square root oblivious RAM construction of [GO96] to implement array data structures in MPC where both index and data are secret-shared. For an array of size N, oblivious read operations have complexity at most $O(\sqrt{N})$ and oblivious write operations complexity at most $O(N \log N)$, which is *amortized* over $O(\sqrt{N})$ write operations. We will assume that any secret index is guaranteed to be within the array bounds.

The results of example 2.14 can be directly applied to show how the element at index i in an array of size N can be extracted using $O(\sqrt{N})$ bilinear operations on secret-shares if both the index and the values in the array are

secret-shared. This is achieved by determining the polynomial interpolating the secret-shared values in the array, which is a linear operation, and evaluating this polynomial at i, which requires $O(\sqrt{N})$ bilinear operations. Note that even though we disregard the complexity of interpolating the polynomial as this is a linear operation, the interpolated polynomial can be cached and, as will become clear in the following paragraphs, will only have to be recomputed after every $O(\sqrt{N})$ write operations.

Writing to an oblivious array at a secret index can, depending on the model of security, be shown to be impossible in a sublinear (in N) number of secure operations, since this problem can be considered equivalent to the problem of computing the ith unit vector which produces $N - 1$ linearly independent secret-shared binary values. Because these values should be independent of each other, each of these requires a separate secure operation. Therefore we apply the commonly used approach of a small *stash* of size $O(\sqrt{N})$ for writing to a secret index.

Concretely, the stash consist of an, initially empty, list of index-delta pairs. Let L be the size of the stash. When a secure write operation of value v at index i is performed on the oblivious array, first, for each index-delta pair (i_j, v_j) at position $1 \leq j \leq L$ in the stash, the value of i_j is obliviously compared to i and, if these are equal, obliviously replaced by $N + j$. This ensures that every index in the stash is unique and the value i does not occur. Furthermore, any stashed index does not exceed the combined array and stash size. Then, the interpolating polynomial p is obliviously evaluated at i, as if performing an oblivious read operation on the array while ignoring the stash. Finally, the index i and the *difference* $\delta = v - p(i)$ are appended to the index-delta pair stash.

The complexity of the oblivious write operation is $O(\sqrt{N} + L)$, where L is the size of the stash, which increases by 1 for every write operation performed. To ensure that the complexity does not grow without bounds, the stashed values should be incorporated into the array at appropriate times so that L can be reset to 0.

To incorporate the stash into the array, first a random permutation of size $N+L$ is sampled obliviously. Recall that the L indices in the stash are all unique and within the bound of $N+L$. Using a combination of polynomial interpolation and evaluation, the permutation is applied to the stashed indices. Note that this does not mean the list of stashed indices is permuted, rather that for each index the corresponding index after permutation is computed. This can be done using $O(\sqrt{N + L})$ secure operations per stashed index and each of these can be evaluated in parallel. Then the permuted indices are revealed to all

parties. Because of the uniqueness and the random permutation, the revealed permuted indices are a uniformly random selection of L out of $N + L$ and therefore reveal no information about the secure write operations represented by the stash. Finally, an array of size $N + L$ is composed, consisting mostly of zeroes, but containing the secret-shared deltas at positions indicated by the permuted indices, to which then the inverse permutation is applied. The first N elements of the permuted delta array are then added element-wise to the oblivious array. The stash is then reinitialized to the empty stash.

The complexity of incorporating the stash into the array depends on both N and L and is given by

$$O(L\sqrt{N+L} + (N+L)\log(N+L)). \tag{3.7}$$

By bounding $L \leq \sqrt{N}$ the complexity of a single oblivious write is $O(\sqrt{N})$ and the complexity of incorporating the stash is $O(N \log N)$. The stash then has to be incorporated after every $\sqrt{N}$ write operations, leading to an amortized $\widetilde{O}(\sqrt{N})$ complexity for oblivious write-and-incorporate operations.

The oblivious read operation should also take into account any stashed values. An oblivious read at index i can simply be carried out by performing polynomial interpolation and evaluation on the main array and comparing each stashed index to i to obliviously determine which, if any, delta should be added to the result. Note that any oblivious equality tests mentioned above could be more efficiently implemented than using a secure protocol for general equality, since it can be assumed that the index values lie within the bounds of the combined array and stash.

3.6 Operations on Permutations

Using the Rearrange protocol it is possible to carry out operations on secure permutations themselves. We show how to securely compute the inverse of a given permutation and how to securely compute the composition of two given permutations.

First, to establish a secret-shared representation of the inverse of a secret-shared permutation, we can use the InverseRearrange protocol. By applying InverseRearrange to any vector, $\boldsymbol{b}$ of N pairwise distinct coefficients, and any secret-shared representation of a permutation π we obtain a pair $(\boldsymbol{b}, [\![\hat{\boldsymbol{b}}]\!])$ such that $\hat{\boldsymbol{b}} = \pi^{-1}\boldsymbol{b}$. By augmenting this pair with the polynomial product tree of the vector $(x - [\![\hat{b}_i]\!])_{i=1}^{N}$, the resulting triple forms a valid representation of the

Protocol 3.12 InversePermutation$((\boldsymbol{a}, [\![\hat{\boldsymbol{a}}]\!], [\![A]\!]))$

1: $[\![\hat{\boldsymbol{b}}]\!] \leftarrow \mathsf{InverseRearrange}(\boldsymbol{a}, (\boldsymbol{a}, [\![\hat{\boldsymbol{a}}]\!], [\![A]\!]))$
2: $[\![B]\!] \leftarrow \mathsf{PolTree}((x - [\![\hat{b}_i]\!])_{i=1}^{N})$
3: **return** $(\boldsymbol{a}, [\![\hat{\boldsymbol{b}}]\!], [\![B]\!])$

Protocol 3.13 Compose$((\boldsymbol{a}, [\![\hat{\boldsymbol{a}}]\!], [\![A]\!]), (\boldsymbol{b}, [\![\hat{\boldsymbol{b}}]\!], [\![B]\!]))$

1: $[\![\hat{\boldsymbol{c}}]\!] \leftarrow \mathsf{Rearrange}([\![\hat{\boldsymbol{a}}]\!], (\boldsymbol{b}, [\![\hat{\boldsymbol{b}}]\!], [\![B]\!]))$
2: $[\![C]\!] \leftarrow \mathsf{PolTree}((x - [\![\hat{c}_i]\!])_{i=1}^{N})$
3: **return** $(\boldsymbol{a}, [\![\hat{\boldsymbol{c}}]\!], [\![C]\!])$

permutation π^{-1}. For completeness, the InversePermutation protocol is given in Protocol 3.12.

In the same manner, it is also possible to compose secret-shared permutations. Given two permutations π_a and π_b, the first of it is represented as $(\boldsymbol{a}, [\![\hat{\boldsymbol{a}}]\!], A)$ we can derive the composition $\pi_c = \pi_b \pi_a$ using the Rearrange protocol. We rearrange $\hat{\boldsymbol{a}}$ by π_b to obtain $\hat{\boldsymbol{c}} = \pi_b \hat{\boldsymbol{a}} = \pi_b \pi_a \boldsymbol{a}$. The pair $(\boldsymbol{a}, [\![\hat{\boldsymbol{c}}]\!])$, augmented with the polynomial product tree of $(x - \hat{c}_i)_{i=1}^{N}$ forms a secret-shared representation of the permutation $\pi_b \pi_a$. Protocol Compose is displayed in Protocol 3.13.

3.7 Imposing a Cycle Structure

In this section we will study the issue of how to securely sample a uniformly random permutation with a given cycle structure. The results of this section are motivated by the problem of secret Santa. The secret Santa problem is the problem of sampling a uniformly random permutation without any fixed points. One generalization of this problem is that of sampling uniformly random permutations that do not contain cycles shorter than some given length. A variant is that of sampling uniformly random cycles of full length, or, more generally, sampling uniformly random permutations with a given cycle structure.

For the original secret Santa problem, and its generalization avoiding short cycles, we do not propose any solution, other than a rejection sampling approach, in which random permutations are sampled and those containing fixed points rejected. Note that secure public fixed point detection is straightforward given secure public equality testing. Similarly, detection of short cycles can be performed efficiently, but at cost linear in the cycle length to be de-

Protocol 3.14 $\mathsf{Conjugate}((\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!]), \sigma)$

1: $[\![\hat{\boldsymbol{b}}]\!] \leftarrow \mathsf{InverseRearrange}(\sigma[\![\hat{\boldsymbol{r}}]\!], (\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!]))$
2: $[\![B]\!] \leftarrow \mathsf{PolTree}((x - [\![\hat{b}_i]\!])_{i=1}^{N})$
3: **return** $(\boldsymbol{r}, [\![\hat{\boldsymbol{b}}]\!], [\![B]\!])$

tected, by applying the permutation to itself and publicly testing whether the intermediate and final results contain any fixed points.

Instead, we solve the problem of securely sampling a random permutation with a given cycle structure. It is known from the theory of symmetric groups that all permutations of a given cycle structure form a *conjugacy class*, i.e., permutations a and b, both of length N, have the same cycle structure if and only if there exists a permutation t of length N, such that $b = t^{-1}at$. The protocol for sampling a uniformly random permutation with a given cycle structure consists of sampling a uniformly random permutation t and computing the composition $t^{-1}at$ where a is any publicly known permutation with the given cycle structure.

Protocol Conjugate, displayed in Protocol 3.14, securely computes a secret-shared representation $(\boldsymbol{b}, [\![\hat{\boldsymbol{b}}]\!], [\![B]\!])$ of the permutation $\pi^{-1}\sigma\pi$, given a permutation σ and secret-shared representation $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$ of permutation π. For correctness, we want that $\hat{\boldsymbol{b}} = \pi^{-1}\sigma\pi\boldsymbol{b}$. By letting $\boldsymbol{b} = \boldsymbol{r}$, we have $\pi\boldsymbol{b} = \hat{\boldsymbol{r}}$ and since σ is publicly known, we can compute $\sigma\pi\boldsymbol{b} = \sigma\hat{\boldsymbol{r}}$ without any interaction. To compute $\hat{\boldsymbol{b}}$, we make use of protocol InverseRearrange.

We claim that protocol Conjugate yields a uniformly random permutation of the given cycle structure, when called with a uniformly random permutation as its first input. What remains to be proved is that $\pi^{-1}\sigma\pi$ gives a uniformly random element from the conjugacy class of σ, when π is a uniformly random permutation. This is implied by the following lemma.

Lemma 3.6. *Let a and b be permutations from the same conjugacy class. Let $\mathcal{C}_{a\to b}$ be the set of all permutation such that $b = \pi^{-1}a\pi$ for $\pi \in \mathcal{C}_{a\to b}$. Then $|\mathcal{C}_{a\to b}| = |\mathcal{C}_{a\to a}|$.*

Proof. Since a and b are of the same conjugacy class, $\mathcal{C}_{a\to b}$ is not the empty set. Let $\pi \in \mathcal{C}_{a\to b}$. Then for any $\iota \in \mathcal{C}_{a\to a}$, we have $b = \pi^{-1}a\pi = (\iota\pi)^{-1}a(\iota\pi)$. Due to the invertibility of permutations each of the $\iota\pi$ is a distinct element of $\mathcal{C}_{a\to b}$ and we have $|\mathcal{C}_{a\to b}| \geq |\mathcal{C}_{a\to a}|$. Conversely, for any $\phi \in \mathcal{C}_{a\to b}$, we have that $a = \phi b\phi^{-1} = (\pi\phi^{-1})^{-1}a(\pi\phi^{-1})$. Here each $\pi\phi^{-1}$ is a distinct element of $\mathcal{C}_{a\to a}$, which gives $|\mathcal{C}_{a\to b}| \leq |\mathcal{C}_{a\to a}|$. □

The above result also shows that the number of elements in a conjugacy class of permutations of size N divides the number of permutations, $N!$. Because for all but trivial permutation sizes there exists more than one conjugacy class, the number of elements in a conjugacy class strictly divides $N!$ for $N > 1$. Our method of sampling a random element from a given conjugacy class uses a random permutation of the same size and therefore 'consumes' more entropy than strictly necessary. We leave as an open question whether an efficient secure protocol with a simple and elegant description exists for randomly sampling an element of a given conjugacy class using strictly less entropy than required for a random permutation.

3.8 Partially Known Permutations

In this section we will consider an efficient variant of the RandomPermutation protocol in which players learn part of the permutation in the sense that each position of the random permutation is revealed to at least one player and it is publicly known for each position to which player or players it is revealed. This problem admits a more efficient solution than can be achieved through Protocol 3.8.

Two obvious applications of such partially known permutations are secure secret Santa, in which each player's name is drawn and revealed to exactly one player, and dealing cards for games such as bridge, in which each card is dealt to exactly one player. More generally, partially known permutations can be used as a building block in secure protocols for anonymous publishing, from which in turn protocols for securely and randomly partitioning data amongst players can be constructed. An example application would be to obliviously partition the records of a joint database amongst all players such that each record is randomly revealed to one player, who can then locally analyse their partition of the database without revealing the origin of the records.

In the following we will restrict ourselves to partially known permutations in which each position is revealed to exactly one player for simplicity. Furthermore, since partially known permutations are uniformly random, it does not matter *which* positions are revealed to each player, only the number.

Secure partially known permutations can be obtained by sampling a random permutation using Protocol 3.8 and then immediately revealing each of the secret-shared roots to the appropriate player. If, instead of all players jointly sampling each of the roots, each player would randomly sample their own roots in Protocol 3.8, then each player could input their own product polynomial

and the multiplicative complexity would be reduced to $O(N \log q)$, where q is the number of players. Similarly, the round complexity would be reduced to $O(\log q)$.

Letting the players sample their own roots in Protocol 3.8 improves both the multiplicative and round complexities by a factor of $\log_q N$. However, for a truly more efficient protocol, we must abandon our base of general multiparty computation and turn to a scheme that is especially suitable for this scenario.

The secure protocol for obliviously sampling a partially known permutation is displayed in Protocol 3.15. The main idea remains the same as in Protocol 3.8: the players sample a number of random roots and obliviously compute the corresponding polynomial. Instead of using Shamir secret sharing, however, in this protocol secrets are shared such that the secret value is equal to the *product* of the shares, i.e., the protocol uses additive secret sharing on the multiplicative group of the field. This form of secret sharing allows any non-zero element to be shared and requires all players to participate in reconstruction of the secret. We shall call this the 'multiplication secret sharing scheme', since the term multiplicative secret sharing already carries a different meaning and the term additive secret sharing, although technically correct, can be confusing in this context. Note that, due to the use of the multiplication secret sharing scheme, we abandon our usual notation denoting a secret-shared value v as $[\![v]\!]$ and specify the protocol explicitly for each player i.

In this protocol, $\otimes$ denotes element-wise product of vectors, i.e., if $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are vectors of length N, and $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$, then $c_i = a_i b_i$ for all $1 \leq i \leq N$. Similarly, by $\boldsymbol{a}^{-1}$ we denote the element-wise inverse of $\boldsymbol{a}$, such that $\boldsymbol{a} \otimes \boldsymbol{a}^{-1}$ is the all-ones vector of length N. To *exchange* a pair (a, b) with a player means sending a to that player and receiving b from that same player, communicated via a private channel.

The advantage of the multiplication secret sharing scheme is that multiplication of shares is a local operation. This completely eliminates the multiplicative complexity of Protocol 3.8. The only communication required in Protocol 3.15 is for sharing secret values (step 6) and revealing shared secrets (step 9), which gives linear communication complexity and constant round complexity. The reason this protocol cannot be used for completely obliviously sampling a random permutation is that the multiplication secret sharing scheme does not permit linear operations on secret-shared values to be computed locally. For partially known permutations the required linear operations are carried out by the player who knows the plaintext of his random roots, but in general this is not possible.

To efficiently multiply polynomials, the polynomials are evaluated at mul-

Protocol 3.15 RandomPKPermutation$(\boldsymbol{n}, \boldsymbol{e})$ for each player i

Input: • $\boldsymbol{L}$, where ℓ_j is the number of cards to be dealt to the player j
• $\boldsymbol{e}$, a list of $N = \sum_j \ell_j$ different points

1: $\boldsymbol{r}_i \xleftarrow{\$} (\mathbb{F}^* \setminus \{e_1, \ldots, e_n\})^{\ell_i}$ ▷ pairwise distinct
2: $f_i(x) \leftarrow \prod_{j=1}^{\ell_i}(x - r_{ij})$
3: $\boldsymbol{v}_i \leftarrow f_i(\boldsymbol{e})$
4: **for** each other player $j \in \{1, \ldots, q\} \setminus \{i\}$ **do**
5: $\quad \boldsymbol{m}_{ij} \xleftarrow{\$} (\mathbb{F}^*)^N$
6: $\quad$ exchange $(\boldsymbol{m}_{ij}, \boldsymbol{m}_{ji})$ with j
7: $\boldsymbol{w}_i \leftarrow \boldsymbol{v}_i \otimes \bigotimes_{\substack{j=1 \\ j \neq i}}^{q} (\boldsymbol{m}_{ij} \otimes \boldsymbol{m}_{ji}^{-1})$
8: **for** each other player $j \in \{1, \ldots, q\} \setminus \{i\}$ **do**
9: $\quad$ exchange $(\boldsymbol{w}_i, \boldsymbol{w}_j)$ with j
10: $\boldsymbol{t} \leftarrow \bigotimes_{j=1}^{q} \boldsymbol{w}_j$
11: $f \leftarrow$ Interpolate$(\boldsymbol{e}, \boldsymbol{t})$
12: find the (ordered) list of roots $\boldsymbol{s}$ of f
13: **if** $s_j = s_k$ for any $j \neq k$ **then**
14: $\quad$ abort with failure
15: **else**
16: $\quad$ **return** $(\boldsymbol{s}, \boldsymbol{r}_i)$

tiple points and the evaluations are multiplied. Unlike Protocol 3.8, in the case of Protocol 3.15 the players must agree on the set of points in which the polynomials are to be evaluated. This is why the set of evaluation points, $\boldsymbol{e}$, is explicitly given as input to the protocol. Since the multiplication secret sharing scheme does not allow zeroes to be shared, the protocol would fail if the polynomials have a root at any of the evaluation points. Because the players sample their random roots in plaintext, they can simply exclude the evaluation points to avoid this possibility. Similarly, each player also ensures that there are no duplicates within its own set of roots. Although the possibility of overlap between players' roots is not eliminated, this reduces the probability of the protocol failing.

To share a secret in the multiplication secret sharing scheme, a player sends each other player a uniformly random value and sets up its own share as the inverse of the product of all other shares, multiplied with the secret value. Because the information that is communicated with other players is independent

of the secret value, it is possible to generate this information pseudorandomly, thereby eliminating the need for any communication to share a secret. This means that, apart from setting up the necessary keys for pseudorandom secret sharing, the only communication required in Protocol 3.15 is revealing n shares in step 9.

Unlike Protocol 3.8, which inherits its security from the underlying schemes, Protocol 3.15 is not actively secure. The players can obviously exploit their ability to select their own roots to manipulate their positions in the permutation. This can be prevented by randomly permuting the recovered roots at the end of the protocol. This permutation can be carried out in the clear, but it is important that this permutation is unknown to the players at the time they select their roots and that it is sampled uniformly randomly. This can be achieved with straightforward commit-and-reveal techniques. Further requirements against malicious players are that the players check that there are exactly N roots and that these contain the roots they contributed to the product polynomial.

3.9 Playing Card Games

Perhaps the most well-known example of a random permutation is a shuffled deck of playing cards. Our random permutation protocol is suitable to shuffle a virtual deck of cards for, e.g., online poker games, and, in fact, the general permutation protocol was originally developed from more a restricted protocol that was closer to the RandomPKPermutation protocol, specifically to satisfy the requirements of virtual poker games. Compared to the RandomPKPermutation protocol, these requirements can be summarized loosely as the ability to shuffle a full deck of cards such that each card can be dealt *separately* and *dynamically* as the game progresses.

In this section we will briefly and informally outline how the protocols described in this chapter can be applied to implement secure card games. The techniques described in this section may also find application in general secure computation.

In physical card games, each card has a single, well-defined location, often part of a pile of cards or a player's hand. We will restrict our attention to games in which each player is able to know the location of each card, disregarding games involving sleight-of-hand. We can simulate such card games by keeping track of the location of each card. Even if the location of each card is known to all players, the face value may not be. This is simulated by secret sharing the

face value of the card. The secret-shared face value of a card can be selectively revealed to players when appropriate.

For simplicity, we will assume that, at least, the universe of cards is publicly known, i.e., every player knows all cards that can occur during a game. For example, for a game of bridge the players know all cards of the standard 52-card deck and no cards from outside of this deck, such as, e.g., a joker, the ace of coins from an Italian deck, or a custom card secretly mixed in by one of the players, can occur. However, this is not a major restriction and, in fact, games with custom cards can be implemented using, for example, a combination of anonymous publishing, symmetric encryption and secret-shared encryption keys as the face value of cards.

The location of each card, i.e., a player's hand or a pile of cards, is known to all players at all times. It is precisely when the face value of a card is known to some player, but should not be, that we can apply our secure random permutation and rearrangement protocols to all cards in the same location to uniformly randomly redistribute the face values of all cards in the particular location. Most actions that are part of playing a game of cards can then be simulated as alterations to the location of cards, revealing the secret-shared face values of cards, and securely shuffling cards in the same location.

Note that, under certain circumstances, the face values can be shuffled more efficiently than using the full shuffle protocol. For example, if the list of face values to be shuffled is publicly known, and all face values are (or can safely considered to be) distinct, the list can be randomized using just a secure random permutation and the secure rearrangement is superfluous. In that case a public mapping from the random roots representing the secret-shared permutation to the face values can be established without additional communication. Also in the case of a single player shuffling their hand, more efficient solutions exists, by letting the player choose the face values of the cards they are holding, or the permutation to apply to their hand. Note that for these solutions the player should prove that the modification of their hand is legal to obtain an actively secure protocol.

There is one kind of manipulation of cards in games that the author can think of that is not covered by the above and that is inserting a card in a secret position in a pile and generalizations thereof. This is a slightly more complicated operation, which can be implemented by virtually putting the card on top of the pile and then letting the player secret share the permutation matrix, S, representing the linear operation of taking the top card of the pile and inserting it at a given position. The matrix S that represents the operation

Protocol 3.16 ConfirmUnitVector($[\![\boldsymbol{u}]\!]$)

Input: $\boldsymbol{u}$, a vector of length n

1: $\boldsymbol{r} \xleftarrow{\$} (\mathbb{F}^*)^n$
2: $p(X) \leftarrow \prod_{i=1}^{n}(X - r_i)$
3: $[\![z]\!] \leftarrow p(\boldsymbol{r} \cdot [\![\boldsymbol{u}]\!])$
4: $z \leftarrow \text{reveal}[\![z]\!]$
5: **if** $z \neq 0$ **then**
6: abort with failure

of moving the top card of a pile of n cards to position p is given by

$$s_{i,j} = \begin{cases} 1 & \text{if } i < p \wedge j = i + 1; \\ 1 & \text{if } i = p \wedge j = 1; \\ 1 & \text{if } i > p \wedge j = i; \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

This approach lets the players keep the position at which the card is inserted secret, but this approach is obviously not secure against a malicious player, as the player can secret share an arbitrary matrix, which may enable them to manipulate the deck in an arbitrary manner. To make this approach actively secure, first note that it is sufficient for the player to only secret share the first column of this matrix, as the remaining columns are a linear function of the first: for $j > 1$

$$s_{i,j} = \begin{cases} 1 - \sum_{\ell=i}^{n} s_{i,1} & \text{if } j = i; \\ 1 - \sum_{\ell=1}^{i} s_{i,1} & \text{if } j = i + 1; \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

For an actively secure protocol, we then only need to ensure that the player secret shares a unit vector rather than an arbitrary vector. There are two obvious ways to achieve this. First, we could use an actively secure protocol to sample a random unit vector, reveal it to the player and let the player publicly announce a rotation that permutes the random unit vector into the desired unit vector. Alternatively, the player can provide a proof that the vector it has shared is a unit vector. There exists a very straightforward way to prove this with MPC. Protocol ConfirmUnitVector (Protocol 3.16) allows the players to prove that a given secret-shared vector is a unit vector.

If the given vector, $\boldsymbol{u}$ is a unit vector, the protocol terminates normally. If $\boldsymbol{u}$ is the null vector, the protocol is guaranteed to abort. The probability of the

protocol completing without abort, if u is neither a unit vector, nor the null vector, is $n/|\mathbb{F}|$. If necessary, the protocol can be repeated multiple times in parallel.

One of the things that we have not shown to be possible using our shuffle protocol is to produce *imperfect* shuffles for card games that require that the deck not be randomized "too much" between games. However, since we have no good description of the various classes of distributions of suitable permutations for such games, we deem this issue mostly outside the scope of secure computation.

Chapter 4

Secure Multi-Pivot Quicksort

4.1 Introduction

In this chapter we will describe a protocol for securely sorting lists of secret-shared values. Hamada et al. [HKI+12] show that any *comparison sort* algorithm is data-oblivious (meaning that the algorithm's execution time and branching behavior are independent of secret input data) if the input list is first securely shuffled and all elements of the list are pairwise distinct. They also show that if the input does not satisfy the distinctness assumption, that an oblivious sort protocol can be obtained by obliviously associating a unique label with each value and obliviously ordering two values by their respective labels in case the values are identical.

Our secure sort protocol combines the oblivious sort technique of [HKI+12] with our secure shuffle protocol of Chapter 3. Although the technique for secure sorting presented in this chapter is not novel, we believe our treatment of how to handle duplicate values is more comprehensive than [HKI+12]. The main contribution of this chapter is, in fact, our variant of the quicksort algorithm, which allows for a trade-off between the complexities in terms of the number of secure comparisons operations and the number of secure comparison rounds. In particular, our multi-pivot quicksort allows for choosing the number of comparison rounds independently from the size of the input list.

Related Work

The prevalent method for oblivious sorting is through the use of sorting networks. Sorting networks are circuits composed of compare-and-exchange oper-

ators, which are operators with two input wires and two output wires such that the first output wire carries the minimum of the inputs and the second output wire the maximum. Given a protocol for secure comparison, secure compare-and-exchange operators can be efficiently instantiated. Since the structure of a sorting network for a given input size, which we consider publicly known in this chapter, is independent of the input data, and compare-and-exchange operators can be instantiated securely and efficiently, sorting networks are suitable as a basis for secure sorting. The relevant properties of a sorting network determining the complexity of the secure sorting protocol derived from it are the number of compare-and-exchange operators, which directly translates to the number of secure comparisons, and the depth of the network, since the round complexity scales linearly with the depth of the sorting network.

Ajtaj, Komlós, and Szemerédi [AKS83] show that there exist sorting networks of depth $O(\log N)$, where N is the size of the input. Such networks therefore have $O(N \log N)$ compare-and-exchange operators. This result, which has become known as the AKS sorting network, asymptotically matches the optimum for comparison sort, however, their result is of theoretical interest only, as the hidden constant in the big O is very large. Zig-zag sort, proposed by Goodrich [Goo14], also features $O(N \log N)$ compare-and-exchange operators, without the impractical hidden constant of the AKS sorting network. However, since the depth of zig-zag sort is also $O(N \log N)$, this network is also not considered of practical interest. Instead, one of several sorting networks with depth $O(\log^2 N)$ and $O(N \log^2 N)$ compare-and-exchange operators, such as Batcher's odd-even mergesort or bitonic sort [Bat68], are used in practice.

Leighton and Plaxton propose a relaxation of sorting networks that correctly sort the *majority* of the input sequences, but fail on some inputs [LP98]. Thus, with the appropriate randomization of the input, their construction can be viewed as a Monte Carlo algorithm. The depth of their circuit is about $7.44 \log_2 N$. Goodrich obtains similar results with randomized Shellsort, in which not the input sequence, but the sorting network itself is randomized [Goo11]. The depth of randomized Shellsort is $6 \log_2 N$. Furthermore, the algorithm admits a simple description. In both of these, each element is compared to exactly one other element in each round, so both of these algorithms clearly feature $O(N \log N)$ comparison complexity. Note that, in contrast to the technique discussed in this chapter, these approaches would make use of a publicly sampled random permutation and secure comparison with private output. The concrete complexities of the approaches by Leighton and Plaxton, and Goodrich are very low. Our approach still compares favorably, however, and permits more flexibility in that the comparison depth can be chosen freely.

As an alternative to using sorting networks, the method of shuffling the input and sorting based on secure comparison with public output, which we employ in this chapter, was first described by Hamada et al. [HKI+12]. In this work the sorting algorithm used is ordinary quicksort. Later works using the same technique include [HICT14] and [BLT14], which explore the application of secure radixsort. We improve on these results in two ways: our method of shuffling the input list, described in Chapter 3, is more efficient, and we generalize the quicksort algorithm to allow for a trade-off between the number of comparison operations and the number of rounds of comparisons.

In this chapter we describe a novel multi-pivot quicksort algorithm and analyze its complexity. Note that similar variants of quicksort with 2 or more pivots have been studied in practical implementations, because these may alleviate efficiency problems arising in concrete implementations of ordinary quicksort due to cache misses and difficulties with branch prediction. For these reasons, 2-pivot quicksort was introduced into the Java standard library for sorting primitives, for example. Our motivation for studying multi-pivot quicksort is to take advantage of the (apparent) parallelism that is inherent in secure multiparty computation and we therefore focus on a different notion of efficiency. For this reason we do not include any existing results on variants of quicksort using multiple pivots in this section. Furthermore, the aforementioned results are typically discussed on, e.g., implementers' mailing lists, rather than reported in academic literature.

4.2 Multi-Pivot Quicksort

In this section we will describe our multi-pivot quicksort algorithm. The algorithm is a comparison sort algorithm, meaning that it determines the ordering of the input list using only comparison operations between pairs of values. As shown by Hamada et al. [HKI+12], if the input list is obliviously shuffled, then any comparison sort algorithm with the comparison operators instantiated by a secure comparison protocol with public output forms an oblivious sorting protocol provided that all elements in the input list are pairwise distinct.

Because security only follows if the input list is randomly ordered and all elements are unique, we design an algorithm that is efficient under these conditions, i.e., our algorithm does not have to be efficient if either condition does not hold and, in fact, in that case, our algorithm can exhibit quadratic complexity, which is the worst-case behavior of quicksort.

We have chosen to base our algorithm on quicksort, because it both features

expected $O(N \log N)$ comparison complexity and expected $O(\log N)$ depth complexity for inputs of length N. The comparison complexity is simply the number of comparisons that need to be carried out to determine the order of the input list. The depth complexity is the number of rounds of comparisons that need to be carried out, where comparisons that do not depend on the outcome of each other are executed in parallel. Both the comparison complexity and depth complexity have small and practical concrete constants. However, the typical execution of quicksort starts with a few rounds in which many comparisons are performed, followed by a relatively large number of rounds with only few comparisons.

As a quick recap, the basic quicksort algorithm sorts a list by picking the first element of the list as pivot, then comparing each other element to this pivot and separating those into two list of elements respectively smaller than and greater than (or equal to) the pivot. Quicksort is then applied recursively on the two lists as long as they are not empty, which returns a sorted list of all elements smaller than the pivot and a sorted list of all elements greater than or equal to the pivot. Finally, these two sorted lists are concatenated with the pivot in between, resulting in the sorted list of the input. Although the worst case running time of quicksort is quadratic in the input size, it achieves $O(N \log N)$ in the average case and performs fewer comparisons in practice than algorithms with worst case $O(N \log N)$ running time. If the input list is randomly ordered and all its elements are unique, we can expect quicksort to behave as in the average case. Otherwise it may exhibit worst case performance with $O(N^2)$ comparison complexity and $O(N)$ depth complexity.

Note that the all comparisons made against a single pivot can be carried out in parallel, which enables us to use quicksort as a base for a protocol with low round complexity. Although using quicksort without modifications would already give us expected logarithmic round complexity the situation is not ideal. In a typical run of quicksort at some point most of the list is sorted, but some unsorted 'pockets' still exist. Sorting these requires only few comparisons, but relatively many rounds. Furthermore, quicksort gives us an expected round complexity, but no guarantees.

Our algorithm is a variant of quicksort in which the round complexity is fixed in advance. To achieve this property, we have to perform a slightly higher number of comparisons. As with the permutation and shuffling protocols, the user is free to choose the round complexity, and from this the communication and computation complexities follows. We fix the round complexity of quicksort by compressing multiple rounds into one. Instead of taking a single pivot, we take multiple pivots at the same time and make all comparisons necessary

to determine these pivots' position in the sorted list. The additional pivots selected this way can be seen as those pivots that would have been selected by ordinary quicksort in future rounds. Because we do not yet have information about the relation between the additional pivots, we have to compare them to *all* remaining elements, rather than first reducing the size of the list, giving rise to a higher expected number of comparisons compared to ordinary quicksort.

An informal description of the multi-pivot quicksort algorithm is displayed in Algorithm 4.1. In addition to the input list, the multi-pivot quicksort algorithm takes a (maximum) comparison depth as input. The number of comparison rounds is guaranteed not to exceed the given maximum depth. If the maximum comparison depth is equal to one, then the only possibility to sort the input list with certainty is by comparing every single pair of elements in parallel. Otherwise, the optimum number of pivots, p, i.e., the number of pivots leading to the smallest expected number of comparisons is computed numerically. Then, the list is partitioned into two lists of p pivots and $N-p$ non-pivots and each pivot is compared to each non-pivot in parallel. Note that the pivot and non-pivot partitions are symmetrical; the term pivot is only used in keeping with the ordinary quicksort algorithm. The list is then reordered according to the outcome of the comparisons. It is very unlikely that the list is completely sorted at this point, so the list consists of bins of elements that, although the bins are in the correct position in the list, the order of the elements within each bin is completely unknown with respect to the other elements in the bin. Therefore, as a final step, each bin is sorted recursively and in parallel by multi-pivot quicksort, where the maximum comparison depth has been decreased by one.

By similarity to ordinary quicksort, it should be clear that multi-pivot quicksort sorts the input list correctly. It is also clear that the multi-pivot quicksort algorithm has depth complexity at most equal to the given maximum comparison depth. Since multi-pivot quicksort is a comparison sort algorithm, the algorithm can be instantiated as a secure protocol if the input list is obliviously shuffled and all elements are pairwise distinct.

The optimal number of pivots, computed on line 5, can be computed numerically. Although the complexity of computing this number naïvely is asymptotically greater than the complexity of sorting itself, we will neither give a procedure for computing the optimal number of pivots, nor consider the computational cost of doing so in our analysis. Our reason for this is that the computation of the optimal number of pivots is not a secure computation, and we consider the arithmetic operations needed for this computation orders of magnitudes faster than the secure comparisons needed for sorting. Furthermore,

Algorithm 4.1 MPQSort($\boldsymbol{v}, d$)

Input: • $\boldsymbol{v}$, a list of N values
• d, the maximum comparison depth

1: **if** $d = 1$ **then**
2: pairwise compare all elements in $\boldsymbol{v}$
3: reorder all elements so that the list is ordered
4: **else**
5: determine the optimal number of pivots p
6: pairwise compare each of the first p elements of $\boldsymbol{v}$ to each of the last $N - p$ elements of $\boldsymbol{v}$
7: reorder all elements in the list into bins in increasing order
8: for each bin $\boldsymbol{v}'$ apply MPQSort($\boldsymbol{v}', d - 1$) in parallel

the computation of the optimal number of pivots can make use of precomputed tables and well-known techniques for search problems, such as dynamic programming and approximation, all of which are outside the scope of this work.

Despite not explicitly describing a method for determining the optimal number of pivots in each round, we must determine the comparison complexity of multi-pivot quicksort, which depends on the pivot selection strategy. We bound the comparison complexity by describing a variant of the algorithm, FixedMPQSort. Although the algorithm, displayed in Algorithm 4.2, is described iteratively, rather than recursively, and it returns the permutation that would sort the input list, rather than directly sorting the list, it essentially performs the same as MPQSprt, apart from the pivot selection strategy. The iterative presentation allows to more clearly analyse the complexity and for a more direct translation to secure computation. Returning the permutation that sorts the input list allows us to compactly describe how to securely sort complex data structures by a sort key in Section 4.4

FixedMPQSort differs from MPQSort in its pivot selection strategy in two ways. First, rather than determining the number of pivots selected in each round dynamically based on the size of the remaining work, the number of pivots selected in each round is fixed and given as input to the algorithm. This means that the number of pivots to select in each bin cannot be determined independent from the other bins. The second way in which the pivot selection strategy of FixedMPQSort differs from MPQSort is that the elements of the input list are not explicitly partitioned into bins depending on the outcome

Algorithm 4.2 FixedMPQSort$(\boldsymbol{v}, \pi)$

Input: • $\boldsymbol{v}$, a list of N values
• π, a pivot selection sequence

1: $\boldsymbol{p} \leftarrow 0^N$
2: **for** $\ell \leftarrow 1$ **to** $|\pi|$ **do**
3: $\quad \boldsymbol{d} \leftarrow 0^N$
4: $\quad C \leftarrow 0^{N \times N}$
5: $\quad$ **for** $i \leftarrow \pi_{\ell-1} + 1$ **to** π_ℓ **do**
6: $\quad\quad$ **for** $j \leftarrow i + 1$ **to** N **do**
7: $\quad\quad\quad$ **if** $p_i = p_j$ **then**
8: $\quad\quad\quad\quad c_{ij} \leftarrow v_j \overset{?}{<} v_i$
9: $\quad\quad\quad\quad c_{ji} \leftarrow 1 - c_{ij}$
10: $\quad$ **for** $i \leftarrow \pi_{\ell-1} + 1$ **to** π_ℓ **do**
11: $\quad\quad d_i \leftarrow \sum_j c_{ij}$
12: $\quad$ **for** $j \leftarrow \pi_\ell + 1$ **to** N **do**
13: $\quad\quad d_j \leftarrow \max_i((1 - c_{ij})d_i) + 1$
14: $\quad \boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{d}$
15: **return** $\boldsymbol{p}$

of previous comparisons. Instead, simply a predetermined number of elements that have not been selected as pivots previously are selected from the global list and compared to the other remaining elements in their respective bins, regardless of the size of these bins.

Because the bins form non-overlapping sub-problems, the strategy of selecting pivots globally, rather than per bin, cannot be expected to perform more efficiently than the strategy of MPQSort. Furthermore, any fixed sequence of number of pivots to select in each round can at best be expected to lead to the same comparison complexity as the optimal such sequence. We do not attempt to find the optimal sequence, and instead show that simply using an exponential sequence leads to expected $O(N \log N)$ comparison complexity and $O(\log N)$ depth complexity.

Algorithm 4.2 is a far cry from the elegant description ordinary quicksort permits. Furthermore, the complexity of local operations, as described, is strictly $O(N^2)$. However, this description clearly indicates which secure comparison are needed and when these are to be carried out. To instantiate this algorithm as a secure protocol in the shuffle-and-sort setting, only the compar-

ison on line 8 has to be instantiated with a secure comparison protocol. Since the output of the secure comparison is public, none of the state of the algorithm has to be kept secret, apart from the input list, which is given in secret-shared form. Note that the apparent $O(N^2)$ complexity of local operations is only for the sake of simplified presentation. In reality, these operations can be implemented using more efficient data structures and the complexity of these local operations scales linearly with the number of comparisons performed, which is one of the complexity measures of interest.

In addition to the input list which is to be sorted, Algorithm 4.2 also takes a pivot selection sequence, π as input, which determines the number of pivots selected in each round. The pivot selection sequence is cumulative, in that the ith term of this sequence describes how many pivots should have been selected as pivot in total after the ith round. For convenience, we let $\pi_0 = 0$. Because all but one element has to be selected as pivot to ensure the list is completely sorted, we require that $\pi_k = N - 1$, where $k = |\pi|$ is the number of comparison rounds.

Theorem 4.1. *The expected number of comparisons made by Algorithm 4.2 when sorting a list of N pairwise distinct elements using pivot selection sequence π is*

$$\sum_{i=1}^{|\pi|} (\pi_i - \pi_{i-1}) \frac{2N - \pi_i - \pi_{i-1} - 1}{\pi_{i-1} + 2},$$

where the expected value is taken over all permutations of the input list.

Proof. Because we assume the elements in the input list are pairwise distinct, each element has a well-defined position in the ordered list consisting of the same elements. Consider the pair of elements in the unordered list which are at positions a and b in the ordered list. Let $d = |b - a| + 1$ be the distance between these two elements in the ordered list, including the elements themselves. As the protocol is executed, these elements will be compared to each other if and only if either of them is selected as a pivot in a particular round *and* none of the elements that are on positions between a and b, inclusive, in the ordered list have been selected as a pivot in any previous round. We can express the probability that two elements at distance d will be compared in the ith round as

$$\frac{\binom{d}{0}\binom{N-d}{\pi_{i-1}}}{\binom{N}{\pi_{i-1}}} \left(1 - \frac{\binom{2}{0}\binom{N-\pi_{i-1}-2}{\pi_i - \pi_{i-1}}}{\binom{N-\pi_{i-1}}{\pi_i - \pi_{i-1}}} \right). \tag{4.1}$$

Recall that

$$\frac{\binom{\hat{n}}{\hat{k}}\binom{n-\hat{n}}{k-\hat{k}}}{\binom{n}{k}} \tag{4.2}$$

is the probability of drawing $\hat{k}$ elements from a subset of size $\hat{n}$ when drawing k elements from a population of size n. The left hand fraction in Equation (4.1) then represents the probability of not having selected any pivot in the range of distance d in any round preceding the ith, and the right hand fraction is the probability of not selecting either end point as a pivot in this particular round. Taking the complement of the latter and multiplying by the former yields the probability of not having selected any pivot in the range of distance d in previous rounds and selecting at least one of the end points in round i. Note that this probability does not depend on the final position of the elements, only on their distance in the sorted list.

To find the total expected number of comparisons made in the ith round, we sum Equation (4.1) over all pairs of elements. As there are $N - d + 1$ pairs at distance d apart with d ranging from 2 to N, this can be expressed as

$$\sum_{d=2}^{n}(N-d+1)\frac{\binom{d}{0}\binom{N-d}{\pi_{i-1}}}{\binom{N}{\pi_{i-1}}}\left(1-\frac{\binom{2}{0}\binom{N-\pi_{i-1}-2}{\pi_i-\pi_{i-1}}}{\binom{N-\pi_{i-1}}{\pi_i-\pi_{i-1}}}\right). \tag{4.3}$$

We can simplify this expression by applying elementary manipulations and the well-known fact that $\sum_{i=0}^{n}\binom{i}{k} = \binom{n+1}{k+1}$. First, consider the following sum, as

the right hand factor of Equation (4.3) is independent of d.

$$\begin{aligned}\sum_{d=2}^{N}(N-d+1)\frac{\binom{d}{0}\binom{N-d}{\pi_{i-1}}}{\binom{N}{\pi_{i-1}}} &= \frac{\pi_{i-1}+1}{\binom{N}{\pi_{i-1}}}\sum_{d=2}^{N}\binom{N-d+1}{\pi_{i-1}+1}\\ &= \frac{\pi_{i-1}+1}{\binom{N}{\pi_{i-1}}}\binom{N}{\pi_{i-1}+2}\\ &= \frac{\pi_{i-1}+1}{\binom{N}{\pi_{i-1}}}\binom{N}{\pi_{i-1}}\frac{(N-\pi_{i-1})(N-\pi_{i-1}-1)}{(\pi_{i-1}+1)(\pi_{i-1}+2)}\\ &= \frac{(N-\pi_{i-1})(N-\pi_{i-1}-1)}{\pi_{i-1}+2}.\end{aligned}$$

Then, since

$$\binom{N-\pi_{i-1}-2}{\pi_i-\pi_{i-1}} = \frac{(N-\pi_i)(N-\pi_i-1)}{(N-\pi_{i-1})(N-\pi_{i-1}-1)}\binom{N-\pi_{i-1}}{\pi_i-\pi_{i-1}},$$

we can combine these two factors of Equation (4.3)

$$\begin{aligned}&\frac{(N-\pi_{i-1})(N-\pi_{i-1}-1)}{\pi_{i-1}+2}\left(1-\frac{(N-\pi_i)(N-\pi_i-1)}{(N-\pi_{i-1})(N-\pi_{i-1}-1)}\right)\\ =\ &\frac{(N-\pi_{i-1})(N-\pi_{i-1}-1)-(N-\pi_i)(N-\pi_i-1)}{\pi_{i-1}+2}\\ =\ &(\pi_i-\pi_{i-1})\frac{2N-\pi_i-\pi_{i-1}-1}{\pi_{i-1}+2}.\end{aligned}$$

The total expected number of comparisons in all rounds is then simply the sum of this number over all rounds, which proves the theorem. □

Theorem 4.2. *Algorithm 4.2 can be used to sort a list of n pairwise distinct elements in $O(\log_M N)$ rounds of comparisons using expected $O(NM\log_M N)$ comparison operations, where M can be chosen freely and the expected value is taken over all permutations of the input list.*

Proof. Let $\pi_i = M^i$ for all i, but $\pi_k = N-1$, where $k = \lceil \log_M N \rceil$. Then π is a valid pivot selection sequence of length k, i.e., it is non-negative, increasing,

and the final term equals $N-1$. Again, for convenience, we let $\pi_0 = 0$. Then, applying the result of Theorem 4.1 to this pivot selection sequence results in an expected number of comparisons of

$$\begin{aligned}\sum_{i=1}^{k}(\pi_i - \pi_{i-1})\frac{2N - \pi_i - \pi_{i-1} - 1}{\pi_{i-1}+2} &< NM + \sum_{i=2}^{k} 2N\frac{M^i - M^{i-1}}{M^{i-1}} \\ &= NM + 2(k-1)N(M-1) \\ &< NM + 2(M-1)N\log_M N.\end{aligned}$$

As $k = \lceil \log_M N \rceil$ is the number of rounds, this proves the theorem. Furthermore, this shows that the constants in the big O notation are small. □

4.3 Dealing with Duplicates

So far we have assumed that the elements in the input list are distinct. In this section we will consider the issues that arise if the input list holds duplicate values. We will attempt to give a complete overview of the problems that arise and how to resolve these efficiently. Note that this issue was already identified and mitigated in the work by Hamada et. al. [HKI+12]. We include this section for completeness and believe that our discussion of the issue is more comprehensive. In this section we reference both the multi-pivot quicksort algorithm and the secure protocol derived from it using the shuffle-and-sort technique for which we will give an explicit description in Section 4.4.

The case in which there are duplicate values in the input list has negative implications for both the comparison complexity of the multi-pivot quicksort algorithm, and the security of the secure protocol derived from it using the shuffle-and-sort technique. Because the shuffle-and-sort technique only permits the use of inequality comparison (smaller than, etc.), but not equality, *any* sorting algorithm is susceptible to the same problem. When sorting lists containing duplicates, the (public) placement of the pivots each round will reveal information about the number of duplicates, as well as about their value relative to the other elements. Because the comparison operation does not test for equality, when a particular value occurs many times in the input list, that particular value, selected as pivot, would be ordered below all other instances of that value as non-pivots (or pivots at a higher position in the input list). This means that all remaining such values are sorted into the same bin, which would then recursively suffer the same problem. The number of comparisons needed to sort the list would in this case therefore tend to the worst case of

$O(N^2)$. The importance of this issue depends on the application and the benefits of preventing this kind of leakage and loss of performance must be weighed against the additional complexity of the protocol.

To show that duplicate values constitute a problem for any comparison sort algorithm, consider an input list consisting only of pairwise distinct elements. Then the sequence or *trace* of all comparison outcomes obtained when sorting an input list uniquely determines the permutation of the input list and vice versa (for any fixed sequence of random coins for randomized algorithms). In contrast, if the input list consists of all identical elements, then the outcomes of comparison operations will be independent of the permutation of the input list. For lists of sufficient length, this would allow us to distinguish lists of identical elements from lists of pairwise distinct elements.

The performance problem that repeated elements cause for quicksort and variants thereof is known as the Dutch national flag problem. Although the problem is well understood, we cannot make use of efficient solutions that rely on comparison operations that distinguish equality, because publicly announcing equality results from secure comparisons would directly reveal the number and relative value of these duplicate elements. The performance problem can be resolved using randomness by arranging the comparisons in such a way that we can expect half of all elements equal to a pivot to be ordered below it and half above. Simply arranging for half of the comparisons involving a particular pivot to be "smaller than" and the other half "smaller than or equal" would suffice, since the input list is randomly permuted.

This would not resolve the security problem, however, since in this case the trace of the comparison outcomes is still independent of the permutation of the input list. Even determining the type of comparison (smaller than versus smaller than or equal) obliviously and independently randomly would not lead to a secure solution as the position of pivots within the ordered list would follow a binomial distribution, rather than the uniform distribution for input lists of pairwise distinct elements. Furthermore, when comparing multiple values to multiple pivots, all of which are identical, the results are likely to be inconsistent and thereby reveal equality of these elements with certainty.

To resolve this issue securely, we must ensure that the public comparison results place the pivots at uniformly random positions within the elements of equal value. We must therefore break ties in a collective and consistent manner. A straightforward solution is to attach a secret-shared tie breaker *label* to each input element. Whenever two elements are compared, the labels are also compared. The comparison of the two elements is made public, unless they are equal, in which case the comparison result of the tie breaker values

is obliviously selected instead. This solution will increase the number of secure operations and rounds by at most a constant factor.

The most straightforward way to efficiently arrange these tie breaker labels in an obviously consistent manner is to assign a secret-shared counter to each element of the input list. The input list and counter list are then rearranged using the same random permutation. Using counters ensures that no two elements of the labeled input list are equal, making the process of sorting any list indistinguishable from sorting a list in which all values are unique. A possible added advantage is that this protocol performs a stable sort, which can be relevant when sorting composite objects by some key. If a stable sort is undesirable, then the list of counters should be shuffled using a different random permutation than the input list. Alternatively, the labels can be chosen randomly, with sufficient entropy. The advantage of this would be that the labels do not need to be rearranged, however, this requires a greater range of values to ensure the probability of duplicates occurring in the labels is sufficiently small, which in turn may require a more complex secure comparison protocol. Note that using counter labels does not increase the round complexity, as sampling the additional random permutation, if necessary, and the rearrangement operation can be carried out in parallel to shuffling the input list.

It is important to note that any solution involving labels introduces a hidden dependency on the size of the input list, N, into the complexity of the secure comparison protocol. Only under the assumption that N is within a constant factor of the size of the allowed range for comparable values can we argue that the overhead introduced by comparing labels is at most a constant factor. Our claim that our protocol achieves an expected $O(N \log N)$ secure comparisons remains valid, regardless. However, the complexity of the comparison operations themselves carry a hidden dependency on N if the assumption were not to hold. We consider it reasonable to assume that the number of distinct values that can be represented and compared is not much smaller than the size of the input list for general comparison sort to be naturally applicable. For the problems of sorting lists far longer than the allowed range of values, specialized sorting protocols should be considered.

If the range of allowed values in the input list is smaller than the range of values allowed by the comparison operation by a factor of at least $N + 1$, we may eliminate the information leakage issue by combining the input values and labels into a single secret-share. The advantage of this variant is that to compare two labeled values, we only need to perform the ordinary comparison protocol once, rather than performing a three-valued comparison protocol on the input values in addition to a comparison on the label and the multiplications

Protocol 4.3 SecureMPQSort($[\![\boldsymbol{v}]\!], d,$ Precomp)

Input: • $\boldsymbol{v}$, a list of N values
• d, the maximum comparison depth
• Precomp, a secure protocol preparing (value, label) pairs

1: $(\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!]) \leftarrow$ RandomPermutation(N)
2: $[\![\hat{\boldsymbol{v}}]\!] \leftarrow$ Rearrange($[\![\boldsymbol{v}]\!], (\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$)
3: $[\![\hat{\boldsymbol{\ell}}]\!] \leftarrow$ Rearrange($(i)_{i=0}^{N-1}, (\boldsymbol{r}, [\![\hat{\boldsymbol{r}}]\!], [\![T]\!])$)
4: **for** $i \leftarrow 1$ **to** N **do**
5: $\quad [\![w_i]\!] \leftarrow$ Precomp($\hat{v}_i, \hat{\ell}_i$)
6: $\sigma \leftarrow$ MPQSort′($[\![\boldsymbol{w}]\!], d$)
7: $[\![\boldsymbol{r}']\!] \leftarrow \sigma[\![\hat{\boldsymbol{r}}]\!]$
8: $[\![T']\!] \leftarrow$ PolTree($(x - [\![r'_i]\!])_{i=1}^{N}$)
9: **return** $(\boldsymbol{r}, [\![\boldsymbol{r}']\!], [\![T']\!])$

needed to obliviously select the results. Especially in this case, the increase in complexity of the comparison protocol should not be ignored.

4.4 The Secure Sorting Protocol

For completeness, we briefly describe the full secure protocol for sorting a secret-shared list of values. We give a fairly general version of the protocol containing several optional features, e.g., stability, that can be omitted or adapted as desired. The secure protocol SecureMPQSort is displayed in Protocol 4.3.

Here MPQSort′ denotes the secure protocol obtained by instantiating the comparison operator in the multi-pivot quicksort algorithm with a secure comparison protocol with public output. The MPQSort′ protocol is assumed to return the permutation which orders the input list, as we have also described for the FixedMPQSort algorithm (Algorithm 4.2), but unlike the informal description of the MPQSort algorithm (Algorithm 4.1). This permutation is computed publicly and composed with the secret permutation used to permute the input list. The result is a secret permutation that orders the original, unshuffled input list. For simply sorting an input list, this is unnecessary, but this allows to obliviously sort compound data by some key, in which only the key is input to SecureMPQSort, and the resulting secret permutation can be used to rearrange the compound data using the Rearrange protocol.

The protocol starts by obliviously sampling a uniformly random permutation (line 1). This permutation is applied, in parallel, to the input list (line 2),

as well as the list of counter labels (line 3). Using counter labels in this manner will lead to a stable sort. For alternatives to using counter labels see Section 4.3. Then, the values and labels that have been rearranged on lines 2 and 3 are paired and an (optional) secure precomputation protocol, Precomp, is run on each pair in parallel (line 5). This precomputation served to convert the (value, label) pairs into a representation which permits efficient comparison. For example, the precomputation can be a decomposition into medium-sized integers so that the comparison can be carried out very efficiently by combining the medium-sized integer comparison protocol of [ABSdV19] with the constant round bitwise less-than protocol of [DFK$^+$06]. After the precomputation, the permutation which orders the shuffled input list is determined using the multi-pivot quicksort protocol, in which the comparison operations have been instantiated by a secure comparison protocol (line 6). Finally, the secret permutation that orders the *unshuffled* input list is computed (lines 7 and 8) and this permutation is returned. The secret permutation that sorts the input list is computed in the same representation as the secret permutations described in Chapter 3 and is therefore compatible for further computation with the techniques described therein.

Chapter 5

Secure Moore–Penrose Pseudoinverse

Note. *This chapter is based on* A Practical Approach to the Secure Computation of the Moore–Penrose Pseudoinverse over the Rationals *[BdV20] with the following major modifications.*

- *The need for the preconditioner of Section 5.4 to be invertible for it to be removable has been relaxed. Although the improvement to the success probability of the protocol is not significant, this relaxation of the requirement has been included as it allows for a slightly more elegant proof of correctness of the protocol.*
- *The appendix of the full version of the paper [BdV19] containing the complexity proof of the complete protocol has been incorporated into the main text.*
- *An error in our complexity analysis has been corrected.*

Solving linear systems of equations is a universal problem. In the context of secure multiparty computation (MPC), a method to solve such systems, especially for the case in which the rank of the system is unknown and should remain private, is an important building block.

We devise an efficient and *data-oblivious* algorithm (meaning that the algorithm's execution time and branching behavior are independent of all secrets) for solving a bounded integral linear system of unknown rank over the

rational numbers via the Moore–Penrose pseudoinverse, using finite-field arithmetic. I.e., we compute the Moore–Penrose pseudoinverse over a finite field of sufficiently large order, so that we can recover the rational solution from the solution over the finite field. While we have designed the algorithm with an MPC context in mind, it could be valuable also in other contexts where data-obliviousness is required, like secure enclaves in CPUs.

Previous work by Cramer *et al.* [CKP07], proposes a constant-rounds protocol for computing the Moore–Penrose pseudoinverse over a finite field. The asymptotic complexity (counted as the number of secure multiplications) of their solution is $O(m^4 + n^2 m)$, where m and n are the dimensions of the linear system, with $m \leq n$. To reduce the number of secure multiplications, we sacrifice the constant-rounds property and propose a protocol for computing the Moore–Penrose pseudoinverse over the rational numbers in a linear number of rounds, requiring only $O(m^2 n)$ secure multiplications.

To obtain the common denominator of the pseudoinverse, required for constructing an integer-representation of the pseudoinverse, we generalize a result by Ben-Israel for computing the squared volume of a matrix. Also, we show how to precondition a symmetric matrix to achieve generic rank profile while preserving symmetry and being able to remove the preconditioner after it has served its purpose. These results may be of independent interest.

5.1 Introduction

Motivated by the goal of performing elementary statistical tasks such as linear regression *securely*, we revisit the topic of secure linear algebra. In this chapter, "securely" refers to *secure multiparty computation* (MPC) [CDN15], however, our results might be of use in other settings as well, for example, for mitigating certain side-channel attacks in trusted execution environments in CPUs.

Secure linear algebra goes back to the work of Cramer and Damgård [CD01], who proposed constant-rounds MPC protocols for various basic tasks in linear algebra. In that paper, as well as in later papers in the same line of work, like [NW06, KMWF07, CKP07, MW08], the focus is on linear algebra *over a finite field*.

Our goal is to obtain, in an "MPC-friendly" way, an (approximate) solution to a linear system *over the real numbers*. In this chapter we choose to approximate real arithmetic by (exact) rational arithmetic, or, in fact, integer arithmetic, using appropriate scaling. Our main reason behind this choice is the close connection between the finite field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ (where p is prime) and inte-

ger arithmetic, since we target MPC schemes that offer finite-field arithmetic. Hence, the protocols that we propose in this chapter will employ finite-field arithmetic *as a tool, rather than as a goal*. We note that there are various papers targeting the same problem that explore other choices, such as secure fixed-point arithmetic (see, e.g., [NWI+13, GSB+17]) or secure floating-point arithmetic (e.g., [BKLS18]).

In an earlier joint work with Blom and Schoenmakers [BBSdV19], we focused on the case of solving full-rank systems. In this chapter, we focus on the more general case of solving linear systems whose rank is unknown. Also, we would like to obtain meaningful solutions in case the system is over- or underdetermined. The *Moore–Penrose pseudoinverse* gives natural solutions in both cases: in the overdetermined case, which is the relevant case for linear regression, it yields the least-squares solution; in the underdetermined case it gives the minimum-norm solution. Another application of the Moore–Penrose pseudoinverse is to compute the condition number of a matrix that is not, or not-necessarily, invertible.

Concretely, given a matrix A of unknown rank with integral elements, we propose a protocol for computing the Moore–Penrose pseudoinverse over the rational numbers in a linear number of rounds. The computational complexity, counted as the number of secure multiplications, is $O(m^2 n)$, where m and n are the dimensions of the system and $m \leq n$. In some multiplicative-linear-secret-sharing-based MPC schemes, such as Shamir's scheme, we may count a secure dot product as a single secure multiplication; in that case the complexity reduces to $O(mn)$.

It should be rather easy to implement our protocol in any finite-field-based arithmetic secret sharing MPC framework; beyond elementary finite-field arithmetic our protocol merely requires secure subprotocols for sampling (public) random elements, performing a zero test on a secret-shared field element, computing the reciprocal of a secret-shared field element, and computing the determinant of an invertible secret-shared matrix.

Circumventing Rational Reconstruction

It is well known that one can perform (bounded) rational arithmetic via arithmetic in $\mathbb{F}_p$, essentially as follows: (i) represent the rational inputs as finite-field elements, i.e., an input of the form a/b, for integers a and b and such that $|a|, |b| \leq \sqrt{p/2}$, is encoded as the element $x = a \cdot b^{-1} \in \mathbb{F}_p$, (ii) perform the computation in integer arithmetic modulo p, (iii) reconstruct the numerators and denominators of the results of the computation, elementwise, in the

following manner. Let $y \in \mathbb{F}_p$ be an output of the computation, that corresponds to the fraction c/d for integers c and d. Then, if $|c|, |d| \leq \sqrt{p/2}$, we can uniquely reconstruct c and d from y by reducing the two-dimensional lattice basis $\{(p, 0), (y, 1)\}$ using the Lagrange–Gauss algorithm, in the sense that the reduced basis will contain the vector (c, d). This reconstruction procedure is known as *rational reconstruction* (see, e.g., [Wan81]).

An important drawback of the use of rational reconstruction in our scenario is that we essentially would need to *double* the bit-length of the finite field modulus p to guarantee unique reconstruction, compared to a route without rational reconstruction (for more details, see Figure 5.1). Because arithmetic in a larger finite field is computationally more expensive, we would like to avoid the use of rational reconstruction.

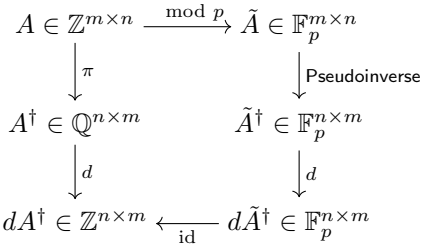
In [BBSdV19], a key trick for obtaining the inverse of an invertible integer matrix B over the rational numbers from the corresponding inverse over the finite field $\mathbb{F}_p$ *without requiring rational reconstruction*, was to form the integer-valued *adjugate matrix* by multiplying B^{-1} by $\det B$. In a similar spirit, we compute the pseudoinverse $A^\dagger$ over the finite field $\mathbb{F}_p$ and identify the conditions under which it corresponds to the pseudoinverse over the rational numbers. Essentially, this comes down to choosing p sufficiently large; see Section 5.4. We can then obtain an integer representation of the pseudoinverse by forming the pair $(dA^\dagger, d)$, where $dA^\dagger$ is an integer matrix containing the numerators of the pseudoinverse and d is the common denominator of the pseudoinverse, which coincides with the squared *volume* of A [Ben92], which we write as $(\operatorname{vol} A)^2$. Figure 5.1 illustrates our approach and compares it to the alternative route of rational reconstruction.

Although taking the square of the volume is rather excessive in certain cases (for example, the magnitude of the common denominator of B^{-1}, for any *invertible* matrix B, equals $|\det B| = \operatorname{vol} B$), it is essentially the price we have to pay for not knowing whether we are dealing with such a special case.

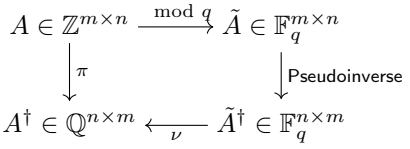
Computing the Pseudoinverse and Its Common Denominator

To compute the Moore–Penrose pseudoinverse $A^\dagger$ of A obliviously, we first compute a *reflexive generalized inverse* of the symmetric product $AA^\mathsf{T}AA^\mathsf{T}$ by means of block-recursive elimination. We then compute the Moore–Penrose pseudoinverse from this generalized inverse.

Springer computes the common denominator $(\operatorname{vol} A)^2$ of the coefficients of $A^\dagger$ via an integer-preserving rank decomposition [Spr83]. To circumvent the need for constructing such a rank decomposition, we seek a simpler alternative.



(a) Our approach. The map d represents scalar multiplication by $d = (\operatorname{vol} A)^2$ and id represents the identity map. The solutions $dA^\dagger$ and $d\tilde{A}^\dagger$ coincide, provided that p is chosen large enough, i.e., according to Lemma 5.10.



(b) Approach using rational reconstruction. The map ν represents the elementwise rational reconstruction procedure. All reconstructed fractions will be in lowest terms (numerator and denominator have no common nontrivial factors). There is, however, a price to be paid, in that $q \geq 2p^2$. Also, the map ν (the Lagrange–Gauss algorithm) is not "MPC-friendly".

Figure 5.1: Comparison between our approach and the approach via rational reconstruction. In the diagrams, the map $\pi : \mathbb{Q}^{m\times n} \to \mathbb{Q}^{n\times m}, A \mapsto A^\dagger$ applies the Moore–Penrose pseudoinverse over the rationals.

Ben-Israel gives a method for computing $(\operatorname{vol} A)^2$ that requires an orthonormal basis for the left null space of A [Ben92]. Although an *orthonormal* basis might not even exist over a finite field, we can easily construct a matrix K whose columns span the left null space of A. We generalize Ben-Israel's result so that we can compute $(\operatorname{vol} A)^2$ from A and K.

Preconditioning for Computing Pseudoinverses

As noted above, we will compute the Moore–Penrose Pseudoinverse via a generalized inverse that is obtained using block-recursive elimination.

Deterministic elimination algorithms typically employ pivoting to avoid problems like division by zero. Pivoting involves searching for and applying suitable row and/or column swaps prior to each elimination step. In secure computation, however, we aim to avoid pivoting because searching for particular elements and applying data-dependent row and column swaps *obliviously* is expensive (in a computational- and round-complexity sense).

An MPC-friendly alternative is to transform the matrix to be eliminated into an equivalent matrix for which the elimination procedure will succeed

without any pivoting; this approach is called *preconditioning*. In case of Gaussian elimination, for example, the condition of *generic rank profile* guarantees that pivoting can be omitted. A matrix A of rank r has *generic rank profile* if and only if all upper-left square submatrices of A up to dimension $r \times r$ are invertible. We prove that generic rank profile is also a sufficient condition for correctness of the particular block-recursive elimination algorithm that we use.

When dealing with a square, full rank matrix B over a finite field $\mathbb{F}$ with large order, one way to achieve generic rank profile with high probability is by pre-multiplying B by a preconditioner matrix R that is chosen uniformly at random from the set of all invertible matrices having the same size as B. When computing the inverse of RB, we can apply the rule $(RB)^{-1} = B^{-1}R^{-1}$, which we will refer to as the *reverse order law* for matrix inversion, to show that the inverse of the preconditioner can easily be removed by post-multiplying by R. For a matrix A with arbitrary rank r, pre-multiplying by a randomly chosen invertible matrix R (of appropriate size) is not sufficient for achieving generic rank profile; we additionally need to mix A's columns by multiplying A by a preconditioner matrix from the right.

A major problem that arises when trying to remove a preconditioner when computing the pseudoinverse, is that the reverse order law for pseudoinverses does not hold in general [Gre66, Har86]. In particular, unfortunately, we have that $(LAR)^\dagger$ does not necessarily equal $R^\dagger A^\dagger L^\dagger$ for invertible preconditioner matrices L and R. Hence, we cannot simply extract $A^\dagger$ from $(LAR)^\dagger$ like we could do above for B^{-1}. We circumvent this problem by applying the preconditioner only to $AA^\mathsf{T}AA^\mathsf{T}$ and removing the preconditioner immediately after computing the reflexive generalized inverse, for which the reverse-order law does hold.

An additional constraint in our setting where we apply preconditioning to $AA^\mathsf{T}AA^\mathsf{T}$, rather than to A directly, is that the preconditioner should preserve symmetry, since the symmetry property enables significant computational savings during elimination. A preconditioner for this particular scenario seems to be lacking in the literature. We resolve this by proving that the preconditioner $X \mapsto UXU^\mathsf{T}$ for a uniformly random matrix U fulfills all our constraints.

Interestingly, and unlike Gaussian elimination, when working over the real or complex numbers, the particular block-recursive algorithm that we use for computing the reflexive generalized inverse does not even require its input to have generic rank profile, hence no preconditioning is needed in this case. Nonetheless, in fields with positive characteristic, the condition emerges from the phenomenon of self-orthogonality.

Related Work

Cramer *et al.* [CKP07] propose a constant-rounds protocol for securely computing the Moore–Penrose pseudoinverse over a finite field. Their approach is to first compute the characteristic polynomial of the Gram matrix $A^{\mathsf{T}}A$, from which they then compute the rank of A (via a technique by Mulmuley [Mul87]) as well as the pseudoinverse of A (via the Cayley–Hamilton theorem).

An important theme in [CKP07] is to ensure that A (and A^{T}) are *suitable*, which guarantees, informally speaking, that certain subspaces that are orthogonal over a field orthogonal over fields with positive characteristic. In our work, where we focus on the setting where the modulus (hence the field's characteristic) is chosen sufficiently large, existence of the pseudoinverse is guaranteed by a result in [BRP90]. (We state this result in the next section.) Nonetheless, as described in the previous section, we do take special precautions, namely, applying preconditioning, to avoid problems related to working over a field with positive characteristic when computing a reflexive generalized inverse.

For an $m \times n$ matrix where $m \leq n$, the complexity (number of secure multiplications) of Cramer *et al.*'s solution is $O(m^4+n^2m)$. Our solution, albeit not constant-rounds, has complexity $O(m^2n)$, and even $O(mn)$ when assuming availability of a "cheap dot product", where the hidden constants in the Big-Oh of our solution are single-digit integers. By "cheap dot product", we mean that an dot product between two vectors of the same but arbitrary length has the same communication and round complexity as a single secure multiplication. It is possible to perform multiplication of an $m \times \ell$ matrix by an $\ell \times n$ matrix using no more than mn "cheap dot products". Because the coefficients of the result matrix may all be mutually independent, it is reasonable to take the complexity of such a matrix product to be equal to mn.

We leave it to further research to compare the practical performance of our method to that of [CKP07] in various application scenarios (i.e., various matrix-dimension regimes, network latency, bounded computational resources and storage space, etc.).

Relation to the LEU Decomposition

In an earlier work [BdV18], we propose to use Malaschonok's *LEU* decomposition [Mal10] for solving linear systems of unknown rank in the context of secure computation. (Note that [BdV18] does not deal with the problem of computing the Moore–Penrose pseudoinverse.) Our new protocol Pseudoinverse is superior to the *LEU*-decomposition-based protocol from [BdV18]; in terms of round

complexity, $O(m)$ versus $O(m^{1.59})$, as well as in terms of the asymptotic computational complexity, $O(m^2)$ versus $O(m^2 \log m)$ secure dot products for a square $m \times m$ matrix.

5.2 Preliminaries

Secret Sharing and Secure Computation

Let $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, where p is prime. We use $\mathbb{F}$ to denote an arbitrary field. We assume the use of an MPC protocol based on arithmetic secret sharing over $\mathbb{F}_p$. Our protocols will inherit the security properties (passive vs. active) from the underlying MPC protocol and of the subprotocols invoked by our protocol. The notation $[\![x]\!]$ represents an element $x \in \mathbb{F}_p$ that is secret-shared among the parties in the MPC protocol. Notation for secure arithmetic then follows naturally, for example, $[\![c]\!] \leftarrow [\![a]\!] + [\![b]\!]$ describes the addition of a and b where the result is stored in a new secret-shared element c, and $[\![d]\!] \leftarrow [\![a]\!][\![b]\!]$ describes an invocation of the multiplication protocol to securely compute the product of a and b and store the result in d. For arbitrary integer matrices A and B, the notation $[\![A]\!]$ expresses that all elements of A are secret-shared over $\mathbb{F}_p$, and $[\![A]\!] + [\![B]\!]$ and $[\![A]\!][\![B]\!]$ represent secure matrix addition (which coincides with elementwise addition) and secure matrix multiplication, respectively. Our protocols assume the availability of subprotocols for securely sampling private as well as public random field elements (e.g., [CDI05]), denoted as $[\![a]\!] \xleftarrow{\$} \mathbb{F}_p$ and $a \xleftarrow{\$} \mathbb{F}_p$ respectively, for securely inverting a field element (see [BB89]), and for performing a secure zero test [DFK+06, NO07]. The latter two are denoted as protocols Reciprocal and IsZero, respectively. We require protocol Reciprocal to be secure for all nonzero inputs (i.e., the protocol is allowed to leak information when run on a secret-share of zero). Protocol IsZero returns $[\![1]\!]$ if its argument equals zero and returns $[\![0]\!]$ otherwise.

Generalized Inverses

A *generalized inverse* of a matrix A is a matrix X associated to A that exists for a class of matrices larger than the class of invertible matrices, shares some properties with the ordinary inverse, and reduces to the ordinary inverse when A is non-singular. In this chapter, we classify generalized inverses using the

following four properties, also known as the *Penrose equations*:

$$AXA = A \tag{5.1}$$
$$XAX = X \tag{5.2}$$
$$(AX)^{\mathsf{T}} = AX \tag{5.3}$$
$$(XA)^{\mathsf{T}} = XA. \tag{5.4}$$

The matrix X that satisfies all four Penrose equations for a given matrix A is called the *Moore–Penrose pseudoinverse*, or simply *pseudoinverse* of A, which we denote as $A^{\dagger}$. The Moore–Penrose pseudoinverse of A over $\mathbb{F}$ exists if and only if $\operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(A^{\mathsf{T}}A) = \operatorname{rank} A$ [Pea68, Thm 1], and if it exists it is unique. We will also focus on generalized inverses of A which only satisfy equations (5.1) and (5.2); such generalized inverses are called *reflexive generalized inverses* and we denote any reflexive generalized inverse of A by A^{-}. Note that reflexive generalized inverses are not necessarily unique. For an extensive treatment of generalized inverses, the reader is referred to [BG03].

For a square matrix A partitioned as

$$A = \begin{pmatrix} E & F \\ G & H \end{pmatrix} \tag{5.5}$$

such that E is square, A/E denotes the *generalized Schur complement*

$$A/E = H - GE^{-}F. \tag{5.6}$$

Despite the fact that E^{-} is not necessarily unique, the generalized Schur complement *is*, although we shall not make use of this.

Submatrices, Their Determinants and Rank Properties

For any $n \in \mathbb{N}$, we write $[n]$ for the set $\{1, \ldots, n\}$. For any $m \times n$ matrix A and index sets $\mathcal{I} \subset [m]$ and $\mathcal{J} \subset [n]$, we denote the determinant of the submatrix of A obtained by selecting all rows in $\mathcal{I}$ and all columns in $\mathcal{J}$ by $[A]_{\mathcal{I},\mathcal{J}}$. Furthermore, $A_{[k]}$ denotes the leading principal submatrix of order k, i.e., the matrix obtained by taking the first k rows and first k columns of A, and we use $[A]_k$ as shorthand for $[A]_{[k],[k]}$, i.e., the leading principal minor of order k. Thus, it holds that $\det A_{[k]} = [A]_k$.

Let A be a matrix of rank r. We say that a matrix A has *generic rank profile* [KL96] if for all $k \in [r]$, it holds that A's leading principal minor of order k is nonzero, i.e., if $[A]_k \neq 0$ for all $k \in [r]$.

Let A be partitioned as in (5.5). If $\det E \neq 0$, then *Schur's determinant formula* asserts that

$$\det A = \det(E) \det(A/E) = \det(E) \det(H - GE^{-1}F). \tag{5.7}$$

A direct consequence of [MS74a, Thm 19] is that

$$\operatorname{rank} A \geq \operatorname{rank} E + \operatorname{rank}(A/E). \tag{5.8}$$

Hence, if A has generic rank profile and E has at least dimension $r \times r$ where $r = \operatorname{rank} A$, then A/E is the null matrix.

The Volume of a Matrix

For any matrix A with rank r and nonzero singular values $\sigma_1, \ldots, \sigma_r$, its *volume* is defined as $\operatorname{vol} A = \prod_{i=1}^{r} \sigma_i$. Note that this definition implies that we define the volume of the zero matrix to be one, which will be convenient for our purpose but deviates from Ben-Israel's definition of matrix volume for this special case [Ben92]. A matrix over an integral domain has a pseudoinverse if and only if its squared volume is a unit (i.e., an invertible element) of the integral domain [BRP90]. The fact that, for any matrix $A \in \mathbb{R}^{m \times n}$, the singular values of AA^{T} are the squares of the singular values of A leads to the following equation:

$$\operatorname{vol}(AA^{\mathsf{T}}) = (\operatorname{vol} A)^2, \tag{5.9}$$

which holds over an arbitrary field. In case A is a square nonsingular matrix, i.e., $m = n$ and $\det A \neq 0$, its volume coincides with the absolute value of its determinant:

$$\operatorname{vol} A = |\det A|. \tag{5.10}$$

Combining the two preceding equations gives

$$(\operatorname{vol} A)^2 = \det(AA^{\mathsf{T}}), \tag{5.11}$$

in the case that $\operatorname{rank} A = m$.

5.3 Block-Recursive Elimination

In this section we present ObliviousRGInverse, our oblivious protocol for computing a reflexive generalized inverse of any symmetric matrix over $\mathbb{F}_p$ that has generic rank profile. Although we could easily devise a protocol that also

Protocol 5.1 ScalarRGInverse($[\![a]\!]$)

Input: $a \in \mathbb{F}_p$
1: $[\![z]\!] \leftarrow$ IsZero($[\![a]\!]$)
2: **return** Reciprocal($[\![a + z]\!]$) $- [\![z]\!]$

works for non-symmetric matrices, we deliberately restrict to symmetric matrices, for the following two reasons: (i) by doing so, we achieve a significant computational saving (essentially a factor of two); and (ii) for our application we anyway only need to compute a reflexive generalized inverse of a symmetric matrix.

First, we define the *extended reciprocal* of an element $c \in \mathbb{F}$ as zero if $c = 0$ and c^{-1}, i.e., the (ordinary) reciprocal, otherwise. Note that the (unique) reflexive generalized inverse of a 1×1 matrix is equal to the 1×1 matrix containing the extended reciprocal of its only coefficient. ScalarRGInverse (Protocol 5.1) is a secure protocol for computing the extended reciprocal.

ObliviousRGInverse is given as Protocol 5.2. On line 4, the partitioning is done such that E and H are square and their dimensions differ by at most one. The way the matrix is partitioned has no bearing on the correctness of the protocol, but partitioning as evenly as possible results in the lowest complexity. We remark that the side notes with label "symmetric" in ObliviousRGInverse indicate that the resulting matrix is symmetric, which is to be exploited in an implementation.

It is straightforward to see that protocol ObliviousRGInverse is oblivious: it only branches on the dimensions of the matrix, which are considered public, and otherwise only performs elementary arithmetic operations, and calls to secure subprotocols (including recursive calls to itself).

Correctness Analysis

Rohde [Roh65] shows that a reflexive generalized inverse A^- of a symmetric, positive-semidefinite matrix *over the real numbers*[1]

$$A = \begin{pmatrix} E & F \\ F^\mathsf{T} & H \end{pmatrix} \tag{5.12}$$

[1] Rohde [Roh65] actually shows his result for complex matrices, but for our purposes it is more convenient to state his result for real matrices.

Protocol 5.2 ObliviousRGInverse($[\![A]\!]$)

Input: $A \in \mathbb{F}_p^{m\times m}$, symmetric, with generic rank profile

1: **if** $m = 1$ **then**
2: **return** ScalarRGInverse($[\![a_{1,1}]\!]$)
3: **else**
4: $\begin{pmatrix} [\![E]\!] & [\![F]\!] \\ [\![F^{\mathsf{T}}]\!] & [\![H]\!] \end{pmatrix} \leftarrow [\![A]\!]$ ▷ split as evenly as possible
5: $[\![X]\!] \leftarrow$ ObliviousRGInverse($[\![E]\!]$)
6: $[\![XF]\!] \leftarrow [\![X]\!][\![F]\!]$
7: $[\![H - F^{\mathsf{T}}XF]\!] \leftarrow [\![H]\!] - [\![F^{\mathsf{T}}]\!][\![XF]\!]$ ▷ symmetric
8: $[\![Y]\!] \leftarrow$ ObliviousRGInverse($[\![H - F^{\mathsf{T}}XF]\!]$)
9: $[\![XFY]\!] \leftarrow [\![XF]\!][\![Y]\!]$
10: $[\![X + XFYF^{\mathsf{T}}X]\!] \leftarrow [\![X]\!] + [\![XFY]\!][\![XF]\!]^{\mathsf{T}}$ ▷ symmetric
11: **return** $\begin{pmatrix} [\![X + XFYF^{\mathsf{T}}X]\!] & -[\![XFY]\!] \\ -[\![XFY]\!]^{\mathsf{T}} & [\![Y]\!] \end{pmatrix}$

can be expressed in Banachiewicz–Schur form as

$$A^- = \begin{pmatrix} E^- + E^-FS^-F^{\mathsf{T}}E^- & -E^-FS^- \\ -S^-F^{\mathsf{T}}E^- & S^- \end{pmatrix}, \tag{5.13}$$

where E^- is a reflexive generalized inverse of E and S^- is a reflexive generalized inverse of $S = A/E = H - F^{\mathsf{T}}E^-F$. This form allows for a block-recursive algorithm for computing the reflexive generalized inverse over the real numbers. As proved by Marsaglia and Styan, the correctness of Rohde's result *over an arbitrary field* depends on the following additional condition.

Lemma 5.1 ([MS74b], statement tailored to our needs)**.** *Over an arbitrary field, Equation* (5.13) *is a reflexive generalized inverse of A if and only if*

$$\operatorname{rank} A = \operatorname{rank} E + \operatorname{rank} S, \tag{5.14}$$

or, equivalently, the following three conditions are satisfied simultaneously

$$(I - EE^-)F(I - S^-S) = 0 \tag{5.15}$$

$$(I - SS^-)F^{\mathsf{T}}(I - E^-E) = 0 \tag{5.16}$$

$$(I - EE^-)FS^-F^{\mathsf{T}}(I - E^-E) = 0, \tag{5.17}$$

where E^- and S^- are reflexive generalized inverses of E and $S = A/E$ respectively.

Lemma 5.2. *Over an arbitrary field, a sufficient condition for Equation* (5.13) *to be a reflexive generalized inverse of a symmetric matrix A is that A has generic rank profile.*

Proof. We partition A as in Equation (5.12) arbitrarily but such that E is square. Now we can make a case distinction on E. If E is invertible, then E^- coincides with the ordinary inverse and it immediately follows that $(I - EE^-) = (I - E^-E) = 0$, thus satisfying (5.15)–(5.17) from Lemma 5.1. If E is not invertible, then, since A has generic rank profile, it then immediately follows that $\operatorname{rank} A = \operatorname{rank} E$ and furthermore that $\operatorname{rank} S = 0$, due to (5.8), thus satisfying (5.14). □

Lemma 5.3. *For any $m \times n$ matrix A over an arbitrary field, any k such that $A_{[k]}$ is invertible, and any i such that $0 \leq i \leq \min(m, n) - k$ it holds that*

$$A_{[k+i]}/A_{[k]} = (A/A_{[k]})_{[i]}. \tag{5.18}$$

Proof. Let

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}, \tag{5.19}$$

where $A_{11} = A_{[k]}$ is an invertible $k \times k$ matrix and A_{22} is an $i \times i$ matrix. Then

$$\begin{aligned} (A/A_{[k]})_{[i]} &= \left(\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix} A_{11}^{-1} \begin{pmatrix} A_{12} & A_{13} \end{pmatrix} \right)_{[i]} \\ &= A_{22} - A_{21} A_{11}^{-1} A_{12} \\ &= A_{[k+i]}/A_{[k]}. \end{aligned} \tag{5.20}$$

□

Corollary 5.4. *Protocol* ObliviousRGInverse*, when run on a symmetric matrix A over $\mathbb{F}_p$ having generic rank profile, correctly computes a reflexive generalized inverse.*

Proof. For the base case, we have already argued correctness of the extended reciprocal near the beginning of Section 5.3. For the recursive step applied to A, note that for an arbitrary partitioning but such that E is a $k \times k$ matrix for some integer k, it is easy to see that E is symmetric and has generic rank profile. Correctness then follows from Lemma 5.2.

We prove that S is symmetric and has generic rank profile by distinguishing two cases. If E is not invertible, then $\operatorname{rank} A = \operatorname{rank} E$ and S is necessarily the (square) null matrix, which is symmetric and has generic rank profile. Otherwise, E is invertible and $S = A/E = G - F^T E^{-1} F$, which is clearly symmetric. For generic rank profile, we can apply Schur's determinant formula to the leading principal minors of A: for any i such that $0 \le i \le \operatorname{rank} A - k$ we have $0 \neq \det(A_{[k+i]}) = \det(E)\det(A_{[k+i]}/E)$. Then, applying Lemma 5.3 gives $\det(A_{[k+i]}/E) = \det((A/E)_{[i]}) \neq 0$, i.e., A/E has generic rank profile. In both cases, correctness now follows from Lemma 5.2. □

Remark. We have proved that generic rank profile is sufficient for correctness—we did not prove that this condition is necessary. This leaves open the possibility that a weaker condition on the input matrix (weaker than generic rank profile) would suffice for correctness of ObliviousRGInverse. In the next section we will compute $(AA^{\mathsf{T}}AA^{\mathsf{T}})^{-}$, from which we construct $A^{\dagger}$. To ensure the correctness of ObliviousRGInverse we will actually randomize its input, $AA^{\mathsf{T}}AA^{\mathsf{T}}$, so that it has generic rank profile with high probability and then undo the randomization on the result. One might raise the question whether choosing the modulus p large enough to guarantee the existence of $A^{\dagger}$, could immediately guarantee correctness of ObliviousRGInverse on $AA^{\mathsf{T}}AA^{\mathsf{T}}$, without requiring $AA^{\mathsf{T}}AA^{\mathsf{T}}$ to have generic rank profile. We answer this question in the negative. As will be shown in Section 5.4, the modulus $p = 17$ suffices to ensure that the Moore–Penrose pseudoinverse exists over $\mathbb{F}_{17}$ and corresponds with $A^{\dagger}$ over the rational numbers for matrices A of rank 2 and Frobenius norm $\|A\| \le \sqrt{2(p-1)} \approx 5.66$. The matrix $\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ satisfies these requirements, yet algorithm ObliviousRGInverse would fail on input $AA^{\mathsf{T}}AA^{\mathsf{T}} \equiv \begin{pmatrix} 0 & 5 \\ 5 & 2 \end{pmatrix} \pmod{17}$, because $\operatorname{rank} A = 2$, but the upper left corner has $\operatorname{rank} E = 0$ and its Schur complement $\operatorname{rank} A/E = 1$, and therefore equation (5.14) is not satisfied.

Complexity Analysis

We first state the complexity (in terms of the number of secure operations) of protocol ObliviousRGInverse when run on a square matrix whose dimensions are a power of two.

Proposition 5.5. *Protocol* ObliviousRGInverse*, when run on an $m \times m$ matrix over $\mathbb{F}_p$, where $m = 2^k$ for integer k, requires $\frac{3}{2}m(m-1) + \frac{1}{2}m\log_2 m$ secure dot products and m invocations of* ScalarRGInverse*.*

If the dimensions of the matrix, m, are not a power of two, it is not always possible to divide the matrix evenly in step 4 of the protocol. In these cases the number of secure dot products required is slightly greater than the number stated in Proposition 5.5. For general dimensions, we prove the following proposition. We note that this bound is not tight.

Proposition 5.6. *Protocol* ObliviousRGInverse*, when run on an $m \times m$ matrix over $\mathbb{F}_p$, requires fewer than $\frac{3}{2}m(m-1) + m\log_2 m$ secure dot products and exactly m invocations of* ScalarRGInverse*.*

We also express the complexity of protocol ObliviousRGInverse in terms of elementary secure multiplications, rather than secure dot products, for MPC schemes for which the "cheap dot product" is not available. Note that the bound given here is exact if we assume the naïve algorithm for matrix multiplication. A more advanced algorithm would result in sub-cubic, but still super-quadratic complexity.

Proposition 5.7. *Protocol* ObliviousRGInverse*, when run on an $m \times m$ matrix over $\mathbb{F}_p$, requires at most $\frac{1}{2}m^3 + \frac{1}{2}m^2 - m$ secure multiplications and exactly m invocations of* ScalarRGInverse*.*

Proof of Propositions 5.5, 5.6 and 5.7. The proof is by complete induction. It is clear that running ObliviousRGInverse on an $m \times m$ matrix requires exactly m invocations of ScalarRGInverse.

Let $D(m)$ denote the number of secure dot products required to run protocol ObliviousRGInverse on an $m \times m$ matrix. Similarly, let $M(m)$ be the number of secure multiplications required, in case a "cheap dot product" is not available. Then, we have to show that

$$D(2^k) = \frac{3}{2}2^k(2^k - 1) + \frac{1}{2}k2^k \qquad \text{(Proposition 5.5);} \tag{5.21}$$

$$D(m) < \frac{3}{2}m(m-1) + m\log_2 m \qquad \text{(Proposition 5.6); and} \tag{5.22}$$

$$M(m) \leq \frac{1}{2}m^3 + \frac{1}{2}m^2 - m \qquad \text{(Proposition 5.7),} \tag{5.23}$$

where we take $m = 2^k$ for Proposition 5.5.

Inspection of the protocol shows that

$$D(1) = 0; \tag{5.24}$$

$$D(2\ell) = 2D(\ell) + 3\ell^2 + \ell; \text{ and} \tag{5.25}$$

$$D(2\ell + 1) = D(\ell) + D(\ell + 1) + 3\ell^2 + 4\ell + 1, \tag{5.26}$$

where we distinguish between even ($m = 2\ell$) and odd ($m = 2\ell+1$) dimensions. Similarly, for $M(m)$, we have

$$M(1) = 0; \tag{5.27}$$

$$M(2\ell) \leq 2M(\ell) + 3\ell^3 + \ell^2; \text{ and} \tag{5.28}$$

$$M(2\ell+1) \leq M(\ell) + M(\ell+1) + 3\ell^3 + \frac{11}{2}\ell^2 + \frac{5}{2}\ell. \tag{5.29}$$

The inequalities in (5.28) and (5.29) can be replaced with equalities in case the naïve algorithm for matrix multiplication is used.

In the base case $m = 1$, the propositions clearly hold. Assume the propositions hold for all $m' < m$. Then for odd $m = 2\ell + 1$ substitution of (5.22) in (5.26) yields

$$\begin{aligned} D(2\ell+1) &< \frac{3}{2}m(m-1) + \ell + 1 + \ell\log_2\ell + (\ell+1)\log_2(\ell+1) \\ &= \frac{3}{2}m(m-1) + \log_2\left(\ell^\ell(\ell+1)^{\ell+1}2^{\ell+1}\right) \\ &< \frac{3}{2}m(m-1) + (2\ell+1)\log_2(2\ell+1), \end{aligned} \tag{5.30}$$

where the last inequality follows from monotonicity of the logarithm and from the following inequality:

$$\begin{aligned} 2^{\ell+1}\ell^\ell(\ell+1)^{\ell+1} &= 2^{1-\ell}(2\ell+1-1)^\ell(2\ell+1+1)^\ell(\ell+1) \\ &= 2^{1-\ell}((2\ell+1)^2-1)^\ell(\ell+1) \\ &< 2^{1-\ell}(2\ell+1)^{2\ell}(\ell+1) \\ &< (2\ell+1)^{2\ell+1}. \end{aligned} \tag{5.31}$$

For the case of even m, Proposition 5.5 follows immediately by substituting equation (5.21) in (5.25). Similarly, Proposition 5.6 follows from the substitution of (5.22) in (5.25) and Proposition 5.7 from the substitution of (5.23) in (5.28) and (5.29). □

5.4 Computing the Moore–Penrose Pseudoinverse

We will compute the Moore–Penrose pseudoinverse using a formula (see, e.g., [RM71, p. 207]) that computes $A^\dagger$ in terms of a reflexive generalized inverse:

$$A^\dagger = A^\mathsf{T}(AA^\mathsf{T}AA^\mathsf{T})^- AA^\mathsf{T}. \tag{5.32}$$

Before proposing our protocol Pseudoinverse, we deal with three remaining questions, namely how to compute the common denominator, how to choose an appropriate modulus, and how to reliably compute $(AA^{\mathsf{T}}AA^{\mathsf{T}})^{-}$, as $AA^{\mathsf{T}}AA^{\mathsf{T}}$ does not necessarily have generic rank profile, which is required by protocol ObliviousRGInverse for correctness.

Computing the Common Denominator

Over the rational numbers, a common denominator d such that $dA^{\dagger}$ is integer-valued if A is integer-valued is $d = (\operatorname{vol} A)^2$ [Spr83, Satz 10]. The squared volume is minimal in the sense that there exist matrices for which it is the smallest possible common denominator.

If we would have an *orthonormal* basis for the left or right null space of A, then we could use [Ben92, Thm. (4.1)] to compute $(\operatorname{vol} A)^2$ directly. An orthonormal basis does not necessarily exist over an arbitrary field. Instead, we generalize [Ben92, Thm. (4.1)] by relaxing the requirements on the null space basis.

Lemma 5.8. *Let $A \in \mathbb{F}^{m\times k}$ be a matrix of rank r. Let $B \in \mathbb{F}^{m\times \ell}$ be a matrix of rank $m-r$ such that its columns are orthogonal to the columns of A, i.e., $B^{\mathsf{T}}A = 0$. Then,*

$$\det(AA^{\mathsf{T}} + BB^{\mathsf{T}}) = (\operatorname{vol} A)^2(\operatorname{vol} B)^2. \tag{5.33}$$

Proof. Note that $AA^{\mathsf{T}} + BB^{\mathsf{T}} = \begin{pmatrix} A & B \end{pmatrix}\begin{pmatrix} A & B \end{pmatrix}^{\mathsf{T}}$. Because the columns of A are orthogonal to those of B, the matrix $\begin{pmatrix} A & B \end{pmatrix}$ has rank $r + (m-r) = m$ and hence

$$\det(\begin{pmatrix} A & B \end{pmatrix}\begin{pmatrix} A & B \end{pmatrix}^{\mathsf{T}}) = (\operatorname{vol}\begin{pmatrix} A & B \end{pmatrix})^2 = (\operatorname{vol} A)^2(\operatorname{vol} B)^2, \tag{5.34}$$

where the first equality holds by equation (5.11), and the second equality is [Ben92, Example 5.1]. □

Theorem 5.9. *Let $A \in \mathbb{F}^{m\times n}$ be a matrix of rank r. Let $K = I - AA^{\dagger} \in \mathbb{F}^{m\times m}$. Then,*

$$(\operatorname{vol} A)^2 = \det(AA^{\mathsf{T}} + K). \tag{5.35}$$

Proof. By property (5.3) of the pseudoinverse, we have that $K = K^{\mathsf{T}}$. This fact, and property (5.1) of the pseudoinverse imply that $KK^{\mathsf{T}} = KK = K$ and $K^{\mathsf{T}}A = 0$, i.e., K is idempotent and its columns are orthogonal to the columns of A.

Combining equation (5.9) with the fact that K is idempotent and symmetric gives us that $\operatorname{vol} K = \operatorname{vol}(KK^{\mathsf{T}}) = (\operatorname{vol} K)^2$. Since the volume of a matrix is nonzero, we conclude that $\operatorname{vol} K = 1$.

Orthogonality of the columns of K and A implies that $\operatorname{rank} K \leq m - r$ and

$$\operatorname{rank} K = \operatorname{rank}(I - AA^{\mathsf{T}}) \geq \operatorname{rank} I - \operatorname{rank}(AA^{\mathsf{T}}) = m - r \tag{5.36}$$

follows from subadditivity of matrix rank. Applying Lemma 5.8 gives us

$$\det(AA^{\mathsf{T}} + K) = \det(AA^{\mathsf{T}} + KK^{\mathsf{T}}) = (\operatorname{vol} A)^2 (\operatorname{vol} K)^2 = (\operatorname{vol} A)^2. \tag{5.37}$$

□

Bound on the Modulus

Springer [Spr83] has proved the following upper bound on the magnitudes of the numerators and the common denominator of the pseudoinverse. Choosing p larger than twice this bound will guarantee that: (i) $d = (\operatorname{vol} A)^2$ is an invertible element in $\mathbb{F}_p$, which is a necessary and sufficient condition for existence of $A^\dagger$ over $\mathbb{F}_p$ [BRP90] (see also Section 5.2), (ii) that the pair $(dA^\dagger, d)$ over $\mathbb{F}_p$ coincides with $(dA^\dagger, d)$ over $\mathbb{Z}$ (see Lemma 5.10 below), and (iii) that the product $AA^{\mathsf{T}}AA^{\mathsf{T}}$ occurring in Equation (5.32) has the same rank as A (which we will need in Theorem 5.14, and note that (iii) is implied by applying (i) to the upcoming Proposition 5.11).

Lemma 5.10 ([Spr83, Satz 12])**.** *Let $N_0 = (\operatorname{vol} A)^2$ and $Z_0 = (z_{ij}) \in \mathbb{Z}^{m \times n}$ be an integer matrix of rank r such that $A^\dagger = \frac{1}{N_0} Z_0$. Let $\mu = \min(m, n)$. Then,*

$$\max(|N_0|, \max_{i,j} |z_{ij}|) \leq \max\left(\frac{\|A\|_F^{2r}}{r^r}, \frac{\|A\|_F^{2r-1}}{\sqrt{r^r (r-1)^{r-1}}} \right), \tag{5.38}$$

and

$$\max(|N_0|, \max_{i,j} |z_{ij}|) \leq \max\left(\frac{\|A\|_F^{2\mu}}{\mu^\mu}, \frac{\|A\|_F^{2\mu-1}}{\sqrt{\mu^\mu (\mu-1)^{\mu-1}}} \right), \tag{5.39}$$

where $\|A\|_F = \sqrt{\sum_{ij} |a_{ij}|^2}$ is the Frobenius norm of A.

Remark. In a setting in which the rank r is unknown, one would use (5.39).

For our construction, we further require that

$$\operatorname{rank}(AA^{\mathsf{T}}AA^{\mathsf{T}}) = \operatorname{rank} A. \tag{5.40}$$

This requirement holds unconditionally over fields of characteristic zero, but not necessarily over finite fields. Nonetheless, as we show below, it turns out that existence of the Moore–Penrose pseudoinverse already implies (5.40).

Proposition 5.11. *Let A be an arbitrary matrix over $\mathbb{F}$. The Moore–Penrose pseudoinverse of A exists if and only if*

$$\operatorname{rank}(AA^{\mathsf{T}}AA^{\mathsf{T}}) = \operatorname{rank} A. \tag{5.41}$$

Proof. Recall from Section 5.2 that the Moore–Penrose pseudoinverse exists over $\mathbb{F}$ if and only if $\operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(A^{\mathsf{T}}A) = \operatorname{rank} A$. Note that

$$\operatorname{rank}(AA^{\mathsf{T}}AA^{\mathsf{T}}) = \operatorname{rank} A \implies \operatorname{rank} A = \operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(A^{\mathsf{T}}A), \tag{5.42}$$

so we only have to prove the converse.

Let $A = VW$ be a rank decomposition of A, i.e., V and W have full column-rank and full row-rank, respectively. Over an arbitrary field, a rank decomposition exists but is not necessarily unique; see, e.g., [Rao73]. Then,

$$\operatorname{rank} A = \operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(VWW^{\mathsf{T}}V^{\mathsf{T}}) \implies \operatorname{rank}(WW^{\mathsf{T}}) \geq \operatorname{rank} A, \tag{5.43}$$

and similarly,

$$\operatorname{rank} A = \operatorname{rank}(A^{\mathsf{T}}A) = \operatorname{rank}(W^{\mathsf{T}}V^{\mathsf{T}}VW) \implies \operatorname{rank}(V^{\mathsf{T}}V) \geq \operatorname{rank} A. \tag{5.44}$$

Also note that both WW^{T} and $V^{\mathsf{T}}V$ have dimension $r \times r$ with $r = \operatorname{rank} A$, therefore, they are invertible. We now write $AA^{\mathsf{T}}AA^{\mathsf{T}}$ in terms of V and W, and multiply by V^{T} from the left and by V from the right, by which we obtain:

$$V^{\mathsf{T}}AA^{\mathsf{T}}AA^{\mathsf{T}}V = (V^{\mathsf{T}}V)(WW^{\mathsf{T}})(V^{\mathsf{T}}V)(WW^{\mathsf{T}})(V^{\mathsf{T}}V), \tag{5.45}$$

the rank of which bounds $\operatorname{rank}(AA^{\mathsf{T}}AA^{\mathsf{T}})$ from below.

Therefore, we can conclude that $\operatorname{rank}(AA^{\mathsf{T}}AA^{\mathsf{T}}) = \operatorname{rank} A$, if and only if $\operatorname{rank} A = \operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(A^{\mathsf{T}}A)$. □

Symmetric Preconditioning

A *preconditioner* is a mapping $A \mapsto h(A)$ for matrices A from a given class, where the goal is to achieve a certain property, either with certainty or with high probability. This property is typically an input condition from some computational technique. For a more elaborate and formal introduction into preconditioning we refer to [CEK+02]. Here, we restrict to preconditioners for achieving *generic rank profile* for symmetric matrices of the form $A = BB^{\mathsf{T}}$ over an arbitrary field of positive characteristic.

To ensure correctness of protocol ObliviousRGInverse, we need a preconditioner that

(i) achieves generic rank profile with high probability;

(ii) preserves symmetry, i.e., $h(A)$ is symmetric; and

(iii) is *removable* with high probability. Informally speaking, this means that the preconditioner can be efficiently removed once "it has done its job". Formally, a preconditioner is removable with respect to computing a reflexive generalized inverse if there exists an efficiently computable mapping g such that $g(h(A)^-)$ is a reflexive generalized inverse of A.

Although several preconditioners for achieving generic rank profile have been proposed in the literature, we are not aware of an existing result that covers all of the above properties simultaneously. For example, the Toeplitz preconditioner by Kaltofen and Saunders [KS91] fails to satisfy (ii), and the diagonal preconditioner proposed in [EK97] (combined with a suitable linear-independence preconditioner, see [CEK+02]) fails to satisfy (iii).

In this section we will show that for a symmetric matrix A, the preconditioner $h(A) = UAU^{\mathsf{T}}$ with U a uniformly random (invertible) matrix is sufficient for satisfying (i)–(iii). It is easy to see that (ii) holds. We first prove the necessary condition for removability in Lemma 5.13 and then prove properties (i) and (iii) in Theorem 5.14.

Lemma 5.12 (Schwartz–Zippel)**.** *Let $g \in \mathbb{F}[x_1, \ldots, x_n]$ be a nonzero polynomial of total degree $d \geq 0$ over a field $\mathbb{F}$. Let $\mathcal{S} \subseteq \mathbb{F}$ and let $\alpha_1, \ldots, \alpha_n$ be chosen independently and uniformly at random from $\mathcal{S}$. Then,*

$$\Pr[g(\alpha_1, \ldots, \alpha_n) = 0] \leq \frac{d}{|\mathcal{S}|}. \tag{5.46}$$

We will now show that the reverse order law holds for generalized reflexive inverses under the condition that the product is rank-preserving.

Lemma 5.13. *Let $A \in F^{m\times\ell}$ and $B \in \mathbb{F}^{\ell\times n}$ be arbitrary and let $Z \in \mathbb{F}^{n\times m}$ be a generalized reflexive inverse of AB. Then, BZ is a generalized reflexive inverse of A if and only if* $\operatorname{rank} A = \operatorname{rank}(AB)$ *and, similarly, ZA is a generalized reflexive inverse of B if and only if* $\operatorname{rank} B = \operatorname{rank}(AB)$.

Proof. We will only show the first part; the second part follows immediately by transposition. Let $r = \operatorname{rank} A$. Let $V \in \mathbb{F}^{m\times r}$ and $W \in \mathbb{F}^{r\times\ell}$ be a rank decomposition of A, i.e., $A = VW$. If $\operatorname{rank} A = \operatorname{rank}(AB)$, then $AB = VWB$, where $\operatorname{rank}(WB) = r$ and $WB \in \mathbb{F}^{r\times n}$, i.e., WB has full row rank and therefore there exists a right inverse $X \in \mathbb{F}^{n\times r}$ of WB, such that $WBX = I \in \mathbb{F}^{r\times r}$. Then, that BZ is a generalized reflexive inverse of A follows from

$$\begin{aligned} A(BZ)A &= VWBZVW \\ &= (VWBZV)(WBX)W \\ &= VWBXW \\ &= VW \\ &= A \end{aligned} \tag{5.47}$$

and

$$(BZ)A(BZ) = BZ. \tag{5.48}$$

Conversely, if $\operatorname{rank} A \neq \operatorname{rank}(AB)$, then

$$\begin{aligned} \operatorname{rank}(BZ) &\leq \operatorname{rank} Z \\ &= \operatorname{rank}(AB) \\ &< \operatorname{rank}(A), \end{aligned} \tag{5.49}$$

therefore BZ is not a generalized reflexive inverse of A. □

Theorem 5.14. *Let $A \in \mathbb{F}^{m\times n}$ be an arbitrary matrix of rank r, such that* $\operatorname{rank}(AA^{\mathsf{T}}) = r$. *Let $U \in \mathbb{F}^{m\times m}$ be chosen uniformly at random. Then, the probability that U is removable and $UAA^{\mathsf{T}}U^{\mathsf{T}}$ has generic rank profile is*

$$\Pr_U\left([UAA^{\mathsf{T}}U^{\mathsf{T}}]_k \neq 0 \quad \forall k \in [r]\right) > 1 - \frac{r(r+1)}{|\mathbb{F}|}. \tag{5.50}$$

Proof. The probability in equation (5.50) expresses both that $UAA^{\mathsf{T}}U^{\mathsf{T}}$ has generic rank profile, and that $\operatorname{rank}(AA^{\mathsf{T}}) = \operatorname{rank}(UAA^{\mathsf{T}}U^{\mathsf{T}})$, i.e., that the preconditioner is removable by applying Lemma 5.13 twice.

We view $U = (u_{i,j})$ as a polynomial matrix with $u_{i,j}$ as indeterminates. For every $1 \le k \le r$, we apply the Cauchy–Binet formula to obtain an expression for the leading principal minor of order k of the matrix $UAA^{\mathsf{T}}U^{\mathsf{T}}$, which is a polynomial in the variables $u_{i,j}$, where we let $\mathcal{K} = [k]$,

$$\begin{aligned} f_k(u_{1,1}, \ldots, u_{i,j}, \ldots, u_{m,m}) &= [UAA^{\mathsf{T}}U^{\mathsf{T}}]_{\mathcal{K},\mathcal{K}} && (5.51)\\ &= \sum_{\substack{\mathcal{I}\subset[m]\\|\mathcal{I}|=k}} [UA]_{\mathcal{K},\mathcal{I}}[A^{\mathsf{T}}U^{\mathsf{T}}]_{\mathcal{I},\mathcal{K}} && (5.52)\\ &= \sum_{\substack{\mathcal{I}\subset[m]\\|\mathcal{I}|=k}} \big([UA]_{\mathcal{K},\mathcal{I}}\big)^2 && (5.53)\\ &= \sum_{\substack{\mathcal{I}\subset[m]\\|\mathcal{I}|=k}} \Big(\sum_{\substack{\mathcal{J}\subset[m]\\|\mathcal{J}|=k}} [U]_{\mathcal{K},\mathcal{J}}[A]_{\mathcal{J},\mathcal{I}}\Big)^2. && (5.54) \end{aligned}$$

It follows immediately from the structure of this formula that the total degree of f_k is $2k$.

Let us now prove that none of the polynomials f_k for all $1 \le k \le r$ is equal to the zero polynomial. Because AA^{T} is symmetric, there exists an invertible matrix $S = (s_{i,j})$ such that $SAA^{\mathsf{T}}S^{\mathsf{T}} = \Lambda$ where $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_r, 0, \ldots, 0)$ with $\lambda_i \neq 0$ for all $1 \le i \le r$ [Alb38, Thm. 6]. Hence,

$$f_k(s_{1,1}, \ldots, s_{i,j}, \ldots, s_{m,m}) = \prod_{i=1}^{k} \lambda_i \neq 0 \qquad \forall k \in [r]. \tag{5.55}$$

The Schwartz–Zippel lemma asserts that $\Pr[f_k(U_{1,1}, \ldots, U_{m,m}) = 0] \le \frac{2k}{|\mathbb{F}|}$, where the $U_{i,j}$ represent the elements of U when viewed as (uniformly random and independent) random variables. Hence, by applying the union bound over k we obtain

$$\Pr[f_1(U) \neq 0 \wedge \cdots \wedge f_r(U) \neq 0] \ge 1 - \frac{\sum_{k=1}^{r} 2k}{|\mathbb{F}|} = 1 - \frac{r(r+1)}{|\mathbb{F}|}. \tag{5.56}$$

□

Protocol 5.3 Pseudoinverse($[\![A]\!]$)

Input: $A \in \mathbb{F}_p^{m \times n}$

1: **if** $m > n$ **then**
2: **return** Pseudoinverse($[\![A]\!]^\mathsf{T})^\mathsf{T}$
3: $[\![AA^\mathsf{T}]\!] \leftarrow [\![A]\!][\![A]\!]^\mathsf{T}$ ▷ symmetric
4: $[\![AA^\mathsf{T}AA^\mathsf{T}]\!] \leftarrow [\![AA^\mathsf{T}]\!][\![AA^\mathsf{T}]\!]$ ▷ symmetric
5: $U \xleftarrow{\$} \mathbb{F}_p^{m \times m}$
6: $[\![X]\!] \leftarrow U^\mathsf{T}$ ObliviousRGInverse$(U[\![AA^\mathsf{T}AA^\mathsf{T}]\!]U^\mathsf{T})U$
7: $[\![XAA^\mathsf{T}]\!] \leftarrow [\![X]\!][\![AA^\mathsf{T}]\!]$
8: $[\![A^\dagger]\!] \leftarrow [\![A^\mathsf{T}]\!][\![XAA^\mathsf{T}]\!]$
9: $[\![K]\!] \leftarrow I - [\![AA^\mathsf{T}]\!][\![XAA^\mathsf{T}]\!]$ ▷ symmetric; in parallel with $[\![A^\dagger]\!]$
10: $[\![(\mathrm{vol}\, A)^2]\!] \leftarrow$ Determinant$([\![AA^\mathsf{T}]\!] + [\![K]\!])$
11: **return** $([\![A^\dagger]\!], [\![(\mathrm{vol}\, A)^2]\!])$

Construction

Our protocol Pseudoinverse, on input of a secret-shared matrix $[\![A]\!] \in \mathbb{F}_p^{m \times n}$, computes the pair $([\![A^\dagger]\!], [\![(\mathrm{vol}\, A)^2]\!])$ and is given as Protocol 5.3.

We note that the rank of A is given by $\mathrm{Tr}(AA^\dagger)$ [BO71]. It can be computed obliviously in Pseudoinverse as $[\![r]\!] = m - \mathrm{Tr}([\![K]\!])$.

Protocol Pseudoinverse makes use of a secure subprotocol Determinant for computing the determinant of an invertible matrix in $\mathbb{F}_p^{m \times m}$ in secret-shared form. A possible instantiation of Determinant can be found in [CD01], where it is called protocol Π_0. See also [BBSdV19], which slightly modifies this protocol to reduce its randomness complexity.

Another possibility for computing the determinant of a matrix is given by a small modification to the ObliviousRGInverse protocol, compute a reflexive generalized inverse, but the determinant of its input as well. This modification is shown in Protocol 5.4 as protocol ObliviousRGInverse′ and requires m additional multiplications, compared to ObliviousRGInverse. Correctness follows from Schur's determinant formula (equation (5.7)).

Corollary 5.15. *Protocol* Pseudoinverse*, when run on an arbitrary $m \times n$ matrix over $\mathbb{F}_p$, correctly computes the Moore–Penrose pseudoinverse with probability at least*

$$\Pr(\mathit{success}) \geq \left[1 - \frac{m(m+1)+2}{|\mathbb{F}|}\right] \cdot P_{\mathsf{Determinant}}, \tag{5.57}$$

Protocol 5.4 $\mathsf{ObliviousRGInverse}'(\llbracket A \rrbracket)$

Input: $A \in \mathbb{F}_p^{m \times m}$ with generic rank profile

1: **if** $m = 1$ **then**
2: **return** $(\mathsf{ScalarRGInverse}(\llbracket a_{1,1} \rrbracket), \llbracket a_{1,1} \rrbracket)$
3: **else**
4: $\begin{pmatrix} \llbracket E \rrbracket & \llbracket F \rrbracket \\ \llbracket F^{\mathsf{T}} \rrbracket & \llbracket H \rrbracket \end{pmatrix} \leftarrow \llbracket A \rrbracket$ ▷ split as evenly as possible
5: $(\llbracket X \rrbracket, \llbracket d_1 \rrbracket) \leftarrow \mathsf{ObliviousRGInverse}'(\llbracket E \rrbracket)$
6: $\llbracket XF \rrbracket \leftarrow \llbracket X \rrbracket \llbracket F \rrbracket$
7: $\llbracket H - F^{\mathsf{T}} XF \rrbracket \leftarrow \llbracket H \rrbracket - \llbracket F^{\mathsf{T}} \rrbracket \llbracket XF \rrbracket$ ▷ symmetric
8: $(\llbracket Y \rrbracket, \llbracket d_2 \rrbracket) \leftarrow \mathsf{ObliviousRGInverse}'(\llbracket H - F^{\mathsf{T}} XF \rrbracket)$
9: $\llbracket XFY \rrbracket \leftarrow \llbracket XF \rrbracket \llbracket Y \rrbracket$
10: $\llbracket d_1 d_2 \rrbracket \leftarrow \llbracket d_1 \rrbracket \llbracket d_2 \rrbracket$ ▷ in parallel with $\llbracket XFY \rrbracket$
11: $\llbracket X + XFYF^{\mathsf{T}} X \rrbracket \leftarrow \llbracket X \rrbracket + \llbracket XFY \rrbracket \llbracket XF \rrbracket^{\mathsf{T}}$ ▷ symmetric
12: **return** $\left(\begin{pmatrix} \llbracket X + XFYF^{\mathsf{T}} X \rrbracket & -\llbracket XFY \rrbracket \\ -\llbracket XFY \rrbracket^{\mathsf{T}} & \llbracket Y \rrbracket \end{pmatrix}, \llbracket d_1 d_2 \rrbracket \right)$

where $P_{\mathsf{Determinant}}$ *denotes the success probability of protocol* Determinant.

Complexity Analysis

Proposition 5.16. *Protocol* Pseudoinverse*, when run on an arbitrary* $m \times n$ *matrix over* $\mathbb{F}_p$*, requires* $mn + \frac{5}{2}m^2 + \frac{3}{2}m$ *secure dot products (or:* $\frac{3}{2}m^2 n + 2m^3 + \frac{1}{2}mn + m^2$ *secure multiplications), one invocation of protocol* Determinant *on a symmetric* $m \times m$ *matrix and one invocation of* ObliviousRGInverse *on a symmetric* $m \times m$ *matrix.*

Protocol Determinant, instantiated as in [BBSdV19], when invoked on a $m \times m$ matrix, requires secure sampling of m^2 random elements, and performing $2m^2 + m - 1$ secure dot products (or: $\frac{4}{3}m^3 + \frac{2}{3}m - 1$ secure multiplications) and m^2 open operations.

The field inversion technique from Bar-Ilan and Beaver [BB89] requires secure sampling of one random element and one secure multiply-and-open operation.

Subprotocol IsZero can be instantiated with the probabilistic secure zero test from Nishide and Ohta [NO07]. This secure zero test is constant round and requires 2κ secure multiplications, 4κ secure multiply-and-open operations

and secure sampling of 5κ random elements, where κ is a security parameter and the protocol may fail with probability $2^{-\kappa} + 1/p$.

Corollary 5.17. *Protocol* Pseudoinverse*, when run on an arbitrary $m \times n$ matrix over $\mathbb{F}_p$, with protocol* Determinant *instantiated as in [BBSdV19], requires in total $nm+6m^2+o(m^2)$ secure dot products (or: $\frac{3}{2}nm^2+\frac{23}{6}m^3+\frac{1}{2}nm+o(m^3)$ secure multiplications), m^2 public random elements, m^2 private random elements, m^2 openings, m secure zero tests and m secure field inversions.*

If protocols IsZero *and* Reciprocal *are instantiated as the probabilistic zero test from [NO07] and as in [BB89], respectively, the m secure zero tests and field inversions require $O(\kappa m)$ secure multiplications, random elements and openings.*

Remark. It is straightforward to adapt Protocol 5.3 such that in line 8 it computes the vector $A^\dagger \boldsymbol{b}$ instead of the matrix $A^\dagger$, i.e., directly solving the linear system $A\boldsymbol{x} = \boldsymbol{b}$ for the vector $\boldsymbol{x}$. By replacing line 8, in case $m \leq n$, with the two lines $[\![XAA^\mathsf{T}\boldsymbol{b}]\!] \leftarrow [\![XAA^\mathsf{T}]\!][\![\boldsymbol{b}]\!]$ and $[\![A^\dagger\boldsymbol{b}]\!] \leftarrow [\![A^\mathsf{T}]\!][\![XAA^\mathsf{T}\boldsymbol{b}]\!]$, one can avoid the matrix-matrix product that gives rise to the mn term. Namely, the complexity (number of secure dot products) becomes $O(n+m^2)$. If $m > n$, then we would transpose the system to be solved: $\boldsymbol{x}^\mathsf{T}A^\mathsf{T} = \boldsymbol{b}^\mathsf{T}$. In this case, line 8 would be replaced by two secure products in which the matrix is multiplied from the *left* by the vector and this would result in a complexity of $O(n^2)$ secure dot products. Note, however, that this adaptation imposes an additional constraint on the size of the modulus; the field should now be large enough to uniquely represent the coefficients of the vector $dA^\dagger\boldsymbol{b}$.

Chapter 6

Trinocchio

Note. *This chapter is based on* Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation *[SVdV16] with the following major modifications.*

- *The case study demonstrating the performance of Trinocchio has been removed, because this was not carried out with involvement of the author.*
- *The appendix on QAP based proofs and Pinocchio in particular has been incorporated into the main text.*
- *The appendix of the full version of the paper [SVdV15] containing the more detailed description and security proofs of the multi-client version of the protocol has been incorporated into the main text.*
- *An issue with our application of the d-PKE assumption in the security proof of the full protocol has been identified and remarked upon.*

Verifiable computation allows a client to outsource computational tasks to a worker with a cryptographic proof of correctness of the result that can be verified faster than performing the computation. Recently, the Pinocchio system achieved faster verification than computation in practice for the first time. Unfortunately, Pinocchio and other efficient verifiable computation systems require the client to disclose the inputs to the worker, which is undesirable for sensitive inputs. To solve this problem, we propose Trinocchio: a system that distributes Pinocchio to three (or more) workers, that each individually do not learn which inputs they are computing on. We fully exploit the almost linear

structure of Pinocchio proofs, letting each worker essentially perform the work for a single Pinocchio proof; verification by the client remains the same. Moreover, we extend Trinocchio to enable joint computation with multiple mutually distrusting inputters and outputters and still very fast verification.

6.1 Introduction

Recent cryptographic advances are starting to make verifiable computation more and more practical. The goal of verifiable computation is to allow a client to outsource a computation to a worker and cryptographically verify the result with less effort than performing the computation itself. Based on recent ground-breaking ideas [Gro10, GGPR13], Pinocchio [PHGR13] was the first implemented system to achieve this for some realistic computations. Recent works have improved the state-of-the-art in verifiable computation, e.g., by considering better ways to specify computations [BCG+13], or adding access control [AJCC15].

However, one feature not yet available in practical verifiable computation is privacy, meaning that the worker should not learn the inputs that it is computing on. This feature would enable a client to save time by outsourcing computations, even if the inputs of those computations are so sensitive that it does not want to disclose them to the worker. Also, it would allow verifiable computation to be used in settings where multiple clients do not trust the worker or each other, but still want to perform a joint computation over their respective inputs and be sure of the correctness of the result.

While privacy was already defined in the first paper to formalize verifiable computation [GGP10], it has not been shown so far how it is efficiently achieved, with existing constructions relying on inefficient cryptographic primitives. By outsourcing a computation to multiple workers, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness, but existing constructions from the multiparty literature lose the most appealing feature of verifiable computation: namely, that computations can be verified very quickly, even in time independent from the computation size. This leads to the central question of this chapter: can we perform verifiable computation with the *correctness* and *performance* guarantees of [PHGR13], but while also getting *privacy* against corrupted workers?

Our Contributions

In this chapter, we introduce Trinocchio to show that indeed, it is possible to outsource a computation in a privacy-preserving way to multiple workers, while retaining the fast verification offered by verifiable computation. Trinocchio uses state-of-the-art [PHGR13]-style proofs, but distributes the computation of these proofs to, e.g., three workers such that no single worker learns anything about the inputs. The client essentially gets a normal Pinocchio proof, so we keep Pinocchio's correctness guarantees and fast verification. The critical observation is that the almost linear structure of Pinocchio proofs (supporting verification based on bilinear maps) allows us to distribute the computation of Pinocchio proofs such that individual workers perform essentially the same work as a normal Pinocchio prover in the non-distributed setting. Specifically, our contributions are:

- We show how production of Pinocchio proofs can be distributed to multiple workers in a privacy-preserving way, thereby achieving privacy-preserving verifiable computation in the setting with one client
- We extend our system to settings with multiple distrusting input and result parties
- We provide a precise security model capturing the security guarantees of our protocols: privacy, correctness, but also input independence

While our Trinocchio protocol ensures correct function evaluation, it only fully protects privacy against semi-honest workers. This is a realistic attacker model; in particular, it means that side channel attacks on individual workers are ineffective because each individual worker's communication and computation are completely independent from the sensitive inputs. However, even if an adversary should be able to obtain sensitive information, they are unable to manipulate the result thanks to the use of verifiable computation. In this way, our protocol *hedges* against the risk of more powerful adversaries.

Related Work

Privacy-preserving outsourcing to single workers has been considered in the literature, but constructions in this setting rely on inefficient cryptographic primitives like fully homomorphic encryption [GGP10, CKKC13, FGP14], functional encryption [GKP+13], and multi-input attribute-based encryption [GKL+15]. (This is not surprising: indeed, even without guaranteeing correctness, letting

a single worker perform a computation on inputs it does not know would intuitively seem to require some form of fully homomorphic encryption.) Some of these works also consider a multi-client setting [CKKC13, GKL+15].

A large body of works considers secure multiparty computation for privacy-preserving outsourcing (see, e.g., [KMR12, PTK13, CLT14, JNO14]). These works do not consider verifiability and achieve correctness at best in the case that *all-but-one* workers are corrupt (due to inherent limitations of the underlying protocols). We stress that this is rather unsatisfactory for the outsourcing scenario, where one naturally wishes to cover the case that *all* workers are corrupt—dispensing of the need to trust any particular worker.

Concerning outsourcing to multiple workers, [ACG+14] presents a verifiable computation protocol combining privacy and correctness; but unfortunately, they guarantee neither privacy nor correctness if all workers are corrupted and may collude; and it places a much higher burden on the workers than, e.g., [PHGR13]. Alternatively, recent works [BDO14, dHSV16, SV15], like us, guarantee correctness independent of worker corruption, but privacy only under some conditions. Our work offers a substantial performance improvement over these works by fully exploiting a set-up that needs to be trusted both for guaranteeing privacy and for guaranteeing correctness.

We should mention that the notion of verifiability exists in various forms and the field has a richer background than presented here, however, we focus entirely on the notion of verifiable computation first formalized by [GGP10], because it is tailored to the outsourcing scenario.

Outline

We first briefly define the security model for privacy-preserving outsourced computation in Section 6.2. We then provide a brief overview of the Pinocchio protocol [PHGR13] for verifiable computation based on quadratic arithmetic programs in Section 6.3. In Section 6.4, we show how Trinocchio distributes the proof computation of Pinocchio in the single-client scenario, and prove security of the construction. We generalise Trinocchio to the setting with multiple, mutually distrusting inputters and outputters. We first give an outline of the generalised protocol and security proof in Section 6.5 and discuss the protocol and security proof in more detail in Sections 6.6 and 6.7, respectively. We finish with a discussion and conclusions in Section 6.8.

6.2 Security Model for Privacy-Preserving Outsourcing

In this section, we define security for privacy-preserving outsourcing. Because we have interactive protocols between multiple parties (as opposed to a cryptographic scheme, like verifiable computation above), we define security using the ideal/real-paradigm [Can00]. In our setting, the parties are several *result parties* that wish to obtain the result of a computation on inputs held by several *input parties*, who are willing to enable the computation, but not to divulge their private input values to anybody else. Therefore, they outsource the computation to several *workers*. (Input and result parties may overlap.) The simplest case is the "single-client scenario" in which one party is the single input/result party.

We consider protocols operating in three phases: an *input phase* involving the input parties and workers; a *computation phase* involving only the workers; and a *result phase* involving the workers and result parties. The work of the input parties and output parties should depend only on the number of other parties and the size of their own in/outputs.

To define security, we will re-use the existing definition framework for secure function evaluation [Can00]. These definitions are not specific to the outsourcing setting, but the outsourcing setting will become apparent when we claim that a protocol, e.g., implements secure function evaluation *if at most X workers are corrupted*. Secure function evaluation is the problem to evaluate $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ with m parties such that the ith party inputs x_i and obtains y_i, and no party learns anything else. (In outsourcing, result parties have non-empty output, input parties have non-empty inputs, and workers have empty in- and outputs.) A protocol π *securely evaluates function* f if the outputs of the parties and adversary $\mathcal{A}$ in a real-world execution of the protocol can be emulated by the outputs of the parties and an adversary $\mathcal{S}_{\mathcal{A}}$ in an idealised execution, where f is computed by a trusted party that acts as shown in Figure 6.1. Security is guaranteed because the trusted party correctly computes the function. Privacy is guaranteed because the adversary in the idealised execution does not learn anything it should not. Secure evaluation also implies *input independence*, meaning that an input party cannot let its input depend on that of another, e.g., by copying the input of another party; this is guaranteed because the adversary needs to provide the inputs of corrupted parties without seeing the honest inputs. Typically, protocols achieve secure function evaluation for a given, restricted class of adversaries, e.g., adversaries that are passive and only corrupt a certain number of workers. Protocols can require set-up assumptions; these are captured by giving protocol participants access to

Secure and correct function evaluation:

- Honest parties send inputs x_i to trusted party
- Adversary sends inputs x_i of corrupted parties to trusted party (active adversary may modify them)
- Trusted party computes function $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ (where $y_1 = \ldots = \perp$ if any $x_i = \perp$)

Secure function evaluation only:

- Trusted party provides outputs y_i for corrupted parties to adversary
- Trusted party provides outputs y_i to honest parties

Correct function evaluation only:

- Trusted party provides all inputs x_i to adversary
- Adversary gives subset of honest parties to trusted party (passive adversary gives all honest parties)
- Trusted party sends outputs y_i to given honest parties, $\perp$ to others

- Honest parties output received value; corrupted parties output $\perp$; adversary chooses own output

Figure 6.1: Ideal-world executions of secure and correct function evaluation. Where the two differ, secure function evaluation is indicated on the left and correct function evaluation on the right

a set of functions $g_1, \ldots, g_k$ that are always evaluated correctly. In this case, we say that the protocol securely evaluates the function *in the* $(g_1, \ldots, g_k)$*-hybrid model*. For details, see [Can00].

We only achieve secure function evaluation if not too many workers are corrupted; we still need to formalise that in all other cases, we still guarantee that the function was evaluated correctly. This weaker security guarantee, which we call *correct function evaluation*, captures security and input independence, as above, but not privacy. It is formalised by modifying the ideal-world execution as shown in Figure 6.1. Namely, after evaluating f, the trusted party provides all inputs to the adversary (modelling that the computation may leak the inputs), who, based on these inputs, can decide which honest parties are allowed to see their outputs. Hence, we guarantee that, *if* an honest party gets a result, then it gets the correct result of the computation on independently chosen inputs, but not that the inputs remain hidden, or that it gets a result at all. Note that, in this definition, the adversary has complete control over which result parties see an output and which ones do not.

6.3 Verifiable Computation from QAPs

In this section, we discuss the protocol for verifiable computation based on quadratic arithmetic programs from [GGPR13, PHGR13].

Modelling Computations as Quadratic Arithmetic Programs

A quadratic arithmetic program, or QAP, is a way of encoding arithmetic circuits, and some more general computations, over a field $\mathbb{F}$ of prime order q. It is given by a collection of polynomials over $\mathbb{F}$.

Definition 6.1 ([PHGR13])**.** A *quadratic arithmetic program* Q over a field $\mathbb{F}$ is a tuple $Q = (\{v_i\}_{i=0}^k, \{w_i\}_{i=0}^k, \{y_i\}_{i=0}^k, t)$, with $v_i, w_i, y_i, t \in \mathbb{F}[x]$ polynomials of degree $\deg v_i, \deg w_i, \deg y_i < \deg t = d$. The polynomial t is called the *target polynomial*. The *size* of the QAP is k; the *degree* is the degree d of t.

In the remainder, for ease of notation, we adopt the convention that $x_0 = 1$.

Definition 6.2. Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP. A tuple $(x_1, \ldots, x_k)$ is a *solution* of Q if t divides $(\sum_{i=0}^k x_i v_i) \cdot (\sum_{i=0}^k x_i w_i) - (\sum_{i=0}^k x_i y_i) \in \mathbb{F}[x]$.

In case t splits, i.e., $t = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_n)$, a QAP can be seen as a collection of rank-1 quadratic equations for $(x_1, \ldots, x_k)$; that is, equations

$v \cdot w - y$ with $v, w, y \in \mathbb{F}[x_1, \ldots, x_k]$ of degree at most one. Namely, $(x_1, \ldots, x_k)$ is a solution of Q if t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, which means exactly that, for every α_j, $(\sum_i x_i v_i(\alpha_j)) \cdot (\sum_i x_i w_i(\alpha_j)) - (\sum_i x_i y_i(\alpha_j)) = 0$: that is, each α_j gives a rank-1 quadratic equation in variables $(x_1, \ldots, x_k)$. Conversely, a collection of d such equations (recall $x_0 \equiv 1$)

$$(v_0^j \cdot x_0 + \ldots + v_k^j \cdot x_k) \cdot (w_0^j \cdot x_0 + \ldots + w_k^j \cdot x_k) - (y_0^j \cdot x_0 + \ldots + y_k^j \cdot x_k)$$

can be turned into a QAP by selecting d distinct elements $\alpha_1, \ldots, \alpha_d$ in $\mathbb{F}$, setting target polynomial $t = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_d)$, and defining v_0 to be the unique polynomial of degree smaller than d for which $v_0(\alpha_j) = v_0^j$, etcetera.

A QAP is said to compute a function $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ if the remaining x_i give a solution exactly if the function is correctly evaluated.

Definition 6.3 ([PHGR13])**.** Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP, and let $f : \mathbb{F}^l \to \mathbb{F}^m$ be a function. We say that Q *computes* f if $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l) \iff \exists\, (x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.

For any function f given by an arithmetic circuit, we can easily construct a QAP that computes the function f. Indeed, we can describe an arithmetic circuit as a series of rank-1 quadratic equations by letting each multiplication gate become one equation. Apart from circuits containing just addition and multiplication gates, we can also express circuits with some other kinds of gates directly as QAPs. For instance, [PHGR13] defines a "split gate" that converts a number a into its k-bit decomposition $a_1, \ldots, a_k$ with equations $a = a_1 + 2 \cdot a_2 + \ldots + 2^{k-1} \cdot a_k$, $a_1 \cdot (1 - a_1) = 0$, …, $a_k \cdot (1 - a_k) = 0$.

Proving Correctness of Computations

If QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ computes a function f, then a prover can prove that $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ by proving knowledge of intermediate wire values $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, i.e., t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$. [PHGR13] gives a construction of a proof system which does exactly this. The proof system assumes discrete logarithm groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ for which the $(4d + 4)$-PDH, d-PKE and $(8d + 8)$-SDH assumptions [PHGR13] hold, with d the degree of the QAP. Moreover, the proof is in the common reference string (CRS) model: the CRS consists of an *evaluation key* used to produce the proof, and a *verification key* used to verify it. Both are public, i.e., provers can know the verification key and the verifiers can know the evaluation key.

To prove that t divides $p = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, the prover computes quotient polynomial $h = p/t$ and basically provides evaluations "in the exponent" of h, $(\sum_i x_i v_i)$, $(\sum_i x_i w_i)$, and $(\sum_i x_i y_i)$ in an unknown point s that can be verified using the pairing. More precisely, given generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$ (written additively) and polynomial $f \in \mathbb{F}[x]$, let us write $\langle f \rangle_1$ for $g_1 \cdot f(s)$ and $\langle f \rangle_2$ for $g_2 \cdot f(s)$. The evaluation key in the CRS, generated using random $s, \alpha_v, \alpha_w, \alpha_y, \beta, r_v, r_w, r_y = r_v \cdot r_w \in \mathbb{F}$, is:

$$\langle r_v v_i \rangle_1, \langle r_v \alpha_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_w \alpha_w w_i \rangle_1, \langle r_y y_i \rangle_1, \langle r_y \alpha_y y_i \rangle_1,$$
$$\langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1, \langle s^j \rangle_1.$$

where i ranges over $l+m+1, l+m+2, \ldots, k$ and j runs from 0 to d, the degree of t. The proof contains the following elements:

$$\begin{array}{rr}
\langle V_{\text{mid}} \rangle_1 = \sum_i \langle r_v v_i \rangle_1 \cdot x_i, & \langle \alpha_v V_{\text{mid}} \rangle_1 = \sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot x_i, \\
\langle W_{\text{mid}} \rangle_2 = \sum_i \langle r_w w_i \rangle_2 \cdot x_i, & \langle \alpha_w W_{\text{mid}} \rangle_1 = \sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot x_i, \\
\langle Y_{\text{mid}} \rangle_1 = \sum_i \langle r_y y_i \rangle_1 \cdot x_i, & \langle \alpha_y Y_{\text{mid}} \rangle_1 = \sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot x_i, \\
\langle Z \rangle_1 = \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot x_i, & \langle H \rangle_1 = \sum_j \langle s^j \rangle_1 \cdot h_j,
\end{array} \tag{6.1}$$

where i ranges over $l + m + 1, l + m + 2, \ldots, k$, and h_j are the coefficients of polynomial $h = p/t$.

To verify that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ and hence $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, a verifier uses the following verification key from the CRS:

$$\langle \alpha_v \rangle_2, \langle \alpha_w \rangle_1, \langle \alpha_y \rangle_2, \langle \beta \rangle_1, \langle \beta \rangle_2, \langle r_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_y y_i \rangle_1, \langle r_y t \rangle_2,$$

where i ranges over $0, 1, 2, \ldots, l + m$[1]. Given the verification key, a proof, and values $x_1, \ldots, x_{l+m}$, the verifier proceeds as follows. First, it checks that

$$\begin{array}{c}
e(\langle V_{\text{mid}} \rangle_1, \langle \alpha_v \rangle_2) = e(\langle \alpha_v V_{\text{mid}} \rangle_1, \langle 1 \rangle_2); \\
e(\langle \alpha_w \rangle_1, \langle W_{\text{mid}} \rangle_2) = e(\langle \alpha_w W_{\text{mid}} \rangle_1, \langle 1 \rangle_2); \\
e(\langle Y_{\text{mid}} \rangle_1, \langle \alpha_y \rangle_2) = e(\langle \alpha_y Y_{\text{mid}} \rangle_1, \langle 1 \rangle_2) :
\end{array} \tag{6.2}$$

intuitively, under the d-PKE assumption, these checks guarantee that the prover must have constructed $\langle V_{\text{mid}} \rangle_1$, $\langle W_{\text{mid}} \rangle_2$, and $\langle Y_{\text{mid}} \rangle_1$ using the elements

[1] In [PHGR13], several terms of the verification key includes a value γ; however, a careful look at [PHGR13]'s proof reveals that γ is actually not needed. We remove it because it simplifies notation, especially for our multi-client protocols.

from the evaluation key. It then checks that

$$e(\langle V_{\text{mid}}\rangle_1 + \langle Y_{\text{mid}}\rangle_1, \langle\beta\rangle_2) \cdot e(\langle\beta\rangle_1, \langle W_{\text{mid}}\rangle_2) = e(\langle Z\rangle_1, \langle 1\rangle_2) : \tag{6.3}$$

under the PDH assumption, this guarantees that the same coefficients x_i were used in $\langle V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, and $\langle Y_{\text{mid}}\rangle_1$. Finally, the verifier computes evaluations $\langle V\rangle_1$ of $\sum_{i=0}^{k} x_i v_i$ as $\langle V_{\text{mid}}\rangle_1 + \sum_{i=0}^{l+m} \langle r_v v_i\rangle_1 \cdot x_i$; $\langle W\rangle_2$ of $\sum_{i=0}^{k} x_i w_i$ as $\langle W_{\text{mid}}\rangle_2 + \sum_{i=0}^{l+m} \langle r_w w_i\rangle_2 \cdot x_i$; and $\langle Y\rangle_1$ of $\sum_{i=0}^{k} x_i y_i$ as $\langle Y_{\text{mid}}\rangle_1 + \sum_{i=0}^{l+m} \langle r_y y_i\rangle_1 \cdot x_i$, and verifies that

$$e(\langle V\rangle_1, \langle W\rangle_2) \cdot e(\langle Y\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2) : \tag{6.4}$$

under the $(8d + 8)$-SDH assumption, this guarantees that, for the polynomial h encoded by $\langle H\rangle_1$, $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ holds.[2]

Theorem 6.4 ([GGPR13], informal)**.** *Given QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ and values $x_1, \ldots, x_{l+m}$, the above is a non-interactive argument of knowledge of $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

Making the Proof Zero-Knowledge

The above proof can be turned into a zero-knowledge proof, that reveals nothing about the values of $(x_{l+m+1}, \ldots, x_k)$ other than that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ for some h, by performing randomisation. Namely, instead of proving that $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, we prove that $t \cdot \tilde{h} = (\sum_i x_i v_i + \delta_v \cdot t) \cdot (\sum_i x_i w_i + \delta_w \cdot t) - (\sum_i x_i y_i + \delta_y \cdot t)$ with $\delta_v, \delta_w, \delta_y$ random from $\mathbb{F}$. Precisely, the evaluation key needs to contain additional elements:

$$\langle r_v t\rangle_1, \langle r_v \alpha_v t\rangle_1, \langle r_w t\rangle_2, \langle r_w \alpha_w t\rangle_1, \langle r_y t\rangle_1, \langle r_y \alpha_y t\rangle_1,$$
$$\langle r_v \beta t\rangle_1, \langle r_w \beta t\rangle_1, \langle r_y \beta t\rangle_1, \langle t\rangle_1.$$

[2] We remark that, as shown in [PHGR13], a verifier who has generated the evaluation and verification keys, can use the randomness from the generation process to save several of the above pairing checks. We do not consider this optimisation here.

Compared to the original proof, we let

$$\begin{aligned}
\langle V'_{\text{mid}}\rangle_1 &= \langle V_{\text{mid}}\rangle_1 + \langle r_v t\rangle_1 \cdot \delta_v,\\
\langle \alpha_v V'_{\text{mid}}\rangle_1 &= \langle \alpha_v V'_{\text{mid}}\rangle_1 + \langle r_v \alpha_v t\rangle_1 \cdot \delta_v,\\
\langle W'_{\text{mid}}\rangle_2 &= \langle W_{\text{mid}}\rangle_2 + \langle r_w t\rangle_2 \cdot \delta_w,\\
\langle \alpha_w W'_{\text{mid}}\rangle_1 &= \langle \alpha_w W_{\text{mid}}\rangle_1 + \langle r_w \alpha_w t\rangle_1 \cdot \delta_w,\\
\langle Y'_{\text{mid}}\rangle_1 &= \langle Y_{\text{mid}}\rangle_1 + \langle r_y t\rangle_1 \cdot \delta_y,\\
\langle \alpha_y Y'_{\text{mid}}\rangle_1 &= \langle \alpha_y Y_{\text{mid}}\rangle_1 + \langle r_y \alpha_y t\rangle_1 \cdot \delta_y,\\
\langle Z'\rangle_1 &= \langle Z\rangle_1 + \langle r_v \beta t\rangle_1 \cdot \delta_v + \langle r_w \beta t\rangle_1 \cdot \delta_w + \langle r_y \beta t\rangle_1 \cdot \delta_y,\\
\langle H'\rangle_1 &= \sum_j \langle s^j\rangle_1 \cdot \widetilde{h}_j,
\end{aligned}$$

with $\widetilde{h}_j$ the coefficients of $h + \delta_v w_0 + \sum_i \delta_v x_i \cdot w_i + \delta_w v_0 + \sum_i \delta_w x_i \cdot v_i + \delta_v \delta_w \cdot t - \delta_y$. Verification remains exactly the same.

Theorem 6.5 ([GGPR13], informal)**.** *Given QAP* $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ *and values* $x_1, \ldots, x_{l+m}$*, the above is a non-interactive zero-knowledge argument of knowledge of* $(x_{l+m+1}, \ldots, x_k)$ *such that* $(x_1, \ldots, x_k)$ *is a solution of* Q*.*

From Arguments of Knowledge to Verifiable Computation

In [PHGR13], the above argument of knowledge is used to construct a *public verifiable computation scheme*. In such a scheme, a client outsources the computation of a function f to a worker, obtaining cryptographic guarantees that the result it gets from the worker is correct. It is defined as follows:

Definition 6.6 ([PHGR13])**.** A *public verifiable computation scheme* $\mathcal{VC}$ consists of three polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Compute}, \mathsf{Verify})$:

- $(\mathrm{EK}_f; \mathrm{VK}_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda)$: a probabilistic *key generation algorithm* that takes as argument a function $f : \mathbb{F}^l \to \mathbb{F}^m$ and a security parameter λ, outputting a public evaluation key EK_f and a public verification key VK_f
- $(\boldsymbol{y}; \pi) \leftarrow \mathsf{Compute}(\mathrm{EK}_f; \boldsymbol{x})$: a probabilistic *worker algorithm* that takes input $\boldsymbol{x} \in \mathbb{F}^l$ and outputs $\boldsymbol{y} = f(\boldsymbol{x}) \in \mathbb{F}^k$ and a proof π of its correctness
- $\{0, 1\} \leftarrow \mathsf{Verify}(\mathrm{VK}_f; \boldsymbol{x}; \boldsymbol{y}; \pi)$: a deterministic *verification algorithm* that outputs 1 if $\boldsymbol{y} = f(\boldsymbol{x})$, 0 otherwise.

To outsource the computation of f, a client runs KeyGen and provides EK_f to the worker. When it needs $f(\boldsymbol{x})$, it provides $\boldsymbol{x}$ to the worker, who runs $\mathsf{Compute}$ and provides the result $\boldsymbol{y} = f(\boldsymbol{x})$ and proof π to the client. The client accepts $\boldsymbol{y}$ if Verify succeeds. We require that worker cannot provide incorrect proofs even if it knows VK_f, which makes this verifiable computation scheme "public". In fact, a trusted party could perform KeyGen once and for all and publish $(\mathrm{EK}_f, \mathrm{VK}_f)$; any client who trusts this party can then use the published VK_f to verify computations. (Trusting this party is needed: the random values used in KeyGen are a trapdoor with which the generator of the keys can produce false proofs.) A public verifiable computation scheme should satisfy *correctness* and *security*. Correctness means that honest workers produce accepting proofs:

Definition 6.7 ([PHGR13])**.** A public verifiable computation scheme $\mathcal{VC}$ is called *correct* if, for all $f : \mathbb{F}^l \to \mathbb{F}^m$ and $\boldsymbol{x} \in \mathbb{F}$:

$$\begin{aligned}&\text{if } (\mathrm{EK}_f; \mathrm{VK}_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\boldsymbol{y}; \pi) \leftarrow \mathsf{Compute}(\mathrm{EK}_f; \boldsymbol{x}),\\ &\quad \text{then } \mathsf{Verify}(\mathrm{VK}_f; \boldsymbol{x}; \boldsymbol{y}; \pi) = 1.\end{aligned}$$

Security means that corrupt workers cannot convince clients of wrong results:

Definition 6.8 ([PHGR13])**.** A public verifiable computation scheme $\mathcal{VC}$ is called *secure* if, for any $f : \mathbb{F}^l \to \mathbb{F}^m$ and probabilistic polynomial time adversary $\mathcal{A}$:

$$\begin{aligned}&\mathsf{Pr}[\ (\mathrm{EK}_f, \mathrm{VK}_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\boldsymbol{x}; \boldsymbol{y}; \pi) \leftarrow \mathcal{A}(\mathrm{EK}_f; \mathrm{VK}_f) :\\ &\quad \boldsymbol{y} \neq f(\boldsymbol{x}) \wedge \mathsf{Verify}(\mathrm{VK}_f; \boldsymbol{x}; \boldsymbol{y}; \pi) = 1\ ] = \mathsf{negl}(\lambda).\end{aligned}$$

Given a QAP Q that computes a function f, the argument of knowledge from Section 6.3 directly gives a public verifiable computation scheme known as Pinocchio [PHGR13]: KeyGen is the computation of the evaluation and verification keys for Q; $\mathsf{Compute}$ computes $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, and proof (6.1); and Verify are the checks (6.2–6.4) for this proof.

Theorem 6.9 (Pinocchio [PHGR13], informal)**.** *Let QAP Q be of degree d. Then the above construction is a secure and correct public verifiable computation scheme under the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions.*

6.4 Distributing the Prover Computation

In this section, we present the single-client version of our Trinocchio protocol for privacy-preserving outsourcing. In Trinocchio, a client distributes computation of a function $x_2 = f(x_1)$ to n workers (we consider here single-valued input and output, but the generalisation is straightforward). Trinocchio guarantees correct function evaluation (regardless of corruptions) and secure function evaluation (if at most θ workers are passively corrupted, where $n = 2\theta + 1$). Trinocchio in effect distributes the proof computation of Pinocchio; the number of workers to obtain privacy against one semi-honest worker is three, hence its name.

Multiparty Computation using Shamir Secret Sharing

To distribute the Pinocchio computation, Trinocchio employs multiparty computation techniques based on Shamir secret sharing [BGW88]. Recall that in (θ, n) Shamir secret sharing, a party shares a secret s among n parties so that $\theta + 1$ parties are needed to reconstruct s. It does this by taking a random degree-$\leq \theta$ polynomial $p(x) = \alpha_\theta x^\theta + \ldots + \alpha x + s$ with s as constant term and giving $p(i)$ to party i. Since $p(x)$ is of degree at most θ, $p(0)$ is completely independent from any θ shares but can be easily computed from any $\theta + 1$ shares by Lagrange interpolation. We denote such a sharing as $[\![s]\!]$. Note that Shamir-sharing can also be done "in the exponent", e.g., $[\![\langle a \rangle_1]\!]$ denotes a Shamir sharing of $\langle a \rangle_1 \in \mathbb{G}_1$ from which $\langle a \rangle_1$ can be computed using Lagrange interpolation in $\mathbb{G}_1$.

Shamir secret sharing is linear, i.e., $[\![a+b]\!] = [\![a]\!] + [\![b]\!]$ and $[\![\alpha a]\!] = \alpha [\![a]\!]$ can be computed locally. When computing the product of $[\![a]\!]$ and $[\![b]\!]$, each party i can locally multiply its points $p_a(i)$ and $p_b(i)$ on the random polynomials p_a and p_b. Because the product polynomial has degree at most 2θ, this is a $(2\theta, n)$ sharing, which we write as $[a \cdot b]$ (note that reconstructing the secret requires $n = 2\theta + 1$ parties). Moreover, the distribution of the shares of $[a \cdot b]$ is not independent from the values of a and b, so when revealed, these shares reveal information about a and b. Hence, in multiparty computation, $[a \cdot b]$ is typically converted back into a random (θ, n) sharing $[\![a \cdot b]\!]$ using an interactive protocol due to [GRR98]. Interactive protocols for many other tasks such as comparing two shared value also exist (see, e.g., [dH12]).

The Trinocchio protocol

We now present the Trinocchio protocol. Trinocchio assumes that Pinocchio's KeyGen has been correctly performed, i.e., formally, Trinocchio works in the KeyGen-hybrid model. Furthermore, Trinocchio assumes pairwise private, synchronous communication channels. To obtain $x_2 = f(x_1)$, a client proceeds in four steps:

- The client obtains the verification key, and the workers obtain the evaluation key, using hybrid calls to KeyGen.
- The client secret shares $[\![x_1]\!]$ of its input to the workers.
- The workers use multiparty computation to compute secret-shares of the output, $[\![x_2]\!]$, and of the Pinocchio proof elements, $[\![\langle V_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_v V_{\text{mid}}\rangle_1]\!]$, $[\![\langle W_{\text{mid}}\rangle_2]\!]$, $[\![\langle \alpha_w W_{\text{mid}}\rangle_1]\!]$, $[\![\langle Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_y Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle Z\rangle_1]\!]$, and $[\langle H\rangle_1]$, as we explain next; and sends these shares to the client.
- The client recombines the shares of the Pinocchio proof elements into $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, $\langle Z\rangle_1$, $\langle H\rangle_1$ by Lagrange interpolation, and accepts x_2 as computation result if Pinocchio's Verify returns success.

Algorithm 6.1 shows in detail how the secret-shares of the function output and Pinocchio proof are computed. The first step is to compute function output $x_2 = f(x_1)$ and values $(x_3, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of the QAP (line 4). This is done using normal multiparty computation protocols based on secret sharing. If function f is represented by an arithmetic circuit, then it is evaluated using local addition and scalar multiplication, and the multiplication protocol from [GRR98]. If f is represented by a circuit using more complicated gates, then specific protocols may be used: e.g., the split gate discussed in Section 6.3 can be evaluated using multiparty bit decomposition protocols [DFK+06, ST06]. Any protocol can be used as long as it guarantees privacy, i.e., the view of any θ workers is statistically independent from the values represented by the shares.

The next task is to compute, in secret-shared form, the coefficients of the polynomial $h = ((\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i))/t \in \mathbb{F}[x]$ that we need for proof element $\langle H\rangle_1$. In theory, this computation could be performed by first computing shares of the coefficients of $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, and then dividing by t, which can be done locally using traditional polynomial long division. However, this scales quadratically in the degree of the QAP and hence

Algorithm 6.1 Trinocchio's Compute protocol

▷ $\mathcal{S} = \{\alpha_1, \ldots, \alpha_d\}$ denotes the roots of the target polynomial of the QAP
▷ $\mathcal{T} = \{\beta_1, \ldots, \beta_d\}$ denotes a list of distinct points different from $\mathcal{S}$
function $\mathsf{Compute}(\mathsf{EK}_f = \{\langle r_v v_i \rangle_1\}_i, \ldots, \{\langle s^j \rangle_1\}_j; [\![x_1]\!])$
  $([\![x_2]\!], \ldots, [\![x_k]\!]) \leftarrow f([\![x_1]\!])$
  $[\![\boldsymbol{v}]\!] \leftarrow \{\sum_i v_i(\alpha_j) \cdot [\![x_i]\!]\}_j$
  $[\![\boldsymbol{w}]\!] \leftarrow \{\sum_i w_i(\alpha_j) \cdot [\![x_i]\!]\}_j$
  $[\![\boldsymbol{y}]\!] \leftarrow \{\sum_i y_i(\alpha_j) \cdot [\![x_i]\!]\}_j$
  $[\![\boldsymbol{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{v}]\!])$
  $[\![\boldsymbol{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{w}]\!])$
  $[\![\boldsymbol{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{y}]\!])$
  $[\![\boldsymbol{v}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{V}]\!])$
  $[\![\boldsymbol{w}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{W}]\!])$
  $[\![\boldsymbol{y}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{Y}]\!])$
  $[\boldsymbol{h}'] \leftarrow \{([\![\boldsymbol{v}'_j]\!] \cdot [\![\boldsymbol{w}'_j]\!] - [\![\boldsymbol{y}'_j]\!])/t(\beta_j)\}_j$
  $[\boldsymbol{H}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h}'])$
  $[\![\langle V_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v v_i \rangle_1 \cdot [\![x_i]\!]$
  $[\![\langle \alpha_v V_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot [\![x_i]\!]$
  $[\![\langle W_{\text{mid}} \rangle_2]\!] \leftarrow \sum_i \langle r_w w_i \rangle_2 \cdot [\![x_i]\!]$
  $[\![\langle \alpha_w W_{\text{mid}} \rangle_1]\!] \sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot [\![x_i]\!]$
  $[\![\langle Y_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y y_i \rangle_1 \cdot [\![x_i]\!]$
  $[\![\langle \alpha_y Y_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot [\![x_i]\!]$
  $[\![\langle Z \rangle_1]\!] \leftarrow \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot [\![x_i]\!]$
  $[\langle H \rangle_1] = \sum_j \langle s^j \rangle_1 \cdot [\boldsymbol{H}_j]$
  return $([\![x_2]\!]; [\![\langle V_{\text{mid}} \rangle_1]\!], [\![\langle \alpha_v V_{\text{mid}} \rangle_1]\!], [\![\langle W_{\text{mid}} \rangle_2]\!], [\![\langle \alpha_w W_{\text{mid}} \rangle_1]\!],$
    $[\![\langle Y_{\text{mid}} \rangle_1]\!], [\![\langle \alpha_y Y_{\text{mid}} \rangle_1]\!], [\![\langle Z \rangle_1]\!], [\langle H \rangle_1])$

leads to unacceptable performance. Hence, we take the approach based on fast Fourier transforms (FFTs) from [BCG+13], and adapt it to the distributed setting. Given a list $\mathcal{S} = \{\omega_1, \ldots, \omega_d\}$ of distinct points in $\mathbb{F}$, we denote by $\boldsymbol{P} = \mathsf{FFT}_{\mathcal{S}}(\boldsymbol{p})$ the transformation from coefficients $\boldsymbol{p}$ of a polynomial p of degree at most $d-1$ to evaluations $p(\omega_1), \ldots, p(\omega_d)$ in the points in $\mathcal{S}$. We denote by $\boldsymbol{p} = \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{P})$ the inverse transformation, i.e., from evaluations to coefficients. Deferring specifics to later, we mention now that the FFT is a linear transformation that, for some $\mathcal{S}$, can be performed locally on secret-shares in $O(d \log d)$.

With FFTs available, we can compute the coefficients of h by evaluating h

in d distinct points and applying FFT^{-1}. Note that we can efficiently compute evaluations $\boldsymbol{v}$ of $v = (\sum_i x_i v_i)$, $\boldsymbol{w}$ of $w = (\sum_i x_i w_i)$, and $\boldsymbol{y}$ of $y = (\sum_i x_i y_i)$ in the zeros $\{\omega_1, \ldots, \omega_d\}$ of the target polynomial (lines 5–7). Namely, the values $v_i(\omega_j)$, $w_i(\omega_j)$, $y_i(\omega_j)$ are simply the coefficients of the quadratic equations represented by the QAP, most of which are zero, so these sums have much fewer than k elements (if this were not the case, then evaluating v, w, and y would take an unacceptable $O(d \cdot k)$). Unfortunately, we cannot use these evaluations directly to obtain evaluations of h, because this requires division by the target polynomial, which is zero in exactly these points ω_i. Hence, after determining $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{y}$, we first use the inverse FFT to determine the coefficients $\boldsymbol{V}$, $\boldsymbol{W}$, and $\boldsymbol{Y}$ of v, w, and y (lines 8–10), and then again the FFT to compute the evaluations $\boldsymbol{v'}$, $\boldsymbol{w'}$, and $\boldsymbol{y'}$ of v, w, and y in another set of points $\mathcal{T} = \{\Omega_1, \ldots, \Omega_k\}$ (lines 11–13). Now, we can compute evaluations $\boldsymbol{h'}$ of h in $\mathcal{T}$ using $h(\Omega_i) = (v(\Omega_i) \cdot w(\Omega_i) - y(\Omega_i))/t(\Omega_i)$. This requires a multiplication of (θ, n)-secret-shares of $v(\Omega_i)$ and $w(\Omega_i)$, hence the result is a $(2\theta, n)$-sharing. Finally, the inverse FFT gives us a $(2\theta, n)$-sharing of the coefficients $\boldsymbol{H}$ of h (lines 14 and 15).

Given secret-shares of the values of x_i and coefficients of h, it is straightforward to compute secret-shares of the Pinocchio proof. Indeed, $\langle V_{\text{mid}} \rangle_1, \ldots, \langle H \rangle_1$ are all computed as linear combinations of elements in the evaluation key, so shares of these proof elements can be computed locally (lines 16–23), and finally returned by the respective workers (lines 24–24).

Note that, compared to Pinocchio, our client needs to carry out slightly more work. Namely, our client needs to produce secret-shares of the inputs and recombine secret-shares of the outputs; and it needs to recombine the Pinocchio proof. However, according to the micro-benchmarks from [PHGR13], this overhead is small. For each input and output, Verify includes three exponentiations, whereas Combine involves four additions and two multiplications; when using [PHGR13]'s techniques, this adds at most a 3% overhead. Recombining the Pinocchio proof involves 15 exponentiations at around half the cost of a single pairing. Alternatively, it is possible to let one of the workers perform the Pinocchio recombining step by using the distributed zero-knowledge variant of Pinocchio (Section 6.3) and the techniques from Section 6.5. In this case, the only overhead for the client is the secret-sharing of the inputs and zero-knowledge randomness, and recombining the outputs.

Parameters for Efficient FFTs To obtain efficient FFTs, we use the approach of [BCG+13]. There, it is noted that the operation $\boldsymbol{P} = \mathsf{FFT}_{\mathcal{S}}(\boldsymbol{p})$ and its

inverse can be efficiently implemented if $\mathcal{S} = \{\omega, \omega^2, \ldots, \omega^d = 1\}$ is a set of powers of a primitive dth root of unity, where d is a power of two. (We can always demand that QAPs have degree $d = 2^k$ for some k by adding dummy equations.) Moreover, [BCG+13] presents a pair of groups $\mathbb{G}_1, \mathbb{G}_2$ of order q such that $\mathbb{F}_q$ has a primitive 2^{30}th root of unity (and hence also primitive 2^kth roots of unity for any $k < 30$) as well as an efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$. Finally, [BCG+13] remarks that for $\mathcal{T} = \{\eta\omega, \eta\omega^2, \ldots, \eta\omega^d = \eta\}$, operations $\mathsf{FFT}_{\mathcal{T}}^{-1}$ and $\mathsf{FFT}_{\mathcal{T}}^{-1}$ can easily be reduced to $\mathsf{FFT}_{\mathcal{S}}$ and $\mathsf{FFT}_{\mathcal{S}}^{-1}$, respectively. In our implementation, we use exactly these suggested parameters.

Security of Trinocchio

Theorem 6.10. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the Trinocchio protocol. Assuming the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the* KeyGen*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* KeyGen*-hybrid model.*

The proof of this theorem is easily derived as a special case of the proof for the multi-client Trinocchio protocol later. Here, we present a short sketch.

Sketch. To prove correct function evaluation, we need to show that for every real-world adversary $\mathcal{A}$ interacting with Trinocchio, there is an ideal-world simulator $\mathcal{S}_{\mathcal{A}}$ that interacts with the trusted party for correct function evaluation such that the two executions give indistinguishable results. The only interesting case is when the client is honest and some of the workers are not. In this case, the simulator receives the input of the honest party, and needs to choose whether to provide the output. To this end, the simulator simply simulates a run of the actual protocol with $\mathcal{A}$, until it has finally obtained function output x_2 and the accompanying Trinocchio proof. If the proof verifies, it tells the trusted party to provide the output to the client; otherwise, it tells the trusted party not to. Finally, the simulator outputs whatever $\mathcal{A}$ outputs. Because Pinocchio is secure, except with negligible probability a verifying proof implies that the real-world output of the client (as given by the adversary) matches the ideal-world output of the client (as computed by the trusted party); and by construction, the outputs of $\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}}$ are distributed identically. This proves correct function evaluation.

For secure function evaluation, again the only interesting case is if the client is honest and some of the workers are passively corrupted. In this case, because corruption is only passive, correctness of the multiparty protocol used to compute f and correctness of the Pinocchio proof system used to compute the proof together imply that real-world executions (like ideal-world executions) result in the correct function result and a verifying proof. Hence, we only need to worry about how $\mathcal{S}_{\mathcal{A}}$ can simulate the view of $\mathcal{A}$ on the Trinocchio protocol without knowing the client's input. However, note that the workers only use a multiparty computation to compute f (which we assume can be simulated without knowing the inputs), after which they no longer receive any messages. Hence simulating the multiparty computation for f and receiving any messages that $\mathcal{A}$ sends is sufficient to simulate $\mathcal{A}$. This proves secure function evaluation. □

Privacy against Active Attacks We remark that actually, Trinocchio in some cases provides privacy against actively corrupted workers as well. Namely, suppose that the protocol used to compute f does not leak any information to corrupted workers in the event of an active attack (even though in this case it may not guarantee correctness). For instance, this is the case for the protocol from [GRR98]: the attacker can manipulate the shares that it sends, which makes the computation return incorrect results; but since the attacker always learns only θ many shares of any value, it does not learn any information. Because the attacker learns no additional information from producing the Pinocchio proof, the overall protocol still leaks no information to the adversary. (And security of Pinocchio ensures the client notices the attacker's manipulation.)

This crucially relies on the workers not learning whether the client accepts the proof: if the workers would learn whether the client obtained a validating proof, then, by manipulating proof construction, they could learn whether a modified version of the tuple $(x_1, \ldots, x_k)$ is a solution of the QAP used, so corrupted workers could learn one chosen bit of information about the inputs (cf. [MF06a]).

6.5 Handling Mutually Distrusting In- and Outputters

We now consider the scenario where there are multiple (possibly overlapping) input and result parties. There are some significant changes between this scenario and the single-client scenario. In particular, we need to extend Pinocchio to allow verification not based on the actual input/output values (indeed, no

Algorithm 6.2 ProofBlock

1: **function** ProofBlock($BK; \boldsymbol{x}; \delta_v, \delta_w, \delta_y$)
2: $\langle V\rangle_1 \leftarrow \langle r_v t\rangle_1 \delta_v + \sum_i \langle r_v v_i\rangle_1 x_i$
3: $\langle V'\rangle_1 \leftarrow \langle r_v \alpha_v t\rangle_1 \delta_v + \sum_i \langle r_v \alpha_v v_i\rangle_1 x_i$
4: $\langle W\rangle_2 \leftarrow \langle r_w t\rangle_2 \delta_w + \sum_i \langle r_w w_i\rangle_2 x_i$
5: $\langle W'\rangle_1 \leftarrow \langle r_w \alpha_w t\rangle_1 \delta_w + \sum_i \langle r_w \alpha_w w_i\rangle_1 x_i$
6: $\langle Y\rangle_1 \leftarrow \langle r_y t\rangle_1 \delta_y + \sum_i \langle r_y y_i\rangle_1 x_i$
7: $\langle Y'\rangle_1 \leftarrow \langle r_y \alpha_y t\rangle_1 \delta_y + \sum_i \langle r_y \alpha_y y_i\rangle_1 x_i$
8: $\langle Z\rangle_1 \leftarrow \langle r_v \beta t\rangle_1 \delta_v + \langle r_w \beta t\rangle_1 \delta_w + \langle r_y \beta t\rangle_1 \delta_y + \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i\rangle_1 x_j$
9: **return** $(\langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$

party sees all of them) but on some kind of representation that does not reveal them. Moreover, we need to use the zero-knowledge variant of Pinocchio (Section 6.3), and we need to make sure that input parties choose their inputs independently from each other.

Multi-Client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio (Section 6.3) with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, and $\langle Z\rangle_1$ encode circuit values $x_{l+m+1}, \ldots, x_k$; in the zero-knowledge variant, these terms are randomised so that they do not reveal any information about $x_{l+m+1}, \ldots, x_k$. In the multi-client case, additionally, the inputs of all input parties and the outputs of all result parties need to be encoded such that no other party learns any information about them. Therefore, we extend the proof with *blocks* of the above seven terms for each input and result party, which are constructed in the same way as the seven proof terms above. Although some result parties could share a block of output values, for simplicity we assign each result party its own block in the protocol.

To produce a block containing values $\boldsymbol{x}$, a party first samples three random field values δ_v, δ_w, and δ_y and then executes ProofBlock, cf. Algorithm 6.2. The BK argument to this algorithm is the *block key*; the subset of the evaluation key terms specific to a single proof block. Because each input party should only provide its own input values and should not affect the values contributed by other parties, each proof block must be restricted to a subset of the wires. This is achieved by modifying Pinocchio's key generation such that, instead of

Algorithm 6.3 CheckBlock

```
1: function CheckBlock(BV; ⟨V⟩_1, ⟨V'⟩_1, ⟨W⟩_2, ⟨W'⟩_1, ⟨Y⟩_1, ⟨Y'⟩_1, ⟨Z⟩_1)
2:     if e(⟨V⟩_1, ⟨α_v⟩_2) = e(⟨V'⟩_1, ⟨1⟩_2) ∧
3:         e(⟨α_w⟩_1, ⟨W⟩_2) = e(⟨W'⟩_1, ⟨1⟩_2) ∧
4:         e(⟨Y⟩_1, ⟨α_y⟩_2) = e(⟨Y'⟩_1, ⟨1⟩_2) ∧
5:         e(⟨Z⟩_1, ⟨1⟩_2) = e(⟨V⟩_1 + ⟨Y⟩_1, ⟨β⟩_2)e(⟨β⟩_1, ⟨W⟩_2)
6:     then
7:         return ⊤
8:     else
9:         return ⊥
```

a sampling a single value β, one such value, β_j, is sampled for each proof block j and the terms $\langle r_v\beta_j v_i + r_w\beta_j w_i + r_y\beta_j y_i\rangle_1$ are only included for wires indices i belonging to block j. That is, the jth block key is

$$BK_j = \{\langle r_v v_i\rangle_1, \langle r_v\alpha_v v_i\rangle_1, \langle r_w w_i\rangle_2, \langle r_w\alpha_w w_i\rangle_1, \langle r_y y_i\rangle_1, \langle r_y\alpha_y y_i\rangle_1, \\ \langle r_v\beta_j v_i + r_w\beta_j w_i + r_y\beta_j y_i\rangle_1, \langle r_v\beta_j t\rangle_1, \langle r_w\beta_j t\rangle_1, \langle r_y\beta_j t\rangle_1\},$$

with i ranging over the indices of wires in the block. Note that ProofBlock only performs linear operations on its $\boldsymbol{x}$, δ_v, δ_w and δ_y inputs. Therefore this algorithm does not have to be modified to compute on secret-shares.

A Trinocchio proof in the multi-client setting now consists of one proof block $\boldsymbol{Q}_i = (\langle V_i\rangle_1, \ldots, \langle Z_i\rangle_1)$ for each input and result party, one proof block $\boldsymbol{Q}_{\text{mid}} = (\langle V_{\text{mid}}\rangle_1, \ldots, \langle Z_{\text{mid}}\rangle_1)$ of internal wire values, and Pinocchio's $\langle H\rangle_1$ element. Verification of such a proof consists of checking correctness of each block, and checking correctness of $\langle H\rangle_1$. The validity of a proof block can be verified using CheckBlock, cf. Algorithm 6.3.

Compared to the Pinocchio verification key, our verification key contains "block verification keys" BV_i (i.e., elements $\langle\beta_j\rangle_1$ and $\langle\beta_j\rangle_2$) for each block instead of just $\langle\beta\rangle_1$ and $\langle\beta\rangle_2$. Apart from the relations inspected by CheckBlock, one other relation is needed to verify a Pinocchio proof: the divisibility check of Equation (6.4). In the protocol, the algorithm that verifies this relation will be called CheckDiv (displayed in Algorithm 6.4 for completeness). We denote the modified setup of the evaluation and verification keys by hybrid call MKeyGen.

Protocol Overview

We will proceed with a protocol overview. Pseudocode and a more detailed description of the protocol are given in Sections 6.6 and 6.7. The multi-client

Algorithm 6.4 CheckDiv

```
1: function CheckDiv(VK; ⟨V⟩_1, ⟨V'⟩_1, ⟨W⟩_2, ⟨W'⟩_1, ⟨Y⟩_1, ⟨Y'⟩_1, ⟨Z⟩_1; ⟨H⟩_1)
2:     if e(⟨V⟩_1, ⟨W⟩_2) · e(⟨Y⟩_1, ⟨1⟩_2)^{-1} = e(⟨H⟩_1, ⟨r_y t⟩_2) then
3:         return ⊤
4:     else
5:         return ⊥
```

variant of our Trinocchio protocol makes use of private channels, just as the single-client variant, to privately communicate in- and output values, and to let the workers carry out the computation. We need some additional communication to ensure input independence and fix the input parties' values. For this we use a bulletin board. To achieve input independence, we first have the input parties commit to a representation of their input and then reveal these, which requires the use of a commitment scheme.

Apart from key set-up there are three phases to the multi-client Trinocchio protocol.

- In the *input phase*, the input parties provide representations of their input on the bulletin board. These representations are later used as part of the proof to verify the computation results. They also serve to ensure that each input party provides its value independent of the other input values. The input parties then secret share their input values to the workers. The workers verify that the secret-shared input values are consistent with their representations on the bulletin board, to prevent malicious input parties from providing a different value.

- The *computation phase* is very similar to the single-client variant of Trinocchio. In this phase the workers perform multi-party computation to carry out the actual computation and obtain secret-shares of intermediate and result wire values. They then use these secret-shared wire values to construct shares of the proof elements. These are then posted on the bulletin board, instead of being communicated directly to the result parties to ensure that all result parties receive a consistent result. In order to prevent these proof elements from revealing any information about the wire values, the zero-knowledge variant of the proof is used (Section 6.3).

- In the *result phase* the workers privately send the shares of the result values to the result parties. The result parties recombine the proof shares

from the bulletin board and check whether the proof verifies. The result parties further check whether the recombined shares of the result are consistent with the information on the bulletin board. The result parties only accept the result received from the workers if both checks are satisfied.

Security of the Trinocchio Protocol

Analogously to the single-client case, we obtain the following result:

Theorem 6.11. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the multi-client Trinocchio protocol. Assuming the d-PKE, $(4d + 4)$-PDH, and $(8d + 8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*

We stress that "at most θ workers are passively corrupted" includes both the case when the adversary is passively corrupted, and corrupts at most θ workers (as well as arbitrarily many input and result parties); and the case when the adversary is actively corrupted, and corrupts no workers (but arbitrarily many input and result parties)

We give a proof of this theorem in Section 6.7. To prove secure function evaluation, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. To prove correct function evaluation, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.

In the single-client case, we remarked that Trinocchio actually provides security against up to θ *actively* corrupted workers. Namely, although θ actively corrupted workers may manipulate the computation of the function and proof, they do not learn any information from this because they do not see the resulting proof that the client gets. In our multi-client protocol, it is less natural to assume that the workers cannot see the resulting proof; and in fact, in our protocol, corrupted workers *do* see the full proof as it is posted on the bulletin board. It should be possible to obtain some privacy guarantees against actively malicious workers (who do not collude with any result parties) by letting the result parties provide proof contributions directly to the result parties instead of posting them on the bulletin board. We leave an analysis for future work.

6.6 Multi-client Protocol

We now present our multi-client Trinocchio protocol of Section 6.5 in more detail. As before, we assume that each input party provides only a single input and each result party receives only a single output; that is, each block from Section 6.5 consists of only one wire. It should be clear from Section 6.5 how this can be generalised.

Communication Model and Notation

We assume synchronous communication; pairwise secure channels between the input parties and workers; between the workers themselves; and between the workers and result parties. To ensure agreement between the parties about the inputs for the computation, we additionally assume a bulletin board. Through this bulletin board, parties can publish messages which can then be retrieved by any other party. Messages on the bulletin board are authenticated. In our protocol, we denote a party posting a message m as $\mathsf{Post}(m)$. For convenience, we don't explicitly denote a party retrieving information from the bulletin board; instead, we take $\mathsf{Post}(m)$ to mean that any party can now use the value for m.

Mixed Commitment Scheme

We use a commitment scheme, which allows a party to commit to a certain value, without revealing that value to other parties, but, when at a later time this value is revealed, the other parties can be certain that the revealed value is equal to the original committed to value. Each party has its own public commitment key k and a commitment to a value v using randomness r is denoted $\mathsf{Commit}_k(v; r)$. Because, given explicit randomness, the commitment algorithm is deterministic, the commitment can be opened by simply revealing (v, r). Then any party can verify the commitment by simply recomputing it. To ensure input independence, the commitment scheme must be non-malleable. Each input party will produce one commitment, so each commitment key is used only once.

In particular, we use a so-called "mixed commitment scheme" [DN02]. In such a scheme, commitment keys can be generated in two ways. First, they can be generated such that the scheme is perfectly binding and computationally hiding, and a trapdoor exists with which the committed value can be extracted. Second, they can be generated such that the scheme is perfectly

Algorithm 6.5 Trinocchio: key set-up

1: **parties** $i \in \mathcal{I}$ **do**
2: $(k_1, \ldots, k_n) \leftarrow \mathsf{ComGen}()$
3: **parties** $i \in \mathcal{I} \cup \mathcal{C} \cup \mathcal{R}$ **do**
4: $(EK = (\{BK_i\}_i, \ldots), VK = (\{BV_i\}_i, \ldots)) \leftarrow \mathsf{MKeyGen}()$

hiding and computationally binding, and a trapdoor exists with which commitments can be opened to any value. Moreover, the keys generated in the two ways should be computationally indistinguishable. In our protocol, commitment keys of the first, i.e., perfectly binding, kind are generated for all input parties by a trusted party (and the trapdoor thrown away), which we model by a hybrid call $k_1, \ldots, k_l \leftarrow \mathsf{ComGen}$. (In the simulator used for the security proof, commitment keys of the first kind are generated for corrupted input parties and commitment keys of the second kind are generated for honest input parties, with the trapdoors used when simulating the adversary.) Mixed commitments can be instantiated efficiently, e.g., using a cryptographic hash function in the random oracle model; or using Paillier encryption [DN02]: in this latter case, perfectly binding commitment keys are $k = (1 + N)^u s^N \bmod N^2$, perfectly hiding commitment keys are $k = s^N \bmod N^2$, and commitments are $k^v r^N \bmod N^2$.

The Protocol

As discussed in the protocol overview (Section 6.5), the protocol starts with hybrid calls to obtain the trusted commitment keys and Trinocchio evaluation and verification keys (given in Algorithm 6.5 for completeness). The remainder of the protocol consists of the *input phase*, in which the input parties provide their inputs to the workers; the *computation phase*, in which the workers compute the function and Pinocchio proof; and the *result phase*, in which the result parties obtain the output from the workers and verify its correctness. In the following, we will describe each of these three phases separately.

Input Phase

The input phase of Trinocchio is displayed in Algorithm 6.6. In the input phase, each input party provides its input to the workers. Compared to the single-client case, in which the input party simply provided secret-shares of its inputs, we need to take several additional steps. Namely, we need each input party to

provide a block for its inputs that other parties can use to verify the proof; and we need to guarantee input independence, namely, that input parties cannot choose their inputs depending on those of others.

To achieve these goals, we proceed as follows. First, each input party computes a block for its input (lines 7 and 8). Having each input party post its block on the bulletin board would break input independence (in effect, it binds the input parties who provide the blocks first). We circumvent this by letting each input party post a commitment to its block first (lines 9–11). After all commitments have been posted, the input parties post the openings to the commitments, i.e., the blocks and commitment randomness (lines 12 and 13). (This guarantees input independence because in the security proof, the inputs of the honest parties can still be changed after the corrupted parties provide their inputs.) After this, the validity of the commitments (line 16) and blocks (line 18) are checked; if any input party provided incorrect information, the computation is aborted. Note that ProofBlock used by the input parties could already be considered a commitment scheme [CFH+15], however, because of the way the CRS is constructed and used in the security proof for the protocol, we cannot make use of the trapdoor that would make it equivocable.

After the input blocks have been posted and checked, the inputs are provided to the workers in the form of $(2\theta, n)$ shares (lines 19 and 20). The shared information is both input $[x_i]$ and block randomness $[\delta_{v,i}]$, $[\delta_{w,i}]$, $[\delta_{y,i}]$: the workers need this latter information to compute the proof's $\langle H \rangle_1$ element. Note that we use $(2\theta, n)$ shares: because $n = 2\theta + 1$, the shares of all workers recombine to a unique value and we do not need to worry about input parties handing out inconsistent shares. The workers check that the shares correspond to the broadcast block by computing additive shares of the block, posting them, and checking if their Shamir recombination (denoted by Combine) matches the value on the bulletin board (lines 22–25). Finally, the $(2\theta, n)$-shares are converted into (θ, n)-shares (each worker (θ, n)-shares its share and applies recombination à la [GRR98]) used for the remainder of the computation (line 26).

Computation Phase

The computation phase of Trinocchio is displayed in Algorithm 6.7. In the computation phase, the workers compute function f, and produce a Pinocchio proof that this computation was performed correctly. The computation of f (line 29) and coefficients $\boldsymbol{H'}$ of the polynomial $h = (v \cdot w - y)/t$ (lines 30–40) are the same as in the single-client case. To generate the proof block for the internal wires, the workers first generate shared random values $[\![\delta_{v,\text{mid}}]\!]$, $[\![\delta_{w,\text{mid}}]\!]$, $[\![\delta_{y,\text{mid}}]\!]$ (line 41):

Algorithm 6.6 Trinocchio: input phase

5: ▷ Continued from Algorithm 6.5
6: **parties** $i \in \mathcal{I}$ **do**
7: $(\delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \xleftarrow{\$} \mathbb{F}^3$
8: $\boldsymbol{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$
9: sample commitment randomness ρ_i
10: $c_i \leftarrow \mathsf{Commit}_{k_i}(\boldsymbol{Q}_i; \rho_i)$
11: $\mathsf{Post}(c_i)$
12: wait until all other input parties have posted commitments
13: $\mathsf{Post}(\boldsymbol{Q}_i, \rho_i)$
14: **for all** $j \in \mathcal{I} \setminus \{i\}$ **do**
15: **if** $c_j \neq \mathsf{Commit}_{k_j}(\boldsymbol{Q}_j; \rho_j)$ **then**
16: abort the protocol
17: **if** $\mathsf{CheckBlock}(BV_j; \boldsymbol{Q}_j) = \perp$ **then**
18: abort the protocol
19: create $(2\theta, n)$-shares $([x_i], [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$
20: distribute shares $([x_i], [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$ to the workers
21: **parties** $\mathcal{C}$ **do**
22: **for all** $i \in \mathcal{I}$ **do**
23: $[\boldsymbol{Q}_i] \leftarrow \mathsf{ProofBlock}(BK_i; [x_i]; [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$
24: $\mathsf{Post}([\boldsymbol{Q}_i])$
25: **if** $\mathsf{Combine}([\boldsymbol{Q}_i]) \neq \boldsymbol{Q}_i$ **then** abort the protocol
26: convert $(2\theta, n)$ shares $([x_i], [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$ to (θ, n) shares $([\![x_i]\!], [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$

for instance, by letting each party share a random value and taking the sum, or using pseudo-random secret sharing. They then call ProofBlock to produce the block using the shared wires and randomness (line 42). The blocks for the result parties are generated in the same way (lines 43–45). The coefficients of the randomised quotient polynomial $\boldsymbol{H}$ are computed from $\boldsymbol{H'}$ analogously to the zero-knowledge variant of Pinocchio (Section 6.3); note that this requires computing overall randomness δ_v, δ_w, δ_y that is the sum of the randomness from all blocks in the proof (lines 46–48). This gives $(2\theta, n)$ shares $[\langle H \rangle_1]$ of proof element $\langle H \rangle_1$ (lines 49–50)

Having computed shares of all proof elements, the workers now post these shares on the bulletin board so that everybody can combine them to obtain

Algorithm 6.7 Trinocchio: computation phase

27: ▷ Continued from Algorithm 6.6
28: **parties** $\mathcal{C}$ **do**
29: compute $([\![x_{l+1}]\!], \ldots, [\![x_k]\!])$ using MPC
30: $[\![\boldsymbol{v}]\!] \leftarrow \{(\sum_i v_i(\omega_j) \cdot [\![x_i]\!]\}_j$
31: $[\![\boldsymbol{w}]\!] \leftarrow \{(\sum_i w_i(\omega_j) \cdot [\![x_i]\!]\}_j$
32: $[\![\boldsymbol{y}]\!] \leftarrow \{(\sum_i y_i(\omega_j) \cdot [\![x_i]\!]\}_j$
33: $[\![\boldsymbol{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{v}]\!])$
34: $[\![\boldsymbol{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{w}]\!])$
35: $[\![\boldsymbol{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{y}]\!])$
36: $[\![\boldsymbol{v'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{V}]\!])$
37: $[\![\boldsymbol{w'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{W}]\!])$
38: $[\![\boldsymbol{y'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{Y}]\!])$
39: $[\boldsymbol{h'}] \leftarrow \{([\![\boldsymbol{v'}_j]\!] \cdot [\![\boldsymbol{w'}_j]\!] - [\![\boldsymbol{y'}_j]\!])/t(\Omega_j)\}_j$
40: $[\boldsymbol{H'}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h'}])$
41: $([\![\delta_{v,\mathrm{mid}}]\!], [\![\delta_{w,\mathrm{mid}}]\!], [\![\delta_{y,\mathrm{mid}}]\!]) \xleftarrow{\$} \mathbb{F}^3$
42: $[\![\boldsymbol{Q}_{\mathrm{mid}}]\!] \leftarrow \mathsf{ProofBlock}(BK_{\mathrm{mid}}; [\![x_{l+m+1}]\!], \ldots, [\![x_k]\!]; [\![\delta_{v,\mathrm{mid}}]\!], [\![\delta_{w,\mathrm{mid}}]\!], [\![\delta_{y,\mathrm{mid}}]\!])$
43: **for all** $i \in \mathcal{R}$ **do**
44: $([\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!]) \xleftarrow{\$} \mathbb{F}^3$
45: $[\![\boldsymbol{Q}_i]\!] \leftarrow \mathsf{ProofBlock}(BK_i; [\![x_i]\!]; [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$
46: $[\delta_v] \leftarrow [\delta_{v,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{v,i}]$
47: $[\delta_w] \leftarrow [\delta_{w,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{w,i}]$
48: $[\delta_y] \leftarrow [\delta_{y,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{y,i}]$
49: $[\boldsymbol{H}] \leftarrow [\boldsymbol{H'}] + [\![\delta_v]\!][\![\boldsymbol{W}]\!] + [\![\delta_w]\!][\![\boldsymbol{V}]\!] + [\![\delta_v]\!][\![\delta_w]\!]\boldsymbol{T} - [\![\delta_y]\!]$
50: $[\langle H \rangle_1] \leftarrow \sum_{j=0}^{d} \langle s^j \rangle_1 [\boldsymbol{H}_j]$
51: $\mathsf{Post}([\![\boldsymbol{Q}_{\mathrm{mid}}]\!] + [\![0]\!])$
52: $\mathsf{Post}([\langle H \rangle_1] + [0])$
53: **for all** $i \in \mathcal{R}$ **do** $\mathsf{Post}([\![\boldsymbol{Q}_i]\!] + [\![0]\!])$

the full proof. Note that the shares of individual workers might statistically depend on information that we do not want to reveal, such as internal circuit wires. To avoid any problems because of this, the workers first re-randomise their proof elements by adding a new random sharing of zero; for instance, obtained by letting each worker share zero or using pseudo-random zero sharing (lines 51–53).

Algorithm 6.8 Trinocchio: result phase

54: ▷ Continued from Algorithm 6.7
55: **parties** $\mathcal{C}$ **do**
56: **for all** $i \in \mathcal{R}$ **do**
57: send $([\![x_i]\!], [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$ to result party i
58: **parties** $i \in \mathcal{R}$ **do**
59: **for all** $j \in \mathcal{R}$ **do**
60: $\boldsymbol{Q}_j \leftarrow \mathsf{Combine}([\boldsymbol{Q}_j])$
61: $\boldsymbol{Q} \leftarrow \mathsf{Combine}([\![\boldsymbol{Q}_{\mathrm{mid}}]\!]) + \sum_{j \in \mathcal{I} \cup \mathcal{R}} \boldsymbol{Q}_j$
62: $\langle H \rangle_1 \leftarrow \mathsf{Combine}([\langle H \rangle_1])$
63: **if** $\mathsf{CheckBlock}(BV_{\mathrm{mid}}; \boldsymbol{Q}_{\mathrm{mid}}) = \bot \vee$
64: $\exists j : \mathsf{CheckBlock}(BV_j; \boldsymbol{Q}_j) = \bot \vee$
65: $\mathsf{CheckDiv}(VK; \boldsymbol{Q}; \langle H \rangle_1) = \bot$
66: **then**
67: output $\bot$ and abort the protocol
68: $(x_i, \delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \leftarrow \mathsf{Combine}([\![x_i]\!], [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$
69: **if** $\boldsymbol{Q}_i \neq \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ **then**
70: output $\bot$ and abort the protocol
71: output x_i

Result Phase

The result phase of Trinocchio is displayed in Algorithm 6.8. In the result phase, the result parties obtain their computation results, and verify them with respect to the information on the bulletin board. First, the result parties obtain secret shares of their output values, and the randomness used in their proof blocks (line 57). Then, they combine the values from the bulletin board into a full multi-client Pinocchio proof (lines 60–62), and verify this proof (lines 63–67). Finally, they recombine their output values (line 68), check if the secret shares of their output values correspond to the posted proof block (lines 69 and 70), and output the computation result (line 71).

6.7 Security Proof

In this section we prove Theorem 6.11, i.e., we show that our multi-client Trinocchio protocol (Algorithms 6.5 through 6.8) correctly (always) and se-

curely (if at most θ workers are passively corrupted) evaluates function f. Theorem 6.11 directly follows from Lemmas 6.12 and 6.13 below.

Trinocchio Correctly Evaluates f

To prove that Trinocchio correctly evaluates f, we construct a simulator that interacts with the trusted party for correct function evaluation shown in Figure 6.1. The simulator $\mathcal{S}_{\text{correct}}$ is given in Algorithms 6.9 through 6.11. The division of the simulator into several algorithms corresponds to the division of the Trinocchio protocol into Algorithms 6.5 through 6.8 for the key set-up, and input, computation and result phases. We do not explicitly specify the simulator for the computation phase, because this phase only involves secure multi-party computation and posting secret shares of the result on the bulletin board, for which it is known that this can be simulated.

Lemma 6.12. *For every probabilistic polynomial-time adversary $\mathcal{A}$, simulator $\mathcal{S}_{\text{correct}}$ is probabilistic polynomial time and the distribution ensembles* $\text{EXEC}^{(\mathsf{ComGen},\mathsf{MKeyGen})}_{\mathsf{Trinocchio},\mathcal{A}}$ *and* $\text{CIDEAL}_{f,\mathcal{S}_{\text{correct}}}$ *are computationally indistinguishable.*

Proof. To prove this lemma, we will start from the EXEC distribution ensemble and introduce increasingly modified distribution ensembles YAD_i, each indistinguishable from the next, to finally show that $\text{EXEC}_{\mathsf{Trinocchio},\mathcal{A}}$ is computationally indistinguishable from $\text{IDEAL}_{f,\mathcal{S}_{\text{correct}}}$. The simulator operates by simulating the protocol with respect to the given adversary $\mathcal{A}$, and finally returning whatever value the simulated adversary $\mathcal{A}$ returned. The lines in the simulator are labelled to explain which parts of the simulator mimic the real protocol, which are needed to interact with the ideal functionality, and which modifications are introduced and explained by the various YAD distributions.

The real protocol is aborted at several places if certain conditions are met. Note that this is always in response to checks on information on the bulletin board that anybody can perform, hence all protocol parties agree on whether the protocol is aborted. If the simulator follows the protocol and the protocol is aborted, the simulator sends $\perp$ to the ideal functionality on behalf of any corrupt input party whose input had not been sent yet, and proceeds to send $\emptyset$ as set of result parties to get the result, disregarding any messages it receives from the ideal functionality. It also completes the simulation of $\mathcal{A}$ to obtain its output. This ensures that the distribution IDEAL is well-defined for aborted protocols.

Algorithm 6.9 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i\in C}, z, \lambda)$ for correct function evaluation: key set-up

1: **for all** $i \in \mathcal{I}$ **do**
2: **if** $i \in C$ **then**
3: generate perfectly binding commitment key k_i, keep the trapdoor ▷ YAD_1
4: **else**
5: generate perfectly hiding commitment key k_i, keep the trapdoor ▷ YAD_1
6: generate modified $(EK = \{\{BK_i\}_i, \ldots\}, VK = \{\{BV_i\}_i, \ldots\})$ ▷ YAD_3
7: whenever the adversary queries ComGen, return $(k_1, \ldots, k_l)$
8: whenever the adversary queries $\mathsf{MKeyGen}$, return (EK, VK)

At various points, the simulator is instructed to terminate the simulation. This is not the same as aborting the simulated protocol. The simulation will be terminated whenever the simulator fails at some computation which is not part of the real protocol, but which is needed to achieve some security property, such as mimicking the real protocol. To terminate the simulation will mean that the output of the adversary in the ideal case will not be consistent with the output in the real case, i.e., it will signal an adversary that it is in fact operating in the ideal case. To show that the termination of the simulation does not enable the distinction between Exec and Ideal, we will show below that each of the conditions which lead to termination of the simulated protocol can only occur with negligible probability.

We will now describe the purpose of each of the increasingly modified distributions YAD_i and show indistinguishability between consecutive distributions.

YAD_1

The distribution ensemble YAD_1 is the Exec distribution ensemble, where the set-up of the protocol is modified such that the commitment keys for the corrupt input parties are generated to be perfectly binding instead of perfectly hiding, and the simulator keeps the trapdoors. This distribution ensemble is computationally indistinguishable from $\text{Exec}_{\mathsf{Trinocchio},\mathcal{A}}$ based on the property of the mixed commitment scheme that the two kinds of commitment keys are indistinguishable.

Algorithm 6.10 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i \in C}, z, \lambda)$ for correct function evaluation: input phase

9: ▷ Continued from Algorithm 6.9
10: **on behalf of honest parties** $i \in \mathcal{I}$ **do**
11: sample commitment randomness ρ_i' ▷ YAD_2
12: $c_i \leftarrow \mathsf{Commit}_{k_i}(\mathbf{0}; \rho_i')$ ▷ YAD_2
13: $\mathsf{Post}(c_i)$ ▷ Exec
14: **for all** $i \in \mathcal{I} \cap C$ **do**
15: extract $\hat{\boldsymbol{Q}}_i$ from c_i using trapdoor ▷ YAD_2
16: **if** $\mathsf{CheckBlock}(BV_i; \hat{\boldsymbol{Q}}_i) = \perp$ **then**
17: $x_i \leftarrow \perp$ ▷ YAD_2
18: **else**
19: use the d-PKE extractor on $\hat{\boldsymbol{Q}}_i$ to obtain field elements $\delta_{v,i}$, $\delta_{w,i}$ and $\delta_{y,i}$ and polynomials $V_i(x)$, $W_i(x)$ and $Y_i(x)$ of degree at most $d-1$; if this fails, terminate the simulation ▷ YAD_4
20: set x_i such that $V_i(x) = x_i v_i(x)$, $W_i(x) = x_i w_i(x)$ and $Y_i(x) = x_i y_i(x)$; if this is not possible, terminate the simulation ▷ YAD_5
21: send x_i to the ideal functionality on behalf of corrupt input party i ▷ Ideal
22: receive $\boldsymbol{x}$ from the ideal functionality ▷ Ideal
23: **on behalf of honest parties** $i \in \mathcal{I}$ **do**
24: $(\delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \xleftarrow{\$} \mathbb{F}^3$ ▷ Exec
25: $\boldsymbol{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ ▷ Exec
26: create ρ_i such that $c_i = \mathsf{Commit}_{k_i}(\boldsymbol{Q}_i; \rho_i)$ using trapdoor ▷ YAD_2
27: simulate lines 13 through 26 of the real protocol on behalf of honest parties ▷ Exec

YAD_2

For the distribution ensemble YAD_2, the protocol is further modified by producing commitments to 0 instead of the input proof blocks on behalf of the honest input parties. When the commitments are opened later in the protocol, the openings to correct proof blocks are created using the trapdoor information. Additionally, the proof blocks produced by corrupt input parties are extracted from their commitments, although the extracted blocks are not used any fur-

Algorithm 6.11 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i \in C}, z, \lambda)$ for correct function evaluation: result phase

28: ▷ Continued from Algorithm 6.10
29: simulate lines 56 through 68 of the real protocol on behalf of honest parties ▷ EXEC
30: $F \leftarrow \emptyset$ ▷ IDEAL
31: **for all** $i \in \mathcal{R} \setminus C$ **do**
32: **if** $\boldsymbol{Q}_i \neq \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ **then**
33: $F \leftarrow F \cup \{i\}$ ▷ IDEAL
34: **for all** $\boldsymbol{Q}' \in \{\boldsymbol{Q}_{\text{mid}}\} \cup \{\boldsymbol{Q}_i\}_{i \in \mathcal{R} \cap C} \cup \{\boldsymbol{Q}_i\}_{i \in F}$ **do**
35: use the d-PKE extractor on $\boldsymbol{Q}'$ to obtain field elements δ'_v, δ'_w and δ'_y and polynomials $V'(x)$, $W'(x)$ and $Y'(x)$ of degree at most $d-1$; if this fails, terminate the simulation ▷ YAD_4
36: set the corresponding entries in $\boldsymbol{x}$ such that $V'(x) = \sum_i x_i v_i(x)$, $W'(x) = \sum_i x_i w_i(x)$ and $Y'(x) = \sum_i x_i y_i(x)$, where i ranges over the indices corresponding to the block $\boldsymbol{Q}'$ belongs to; if this is not possible, terminate the simulation ▷ YAD_5
37: **if** $t(x) \nmid (\sum_i x_i v_i(x))(\sum_i x_i w_i(x)) - \sum_i x_i y_i(x)$ **then**
38: terminate the simulation ▷ YAD_6
39: Send $\mathcal{R} \setminus F$ to the ideal functionality ▷ IDEAL
40: Return the output of the simulated adversary

ther at this stage.

Indistinguishability between YAD_2 and YAD_1 follows directly from the indistinguishability property of the commitment scheme. The commitment scheme also guarantees that commitments produced by the adversary can only be opened to the extracted proof block, i.e., that $\hat{\boldsymbol{Q}}_i = \boldsymbol{Q}_i$ for corrupt input parties i.

YAD_3

For distribution ensemble YAD_3, we will again modify the set-up of the protocol, but this time of the evaluation and verification keys. This happens analogously to [PHGR13]'s security proof. Instead of sampling s, α_v, α_w, α_y, r_v, r_w, β_{mid} and the β_i for $1 \leq i \leq l+m$ uniformly at random and generating the keys from these values, the set-up proceeds as follows.

For a given QAP of degree d, set $q \leftarrow 4d + 4$, then sample $s \xleftarrow{\$} \mathbb{F}$. Next, set

$$\begin{aligned}\mathsf{chal} \leftarrow \{&\langle 1\rangle_1, \langle s\rangle_1, \langle s^2\rangle_1, \ldots, \langle s^q\rangle_1, \langle s^{q+2}\rangle_1, \ldots, \langle s^{2q}\rangle_1 \\ &\langle 1\rangle_2, \langle s\rangle_2, \langle s^2\rangle_2, \ldots, \langle s^q\rangle_2, \langle s^{q+2}\rangle_2, \ldots, \langle s^{2q}\rangle_2\}.\end{aligned}$$

From this point onwards, the value s will not be used directly to compute the keys. Instead, any key element derived from s will be generated from chal. This restriction will be necessary to complete the security proof later.

Randomly draw α_v, α_w, α_y, r'_v and r'_w. Also draw a random polynomial χ_{mid} of degree at most $3d + 3$ such that $\chi_{\text{mid}}(x) \cdot (r'_v v_i(x) + r'_w x^{d+1} w_i(x) + r'_v r'_w x^{2d+2} y_i(x))$ has a zero coefficient in front of x^{3d+3} for all internal wire indices i, and $\chi_{\text{mid}}(x)t(x)$, $\chi_{\text{mid}}(x)x^{d+1}t(x)$ and $\chi_{\text{mid}}(x)x^{2d+2}t(x)$ have a zero coefficient in front of x^{3d+3} as well. Such polynomials exist by Lemma 10 of [GGPR13]. Similarly, for each input and output wire $1 \leq i \leq l + m$, draw random polynomial $\chi_i(x)$ such that $\chi_i(x)$ is of degree at most $3d + 3$ and $\chi_i(x) \cdot (r'_v v_k(x) + r'_w x^{d+1} w_k(x) + r'_v r'_w x^{2d+2} y_k(x))$, $\chi_i(x)t(x)$, $\chi_i(x)x^{d+1}t(x)$ and $\chi_i(x)x^{2d+2}t(x)$ have a zero coefficient in front of x^{3d+3}

Now, we will generate the evaluation and verification keys as if we had used the following

$$\begin{aligned} r_v &= r'_v s^{d+1} \\ r_w &= r'_w s^{2d+2} \\ r_y &= r'_y s^{3d+3} \\ \beta_{\text{mid}} &= s\chi_{\text{mid}}(s) \\ \beta_i &= s\chi_i(s), \end{aligned}$$

where i ranges from 1 to $l + m$. Because we are not allowed to inspect the value of s directly, we cannot compute these values explicitly. However, we can compute the evaluation and verification key elements from chal. Because r_v, r_w and various β's are still distributed uniformly, and $r_y = r_v \cdot r_w$ still holds, the distribution of the keys is statistically indistinguishable from keys generated using the real key generation algorithm.

YAD_4

Distribution ensemble YAD_4 is produced in the same manner as YAD_3, except that the d-PKE extractor is run on the adversarially generated proof blocks that satisfy the $\mathsf{CheckBlock}$ predicate. If the extractor fails then the simulation

is terminated. Because the d-PKE assumption states that the probability of failure is negligible, YAD_4 will be statistically indistinguishable from YAD_3. Therefore an adversary cannot cause the simulation to fail with better than negligible probability in an attempt to distinguish Exec from Ideal and the use of the d-PKE extractor on lines 19 and 35 is justified.

Remark. The use of knowledge of exponent assumptions is subject to many subtleties, as was identified by Yao and Zhao [YZ10]. After publication of the work presented in this chapter, we have discovered that our multiple use of the extractor may not be completely justified under the d-PKE assumption. One of the problems that arises with knowledge of exponent assumptions that an adversary may be able to obtain pairs of group elements that satisfy the relation verified by the pairing check from other, unrelated protocol instances or other parts of the same protocol instance. Such an adversary would be able to compute proof terms that satisfy some of the pairing relations without necessarily knowing the exponents, thereby violating the extractability assumption. The d-PKE assumption is stated in a stand-alone fashion and avoids this problem. We use the extractor multiple times however, which is in violation of the definition of the d-PKE assumption. This is not to say that this constitutes an effective attack on our Trinocchio protocol. However, the security proof is incomplete in this regards and would have to be carefully reworked at this point to claim security under the stated assumptions. For the purpose of this thesis, we leave this as an open issue.

YAD_5

In addition to extracting the contents of all proof blocks, to produce distribution ensemble YAD_5 we will also attempt to retrieve the $\boldsymbol{x}$ values that constitute the extracted $V(x)$, $W(x)$ and $Y(x)$ polynomials. If no $\boldsymbol{x}$ exists such that $V(x) = \sum_i x_i v_i(x)$, $W(x) = \sum_i x_i w_i(x)$ and $Y(x) = \sum_i x_i y_i(x)$, then the simulation is terminated. We will show that an adversary that successfully causes this failure, i.e., with higher than negligible probability, can break the q-PDH assumption, as in the security proof of [PHGR13].

Suppose an adversary manages to produce a proof block $\boldsymbol{Q}$, corresponding to block verification key BK for which $\mathsf{CheckBlock}(VK; \boldsymbol{Q})$ holds and $V(x)$, $W(x)$ and $Y(x)$, as well as δ_v, δ_w and δ_y are successfully extracted, but no $\boldsymbol{x}$ exists satisfying $V(x) = \sum_i x_i v_i(x)$, $W(x) = \sum_i x_i w_i(x)$ and $Y(x) = \sum_i x_i y_i(x)$. Let $\langle Z \rangle_1$ be the final element of $\boldsymbol{Q}$. Then we can write $\langle Z \rangle_1$ as a polynomial

$\sum_i \xi_i x^i$ evaluated at s "in the exponent":

$$\begin{aligned}\langle Z\rangle_1 - &\langle r_v\beta t\rangle_1\delta_v + \langle r_w\beta t\rangle_1\delta_w + \langle r_y\beta t\rangle_1\delta_y \\ &= \sum_j \langle r_v\beta v_j + r_w\beta w_j + r_y\beta y_j\rangle_1 x_j \\ &= \langle s\chi(s)\cdot(r'_v s^{d+1}V(s) + r'_w s^{2d+2}W(s) + r'_v r'_w s^{3d+3}Y(s))\rangle_1 \\ &= \langle \sum_i \xi_i s^i\rangle_1.\end{aligned}$$

By Lemma 10 of [GGPR13], the coefficient ξ_{q+1} for x^{q+1} is non-zero with high probability. We can then compute

$$\langle s^{q+1}\rangle_1 = \xi_{q+1}^{-1}\cdot(\langle Z\rangle_1 - \langle r_v\beta t\rangle_1\delta_v + \langle r_w\beta t\rangle_1\delta_w + \langle r_y\beta t\rangle_1\delta_y - \sum_{i\neq q+1}\xi_i\langle s^i\rangle_1)$$

using only information in the evaluation key.

Recall from YAD_3 that the very first step in generating this distribution ensemble is to create a q-PDH challenge for some secret value s and in the rest of the process any information derived from s is computed based on this challenge. If instead of generating the challenge ourselves, we consider it a given, then the algorithm for generating YAD_5 together with an adversary that successfully causes failure can as a whole be viewed as an algorithm that breaks the q-PDH assumption.

This justifies the extraction of all wire values from proof blocks on lines 20 and 36 of $\mathcal{S}_{\text{correct}}$.

YAD_6

Distribution ensemble YAD_6 is generated as YAD_5, except that if the divisibility check $\mathsf{CheckDiv}$ succeeds, we use the wire values obtained in the normal course of the protocol together with the wire values extracted in YAD_5 to test whether $t(x)$ truly divides $p(x) = (\sum_{i=0}^k x_i v_i(x))(\sum_{i=0}^k x_i w_i(x)) - \sum_{i=0}^k x_i y_i(x)$. If this is not the case then the simulation is terminated. We will show that the probability of an adversary forcing this failure is negligible, as an algorithm that successfully manages to cause such a failure can be used to break the $2q$-SDH assumption, closely following the security proof of [PHGR13].

Let $V(x) = \sum_{i=0}^k x_i v_i(x)$, and, analogously, $W(x) = \sum_{i=0}^k x_i w_i(x)$, and $Y(x) = \sum_{i=0}^k x_i y_i(x)$. Suppose that $t(x)$ does not divide $p(x) = V(x)W(x) - Y(x)$. Let r be a root of $t(x)$ but not of $p(x)$ and let $T(x) = t(x)/(x-r)$.

Let $d(x) = \gcd(t(x), p(x))$ and $a(x)$ and $b(x)$ be polynomials of degree at most $2d-1$ and $d-1$ respectively such that $a(x)t(x) + b(x)p(x) = d(x)$. Set $A(x) = a(x)T(x)/d(x)$ and $B(x) = b(x)T(x)/d(x)$. These polynomials have no denominator since $d(x)$ divides $T(x)$. Then $A(x)t(x) + B(x)p(x) = T(x)$. Dividing by $t(x)$, we have $A(x) + B(x)p(x)/t(x) = 1/(x-r)$. Note that $\langle H \rangle_1 = \langle p/t \rangle_1$. We can now evaluate $\langle A \rangle_1$ and $\langle B \rangle_2$ using terms in the evaluation key. From these we can solve $e(\langle A \rangle_1, \langle 1 \rangle_2) e(\langle H \rangle_1, \langle B \rangle_2) = e(\langle 1 \rangle_1, \langle 1 \rangle_2)^{1/(s-r)}$.

Note that the q-PDH challenge can be considered an incomplete $2q$-SDH challenge. If, as with YAD_5, we again do not generate the challenge ourselves, but consider it a given, the algorithm for generating YAD_6, along with an adversary that successfully causes failure can be viewed as an algorithm which break the $2q$-SDH assumption.

Ideal

The distribution ensembles YAD_1 through YAD_6 are indistinguishable from each other and from Exec. Through the distribution ensembles YAD_1 to YAD_6, we have argued that the distribution of the adversary's interactions with real protocol parties are indistinguishable from its simulation by YAD_i. At the same time, the outputs of the honest result parties in each YAD_i are still according to the protocol. Comparing YAD_6 to $\text{Ideal}_{f,\mathcal{S}_{\text{correct}}}$, we see that the adversary's output is unchanged, but now honest result parties get the value computed by the trusted party instead of the value from the simulated protocol. However, note that if the simulation in YAD_6 is not terminated, then the vector $\boldsymbol{x}$ is in fact a solution to the QAP corresponding to inputs supplied to the trusted party. Hence, because the QAP computes f, the values from $\boldsymbol{x}$ that are output as computation results in YAD_6 are in fact the output of f on the inputs supplied to the trusted party. Therefore, the outputs of the honest result parties in YAD_6 and Ideal are the same.

From Exec to Ideal

Overall, the sequence of distribution ensembles shows that the real- and ideal-world executions of the protocol are computationally indistinguishable, hence the lemma follows. □

Algorithm 6.12 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i\in C}, z, \lambda)$ for secure function evaluation

1: Generate real commitment keys $k_1, \ldots, k_n$ as in the protocol; when $\mathcal{A}$ makes a hybrid call to ComGen, return $k_1, \ldots, k_n$
2: Generate evaluation key EK and verification key VK, keep trapdoor s; when $\mathcal{A}$ makes a hybrid call to $\mathsf{MKeyGen}$, return (EK, VK)
3: **for all** $i \in \mathcal{I} \setminus C$ **do**
4: $\quad x_i \leftarrow 0$
5: Simulate lines 7 to 57 of the real protocol on behalf of honest input parties and workers. If the protocol aborts, send $\perp$ to the ideal functionality on
behalf of corrupt input parties and abort the simulated protocol
6: **for all** $i \in \mathcal{I} \cap C$ **do**
7: $\quad x_i \leftarrow \mathsf{Combine}([\![x_i]\!])$
8: $\quad$ Send x_i to the ideal functionality on behalf of corrupt input party i
9: **for all** $i \in \mathcal{R} \cap C$ **do**
10: $\quad$ Receive result $\hat{x}_i$ from the ideal functionality
11: $\quad \delta_{v,i} \leftarrow \mathsf{Combine}([\![\delta_{v,i}]\!])$
12: $\quad \delta_{w,i} \leftarrow \mathsf{Combine}([\![\delta_{w,i}]\!])$
13: $\quad \delta_{y,i} \leftarrow \mathsf{Combine}([\![\delta_{y,i}]\!])$
14: $\quad \hat{\delta}_{v,i} \leftarrow \delta_{v,i} + (x_i - \hat{x}_i)\frac{v_i(s)}{t(s)}$
15: $\quad \hat{\delta}_{w,i} \leftarrow \delta_{w,i} + (x_i - \hat{x}_i)\frac{w_i(s)}{t(s)}$
16: $\quad \hat{\delta}_{y,i} \leftarrow \delta_{y,i} + (x_i - \hat{x}_i)\frac{y_i(s)}{t(s)}$
17: $\quad$ Create shares $([\![\hat{x}_i]\!], [\![\hat{\delta}_{v,i}]\!], [\![\hat{\delta}_{w,i}]\!], [\![\hat{\delta}_{y,i}]\!])$ such that they are consistent with the shares of $([\![x_i]\!], [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$ held by corrupt computation parties
18: $\quad$ Send $([\![\hat{x}_i]\!], [\![\hat{\delta}_{v,i}]\!], [\![\hat{\delta}_{w,i}]\!], [\![\hat{\delta}_{y,i}]\!])$ to result party i
19: Return the output of the simulated adversary

Private Case

The simulator $\mathcal{S}_{\text{private}}$ for private function evaluation is given in Algorithm 6.12. We show that it works in situations when privacy is guaranteed:

Lemma 6.13. *For every probabilistic polynomial-time adversary $\mathcal{A}$ such that at most θ workers are passively corrupted, $\mathcal{S}_{\text{correct}}$ is probabilistic polynomial time and the distribution ensembles* $\text{EXEC}^{(\mathsf{ComGen},\mathsf{MKeyGen})}_{\mathsf{Trinocchio},\mathcal{A}}$ *and* $\text{IDEAL}_{f,\mathcal{S}_{\text{private}}}$

are computationally indistinguishable.

Proof. The simulator mostly runs the actual protocol, using zero inputs on behalf of honest parties. However, it needs to provide the inputs of the corrupted input parties to the trusted party, and make sure that corrupted result parties obtain the result from the trusted party. For the corrupted inputs, note that the simulator simulates at least $\theta + 1$ honest workers, hence it knows enough shares of the inputs of corrupted input parties to determine them and send them to the trusted party (lines 6–8). In order to manipulate the corrupted results, the simulator simulates normal Trinocchio key generation with respect to the adversary, but keeps trapdoor s (line 2). It can then use s to make sure that the proof block that was generated for the adversary during the protocol run indeed opens to the output value for the result party that the simulator gets from the trusted party (lines 11–18).

To see that the Exec and Ideal distributions are the same, first note that because the workers are all semi-honest, the outputs of the result parties in Exec are always correct, and hence the same as in Ideal. Hence, we only have to worry about the observations made by the adversary.

Now, note that the simulator at no point uses, or even has access to, the honest input parties' private values. Since the simulator follows the real protocol specification up to line 57, the adversary cannot detect any deviations from the real protocol, other than might be caused by the fact that the input values for the honest parties do not match the distribution of real input values. However, the privacy properties of the underlying secure multiparty computation protocol imply that no data exchanged during the computation protocol reveals any information about the input or intermediate wire values. Moreover, the commitment scheme is used as in the protocol, so does not give the adversary any chance of distinguishing the real and ideal world.

The only other information that the adversary learns is what is opened during the multiparty computation protocol, i.e., the shares of the proof blocks ($\boldsymbol{Q}$) and divisibility check term ($\langle H \rangle_1$). First, note that these shares reveal nothing more than the proof blocks and divisibility check term themselves, as these shares are freshly randomised using a zero sharing before they are revealed.

Now consider what the adversary learns from the proof blocks and divisibility check term. As observed in [GGPR13], the first, third and fifth elements of a proof block, $\langle V \rangle_1$, $\langle W \rangle_2$, and $\langle Y \rangle_1$, are uniformly distributed if the δ_v, δ_w and δ_y used to compute those are uniformly distributed as well. This holds regardless of which value $\boldsymbol{x}$ is used. Furthermore, once these three elements are

known, the remaining four elements are fixed due to the verification relations. Because all of the proof blocks generated in the protocol are produced using randomly chosen values for δ_v, δ_w and δ_y, it holds that all proof blocks in the protocol are distributed uniformly randomly and do not reveal any information about the values they are composed from.

We conclude that the adversary sees no information that allows it to distinguish the real and ideal worlds, hence the lemma follows. □

6.8 Discussion and Conclusion

In this chapter, we have presented Trinocchio, a system that adds privacy to the Pinocchio verifiable computation scheme essentially at the cost of replicating the Pinocchio proof production algorithm at three (or more) servers. Trinocchio has the same correctness and security guarantees as Pinocchio; distributing the computation between $2\theta + 1$ workers gives privacy if at most θ of them are corrupted. We have shown in a case study that the overhead is indeed small.

As far as we are aware, our work is the first to deliver efficient verifiable computation (i.e., with cryptographic guarantees of correctness and practical verification times independent of the computation size) with privacy guarantees. Although privacy is only guaranteed if not too many of the workers are corrupt, the use of verifiable computation ensures that the outcome of the protocol cannot be manipulated by the workers. This allows us to hedge against an adversary being more powerful than anticipated in a real world scenario.

As discussed, existing verifiable computation constructions in the single-worker setting [GGP10, GKP+13, FGP14] use very expensive cryptography, while multiple-worker efforts to provide privacy [ACG+14] do not guarantee correctness if all workers are corrupted. In contrast, existing works from the area of multiparty computation [BDO14, SV15, dHSV16] deliver privacy and correctness guarantees, but have much less efficient verification.

A major limitation of Pinocchio-based approaches is that they assume trusted set-up of the (function-dependent) evaluation and verification keys. In the single-client setting, the client could perform this set-up itself, but in the multiple-client setting, it is less clear who should do this. In particular, whoever has generated the evaluation and verification keys can use the values used during key generation as a trapdoor to generate proofs of false statements. Even though key generation can likely be distributed using the same techniques we use to distribute proof production, it remains the case that all generating

parties together know this trapdoor. Unfortunately, this seems inherent to the Pinocchio approach.

Our work is a first step towards privacy-preserving verifiable computation, and we see many promising directions for future work. Recent work in verifiable computation has extended the Pinocchio approach by making it easier to specify computations [BCG+13], and by adding access control functionality [AJCC15]. In future work, it would be interesting to see how these kind of techniques can be used in the Trinocchio setting. Also, recent work has focused on applying verifiable computation on large amounts of data held by the server (and possibly signed by a third party) [CTV15]; assessing the impact of distributing the computation (in particular when aggregating information from databases from several parties) in this scenario is also an important future direction. It would also be interesting to base Trinocchio on the (much faster) Pinocchio codebase [PHGR13] and more efficient multiparty computation implementations, and see what kind of performance improvements can be achieved. Another interesting direction is to investigate the possibility of practical universally composable [Can01, CCL15] distributed verifiable computation; or to use the universal composability framework to obtain a more generic framework for combining multiparty computation with verifiable computation (even with only standalone guarantees).

Chapter 7

Asymptotically Optimal Hash Chain Reversal

7.1 Introduction

One-way hash chains were introduced by Lamport to construct an authentication scheme resistant to eavesdropping attacks [Lam81]. The idea of Lamport's scheme is elegant in its simplicity. The prover generates a hash chain by iterating the evaluation of a hash function on a random seed value. The last value of the chain is registered with the verifier. Any time the prover wishes to authenticate itself, it computes the preimage of the value registered with the verifier, which is possible for the prover as only it knows the seed value and deemed impossible for any other party due to the one-wayness of the hash function. The verifier then verifies that the given preimage indeed hashes to the registered value and updates the registered value to its preimage. In this way, the prover can authenticate itself to the verifier as many times as the length of the original hash chain. In effect, the prover computes the hash chain in reverse order.

The efficiency of Lamport's authentication scheme is determined by the *budget*, which is the number of hash function evaluations required between successive outputs. For long chains, it is not considered practical to compute every element of the reversed chain starting from the seed value as the budget would be equal to the length of the chain. Storing all elements when the chain is initially computed in the forward direction is not considered practical either for long chains, as the required storage would be equal to the size of the chain.

An intermediate solution in which some, but not all positions are stored and the missing positions are recomputed would allow for a trade-off between the budget and storage complexities, but the product of these two would still grow at the same impractical rate.

For the optimal evaluation strategy in both storage and computation complexity, some positions must be initially stored from which the missing positions can be recomputed. However, when one of these stored positions is output, its memory should be reused to compute and store another intermediate position on the chain which, once computed, can be used as the basis for computing many more outputs or even further intermediate positions. Such algorithms, which store some elements on the chain and update these as the evaluation proceeds are known as pebbling algorithms. Jakobsson first presented pebbling algorithms for hash chain reversal with both budget and storage complexity logarithmic in the length of the chain [Jak02].

The problem of reverse automatic differentiation can be seen as a relaxation of hash chain reversal, because it takes the total time, rather than the budget as the time complexity measure. As such the results of Griewank [Gri92] and Grimm et al. [GPRS96] can be considered a natural lower bound on the complexity of hash chain reversal. A consequence of their results is that the space-time complexity of hash chain reversal is bounded from below by $1/4(\log_2 n)^2$ for chains of length n. This fact was also independently discovered in the context of hash chain reversal by Coppersmith and Jakobsson [CJ02].

The space-time complexity of binary partitioning strategies to hash chain reversal, such as those of [Jak02, CJ02, YSEL09, Sch16] is at least $1/2(\log_2 n)^2$. The best of these approaches are therefore within a factor 2 from the optimum. In this chapter we investigate the question whether there exist pebbling algorithms which achieve this lower bound. Using our novel framework, we give a construction that asymptotically approaches the known optimal space-time complexity of $1/4(\log_2 n)^2$ for hash chain reversal, thus answering the question, at least asymptotically, in the affirmative.

In addition to our asymptotic results, we will also show two particularly simple strategies that achieve complexity lower than $1/2(\log_2 n)^2$ of binary partitioning strategies at chain lengths reasonable for real applications. The first is a strategy that uses a partitioning based on Fibonacci numbers, first mentioned by Schoenmakers in [Sch16] as a potentially viable strategy. Using our framework we confirm that, for sufficiently long chains, this strategy indeed features lower space-time complexity than a binary partitioning strategy. Concretely, the space-time complexity converges to about $0.46(\log_2 n)^2$ as the length of the chain increases. We determine numerically that for the 42nd Fi-

bonacci number, at chain length of about 2^{28}, the space-time complexity is lower than that of the binary partitioning strategy at about $0.48(\log_2 n)^2$.

Our second concretely more efficient strategy is a generalization of the binary partitioning strategy which uses b-ary partitioning for any arbitrary integer $b \geq 2$. The space-time complexity converges to about $0.40(\log_2 n)^2$ for $b = 3$ and is already lower than that of binary partitioning at chains of length about 2^{19}. The optimal b-ary partitioning is found at $b = 5$, for which the space-time complexity converges to about $0.37(\log_2 n)^2$, however it is not until chain lengths of about 2^{48} that this strategy outperforms binary partitioning. Of note is also the case of $b = 4$, which uses half as much storage as a binary partitioning strategy, but achieves the same space-time complexity at chain length 2^{28}, which improves for longer chains, converging to $0.375(\log_2 n)^2$.

Related Work

As mentioned above, the problem optimizing only the total amount of computation was solved by Griewank [Gri92] and Grimm et al. [GPRS96], albeit not for hash chain reversal, but for the purpose of reverse automatic differentiation.

Inspired by techniques from Itkis and Reyzin [IR01] for efficient key updates in a forward-secure digital signature scheme, Jakobsson [Jak02] first introduced an efficient pebbling algorithm featuring both $\lceil \log_2 n \rceil$ storage and computation complexity for the reversal of hash chains of length n, i.e., this algorithm has space-time complexity $(\log_2 n)^2 = k^2$ for hash chains of length $n = 2^k$, with k integer. This algorithm favored simplicity over efficiency.

The work by Jakobsson was followed by Coppersmith and Jakobsson [CJ02], giving an algorithm which sacrifices simplicity, but reduces the computation complexity by a factor of 2, while only adding storage overhead of $\lceil \log_2(\log_2 n + 1) \rceil$ compared to [Jak02]. The space-time complexity of this algorithm is $1/2k^2 + O(k \log k)$. Furthermore, the authors independently give a lower bound on the complexities, coinciding with the results of [Gri92, GPRS96], and show that their algorithm is nearly within a factor 2 from the optimum of $1/4k^2$.

The results of Coppersmith and Jakobsson [CJ02] were refined by Yum et al. [YSEL09], demonstrating that the overhead in storage complexity of Coppersmith and Jakobsson's algorithm can be eliminated by using a simple greedy algorithm, giving an algorithm that has space-time complexity of exactly $1/2k^2$.

All pebbling algorithms described so far are so-called binary pebbling algorithms which place the intermediate positions on the chain at powers of 2. Schoenmakers [Sch16, Sch17] gives a framework for describing the above-

mentioned binary pebbling algorithms explicitly, allowing for exact analysis and for the algorithms to be specified using minimal storage overhead, consisting of the hash values stored and a single counter. Furthermore, the explicit description of the minimal binary pebbling algorithm is the first in-place algorithm of its kind. Schoenmakers also is the first to mention that using Fibonacci numbers to determine the placement of intermediate positions, rather than powers of 2, would lead to the lower space-time complexity of $0.46(\log_2 n)^2$.

Sella [Sel03] departs from a binary partitioning scheme for the placement of intermediate positions. Instead of focusing on the space-time complexity, Sella describes an algorithm for any given bound on the budget. The storage complexity of this algorithm is $(m+1)b$, where m is the given bound on the budget and $b = n^{\frac{1}{m+1}}$, for chains of length n. By fixing b and solving for m this results in a b-ary partitioning scheme, though different from our preferred manner of b-ary partitioning. Sella bounds the space-time complexity of this strategy from above by $b(\log_b n)^2 = b(\log_2 n)^2/(\log_2 b)^2$. For the optimal choice of $b = 7$ this is about $0.89(\log_2 n)^2$. This work was later improved by Kim [Kim03], who shows that the storage complexity can be reduced to $(m + 1)(b - 1)$, which, for optimal $b = 5$ gives a space-time complexity of about $0.74(\log_2 n)^2$. Using our framework, we show that the space-time complexity actually converges to $(b-1)^2(\log_2 n)^2/(b(\log_2 b)^2)$, which is minimal for $b = 2$, i.e., the binary partitioning scheme, but allows for a trade-off between space and time complexity. Furthermore, Sella describes a specific algorithm for the case that only a single hash function is permitted each round, i.e., $m = 1$ and proves it optimal for that case. Despite the simplicity and optimality of this solution, it is not suitable for longer chains as the storage complexity grows with the square root of the chain length.

Unlike the above-mentioned works we do not concretely specify an operational algorithm for the evaluation of the hash chain and specify our storage complexity purely in terms of the number of hash values that need to be stored, ignoring any additional state. Therefore, our results cannot directly be compared to practical implementations. We do, however, show how our framework can be used to easily analyze the complexity of existing solutions in Section 7.11.

Roadmap

We will give a brief roadmap of how the various sections of this chapter relate to each other and to the main result.

Our main result regards the space-time complexity. We formally define our notion of space-time complexity in Section 7.5 based on the budget and storage complexities as explained above. Alongside, we will also define the *averaged* space-time complexity notion, which is based on the total amount of computation, rather than the budget and can be seen as a lower bound on the *true* space-time complexity. We present our main result at the end of Section 7.10, stating that we can compose new pebbling algorithms for hash chains of increasing length out of a given *base* pebbling algorithm, such that the space-time complexity of the *composite* algorithms converges (with increasing length of the hash chain) to the *averaged* space-time complexity of the *base* algorithm. As the optimum strategy for the averaged space-time complexity is known, this resolves the question of optimal hash chain reversal in the asymptotic sense.

To arrive at our main result, we first give a completely general introduction of how two pebbling algorithms can be composed into a single pebbling algorithm for reversing a chain of longer length by essentially concatenating the two algorithms in Section 7.3. In Section 7.4 we formally define a restriction of this general composition idea that forms the foundation of our framework. We then formally define our complexity measures in Section 7.5. As our composition of pebbling algorithms essentially concatenates two algorithms, the length of the composite hash chain is the sum of the lengths of the component chains. In Section 7.9 we define a higher order composition, in which the length of the composite chain is equal to the *product* of the lengths of the component chains. We show that it is beneficial to apply this higher order composition repeatedly to the same pebbling algorithm, obtaining the analogue of exponentiation in Section 7.10.

On the way to the main result we have skipped three sections. After formally stating the foundation of our framework and the complexity measure, we state several useful facts in Section 7.6. These useful facts may also serve as examples of how our framework can be applied. In Sections 7.7 and 7.8 we also describe two particular pebbling strategies in terms of our framework, where the partitioning is based on binomial coefficients and Fibonacci numbers, respectively. Both of these serve as more elaborate examples of the application of our framework. These two sections have additional merit. The approach based on binomial coefficient is known to have the minimal *averaged* space time complexity [Gri92, GPRS96]. When we apply our main result to such pebblers we therefore obtain asymptotically optimal pebbling algorithms for hash chain reversal. The approach based on Fibonacci numbers is of historical interest, as it was a first known construction to achieve space-time complexity better than binary partitioning schemes.

7.2 Preliminaries

Informally, we define a *pebbler* as a process that evaluates a hash chain in reverse order. The *evaluation* of a pebbler proceeds in rounds. A round consists of the pebbler being activated in a certain persistent state and the pebbler performing some hash function evaluations modifying the persistent state. At some point during the round the pebbler must output the next element of the reversed hash chain. Pebblers also makes use of some ephemeral state during a round, which is lost at the end of every round.

The persistent state of a pebbler consists of a number of hash values, or *storage units*, and some auxiliary information such as the round counter. For technical reasons we consider the *initial state* of a pebbler as the persistent state stored *after* the first output is generated. In the following, when we refer to the *state* of a pebbler, we mean the persistent state.

The number of storage units in use by a pebbler varies per round. We call the maximum number of storage units in use by a pebbler, taken over all rounds, the *storage size* of the pebbler. We refer to the length of the hash chain evaluated by the pebbler as the *length* of the pebbler. We consider the seed value of the hash chain as *input* to the pebbler.

The number of hash function evaluations required to compute the next output and state of a pebbler also varies per round. We call the maximum number of hash function evaluations required in a single round the *budget* and the total number of hash function evaluations to completely initialize and evaluate a pebbler its *work size*. We are interested in finding pebblers that have low storage size and low budget, which we also call the *resource utilization*; finding pebblers that have low work size is a secondary interest.

We will only consider pebblers that are deterministic and independent of the hash function used. Furthermore, we impose the constraint that no hash image will be computed that is already stored. Therefore, a pebbler can be represented as a sequence of sets of natural numbers that indicate which iterated images of the input under the hash function are stored. We call these numbers the *positions* on the hash chain. Here we assume that no preimage of the input is known, i.e., that we do not store any position below zero. We will further restrict ourselves to pebblers that only store positions below the output position. Of course, to completely reverse a hash chain, the zero position has to be stored at all times until it is output as the very last step of the process. Finally, the output position must be computed in every round, but we do not consider this position stored at the end of that round.

A graphical representation of a pebbler is displayed in Figure 7.1.

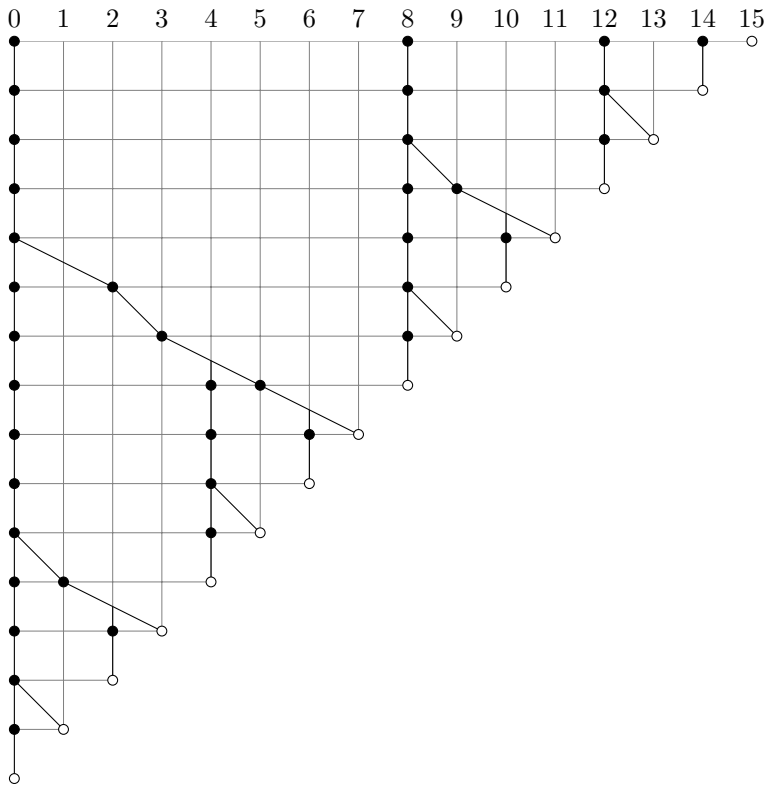


Figure 7.1: Example of a pebbler of length 16, storage size 4 and budget 2. Each row represents one evaluation round. Evaluation proceeds from top to bottom, with the open circle indicating the output position of that round. The solid dots represent the positions stored at the end of the round and the lines connecting them are the hash function evaluations that compute new positions.

Notation

In this chapter we will make extensive use of sequences. We will denote literal sequences using curly braces, where the elements are to be read from left to right, i.e., the sequence $\{1, 2, 3\}$ consists of three elements, the first of which is 1, followed by 2 and finally 3. As our notation overlaps with set notation, we denote the empty sequence by the symbol $\emptyset$. Occasionally we will use superscripts to denote constant sequences of a given length: $\{1\}^n$ denotes the all-ones sequence of length n.

As a convention for clarity, we will use capitals to refer to pebblers and sequences and lower case letters for all other symbols.

We define the arithmetic operations on scalars and sequences as follows. Let a be a scalar and $R = \{r_1, r_2, \ldots, r_m\}$ and $S = \{s_1, s_2, \ldots, r_n\}$ be sequences of length m and n, respectively. Then

$$a + R = \{a + r_1, a + r_2, \ldots, a + r_m\}$$

and

$$R + S = \{r_1 + s_1, r_2 + s_2, \ldots, r_{\min(m,n)} + s_{\min(m,n)}\},$$

i.e., addition of a sequence and a scalar adds the scalar to each element of the sequence and addition of two sequences is pairwise addition of the elements of the sequences up to the length of the shorter of the two sequences. The other arithmetic operations of subtraction, multiplication and division are defined analogously.

Finally, we define two operations on sequences only. Concatenation,

$$R \| S = \{r_1, r_2, \ldots, r_m, s_1, s_2, \ldots, s_n\},$$

extends the left-hand sequence by the right-hand one. The second operation, which we name 'prefer' and denote with the symbol $\not\swarrow$ maps two sequences to a sequence of the same length as the longest of its operands. In each position of the resultant sequence the elements are taken from the left-hand operand at the same position, unless the left-hand operand is too short, in which case the values are taken from the right-hand operand at that position. More formally, let V and W be (possibly empty) sequences such that $S = V \| W$ and the length of V is equal to $\min(m, n)$, then we define

$$R \not\swarrow S = R \| W.$$

7.3 Sequential Composition

A very powerful notion for describing pebblers is sequential composition. We will only define the general notion informally and will define a restriction of the sequential composition of pebblers, called patient pebblers, formally in the next section. We will first define the elementary pebbler.

Definition 7.1. We define the *elementary pebbler*, denoted E, as the pebbler that evaluates the hash chain of length one in reverse by simply outputting its input when evaluated. The elementary pebbler does not store any positions on the hash chain and does not perform any hash evaluations, i.e., both its storage size and its budget are zero.

Let X and Y be pebblers of length n_X and n_Y, respectively. We can construct a pebbler Z, informally denoted $X \oplus Y$, of length $n_Z = n_X + n_Y$ by composing X and Y as follows. Let x be the input of Z. We compute $y = f^{n_X}(x)$ and compute the initial state of Y on input y. We store x together with the initial state of Y on input y as the initial state of Z on input x. Then, to evaluate Z, we first perform the evaluation of Y, initialized on input y. In parallel to the evaluation of Y, we compute the initial state of X on input x. The initialization of X is not performed all at once, but is divided in the same number of rounds as the evaluation of Y. After Y has been evaluated, we perform the evaluation of X, which is now fully initialized on input x. We call this composition of two pebblers *sequential composition*.

Note that sequential composition is not symmetric in its components. In the sequential composition X and Y as described in the preceding paragraph we call Y, the pebbler that is evaluated first, the *right sub-pebbler* and X the *left sub-pebbler*. This corresponds both with the notation $X \oplus Y$ and with the graphical representation of pebblers, in which the left sub-pebbler is displayed to the left of the right sub-pebbler. Note also that we start the evaluation of the composite pebbler in a state in which its right sub-pebbler is fully initialized and we consider only the initialization of the left sub-pebbler as part of the evaluation. Of course, for the composite pebbler $Z \oplus W = (X \oplus Y) \oplus W$, where W is any pebbler, the complete initialization of Z, i.e., the initialization of X *and* Y, must be performed in parallel with the evaluation of W.

We recursively define *composite pebblers* as either the elementary pebbler, or the sequential composition of two composite pebblers. Not every pebbler is a composite pebbler. However, the notion of composite pebblers is general in the sense that for every pebbler, there exists a composite pebbler with storage size and budget not greater than the original pebbler.

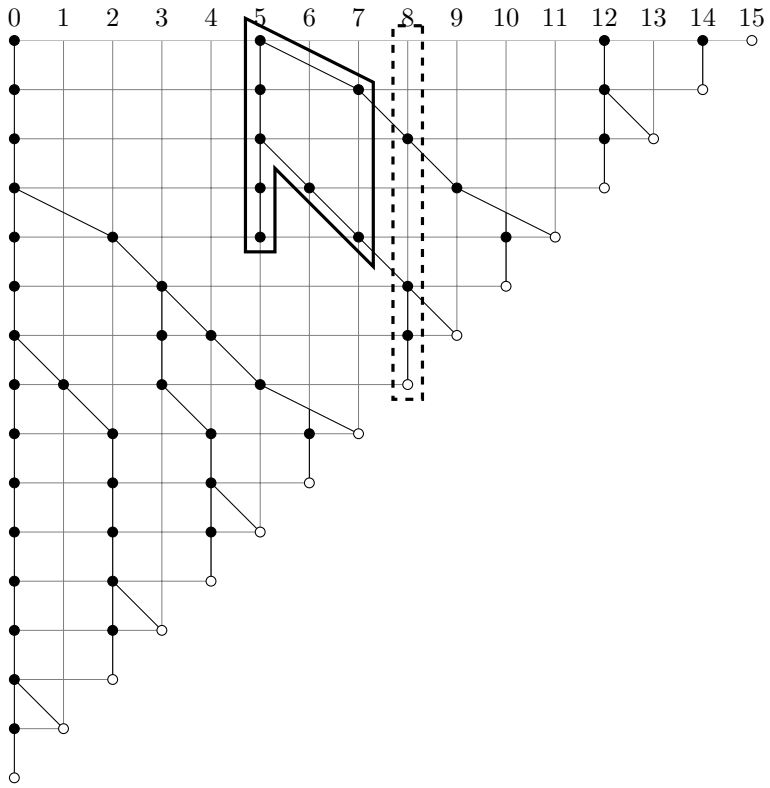


Figure 7.2: Example of a pebbler that is not a composite pebbler. Removing the positions in the area marked with the solid border and storing position 8 from the initial round until it is output, in the area marked with the dashed border, would transform this pebbler into a composite pebbler. Note that there are two kinds of redundancy that make this pebbler not a composite pebbler: position 8 computed twice, whereas once would suffice, and position 5 is eventually discarded without being used for output or further computation.

We illustrate this by example. Consider the pebbler displayed in Figure 7.2. We see that output positions 0 through 7 are derived from position 0 in the initial state and only output positions from 8 onwards are derived from other positions in the initial state. Output position 8 is derived from position 5 in the

initial state and by simply removing all stored positions smaller than 8 that are derived from initial position 5, marked in the figure with a solid border, and replacing these by simply storing position 8 from the initial state onwards until it is output, as marked with the dashed border, would transform this pebbler into a composite pebbler. Doing so would eliminate the hash function evaluations required to compute position 8 and not introduce any new ones. Furthermore, in any round before position 8 is output and in which position 8 is not stored in the original pebbler there is at least one stored position strictly between 0 and 8 that would be eliminated. Eliminating these would free up at least one storage unit, while storing position 8 in these rounds would require exactly one storage unit. Therefore both the storage size and the budget are no greater in the modified pebbler than they are in the original pebbler.

It is clear that the modified pebbler can be decomposed into two sub-pebblers. The right sub-pebbler consists of the first 8 rounds restricted to positions from 8 onwards. The left sub-pebbler consists of the last 8 rounds. We should apply the above procedure recursively to the sub-pebblers as well to obtain a composite pebbler. In the example, the sub-pebblers are already composite. Note that the inefficient computation of position 4 from position 3 twice does not invalidate the fact that the left sub-pebbler is composite, since this is part of the initialization of the left sub-pebbler and our definition of composite pebblers does not consider initialization.

7.4 Patient Pebblers

For composite pebblers, we can conclude that *every* hash function evaluation performed during the evaluation of the pebbler is actually part of the initialization of a sub-pebbler. To fully describe such a composite pebbler, then, is to fully describe its sub-pebblers and to provide a schedule for the initialization of its left sub-pebbler. Interestingly, we can derive bounds on the resource utilization of a pebbler in terms of only its sub-pebbler structure.

The sub-pebbler structure of a composite pebbler imposes a lower bound on the storage size. We will argue the existence of a schedule which achieves this lower bound exactly and give an upper bound on the budget. In the following we will therefore not explicitly specify any initialization schedule and, consequently, not specify any hash function evaluations explicitly.

In this section we will give formal definitions for a restriction of the sequential composition of pebblers, which we call *patient pebblers*. We recursively define patient pebblers as either the elementary pebbler, or pebblers obtained

through patient composition of patient pebblers of smaller length.

Unlike sequential composition in general, not all pebblers can be reduced to patient pebblers. The advantage of patient pebblers is that we can derive the resource utilization of a pebbler in a straightforward manner from the sub-pebbler structure, without needing to specify concrete initialization schedules. Furthermore, our final result, showing the existence of pebblers which asymptotically approach the lower bound on resource utilization, is obtained using patient pebblers.

As the evaluation of the right sub-pebbler of a composite pebbler progresses, it gradually *releases* storage units, i.e., it will require fewer and fewer storage units until, when the evaluation of the sub-pebbler is complete, it does not require any storage at all anymore. The main idea of patient pebblers is that for the initialization of left sub-pebblers, the hash chain is evaluated in the forward direction *exactly once* and those positions that are part of the initial state of the left sub-pebbler are stored by *reusing* storage units released by the right sub-pebbler as much as possible, only using previously unallocated storage units in case the left sub-pebbler's storage requirements exceed the right sub-pebbler's. In patient composition we make the reuse of storage units explicit by not permitting the initialization of the left sub-pebbler to progress until the necessary storage units become available. Furthermore, if the evaluation of the right sub-pebbler releases more storage units than required for the initialization of the left sub-pebbler, the excess number of storage units released first by the sub-pebbler are not reused and are released by the composite pebbler as well, i.e., the initialization of the left sub-pebbler waits patiently until the storage units it must use become available.

The initialization of the left sub-pebbler of a patient pebbler may start later than would be optimal for an arbitrary pebbler of the same size, but, if the patient pebbler is itself used as the right sub-pebbler in another composition, the properties of patient pebblers ensure that the initialization of the pebbler it is composed with, which typically has greater length and resource utilization than its own left sub-pebbler, can start as early as possible while keeping the storage requirements to a minimum.

We are now ready to formally define patient pebblers and patient composition. We define patient pebblers in terms of the properties necessary for performing composition and analyzing resource utilization. As noted before, we completely omit the specification of a schedule for patient pebblers.

Definition 7.2. We define a *patient pebbler* as either the elementary pebbler or a pebbler obtained by the *patient composition* of two patient pebblers of

smaller length. A patient pebbler X is defined in terms of the following defining properties:

- s_X, the *storage size*
- N_X, the *allocation schedule*
- w_X, the *work size*
- R_X, the *release schedule*
- D_X, the *remaining evaluation schedule*
- b_X, the *budget bound*.

N_X, R_X and D_X are sequences.

Definition 7.3. For any patient pebbler $X \neq E$, we define n_X, r_X, and d_X as the last elements of N_X, R_X, and D_X, respectively.

Definition 7.4. For the elementary pebbler, we define the sequence properties N_E, R_E and D_E as the empty sequence. We define the remaining properties as $n_E = 1$ and $s_E = w_E = b_E = r_E = d_E = 0$. Note that n_E, r_E, and d_E are defined, despite their corresponding sequences being empty and that $n_E = 1$.

Definition 7.5. For any patient pebblers X and Y, we define the *patient composition*, denoted $X + Y$ in terms of the properties of pebblers:

$$
\begin{aligned}
s_{X+Y} &= \max(s_X, s_Y + 1) \\
N_{X+Y} &= N_Y \| \{n_Y + n_X\} \\
w_{X+Y} &= w_X + w_Y + n_X \\
R_{X+Y} &= R_X \mathbin{\swarrow} ((\{0\} \| R_Y) + n_X) \\
D_{X+Y} &= D_X \mathbin{\swarrow} ((\{0\} \| D_Y) + w_X) \\
b_{X+Y} &= \max\left(\{b_X, b_Y\} \middle\| \left(\frac{N_X - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)} \mathbin{\swarrow} \frac{N_X + w_Y - n_Y}{n_Y}\right)\right)
\end{aligned}
$$

Corollary 7.6. *For any patient pebblers X and Y*

$$n_{X+Y} = n_X + n_Y$$
$$r_{X+Y} = \begin{cases} r_X & \text{if } s_X > s_Y \\ r_Y + n_X & \text{otherwise} \end{cases}$$
$$d_{X+Y} = \begin{cases} d_X & \text{if } s_X > s_Y \\ d_Y + w_X & \text{otherwise.} \end{cases}$$

For the remainder of this chapter, we will only consider patient pebblers and will use the terms pebbler and patient pebbler interchangeably.

We will now explain the meaning of the properties of patient pebblers and the corresponding patient composition rule one by one.

Storage Size

The storage size s_X of a pebbler X denotes the number of storage units required to evaluate the pebbler. Because the evaluation of patient pebbler $X+Y$ entails the evaluation of both Y and X, and during the evaluation of Y, the seed of X needs to be stored, the storage size s_{X+Y} must at least be $\max(s_X, s_Y + 1)$. A defining characteristic of patient composition is that the initialization of the left sub-pebbler within the evaluation of a patient pebbler reuses the storage units released during the evaluation of right sub-pebbler as much as possible. Therefore, we actually define

$$s_{X+Y} = \max(s_X, s_Y + 1).$$

Allocation Schedule

The allocation schedule N_X of a pebbler X specifies which positions on the hash chain are stored in the initial state of X. The positions are counted from the end of the chain, so each element of N_X can be viewed as the length of a sub-pebbler. The sequence is ordered from small to large, i.e., the positions are ordered from right to left, and the last element of the sequence, n_X, denotes the length of the pebbler itself.

The allocation schedule composition rule,

$$N_{X+Y} = N_Y \| \{n_X + n_Y\},$$

simply consists of the allocation schedule of the right sub-pebbler extended by the length of the composite pebbler, which indicates that the initial state of the

composite pebbler consists of the initial state of its right sub-pebbler, together with the seed for the left sub-pebbler. The initial state of the right sub-pebbler Y can be easily represented as N_Y in N_{X+Y}, since the positions are counted from the end of the chain. The seed of X, which lies at distance n_X from the end of the chain in X, is translated by n_Y, to reflect that it lies at distance $n_X + n_Y$ from the end of the chain in $X + Y$.

Note that the length of the allocation schedule of a patient pebbler $X + Y$, i.e., the size of the initial state of $X + Y$, is equal to $s_Y + 1$, rather than s_{X+Y}, as might be expected. This reflects the fact that the initial state of $X + Y$ consists of the initial state of Y and apart from storing the seed, X remains completely uninitialized. Although it may seem suboptimal not to include a (partial) initialization of X, in case $s_X > s_Y + 1$, omitting such an optimization simplifies our definitions and results. For the main results of this chapter, we will only consider patient pebblers $X + Y$ that obey the regularity requirement that $s_X \leq s_Y + 1$, unless explicitly noted otherwise.

Work Size

The work size w_X of a pebbler X denotes the total number of hash function evaluations required to evaluate the hash chain in reverse, including the computation of the initial state of the pebbler.

The number of hash function evaluations required to evaluate the patient composition of two pebblers is equal to the length of the left sub-pebbler, which is the number of evaluations required to compute the seed of the right sub-pebbler from the seed of the left sub-pebbler, in addition to the sum of the number of hash function evaluations required to evaluate each of the sub-pebblers. This gives us

$$w_{X+Y} = w_X + w_Y + n_X.$$

Release Schedule

The release schedule R_X indicates in which rounds of the evaluation of a pebbler X a storage unit is released, which is necessary to determine in which rounds the initialization of another pebbler that pebbler X may be composed with from the left can progress. Concretely, R_X is an increasing sequence of length s_X. If the kth element of R_X is equal to ℓ, that means that the last ℓ rounds of the evaluation of X require fewer than k storage units. In a sense, ℓ is the length of the remainder of the hash chain in the first round in which fewer than k storage units are required.

We will now explain the composition rule for the release schedule,

$$R_{X+Y} = R_X \swarrow ((\{0\} \| R_Y) + n_X).$$

During the rounds in which the left sub-pebbler of a composite pebbler is being evaluated, the evaluation of the composite pebbler proceeds identically to the evaluation of the left sub-pebbler. Therefore the release schedule of the composite pebbler, R_{X+Y} contains the complete release schedule of the left sub-pebbler, R_X.

If storage units released during the evaluation of the right sub-pebbler would not be reused for the evaluation of the left sub-pebbler, then also $R_Y + n_X$ would have to be included in R_{X+Y}. Here, the n_X term accounts for the fact that the position of the right sub-pebbler is translated by the length of the left sub-pebbler in the composite pebbler.

However, storage units released during the evaluation of the right sub-pebbler *are* reused for the evaluation of the left sub-pebbler. Only the storage units allocated to Y in excess of $s_X - 1$, where the -1 is due to the fact that the storage unit used to store the seed of X is *not* reused, are released by the composite pebbler. Patient composition dictates that the excess storage units released during the evaluation of Y are the first storage units released during the evaluation of Y and that only the storage units released later are reused for the evaluation of X. We can represent this in the composition rule, using the $\swarrow$ operation to combine sequences, because the release schedule is essentially in reverse order, i.e., storage units released last occur first in the release schedule. The last element of the sequence, r_X, therefore indicates the round in which the *first* storage unit released by the pebbler.

To account for the fact that the seed of X is not stored in a reused storage unit, the release schedule of Y is padded on the left with a single element. If $X \neq E$, the value of this padding does not matter, because of the use of the $\swarrow$ operation. However, if $X = E$, we need that the first element of R_{X+Y} is equal to 1. Therefore, we pad the sequence R_Y from the left with a single 0, before translating by n_X, which is equal to 1 in case $X = E$. This fully explains the composition rule for the release schedule.

It is trivial to show by induction that, if it exists, the first element of the release schedule is equal to 1, which corresponds to the fact that the seed of the pebbler is stored until the very last round.

Remaining Evaluation Schedule

The remaining evaluation schedule D_X of a pebbler X describes the number of hash evaluations that are still required to complete the evaluation of a pebbler at the point when a storage unit is released, as indicated by the release schedule R_X. This information is necessary to determine the budget bound b_X. The composition rule for the remaining evaluation schedule is very similar to the composition rule for the release schedule and the derivation is identical, except of the fact that the remaining evaluation schedule of the right sub-pebbler should be translated by the work size, rather than the length, of the left sub-pebbler, because whenever a pebbler releases a storage unit from its right sub-pebbler, the left sub-pebbler remains completely uninitialized. This gives the following composition rule for the remaining evaluation schedule,

$$D_{X+Y} = D_X \swarrow ((\{0\} \| D_Y) + w_X).$$

The length of the remaining evaluation schedule is equal to the storage size, just as the length of the release schedule. The first element of the remaining evaluation schedule, if it exists, must be equal to 0, since this corresponds to the number of hash evaluations still required in the last round. Since the value of the padding of D_Y from the left in the composition rule is only relevant in case $X = E$ and, in this case $w_X = 0$ and the first element of D_{X+Y} should be 0, we pad D_Y from the left with 0.

The last element of the remaining evaluation schedule, d_X, denotes the number of hash evaluations required to complete the evaluation of X at the point when its first storage unit is released.

Budget Bound

The budget bound is the number of hash evaluations per round required to evaluate the pebbler. Note that the number of hash evaluations performed in a round may actually be smaller than the budget bound, but in patient pebblers there must be at least one round in which the number of hash evaluations is equal to the budget bound. The budget bound is computed as a fraction and to obtain the actual budget, the budget bound should be rounded upwards. The composition rule for the budget bound,

$$b_{X+Y} = \max\left(\{b_X, b_Y\} \middle\| \left(\frac{N_X - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)} \swarrow \frac{N_X + w_Y - n_Y}{n_Y}\right)\right)$$

is easily the most complex of the composition rules. It consists of various parts that we will each address separately.

Just as the evaluation of a composite pebbler consists of three parts, to wit, the evaluation of its left sub-pebbler, the evaluation of its right sub-pebbler, and the initialization of its left sub-pebbler (the latter two of which are performed simultaneously), so does the composition rule for the budget bound consist of three parts. The maximum of these three contributions to the budget bound determines the final budget bound.

First of all, the budget bound of a composite pebbler can not be smaller than the budget bounds of its sub-pebblers. Therefore the maximum of b_X and b_Y, corresponding to the budget bounds of the evaluation of the left and right sub-pebblers, respectively, is included in composition rule. The remainder of the composition rule

$$\frac{N_X - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)} \swarrow \frac{N_X + w_Y - n_Y}{n_Y} \tag{7.1}$$

represents the contribution to the budget bound due to the initialization of the left sub-pebbler X, *while the right sub-pebbler* Y *is being evaluated simultaneously.* (7.1) describes a sequence of length s_X. Each element of this sequence describes the number of hash evaluations required to initialize the part of X of length given by the corresponding element of N_X.

The sequence in (7.1) is described with the $\swarrow$ operation. In case $s_X \leq s_Y + 1$, this can be simplified to its left-hand operand,

$$\frac{N_X - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)}, \tag{7.2}$$

which describes the contribution to the budget bound due to the initialization of X which is carried out by reusing storage units released during the evaluation of Y.

Each element of (7.2) is given by the number of hash evaluations required to initialize sections of X, described by $N_X - 1$, while simultaneously evaluating part of Y, which is derived from D_Y as described below. This total number of hash evaluation is divided by the number of rounds in which the initialization needs to be completed to obtain the average number of hash function evaluations required per round. As the initialization of a section of X can only begin when the corresponding storage unit is released during the evaluation of Y, this number of rounds is derived from R_Y.

For each section of X, the number of rounds available to perform the initialization of a section is given by $1 + (\{0\} \| R_Y)$. The $+1$ term stems from the fact that we allow the initialization to be completed in the first evaluation round of

X. The sequence R_Y is padded from the left with a single 0, because the *first* storage unit to be reused is actually used to compute the *second* position of the initial state of X. The last section of X to be initialized, the length of which is described by the first element of N_X is actually computed without storing any state, since the computed position is immediately output, and therefore this has to be computed in a single round, which explains the padding with zero.

Finally, the $\{0\}\|D_Y$ term is due to the fact that while initialization of X is being performed, the evaluation of Y must simultaneously be completed. D_Y describes the number of hash evaluations required to complete the evaluation of Y in the rounds that a storage unit it released. This sequence is also padded with zero from the left, aligning it with $1+\{0\}\|R_Y$ in the denominator, because no more hash evaluations are required to complete the evaluation of Y in the first round of evaluation of X.

Note that (7.2) considers initialization of the entire section of X to the endpoint, not just from one stored position to the next. The budget requirement for one such section given by (7.2) may not actually suffice to initialize the entire section, as there exist pebblers for which the initialization of the left sub-pebbler reaches a stored position and can not continue until another storage unit is released by the evaluation of the right sub-pebbler. However, if this is the case, then another element of (7.2) for the shorter, remaining section will actually give a higher budget requirement. The budget bound of Definition 7.5 suffices, because it includes the maximum taken over the entire sequence.

The right-hand operand of (7.1),

$$\frac{N_X + w_Y - n_Y}{n_Y}, \tag{7.3}$$

simply describes the contribution to the budget bound due to the initialization of X if the storage units used for the initialization of X would not be reused from the evaluation of Y and, instead, the initialization of X starts in the same round as the evaluation of Y. The fraction describes the number of hash evaluations required to initialize (sections of) X, given by $N_X - 1$, while simultaneously evaluating the entirety of Y, given by $w_Y - (n_Y - 1)$, divided by the number of rounds in which the initialization needs to be completed, given by n_Y.

In contract to (7.2), we do not include a $+1$ term in the denominator of (7.3). The reason for this is a rather technical detail, which we nonetheless consistently apply to all our results. As described in the explanation of (7.2), we actually consider the first round of evaluation of a pebbler as part of its initialization and, in fact, consider the initial state of a pebbler equal to the

stored state of the pebbler *at the end of its first output round*. Two (minor) consequences of this are that we do not allow any initialization of a sub-pebbler to be performed during the first round of evaluation (initialization of parallel pebblers is permitted, of course) and we in fact disregard the number of hash evaluations performed in the first output round for the purpose of determining the budget of a pebbler (this is most evident for trivial pebblers, Definition 7.23, where the budget is one less than one might expect).

Example

An example of the 'skeleton' of a patient pebbler is shown in Figure 7.3. The pebbler has the same sub-pebbler structure as the (composite equivalent of) the pebbler in Figure 7.2. As explained above, to analyze patient pebblers we do not need to specify the evaluation schedule, i.e., which hash function evaluations are performed in which rounds, as only the sub-pebbler structure suffices. Therefore, there is no composition rule for the schedule and consequently no schedule is given in Figure 7.3. Instead, only the confines within which the initialization of a sub-pebbler must be completed are given, indicated with dashed lines. The zigzag lines within these confines represent the unspecified hash function evaluations.

Let

$$\begin{aligned} P &= E + E \\ Q &= P + P \\ W &= P + (P + Q) \\ Y &= Q + Q \\ Z &= W + Y. \end{aligned}$$

Pebbler Z is the pebbler shown in Figure 7.3. The properties Z and its sub-pebblers are given in Table 7.1.

Some of the properties of Z can also be immediately recognized in the figure. The fact that $s_Z = 4$ can be seen from the fact that the maximum number of dots occurring on any single line is 4 and $N_Z = \{2, 4, 8, 16\}$ can be found by observing the positions of the dots on the top line, counting backwards from position 16, the length of the pebbler. The release schedule $R_Z = \{1, 3, 5, 7\}$ can be found by determining the lowest line on which the corresponding number of dots occur.

The work size w_Z and consequently the remaining evaluation schedule D_Z cannot be immediately found in the graphical representation of Z. The budget

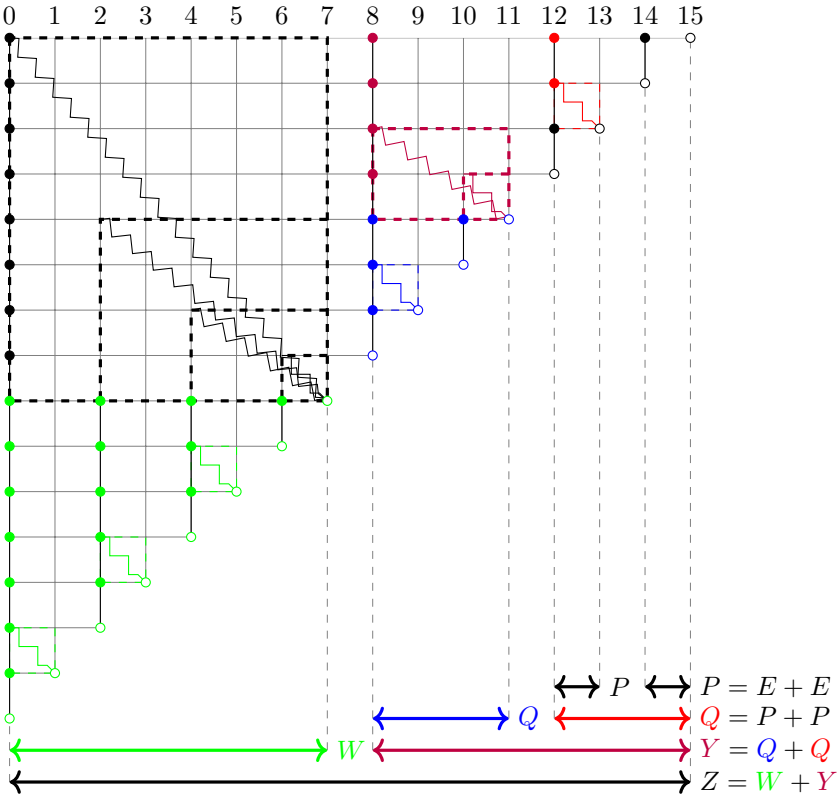


Figure 7.3: A patient pebbler with the same sub-pebbler structure as the example in Figure 7.2. The dashed borders mark the confines within which the hash function evaluations must be carried out to initialize the corresponding sub-pebbler. For patiant pebblers we do not specify exactly how each initialization should proceed, and have consequently indicated these with zigzag lines. Note that the sub-pebblers of W and the blue Q are not shown.

	E	P	Q	Y	$P+Q$	W	Z
s_X	0	1	2	3	3	4	4
N_X	$\emptyset$	$\{2\}$	$\{2,4\}$	$\{2,4,8\}$	$\{2,4,6\}$	$\{2,4,6,8\}$	$\{2,4,8,16\}$
n_X	1	2	4	8	6	8	16
w_X	0	1	4	12	7	10	30
R_X	$\emptyset$	$\{1\}$	$\{1,3\}$	$\{1,3,7\}$	$\{1,3,5\}$	$\{1,3,5,7\}$	$\{1,3,5,7\}$
r_X	0	1	3	7	5	7	7
D_X	$\emptyset$	$\{0\}$	$\{0,1\}$	$\{0,1,5\}$	$\{0,1,2\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
d_X	0	0	1	5	2	3	3
b_X	0	0	1	3/2	1	1	3/2

Table 7.1: Properties of pebbler Z displayed in Figure 7.3 and its sub-pebblers.

bound is also not directly visible, however, the Figure does graphically represent the components from which the budget bound is determined and can make the explanation of its definition above more insightful.

Recall that we define the round number as the number of the position that is output in that round. In the figure, the round number is therefore equal to the position of the open circle on each line.

Consider first sub-pebbler $Y = Q + Q$ on positions 8 through 15. Its right sub-pebbler Q releases a storage unit in rounds 15 and 13. This is consistent with $R_Q = \{1, 3\}$ translated by 12. The storage unit released in round 15 is not reused by the left sub-pebbler Q of Y, but the storage unit released in round 13 is reused. This storage unit is used for the initialization of the left sub-pebbler Q, which covers positions 8 through 11. The initialization requires 3 hash function evaluations, corresponding to the $N_X - 1$ term in the numerator in the definition of the budget bound (here we specifically refer to the second element of N_Q - 1). The initialization has to be completed in 2 rounds, corresponding to the denominator in the budget bound, specifically the second element of $1 + (\{0\} \| R_Q)$. This is represented in Figure 7.3 by the area marked with the dashed border between positions 8 and 11. The 3 hash function evaluations that must be performed in these 2 rounds, but are not specified further as represented by the zigzag line running diagonally through this box.

As part of this initialization position 10 is stored. Position 11 is computed but never stored. Computing position 11 from position 10 requires a single hash function evaluation, which must be performed in round 11 itself. This is represented by the small dashed box between positions 10 and 11 where the zigzag line again represents the hash function evaluation.

Since no evaluations are carried out *within* pebbler Y in parallel to the initialization described above, the budget required to complete the initialization of the left sub-pebbler Q of Y on time is determined by taking the maximum of 3 evaluations in 2 rounds, 1 evaluation in 1 round, and the budget bounds of the sub-pebblers themselves.

To determine the budget bound of Z, we study the initialization of W in parallel to the evaluation of Y in the same way. We see that to initialize W requires 7 hash function evaluations in 8 rounds, as indicated by the dashed area from position 0 to 7. However, in this case, also 5 hash function evaluations must be performed in the evaluation of Y, as indicated by the three marked areas between positions 8 and 9, 8 and 11, and 12 and 13. It does no matter exactly how the initializations within Y are scheduled, because they must be completely performed in parallel to the outermost initialization of W.

This last point illustrates why patient pebblers can be analyzed without fully specifying an evaluation schedule. The evaluation of a pebbler can be broken down into the initialization of its sub-pebblers. Even though multiple sub-pebblers can be in a state of initialization at the same time, due to structure imposed by the patient reuse of storage units, it is not possible to *interleave* these initializations, i.e., it is not possible to *begin* initialization within the sub-pebbler while any initialization is carried out within the right sub-pebbler. Therefore it suffices to only count the total number of hash function evaluations when determining the contribution of the evaluation of the right sub-pebbler to the budget bound. This is reflected by defining the remaining evaluation schedule D_X in terms of the (total) work size w_X of its right sub-pebbler.

In the initialization of W from position 0 to 7, it must store positions 2, 4 and 6. The initialization of W begins as soon as Y releases its first storage unit, but this initialization must store position 2 and cannot continue until Y releases a second storage unit. In this case, the initialization can only continue beyond position 2 in round 11. Therefore, the budget bound also includes the fact that part of W must be initialized using 5 hash function evaluations in 4 rounds. We can see that in this case there is also one hash function evaluation to be carried out in the evaluation of Y. This reasoning must also be applied to the third section of W when the third storage unit is released by Y, and so on. In determining the budget bound each section of initialization must be considered and the maximum must be taken, as the outermost section spanning the entire length of the sub-pebbler is not necessarily the one that requires the highest budget.

There is no need for the initialization to already reach the "threshold" position at the time the next storage unit is released, therefore we take the

entire section up to the final position of a pebbler into account when considering the budget requirements for its initialization. In the figure this represented by having multiple dashed areas overlap with a common bottom right corner.

7.5 Space-Time Product

Coppersmith and Jakobsson propose the product of the storage size and the budget of a pebbler as complexity measure [CJ02].

The results of Griewank [Gri92] and Grimm et al. [GPRS96] (cf. Section 7.7 for more details) imply that for any pebbler X

$$w_X \geq \frac{n_X}{4s_X}(\log_2 n_X)^2. \tag{7.4}$$

A similar result, although with somewhat incompatible definitions to ours, is also claimed independently by Coppersmith and Jakobsson [CJ02].

The work size w_X is defined as the total number of hash function evaluations required to reverse a hash chain and is equal to the *sum* of the number of hash function evaluations over all rounds, *including* the initialization. The budget, instead, is equal to the *maximum* of the number of hash function evaluations over all rounds, *excluding* the initialization. We can therefore conclude for the budget bound b_X of any pebbler X that

$$b_X \geq \frac{w_X - (n_X - 1)}{n_X} > \frac{1}{4s_X}(\log_2 n_X)^2 - 1. \tag{7.5}$$

Coppersmith and Jakobsson [CJ02] also show the existence of pebblers Y using a binary partitioning scheme such that

$$b_Y s_Y = \frac{1}{2}(\log_2 n_Y)^2 + O(\log n_Y \log\log n_Y). \tag{7.6}$$

These so-called *binary pebblers* were refined by Yum et al. [YSEL09] and Schoenmakers [Sch16], showing the existence of pebblers that achieve exactly

$$b_Y s_Y = \frac{1}{2}(\log_2 n_Y)^2. \tag{7.7}$$

In this chapter we are interested in pebblers that achieve both budget and storage size logarithmic in the length of the pebbler with a small constant in the dominant factor. This leads us to the following definition.

Definition 7.7. For any pebbler $X \neq E$, we define the *space-time product factor*, ϕ_X, as

$$\phi_X = \frac{s_X \lceil b_X \rceil}{(\log_2 n_X)^2},$$

and the *averaged space-time product factor*, π_X, as

$$\pi_X = \frac{s_X w_X}{n_X (\log_2 n_X)^2}.$$

Note that we use the actual budget, $\lceil b_X \rceil$, of X in the definition of ϕ_X, rather than the budget bound, as explained in Section 7.4.

Due to (7.5) we can derive a lower bound on the space-time product factor in terms of the averaged space-time product factor.

Corollary 7.8. *For any pebbler $X \neq E$, we have*

$$\phi_X \geq \pi_X - \frac{1}{(\log_2 n_X)^2},$$

Definition 7.7 allows us to state the objective of this chapter more precisely. Our main question is whether there exist sequences of pebblers $X^1, X^2, \ldots$ of increasing length, such that their space-time work factor approaches the known lower bound of $1/4$, i.e., such that $\lim_{k\to\infty} \phi_{X^k} = 1/4$.

7.6 Useful Facts

We will now give some simple facts about patient pebblers. First, we will give some bounds on patient pebbler properties that will be useful in deriving bounds on the resource utilization of complex pebblers. The proofs of the following are all straightforward by induction. All such proofs in this chapter are deferred until Section 7.12.

Lemma 7.9. *For any pebbler X, we have*

$$\begin{aligned} 1 &\leq n_X \\ 2 &\leq N_X \\ N_X &\leq n_X. \end{aligned}$$

Lemma 7.10. *For any pebbler X, we have*

$$n_X - 1 \leq w_X \leq \binom{n_X}{2}.$$

Corollary 7.11. *For any pebbler X, we have*

$$\frac{w_X}{n_X} \leq \frac{n_X - 1}{2}.$$

Lemma 7.12. *For any pebbler X, we have*

$$1 \leq R_X \leq n_X - 1.$$

Lemma 7.13. *For any pebbler X, we have*

$$0 \leq D_X \leq \binom{R_X}{2}.$$

Corollary 7.14. *For any pebbler X, we have*

$$\frac{D_X}{R_X} \leq \frac{r_X - 1}{2}$$

Lemma 7.15. *For any pebbler X, we have*

$$0 \leq b_X \leq n_X - 1.$$

Next, we explicitly list the properties of patient pebblers in which one or both of the constituent pebblers is the elementary pebbler. These serve as simple examples of composite patient pebblers and the application of the composition rules, and form the base cases for the binomial pebblers defined in the upcoming Section 7.7. Note that for pebblers X with $s_X \geq 2$, the pebbler $X+E$ of Corollary 7.17 does *not* satisfy the regularity requirement that $s_X \leq s_E + 1$.

Corollary 7.16. *The pebbler $E + E$ has the following properties.*

$$\begin{aligned} s_{E+E} &= 1 \\ N_{E+E} &= \{2\} \\ w_{E+E} &= 1 \\ R_{E+E} &= \{1\} \\ D_{E+E} &= \{0\} \\ b_{E+E} &= 0 \end{aligned}$$

Corollary 7.17. *For any pebbler $X \neq E$, we have*

$$\begin{aligned} s_{X+E} &= s_X \\ N_{X+E} &= \{n_X + 1\} \\ w_{X+E} &= w_X + n_X \\ R_{X+E} &= R_X \\ D_{X+E} &= D_X \\ b_{X+E} &= n_X - 1 \end{aligned}$$

Corollary 7.18. *For any pebbler $Y \neq E$, we have*

$$\begin{aligned} s_{E+Y} &= s_Y + 1 \\ N_{E+Y} &= N_Y \| \{1 + n_Y\} \\ w_{E+Y} &= w_Y + 1 \\ R_{E+Y} &= (\{0\} \| R_Y) + 1 \\ D_{E+Y} &= \{0\} \| D_Y \\ b_{E+Y} &= b_Y \end{aligned}$$

Finally, for pebblers $X + Y$ that satisfy the regularity requirement that $s_X \leq s_Y + 1$, we can simplify the composition rules for s_{X+Y} and b_{X+Y} as follows.

Corollary 7.19. *For any pebblers X and Y, such that $s_X \leq s_Y + 1$, we have*

$$\begin{aligned} s_{X+Y} &= s_Y + 1 \\ b_{X+Y} &= \max\left(\{b_X, b_Y\} \middle\| \frac{N_X - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)}\right) \end{aligned}$$

In case $s_X \in \{s_Y, s_Y + 1\}$, we can also simplify the definition of R_{X+Y} and D_{X+Y} as given by the following two corollaries.

Corollary 7.20. *For any pebblers X and Y, such that $s_X = s_Y + 1$, we have*

$$\begin{aligned} R_{X+Y} &= R_X \\ D_{X+Y} &= D_X \end{aligned}$$

Corollary 7.21. *For any pebblers X and Y, such that $s_X = s_Y$, we have*

$$\begin{aligned} R_{X+Y} &= R_X \| \{n_X + r_Y\} \\ D_{X+Y} &= D_X \| \{w_X + d_Y\} \end{aligned}$$

7.7 Binomial Pebblers

Griewank and Grimm et al. give the optimal solution for program reversal in terms of memory and computational time usage [Gri92, GPRS96]. This works were performed in the context of reverse automatic differentiation, but can be applied directly to hash chain reversal. In the terms we have introduced for patient pebblers, these results state that the storage size and work size can be simultaneously optimized using a sub-pebbler structure that is defined by binomial coefficients. In this section, we will describe this solution in terms of patient pebblers, which we will call binomial pebblers.

The optimality of binomial pebblers in regards to work size essentially stems from the fact that the pebbler will keep reusing its storage units until the very end of its evaluation, when the remainder of the chain is so short that not all storage units can be usefully allocated. This means, however, that if two such pebblers are composed through patient composition, the initialization of the left sub-pebbler cannot begin until last rounds of evaluation of the right sub-pebbler. It is precisely this ruthless efficiency of binomial pebblers that results in optimal work size, but a very high budget bound.

Since we are interested in pebblers that exhibit low budget bound, binomial pebblers are emphatically not the solution to our problem of hash chain reversal. However, binomial pebblers turn out to be useful building blocks from which more complex pebblers can be constructed and are, in fact, key to proving the asymptotic optimality, with respect to the budget bound rather than to the work size, of our solution.

We will first define binomial pebblers as follows, which is reminiscent of the definition of binomial coefficients in Pascal's triangle.

Definition 7.22. For any $\sigma \geq 1$ and $\beta \geq 1$, we define the binomial pebbler ${}_{\sigma}B_{\beta}$ as

$$ {}_{\sigma}B_{\beta} = {}_{\sigma}\hat{B}_{\beta-1} + {}_{\sigma-1}\hat{B}_{\beta}, $$

where

$$ {}_{\sigma}\hat{B}_{\beta} = \begin{cases} E & \text{if } \sigma = 0 \vee \beta = 0 \\ {}_{\sigma}B_{\beta} & \text{otherwise.} \end{cases} $$

An example of the binomial pebbler ${}_3B_3$ is given in Figure 7.4.

We call σ the storage size of the binomial pebbler since, as will be shown later, it coincides with the notion of storage size as defined for patient pebblers in general. β is called the block size for reasons that will become evident when

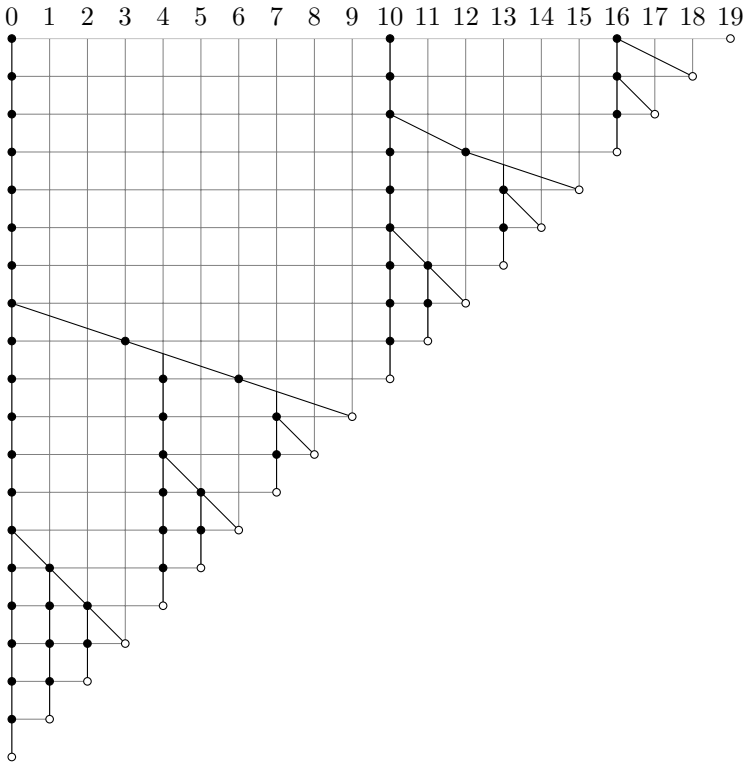


Figure 7.4: Binomial pebbler ${}_3B_3$ with possible schedule. Note that the initialization of each sub-pebbler is completed in only σ rounds. This illustrates that binomial pebblers have a high budget bound, despite minimizing the work size.

combining the notion of trivial pebblers defined immediately below with the notion of power pebblers as defined later in Section 7.10.

Definition 7.23. We call a binomial pebbler with $\sigma = 1$ a *trivial* pebbler and a binomial pebbler with $\beta = 1$ a *naïve* pebbler.

The naïve pebbler is the pebbler that stores all positions on the hash chain as part of its initialization. For evaluation, this pebbler does not have to recompute any position on the chain and simply outputs the appropriate stored value. The work size of this pebbler is therefore minimal, within all pebblers of the same length, but the storage size grows linearly with the length.

The trivial pebbler, on the other hand, stores only its seed value and must therefore recompute the entire (remainder of the) chain for each output. It has the minimum storage size 1 at every length, but the work size scales quadratically with the length, and the budget bound linearly.

Due to the super-logarithmic, in fact, linear resource utilization of these pebblers neither of these pebblers is suitable in practice for reversing long hash chains. However, as mentioned before, binomial pebblers in general can serve as a useful building block to construct more complex pebblers. Using trivial or naïve pebblers in our constructions lead to pebblers that are more straightforward to analyze than using binomial pebblers in general.

Next, we will analyze the properties of binomial pebblers.

Lemma 7.24. *For any $\sigma \geq 1$ and $\beta \geq 1$, the binomial pebbler ${}_{\sigma}B_{\beta}$ has the following properties:*

$$
\begin{aligned}
s_{{}_{\sigma}B_{\beta}} &= \sigma \\
N_{{}_{\sigma}B_{\beta}} &= \left\{ \binom{\beta + i}{\beta} \right\}_{i=1}^{\sigma} \\
w_{{}_{\sigma}B_{\beta}} &= \sigma \binom{\beta + \sigma}{\beta - 1} \\
R_{{}_{\sigma}B_{\beta}} &= \{i\}_{i=1}^{\sigma} \\
D_{{}_{\sigma}B_{\beta}} &= \{0\}^{\sigma} \\
b_{{}_{\sigma}B_{\beta}} &= \frac{1}{\sigma} \left(\binom{\beta - 1 + \sigma}{\beta - 1} - 1 \right).
\end{aligned}
$$

Corollary 7.25. *For any $\sigma \geq 1$ and any $\beta \geq 1$, we have*

$$n_{{}_\sigma B_\beta} = \binom{\beta+\sigma}{\beta}$$
$$r_{{}_\sigma B_\beta} = \sigma$$
$$d_{{}_\sigma B_\beta} = 0$$

There are several things of note. First, observe that for a binomial pebbler ${}_\sigma B_\beta$ the storage size is indeed equal to σ and that the length is equal to the binomial coefficient $\binom{\beta+\sigma}{\beta}$. The release schedule consists of rounds 1 through σ, which is due to the fact that the binomial pebblers reuses its storage units until the very end of its evaluation. Correspondingly, the remaining evaluation schedule is equal to zero everywhere, as there is nothing left to evaluate at this point.

For any σ the binomial pebbler ${}_\sigma B_2$ with block size 2 has budget bound $b_{{}_\sigma B_2} = 1$ and length $n_{{}_\sigma B_2} = \binom{\sigma+2}{2}$. This pebbler corresponds to Sella's optimal pebbler with budget bound 1 [Sel03].

For the averaged space-time product factor we have the following.

Corollary 7.26. *For any $\sigma \geq 1$ and $\beta \geq 1$,*

$$\pi_{{}_\sigma B_\beta} = \frac{\beta\sigma^2}{(\sigma+1)\left(\log_2 \binom{\beta+\sigma}{\beta}\right)^2}$$

Note that we can obtain binomial pebblers of length exponential in the storage size and block size, if we let both sizes grow at the same (up to a linear factor) rate. We therefore have the following lemma for the behavior of the averaged space-time product factor in the limit.

Lemma 7.27. *For any $\sigma \geq 1$ and $\beta \geq 1$, we have*

$$\lim_{\ell\to\infty} {}_{\ell\sigma} B_{\ell\beta} = \frac{\beta\sigma}{\left(\beta \log_2 \frac{\beta+\sigma}{\beta} + \sigma \log_2 \frac{\beta+\sigma}{\sigma}\right)^2}.$$

Proof. It is a well-known fact that, for n and k tending to infinity with k growing linearly in n, the approximation

$$\begin{aligned}\log_2 \binom{n}{k} &\sim nH(k/n) \\ &= k \log_2 \left(\frac{n}{k}\right) + (n-k) \log_2 \left(\frac{n}{n-k}\right), \end{aligned} \tag{7.8}$$

where H is the binary entropy function, can be used. Substitution of this approximation in the equation of Corollary 7.26 yields

$$\lim_{\ell\to\infty} {}_{\ell\sigma}B_{\ell\beta} = \lim_{\ell\to\infty} \frac{\ell^3\beta\sigma^2}{(\ell\sigma+1)\left(\ell\beta\log_2\frac{\ell(\beta+\sigma)}{\ell\beta} + \ell\sigma\log_2\frac{\ell(\beta+\sigma)}{\ell\sigma}\right)^2}, \tag{7.9}$$

from which the lemma directly follows. □

Corollary 7.28. *It is straightforward to see that taking $\sigma = \beta$ in Lemma 7.27 then gives*

$$\lim_{\sigma\to\infty} \pi_{\sigma B_\sigma} = \frac{1}{4}. \tag{7.10}$$

In the limit of Lemma 7.27 we may also try to minimize $\pi_{\sigma B_\beta}$ in σ and β for given $\sigma + \beta$. We claim, without proof, that this in fact results in $\sigma = \beta$ for the optimum. This shows that the lower bound for the averaged space-time product factor of $1/4$ as given by [CJ02] is asymptotically approached by binomial pebblers. Note that the results of Griewank [Gri92] and Grimm et al. [GPRS96] imply that the minimal averaged space-time product factor is achieved by binomial pebblers.

However, the averaged space-time product factor π only gives an indication of a lower bound on the space-time product factor ϕ and, as noted before, because of the ruthless efficiency of binomial pebblers with respect to the work size the actual budget bound of a such binomial pebbler scales as $O(n/\log n)$.

7.8 Fibonacci Pebblers

We introduce and analyze Fibonacci pebblers. Fibonacci sequence displays exponential growth by repeatedly composing smaller elements from the same sequence. As such, our framework of patient composition is naturally amenable to the Fibonacci sequence.

As we shall show, Fibonacci pebblers give a first indication that pebblers with space-time product factor smaller than $1/2$ do exist. To allow for generalization of the Fibonacci sequence, we permit the sequence to start with two arbitrary pebblers, rather than the elementary pebbler. This generalization of substituting an arbitrary pebbler for the elementary pebbler in compositions is explored further in Section 7.9 and is fundamental for our final results.

We first give our definition of the Fibonacci sequence for completeness.

Definition 7.29.

$$f_k = \begin{cases} k & \text{if } k \leq 1 \\ f_{k-1} + f_{k-2} & \text{otherwise} \end{cases}$$

Our results also make use of the following sequence, defined in terms of the Fibonacci sequence.

Definition 7.30.

$$g_k = \sum_{i=0}^{k} f_i f_{k-i}$$

Lemma 7.31. *For any k*

$$g_k = \begin{cases} 0 & \textit{if } k \leq 1 \\ g_{k-1} + g_{k-2} + f_{k-1} & \textit{otherwise} \end{cases}$$

Lemma 7.32. *For any k*

$$g_k = \frac{2kf_{k+1} - (k+1)f_k}{5}$$

The proofs of these lemmas are straightforward by induction and are included in Section 7.12 for completeness.

It is a well-known fact that

$$f_k = \frac{\varphi^k - \psi^k}{\sqrt{5}}, \tag{7.11}$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio and $\psi = \frac{1-\sqrt{5}}{2}$. We therefore have the following.

Corollary 7.33.

$$f_k \sim \frac{\varphi^k}{\sqrt{5}}$$
$$g_k \sim \frac{k\varphi^k(2\varphi - 1)}{5\sqrt{5}} = \frac{k\varphi^k}{5}$$

We are now ready to define Fibonacci pebblers.

Definition 7.34. For any pebblers X and Y, and any $k \geq 1$, we define the Fibonacci pebbler, denoted ${}^{Y}_{X}F_k$, as

$${}^{Y}_{X}F_k = \begin{cases} X & \text{if } k = 1 \\ Y & \text{if } k = 2 \\ {}^{Y}_{X}F_{k-1} + {}^{Y}_{X}F_{k-2} & \text{otherwise.} \end{cases}$$

The definition is analogous to that of the Fibonacci numbers. However, for generality, we permit the base pebblers to be arbitrary. Additionally, because patient composition is not commutative, the order of the two component pebblers in the recursive definition is important. At the end of this section we will also briefly consider the reverse definition. We have chosen to treat this order first and more extensively, because it features lower storage size when compared to reverse Fibonacci pebblers.

An example of the Fibonacci pebbler ${}^{E}_{E}F_8$ is given in Figure 7.5.

For our analysis of Fibonacci pebblers, we will focus on Fibonacci pebblers ${}^{Y}_{X}F_k$ whose base pebblers X and Y have the same storage size, i.e., $s_X = s_Y$. This is possible without loss of generality due to the following two corollaries.

Corollary 7.35. *For any pebblers X and Y, such that $s_X < s_Y$ let $X' = Y$ and $Y' = Y + X$. Then, $s_{X'} = s_{Y'}$, and, for any $k \geq 1$,*

$${}^{Y}_{X}F_k = {}^{Y'}_{X'}F_{k-1}.$$

Corollary 7.36. *For any pebblers X and Y, such that $s_X \geq s_Y$ let $X' = Y + X$ and $Y' = (Y + X) + Y$. Then, $s_{X'} = s_{Y'}$, and, for any $k \geq 2$,*

$${}^{Y}_{X}F_k = {}^{Y'}_{X'}F_{k-2}.$$

Note that we did not enforce the regularity constraint that $s_X \leq s_Y + 1$ in our definition of Fibonacci pebblers. The above corollary can also be applied resolve violation of the regularity constraint.

To analyze the properties of Fibonacci pebblers, we distinguish between the even and odd elements of the sequence.

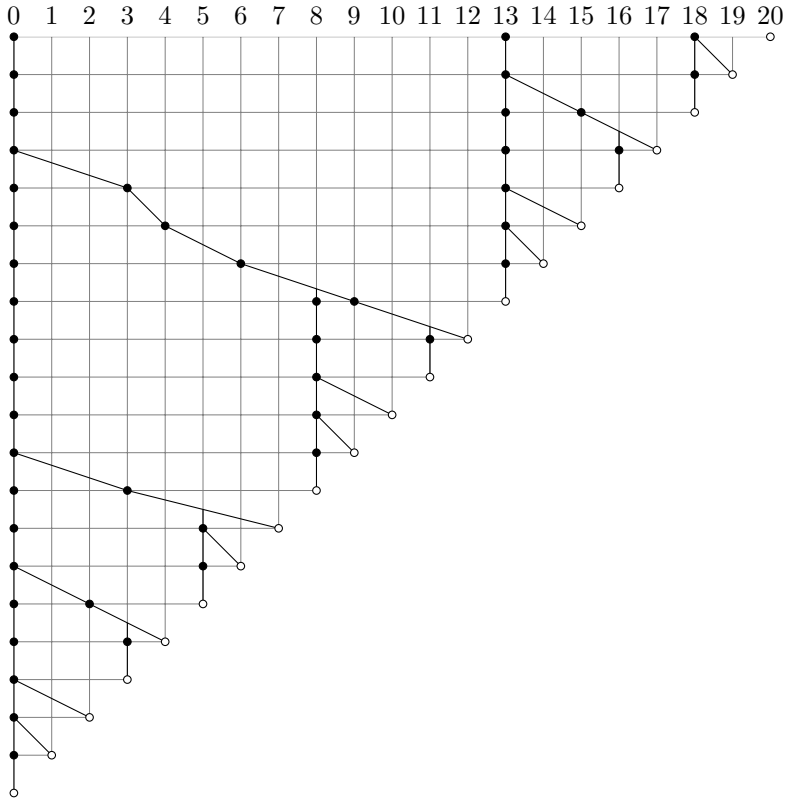


Figure 7.5: Fibonacci pebbler ${}^{E}_{E}F_8$ with possible schedule. Note that the storage size only increases for every second Fibonacci number, e.g., $s_{{}^{E}_{E}F_8} = s_{{}^{E}_{E}F_7}$, where ${}^{E}_{E}F_7$ is the left sub-pebbler of ${}^{E}_{E}F_8$.

Lemma 7.37. *For any pebblers X and Y, such that $s_X = s_Y$, and any $k \geq 1$, we have*

$$\begin{aligned}
s_{{}^{Y}_{X}F_{2k}} &= s_Y + k - 1 \\
N_{{}^{Y}_{X}F_{2k}} &= N_Y \| \{f_{2i} n_X + f_{2i+1} n_Y\}_{i=1}^{k-1} \\
w_{{}^{Y}_{X}F_{2k}} &= f_{2k-2} w_X + f_{2k-1} w_Y + g_{2k-2} n_X + g_{2k-1} n_Y \\
R_{{}^{Y}_{X}F_{2k}} &= R_Y \| \{(f_{2i-1} - 1) n_X + f_{2i} n_Y + r_X\}_{i=1}^{k-1} \\
D_{{}^{Y}_{X}F_{2k}} &= D_Y \| \{(f_{2i-1} - 1) w_X + f_{2i} w_Y + \\
&\qquad (g_{2i-1} - f_{2i-1} + 1) n_X + (g_{2i} - f_{2i}) n_Y + d_X\}_{i=1}^{k-1}
\end{aligned}$$

and

$$\begin{aligned}
s_{{}^{Y}_{X}F_{2k+1}} &= s_X + k \\
N_{{}^{Y}_{X}F_{2k+1}} &= N_X \| \{f_{2i-1} n_X + f_{2i} n_Y\}_{i=1}^{k} \\
w_{{}^{Y}_{X}F_{2k+1}} &= f_{2k-1} w_X + f_{2k} w_Y + g_{2k-1} n_X + g_{2k} n_Y \\
R_{{}^{Y}_{X}F_{2k+1}} &= R_Y \| \{(f_{2i-1} - 1) n_X + f_{2i} n_Y + r_X\}_{i=1}^{k} \\
D_{{}^{Y}_{X}F_{2k+1}} &= D_Y \| \{(f_{2i-1} - 1) w_X + f_{2i} w_Y + \\
&\qquad (g_{2i-1} - f_{2i-1} + 1) n_X + (g_{2i} - f_{2i}) n_Y + d_X\}_{i=1}^{k}
\end{aligned}$$

The proof of this lemma is straightforward (though tedious) by induction and is included in Section 7.12 for completeness.

Observe that the storage size, release schedule and remaining evaluation schedule for even Fibonacci pebblers are the same as for the preceding odd Fibonacci pebbler, i.e., $s_{{}^{Y}_{X}F_{2k+2}} = s_{{}^{Y}_{X}F_{2k+1}}$, as well as $R_{{}^{Y}_{X}F_{2k+2}} = R_{{}^{Y}_{X}F_{2k+1}}$ and $D_{{}^{Y}_{X}F_{2k+2}} = D_{{}^{Y}_{X}F_{2k+1}}$.

We have not included the budget bound in the properties of Fibonacci pebblers given in Lemma 7.37. This is because the budget bound $b_{{}^{Y}_{X}F_{k}}$ does not behave regularly for $k \leq 4$. For $k \geq 5$ we state the budget bound of Fibonacci pebblers in the following lemma.

Lemma 7.38. *For any pebblers X and Y, such that $s_X = s_Y$, and any $k \geq 5$, we have*

$$b_{{}^{Y}_{X}F_{k}} = \max\left(b_{{}^{Y}_{X}F_{k-1}}, \frac{n_{{}^{Y}_{X}F_{k-1}} - 1 + d_{{}^{Y}_{X}F_{k-3}}}{1 + r_{{}^{Y}_{X}F_{k-3}}}\right).$$

Proof. By definition, we have for $k \geq 3$

$$\begin{aligned} b_{{}_X^Y F_k} &= \max\left(\{b_{{}_X^Y F_{k-1}}, b_{{}_X^Y F_{k-2}}\}, \frac{N_{{}_X^Y F_{k-1}} - 1 + \{0\} \| D_{{}_X^Y F_{k-2}}}{1 + \{0\} \| R_{{}_X^Y F_{k-2}}}\right) \\ &= \max\left(\{b_{{}_X^Y F_{k-1}}\}, \frac{N_{{}_X^Y F_{k-1}} - 1 + \{0\} \| D_{{}_X^Y F_{k-2}}}{1 + \{0\} \| R_{{}_X^Y F_{k-2}}}\right), \end{aligned} \tag{7.12}$$

where the second equality holds because $b_{{}_X^Y F_{k-1}} = b_{{}_X^Y F_{k-2} + {}_X^Y F_{k-3}} \geq b_{{}_X^Y F_{k-2}}$ for $k \geq 4$.

For $k \geq 5$, we can split off the last element of the sequence in (7.12):

$$\begin{aligned} &\frac{N_{{}_X^Y F_{k-1}} - 1 + \{0\} \| D_{{}_X^Y F_{k-2}}}{1 + \{0\} \| R_{{}_X^Y F_{k-2}}} \\ &\quad = \underbrace{\frac{N_{{}_X^Y F_{k-3}} - 1 + \{0\} \| D_{{}_X^Y F_{k-4}}}{1 + \{0\} \| R_{{}_X^Y F_{k-4}}}}_{\leq b_{{}_X^Y F_{k-2}}} \Bigg\| \frac{n_{{}_X^Y F_{k-1}} - 1 + d_{{}_X^Y F_{k-3}}}{1 + r_{{}_X^Y F_{k-3}}}, \end{aligned} \tag{7.13}$$

The left hand subsequence is smaller than $b_{{}_X^Y F_{k-2}}$ by definition and is already included in the maximum of (7.12). To see that the equality holds for the last element, we must distinguish between even and odd k. First, for even k, the sequences $N_{{}_X^Y F_{k-1}}$ and $\{0\} \| D_{{}_X^Y F_{k-2}}$ and $\{0\} \| R_{{}_X^Y F_{k-2}}$ (including the padding with zero) have the same length and the last element of the sequence can be expressed in terms of $n_{{}_X^Y F_{k-1}}$, $d_{{}_X^Y F_{k-2}}$, and $r_{{}_X^Y F_{k-2}}$. For odd k, the sequences $N_{{}_X^Y F_{k-1}}$ is one shorter than $\{0\} \| D_{{}_X^Y F_{k-2}}$ and $\{0\} \| R_{{}_X^Y F_{k-2}}$ (including the padding with zero) and the last element of the sequence can be expressed in terms of $n_{{}_X^Y F_{k-1}}$, $d_{{}_X^Y F_{k-4}}$, and $r_{{}_X^Y F_{k-4}}$, i.e., using the *second-to-last* elements of $D_{{}_X^Y F_{k-2}}$ and $R_{{}_X^Y F_{k-2}}$. However, we also make use of the fact that, if k is even, we have $d_{{}_X^Y F_{k-2}} = d_{{}_X^Y F_{k-3}}$ and $r_{{}_X^Y F_{k-2}} = r_{{}_X^Y F_{k-3}}$, whereas for odd k, we have $d_{{}_X^Y F_{k-4}} = d_{{}_X^Y F_{k-3}}$ and $r_{{}_X^Y F_{k-4}} = r_{{}_X^Y F_{k-3}}$. This holds for any $k \geq 5$ by Lemma 7.37. □

For sufficiently large k, we can then give the budget bound more precisely.

Lemma 7.39. *For any pebblers X and Y, such that $s_X = s_Y$, there exists a K such that for any $k \geq K$*

$$b_{{}_X^Y F_{2k+1}} = b_{{}_X^Y F_{2k+2}} = \frac{n_{{}_X^Y F_{2k}} - 1 + d_{{}_X^Y F_{2k-2}}}{1 + r_{{}_X^Y F_{2k-2}}}.$$

Proof (informal). We do not formally prove this lemma, but will argue its correctness informally. It should be obvious that the equality

$$b_{{}_X^Y F_k} = \frac{n_{{}_X^Y F_{k-1}} - 1 + d_{{}_X^Y F_{k-3}}}{1 + r_{{}_X^Y F_{k-3}}} \tag{7.14}$$

must hold for infinitely many k, because if this were not the case, this would imply that there exists an upper bound c for any pebblers X and Y such that $b_{{}_X^Y F_k} < c$ for all k, whereas the length of ${}_X^Y F_k$ would grow unbounded with k and the storage size would grow logarithmically in the length. Such a sequence of pebblers of increasing length would therefore approach space-time product factor 0, which is impossible.

This weaker statement suffices for our purposes; the more complete statement of the lemma is given for concreteness. The remainder of the proof is only given as an informal argument.

Using Corollary 7.33, it can be shown that, as k grows,

$$n_{{}_X^Y F_k} \sim \frac{\varphi^{k-2}(n_X + \varphi n_Y)}{\sqrt{5}} \tag{7.15}$$

$$r_{{}_X^Y F_{2k+1}} = r_{{}_X^Y F_{2k+2}} \sim \frac{\varphi^{2k-1}(n_X + \varphi n_Y)}{\sqrt{5}} \tag{7.16}$$

$$d_{{}_X^Y F_{2k+1}} = d_{{}_X^Y F_{2k+2}} \sim \frac{(2k-1)\varphi^{2k-1}(n_X + \varphi n_Y)}{5}. \tag{7.17}$$

The growth rate in the numerator of (7.14) dominates the growth rate in the denominator. However, the fact that $r_{{}_X^Y F_{2k+2}} = r_{{}_X^Y F_{2k+1}}$ and $d_{{}_X^Y F_{2k+2}} = d_{{}_X^Y F_{2k+1}}$ for any k, also results in $b_{{}_X^Y F_{2k+2}} = b_{{}_X^Y F_{2k+1}}$ for sufficiently large k. □

As an immediate consequence of (7.15) through (7.17), combined with

$$s_{{}_X^Y F_{2k+1}} = s_{{}_X^Y F_{2k+2}} \sim k, \tag{7.18}$$

we find the asymptotic space-time product factor of any Fibonacci pebbler for large k.

Corollary 7.40. *For any pebblers X and Y we have*

$$\lim_{k\to\infty} \phi_{{}_X^Y F_k} = \frac{1}{2\sqrt{5}(\log_2 \varphi)^2} \approx 0.4639.$$

Fibonacci pebblers are a first indication that pebblers with space-time product factor smaller than $1/2$ exist. Numerically it can be confirmed that, when using the elementary pebbler for both bas pebblers, $\phi_{{}^{E}_{E}F_{42}} \approx 0.4848$ and, in fact, $\phi_{{}^{E}_{E}F_{k}} < 1/2$ for every even $k \geq 42$. For odd k, the same holds if $k \geq 61$, except for $k = 77$. The convergence is rather slow, though, as $n_{{}^{E}_{E}F_{42}} = f_{42} \approx 2^{28}$, for example.

Reverse Fibonacci Pebblers

Because the sequential composition of pebblers is not commutative, we can also define the *reverse* Fibonacci Pebblers.

Definition 7.41. For any pebblers X and Y, and any $k \geq 1$, we define the reverse Fibonacci pebbler, denoted ${}^{Y}_{X}G_k$, as

$${}^{Y}_{X}G_k = \begin{cases} X & \text{if } k = 1 \\ Y & \text{if } k = 2 \\ {}^{Y}_{X}G_{k-2} + {}^{Y}_{X}G_{k-1} & \text{otherwise.} \end{cases}$$

An example of the reverse Fibonacci pebbler ${}^{E}_{E}G_8$ is given in Figure 7.6.

For completeness, we state the properties, except the budget bound, of reverse Fibonacci pebblers without proof.

Lemma 7.42. *For any pebblers X and Y, such that $s_X + 1 = s_Y$, and any $k \geq 1$, we have*

$$\begin{aligned}
s_{{}^{Y}_{X}G_{2k}} &= s_Y + 2k - 2 \\
w_{{}^{Y}_{X}G_{2k}} &= f_{2k-2}w_X + f_{2k-1}w_Y + (g_{2k-1} - g_{2k-2})n_X + g_{2k-2}n_Y \\
N_{{}^{Y}_{X}G_{2k}} &= N_Y \| \{f_{i-2}n_X + f_{i-1}n_Y\}_{i=1}^{2k-2} \\
R_{{}^{Y}_{X}G_{2k}} &= R_Y \| \{R'_Y + f_{2i-2}n_X + (f_{2i-1} - 1)n_Y\}_{i=1}^{k-1} \\
D_{{}^{Y}_{X}G_{2k}} &= D_Y \| \{D'_Y + f_{2i-2}w_X + (f_{2i-1} - 1)w_Y + g_{2i-1}n_X + \\
&\qquad (g_{2i-1} - f_{2i} + 1)n_Y\}
\end{aligned}$$

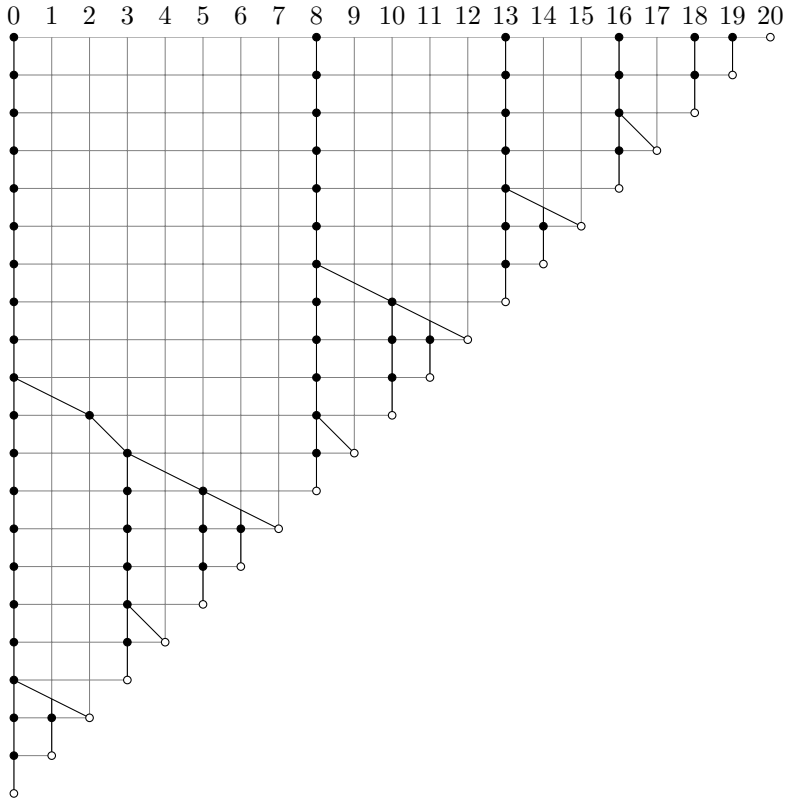


Figure 7.6: Reverse Fibonacci pebbler ${}^{E}_{E}G_8 = {}^{E+E}_{E}G_7$ with possible schedule. Note that the storage size $s_{{}^{E}_{E}G_8} = 2s_{{}^{E}_{E}F_8}$ (cf. Figure 7.5).

and

$$\begin{aligned}
s_{{}_X^Y G_{2k+1}} &= s_Y + 2k - 1 \\
w_{{}_X^Y G_{2k+1}} &= f_{2k-1} w_X + f_{2k} w_Y + (g_{2k} - g_{2k-1}) n_X + g_{2k-1} n_Y \\
N_{{}_X^Y G_{2k+1}} &= N_Y \| \{ f_{i-2} n_X + f_{i-1} n_Y \}_{i=1}^{2k-1} \\
R_{{}_X^Y G_{2k+1}} &= R_X \| \{ R'_Y + f_{2k-1} n_X + (f_{2k} - 1) n_Y \}_{i=1}^{k-1} \\
D_{{}_X^Y G_{2k+1}} &= D_X \| \{ D'_Y + f_{2i-1} w_X + (f_{2i} - 1) w_Y + g_{2i} n_X + \\
&\qquad (g_{2i} - f_{2i+1} + 1) n_Y \}
\end{aligned}$$

where R'_Y and D'_Y denote the last two *elements of R_Y and D_Y, respectively.*

As may be directly observed from the definition of reverse Fibonacci pebblers, the storage size of a reverse Fibonacci pebbler is twice a high when compared to the ordinary Fibonacci pebbler of the same length.

7.9 Nested Composition

In the previous section we defined Fibonacci pebblers recursively as the composition of two smaller Fibonacci pebblers, analogous to the definition of Fibonacci numbers. Instead of taking the elementary pebbler, we allowed for *any* pair of pebblers as the base pebblers for constructing Fibonacci pebblers to allow for more general sequences of pebblers. In this section we will further explore the idea taking an *arbitrary* pebbler, rather than the elementary pebbler, as the elementary building block for composite pebblers. We will call this notion *nested composition*. Intuitively, nested composition can be thought of as replacing every instance of the elementary pebbler in a given pebbler X with another pebbler Y. In terms of our visualization of pebblers, this corresponds to replacing every open circle with a copy of the representation of Y. Formally, we define nested composition as follows.

Definition 7.43. We recursively define the *nested composition* of pebblers X and Y, denoted XY, as follows:

$$XY = \begin{cases} Y & \text{if } X = E \\ (VY) + (WY) & \text{if } X = V + W. \end{cases}$$

Before giving an example of nested composition in Figure 7.7 we will first analyze the pebbler properties, except the budget bound, of nested composition.

Lemma 7.44. *The following holds for pebblers obtained through nested composition:*

$$\begin{aligned} s_{XY} &= s_X + s_Y \\ N_{XY} &= N_Y \| (n_Y N_X) \\ w_{XY} &= n_Y w_X + n_X w_Y \\ R_{XY} &= R_Y \| (n_Y R_X + r_Y) \\ D_{XY} &= D_Y \| (n_Y D_X + w_Y R_X + d_Y) \end{aligned}$$

Corollary 7.45. *For all pebblers X and Y, we have*

$$\begin{aligned} n_{XY} &= n_Y n_X \\ r_{XY} &= n_Y r_Y + r_Y \\ d_{XY} &= n_Y d_Y + w_Y r_Y + d_Y \end{aligned}$$

We have chosen the notation of nested composition suggestive of multiplication to reflect the fact that the length n_{XY} of a pebbler XY equal to the product $n_X n_Y$. This coincides with the intuition behind nested composition that pebbler Y is repeated once in XY for every elementary pebbler of X.

An example of the nested composition XY, where $X = {}_2B_1 + {}_1B_1$ and $Y = {}_1B_1 + {}_1B_1$ is given in Figure 7.7, complete with a possible initialization schedule. Note that the pebbler Y is repeated $n_X = 5$ times, once in each of the marked areas. Furthermore, the marked areas themselves are positioned according to X.

Although we do not explicitly define the initialization schedule for sequential composition (and therefore neither for nested composition), we have given a possible initialization schedule in Figure 7.7. Note that in this case the only possible initialization schedule for the sub-pebbler ${}_2B_1Y$ resulting the minimal budget 2 is such that 2 hash evaluations are performed in *each* of the first 8 rounds. The example gives some indication that simply scaling up the initialization schedule would not necessarily lead to an optimal budget and that the analysis of the budget bound is more complicated. In general, we do not have an exact equation for b_{XY}. Instead we prove the following upper bound.

Lemma 7.46. *For any pebblers X and Y, let*

$$h_{XY} = \max\left(b_{Y+Y}, \frac{n_Y n_X - 1 + d_Y}{r_Y + 1}, \frac{3}{2}n_X + \max\left(\frac{w_Y}{n_Y}, \frac{d_Y}{r_Y}\right)\right).$$

Then

$$b_{XY} \leq h_{XY}.$$

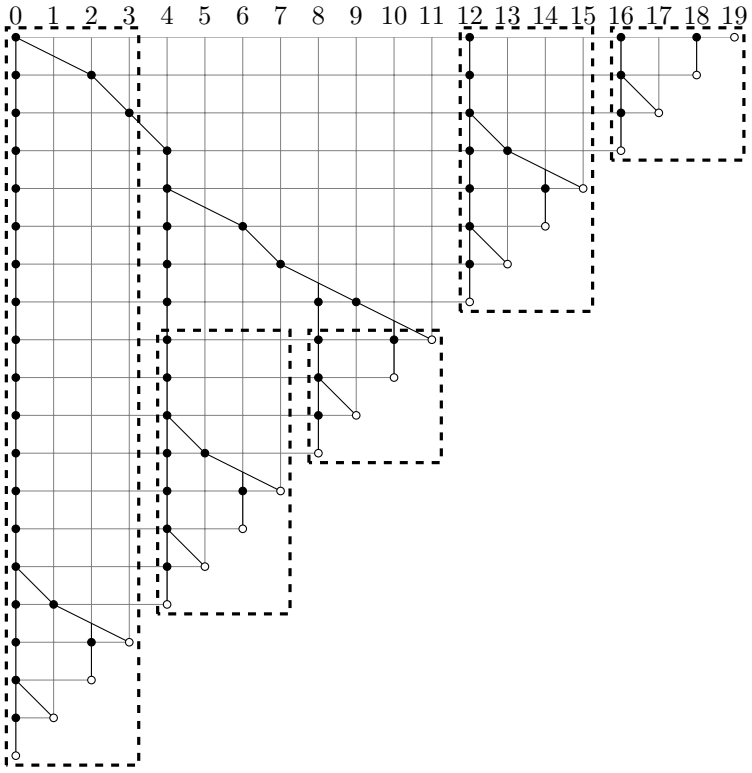


Figure 7.7: The nested pebbler $({}_2B_1 + {}_1B_1)({}_1B_1 + {}_1B_1)$ with possible schedule. Observe how the pebbler ${}_1B_1 + {}_1B_1$ is repeated in each of the $n_{{}_2B_1+{}_1B_1} = 5$ marked areas and how the marked areas themselves have width $n_{{}_1B_1+{}_1B_1} = 4$ and are positioned as in ${}_2B_1 + {}_1B_1$

For some simple pebblers we can work out the nested composition exactly. The following lemma allows us to compare our framework to the results of Sella [Sel03] and Kim [Kim03].

Lemma 7.47. *For any pebbler Y and any $\sigma \geq 1$, we have*

$$b_{\sigma B_1 Y} = b_{Y+Y}.$$

There are further similarities between the ordinary notion of multiplication and nested composition. Indeed, nested composition is defined in terms of left distributivity with the elementary pebbler as the left identity. From this, it follows that the elementary pebbler is also the right identity element under nested composition, and that nested composition is associative, as stated in the following to lemmas. However, nested composition is not commutative, as can readily be seen in the example of Figure 7.7, as it does contain 5 identical sub-pebblers of length 4, but not 4 identical sub-pebblers of length 5.

Lemma 7.48. *E is the identity element under nested composition, i.e., for any pebbler X,*

$$XE = X = EX.$$

Proof. The second equality holds by definition. The first we prove by induction. In the base case, $X = E$, the first equality holds by definition. Suppose the first equation holds for some pebblers V and W such that $s_V \leq s_W + 1$. Then, for $X = V + W$, we then have

$$XE = (VE) + (WE) = V + W = X. \tag{7.19}$$

□

Lemma 7.49. *Nested composition is associative, i.e., for all pebblers X, Y, and Z we have $(XY)Z = X(YZ)$.*

Although nested composition is associative, it is not commutative. However, we *do* have commutativity *with respect to the averaged space-time product factor*, as stated by the following lemma.

Lemma 7.50. *For any pebblers $X \neq E$ and $Y \neq E$, we have*

$$\pi_{XY} = \pi_{YX}$$

7.10 Power Pebblers

The nested composition of distinct pebblers tends to increase the averaged space-time product factor with respect to the minimal averaged space-time product factor of the two base pebblers, as evidenced by the following lemma and its corollary.

Lemma 7.51. *Let $X \neq E$ and $Y \neq E$ be pebblers. Without loss of generality, assume that $\pi_X \leq \pi_Y$. Then*

$$\pi_{XY} \geq \pi_X$$

Proof.

$$\begin{aligned}
\pi_{XY} &= \frac{(s_X + s_Y)(n_Y w_X + n_X w_Y)}{n_X n_Y (\log_2 n_X + \log_2 n_Y)^2} \\
&= \frac{s_X + s_Y}{(\log_2 n_X + \log_2 n_Y)^2} \left(\frac{w_X}{n_X} + \frac{w_Y}{n_Y} \right) \\
&= \frac{s_X + s_Y}{(\log_2 n_X + \log_2 n_Y)^2} \left(\pi_X \frac{(\log_2 n_X)^2}{s_X} + \pi_Y \frac{(\log_2 n_Y)^2}{s_Y} \right) \\
&\geq \pi_X \frac{s_X + s_Y}{(\log_2 n_X + \log_2 n_Y)^2} \left(\frac{(\log_2 n_X)^2}{s_X} + \frac{(\log_2 n_Y)^2}{s_Y} \right) \\
&= \pi_X \left(1 + \frac{(s_X \log_2 n_Y - s_Y \log_2 n_X)^2}{(\log_2 n_X + \log_2 n_Y)^2 s_X s_Y} \right) \\
&\geq \pi_X, \qquad (7.20)
\end{aligned}$$

where the first inequality holds because $\frac{(\log_2 N_Y)^2}{s_Y} > 0$. □

With a slight modification to the proof, we find the conditions under which we have equality in Lemma 7.51.

Corollary 7.52.

$$\pi_{XY} = \pi_X \iff \pi_X = \pi_Y \wedge \frac{s_X}{\log_2 n_X} = \frac{s_Y}{\log_2 n_Y}$$

Proof. In case $\pi_X = \pi_Y$ and $\frac{s_X}{\log_2 n_X} = \frac{s_Y}{\log_2 n_Y}$ the inequalities in the proof of Lemma 7.51 become equalities. Similarly, of either of the conditions does not hold, the corresponding inequality in the proof of Lemma 7.51 becomes a strict inequality. □

Note that in particular, this means that $\pi_{XX} = \pi_X$. In this section we therefore focus on the nested composition of pebblers created from only a single base pebbler. We take this beyond the nested composition of a pebbler with itself, but iterate this process.

Definition 7.53. For any pebbler X and any $k \geq 0$, we define the *power pebbler*, denoted X^k, as

$$X^k = \begin{cases} E & \text{if } k = 0 \\ XX^{k-1} & \text{otherwise.} \end{cases}$$

The process of iteratively taking the nested composition of a pebbler with itself is analogous to exponentiation, we dub these *power pebblers*. As may be expected, the length of a power pebbler is exponential in the length of its base pebbler. The properties of power pebblers are as follows.

Lemma 7.54. *For any pebbler $X \neq E$ and any $k \in \mathbb{N}$, we have*

$$\begin{aligned}
s_{X^k} &= k s_X \\
N_{X^k} &= \left\|_{i=0}^{k-1} \left(n_X^i N_X \right) \right. \\
w_{X^k} &= k n_X^{k-1} w_X \\
R_{X^k} &= \left\|_{i=0}^{k-1} \left(n_X^i R_X + \frac{n_X^i - 1}{n_X - 1} r_X \right) \right. \\
D_{X^k} &= \left\|_{i=0}^{k-1} \left(n_X^i D_X + \frac{n_X^i - 1}{n_X - 1} d_X + \right. \right. \\
&\qquad \left. \left. i n_X^{i-1} w_X R_X + \left(\frac{i n_X^{i-1}}{n_X - 1} - \frac{n_X^i - 1}{(n_X - 1)^2} \right) w_X r_X \right) \right.
\end{aligned}$$

Corollary 7.55. *For any pebbler $X \neq E$ and any $k \in \mathbb{N}$, we have*

$$\begin{aligned}
n_{X^k} &= n_X^k \\
r_{X^k} &= \frac{n_X^k - 1}{n_X - 1} r_X \\
d_{X^k} &= \frac{n_X^k - 1}{n_X - 1} d_X + \left(\frac{k n_X^{k-1}}{n_X - 1} - \frac{n_X^k - 1}{(n_X - 1)^2} \right) w_X r_X
\end{aligned}$$

If we combine our notion of power pebblers with the binomial pebbler ${}_1B_1$ of length 2, we obtain the binary pebbler ${}_1B_1^k$ of length 2^k. Although we do not describe a schedule for the evaluation for such pebblers, and have not yet stated the budget bound for power pebblers in general, these pebblers correspond to the results of [Jak02, CJ02, YSEL09, Sch16].

Lemma 7.56. *For any pebbler $X \neq E$ and any $k \geq 1$, we have*

$$\frac{d_{X^k}}{r_{X^k}} \leq \frac{d_X}{r_X} + k\frac{w_X}{n_X}$$

Just as for nested composition, we do not have an exact expression for the budget bound. Instead, we prove the following inequality.

Lemma 7.57. *For any pebbler $X \neq E$, we have*

$$b_{X^k} \leq n_X^3 + \frac{d_X}{r_X} + (k-2)\frac{w_X}{n_X}.$$

The binary pebbler strategies of [Jak02, CJ02, YSEL09, Sch16] correspond to power pebblers ${}_1B_1^k$ and the pebbler strategy of [Sel03, Kim03] is the generalization to ${}_\sigma B_1^k$ for any σ. For these pebblers, we can work out the budget bound exactly.

Lemma 7.58. *For any $\sigma \geq 1$ and any $k \geq 2$, we have*

$$b_{{}_\sigma B_1^k} = \frac{(k-1)\sigma + 1}{\sigma + 1}.$$

Using the above lemma, we can state the space-time product factor of these pebblers exactly. In particular, this shows that the space-time product factor of binary pebblers is exactly $1/2$ for even k.

Corollary 7.59. *For any $\sigma \geq 1$ and any $k \geq 2$, we have*

$$\phi_{{}_\sigma B_1^k} = \frac{\sigma \left\lceil \frac{(k-1)\sigma+1}{\sigma+1} \right\rceil}{k(\log_2(\sigma+1))^2}$$

In the general case of power pebblers of arbitrary pebblers, we see that, for large k, the upper bound on the budget bound is dominated by $k\frac{w_X}{n_X}$. This leads us to the following theorem, which is the main result of this chapter.

Theorem 7.60. *For any pebbler $X \neq E$*

$$\lim_{k\to\infty} \phi_{X^k} = \pi_X$$

Proof.

$$\begin{aligned}
\lim_{k\to\infty} \phi_{X^k} &= \lim_{k\to\infty} \frac{s_{X^k} b_{X^k}}{(\log_2 n_{X^k})^2} \\
&\leq \lim_{k\to\infty} \frac{k s_X \left\lceil n_X^3 + \frac{d_X}{r_X} + (k-2)\frac{w_X}{n_X} \right\rceil}{k^2 (\log_2 n_X)^2} \\
&= \frac{s_X w_X}{n_X (\log_2 n_X)^2} \\
&= \pi_X
\end{aligned} \tag{7.21}$$

Combined with the lower bound of Corollary 7.8, the theorem follows. □

Theorem 7.60 states that in the limit of large k, the space-time product factor tends to the averaged space-time product factor. Because binomial pebblers feature the optimal averaged space-time product factor and, in fact, binomial pebblers approach the lower bound of the averaged space-time product factor of 1/4, we therefore claim that power pebblers of binomial pebblers asymptotically approach the optimal space-time product factor of 1/4.

7.11 Reflections on Existing Results

In this section we will discuss how existing results on hash chain reversal fit within our framework. In our framework we do not specify an evaluation schedule, rather we only show the existence of an evaluation schedule that satisfies certain complexity bounds. This is in contrast to existing solutions, which specify an algorithm for reversing a hash chain of a particular length (or, more accurately, a sequence of algorithms for reversing hash chains of increasing length). Therefore, our framework can not be used to fully describe existing results. We include this section to show how our framework can be applied to analyze the complexity bounds on existing solutions.

We mention the results of Griewank and Grimm et al. on program reversal for the purpose of reverse automatic differentiation [Gri92, GPRS96] for completeness. In our framework, these results are the binomial pebblers which we have extensively studied in Section 7.7. Binomial pebblers are optimal in terms

of storage size and work size. However, our quantity of interest is the budget, rather than the work size, and binomial pebblers do not show the desired behavior where the length of the pebblers grows exponentially with the storage size and budget.

Most of the results on hash chain reversal use the so-called binary pebbler structure [Jak02, CJ02, YSEL09, Sch16, Sch17]. The binary pebbler structure corresponds to the power pebblers ${}_1B_1^k$ in our framework. Using our framework, it follows immediately that $n_{{}_1B_1^k} = 2^k$ and $\pi_{{}_1B_1^k} = 1/2$. Theorem 7.60 states that $\lim_{k\to\infty} \phi_{{}_1B_1^k} = 1/2$ as well. However, as shown by [YSEL09, Sch16] for binary pebblers ${}_1B_1^k$ the space-time product factor is actually equal to $1/2$ for all even k, not just asymptotically. The equality does not hold for odd k due to rounding. Because our results only include an upper bound on the budget for power pebblers, this does not immediately follow from our framework. Although we do not have an exact solution for the budget bound for power pebblers in general, it is straightforward to derive this for binary pebblers in our framework as well.

The strategy of Sella [Sel03] and Kim [Kim03] can be seen as a generalization of binary pebblers to a b-ary partitioning. In our framework, their results correspond to power pebblers ${}_\sigma B_1^k$, in which $\sigma = b - 1$ can be chose freely. For $\sigma = 1$, these indeed reduce to binary pebblers. It follows immediately from our framework that $n_{{}_\sigma B_1^k} = (\sigma+1)^k$ and $\pi_{{}_\sigma B_1^k} = \frac{\sigma^2}{(\sigma+1)(\log_2(\sigma+1))^2}$. Sella shows that the storage size of pebblers with this structure is at most $k(\sigma+1)$. Kim improves this claim to $s_{{}_\sigma B_1^k} = k\sigma$, which also follows from our framework.

In our framework we can easily express an alternative b-ary partitioning constructed as ${}_1B_\beta^k$, where the blocksize $\beta = b - 1$ can be chosen freely. We call such pebblers *block pebblers*, explaining why we called β the block size of binomial pebblers. Compared to the strategy of Sella and Kim, block pebblers have a lower storage size and necessarily to a greater budget. The space-time product factor of block pebblers converges to $\pi_{{}_1B_\beta^k} = \frac{\beta}{2(\log_2(\beta+1))^2}$ which, unlike the pebbling strategy of Sella and Kim, can be smaller than $1/2$. The optimum is found for $b = 5$ for which $\pi_{{}_1B_4} \approx 0.37$, first announced by Schoenmakers in [Sch17].

Schoenmakers is the first to mention pebbler structure based on Fibonacci numbers as pebblers that have lower space-time complexity than binary pebblers [Sch16]. We discuss Fibonacci pebblers extensively in Section 7.8. The space-time product factor for Fibonacci pebblers converges to $\lim_{k\to\infty} \phi_{{}_Y^X F_k} = \frac{1}{2\sqrt{5}(\log_2 \varphi)^2} \approx 0.4639$.

7.12 Deferred Proofs

Several proofs in this chapter are straightforward by induction, but rather long-winded and tedious. Such proofs have been collected in this section so as not to distract from the main text.

Proof of Lemma 7.9. In the base case, $X = E$, the lemma holds because N_E is the empty sequence and $n_E = 1$ by definition. For the induction hypothesis, assume the lemma holds for pebblers V and W. Let $X = V + W$, then $n_X = n_V + n_W$ and $N_X = N_{V+W} = N_W \| \{n_W + n_V\}$. We then have

$$\begin{aligned} 1 &\leq 1 + 1 \\ &\leq n_V + n_W \end{aligned} \tag{7.22}$$

and

$$\begin{aligned} 2 &\leq \{2\}^{sw} \| \{1 + 1\} \\ &\leq N_W \| \{n_V + n_W\} \\ &\leq \{n_W\}^{sw} \| \{n_V + n_W\} \\ &\leq n_X, \end{aligned} \tag{7.23}$$

where, in fact $n_X \geq 2$, because $X \neq E$. □

Proof of Lemma 7.10. The proof is by induction. In the base case, $X = E$, the lemma holds because $w_E = 0$ by definition. For the induction hypothesis, assume the lemma holds for pebblers V and W. Let $X = V + W$, then $w_X = w_{V+W} = w_V + w_W + n_V$ and

$$\begin{aligned} n_X - 1 &\leq n_V - 1 + n_W - 1 + n_V \\ &\leq w_V + w_W + n_V \\ &\leq \binom{n_V}{2} + \binom{n_W}{2} + n_V \\ &= \binom{n_V + n_W}{2} - n_V(n_W - 1) \\ &\leq \binom{n_X}{2}. \end{aligned} \tag{7.24}$$

□

Proof of Lemma 7.12. The proof is by induction. In the base case, $X = E$, the lemma holds because R_E is the empty sequence by definition. For the induction hypothesis, assume the lemma holds for pebblers V and W. Let $X = V + W$, then $R_X = R_{V+W} = R_V \swarrow ((\{0\} \| R_W) + n_V)$ and

$$\begin{aligned} 1 &\leq \{1\}^{s_V} \swarrow ((\{0\} \| \{1\}^{s_W}) + 1) \\ &\leq R_V \swarrow ((\{0\} \| R_W) + n_V) \\ &\leq \{n_V - 1\}^{s_V} \swarrow ((\{0\} \| \{n_W - 1\}^{s_W}) + n_V) \\ &\leq n_X - 1 \end{aligned} \tag{7.25}$$

□

Proof of Lemma 7.13. The proof is by induction. In the base case, $X = E$, the lemma holds because D_E is the empty sequence by definition. For the induction hypothesis, assume the lemma holds for pebblers V and W. Let $X = V + W$, then $D_X = D_{V+W} = D_V \swarrow ((\{0\} \| D_W) + w_V)$ and

$$\begin{aligned} 0 &\leq \{0\}^{s_V} \swarrow ((\{0\} \| \{0\}^{s_W}) + 0) \\ &\leq D_V \swarrow ((\{0\} \| D_W) + w_V) \\ &\leq \binom{R_V}{2} \swarrow \left(\binom{\{0\} \| R_W}{2} + \binom{n_V}{2} \right) \\ &= \binom{R_V}{2} \swarrow \left(\binom{(\{0\} \| R_W) + n_V}{2} - n_V(\{0\} \| R_W) \right) \\ &\leq \binom{R_X}{2}, \end{aligned} \tag{7.26}$$

where we use the fact that $\binom{0}{2} = 0$. □

Proof of Lemma 7.15. The proof is by induction. In the base case, $X = E$, the lemma holds because $b_E = 0$ and $n_E = 1$ by definition. For the induction hypothesis, assume the lemma holds for pebblers V and W. Let $X = V + W$,

then $b_X = b_{V+W} = \max\left(\{b_V, b_W\} \middle\| \left(\frac{N_V - 1 + (\{0\} \| D_W)}{1 + (\{0\} \| R_W)} \swarrow \frac{N_V + w_W - n_W}{n_W}\right)\right)$ and

$$\begin{aligned}
0 &\leq b_V \\
&\leq \max\left(\{b_V, b_W\} \middle\| \left(\frac{N_V - 1 + (\{0\} \| D_W)}{1 + (\{0\} \| R_W)} \swarrow \frac{N_V + w_W - n_W}{n_W}\right)\right) \\
&\leq \max\left(\left\{n_V - 1, n_W - 1, \frac{n_V + w_W - n_W}{n_W}\right\} \middle\| \frac{n_V - 1 + (\{0\} \| D_W)}{1 + (\{0\} \| R_W)}\right) \\
&\leq \max\left(\left\{n_W - 1, n_V + \frac{n_W - 1}{2} - 1\right\} \middle\| n_V - 1 + \underbrace{\left(\{0\} \| \frac{R_W - 1}{2}\right)}_{< \frac{n_W - 1}{2}}\right) \\
&\leq \max\left(n_W - 1, n_V + n_W - \frac{n_W + 3}{2}\right) \\
&\leq n_X - 1 \qquad (7.27)
\end{aligned}$$

□

Proof of Lemma 7.24. We first prove by induction on β that the lemma holds for all trivial pebblers, i.e., all binomial pebblers ${}_\sigma B_\beta$, where $\sigma = 1$. For the base case, $\beta = 1$, we have ${}_1 B_1 = E + E$ and, by Corollary 7.16,

$$s_{E+E} = 1 = \sigma \qquad (7.28)$$

$$N_{E+E} = \{2\} = \left\{\binom{\beta + i}{\beta}\right\}_{i=1}^{\sigma} \qquad (7.29)$$

$$w_{E+E} = 1 = \sigma\binom{\beta + \sigma}{\beta - 1} \qquad (7.30)$$

$$R_{E+E} = \{1\} = \{i\}_{i=1}^{\sigma} \qquad (7.31)$$

$$D_{E+E} = \{0\} = \{0\}^{\sigma} \qquad (7.32)$$

$$b_{E+E} = 0 = \frac{1}{\sigma}\left(\binom{\beta - 1 + \sigma}{\beta - 1} - 1\right). \qquad (7.33)$$

For the induction hypothesis, assume the lemma holds for $\sigma = 1$ and some

β. Then, for ${}_1B_{\beta+1} = {}_1B_\beta + E$, we apply Corollary 7.17 and obtain

$$\begin{aligned}
s_{{}_1B_{\beta}+E} &= s_{{}_1B_\beta} \\
&= 1 \\
&= \sigma && (7.34)\\
N_{{}_1B_{\beta}+E} &= \{n_{{}_1B_\beta} + 1\} \\
&= \left\{ \binom{\beta+1}{\beta} + 1 \right\} \\
&= \left\{ \binom{\beta+1+i}{\beta+1} \right\}_{i=1}^{\sigma} && (7.35)\\
w_{{}_1B_{\beta}+E} &= w_{{}_1B_\beta} + N_{{}_1B_\beta} \\
&= 1 \cdot \binom{\beta+1}{\beta-1} + \binom{\beta+1}{\beta} \\
&= \sigma \binom{\beta+1+\sigma}{\beta} && (7.36)\\
R_{{}_1B_{\beta}+E} &= R_{{}_1B_\beta} \\
&= \{1\} \\
&= \{i\}_{i=1}^{\sigma} && (7.37)\\
D_{{}_1B_{\beta}+E} &= D_{{}_1B_\beta} \\
&= \{0\} \\
&= \{0\}^{\sigma} && (7.38)\\
b_{{}_1B_{\beta}+E} &= n_{{}_1B_\beta} - 1 \\
&= \binom{\beta+1}{\beta} - 1 \\
&= \frac{1}{\sigma}\left(\binom{\beta+\sigma}{\beta} - 1 \right) && (7.39)
\end{aligned}$$

Next, we prove by induction on σ that the lemma holds for all naïve pebblers, i.e., all binomial pebblers ${}_\sigma B_\beta$, where $\beta = 1$. We have already proved this for the base case, $\sigma = 1$.

For the induction hypothesis, assume the lemma holds for $\beta = 1$ and some

σ. Then, for ${}_{\sigma+1}B_1 = E + {}_{\sigma}B_1$, we apply Corollary 7.18 and we obtain

$$\begin{aligned}
s_{E+{}_{\sigma}B_1} &= s_{{}_{\sigma}B_1} + 1 \\
&= \sigma + 1 && (7.40)\\
N_{E+{}_{\sigma}B_1} &= N_{{}_{\sigma}B_1} \| \{1 + n_{{}_{\sigma}B_1}\} \\
&= \left\{ \binom{1+i}{1} \right\}_{i=1}^{\sigma} \Bigg\| \left\{ 1 + \binom{1+\sigma}{1} \right\} \\
&= \left\{ \binom{\beta+i}{\beta} \right\}_{i=1}^{\sigma+1} && (7.41)\\
w_{E+{}_{\sigma}B_1} &= w_{{}_{\sigma}B_1} + 1 \\
&= \sigma \binom{1+\sigma}{0} + 1 \\
&= (\sigma+1)\binom{\beta+\sigma}{\beta-1} && (7.42)\\
R_{E+{}_{\sigma}B_1} &= (\{0\} \| R_{{}_{\sigma}B_1}) + 1 \\
&= (\{0\} \| \{i\}_{i=1}^{\sigma}) + 1 \\
&= \{i\}_{i=1}^{\sigma+1} && (7.43)\\
D_{E+{}_{\sigma}B_1} &= \{0\} \| D_{{}_{\sigma}B_1} \\
&= \{0\} \| \{0\}^{\sigma} \\
&= \{0\}^{\sigma+1} && (7.44)\\
b_{E+{}_{\sigma}B_1} &= b_{{}_{\sigma}B_1} \\
&= 0 \\
&= \frac{1}{\sigma+1} \left(\binom{\beta+\sigma}{\beta-1} - 1 \right) && (7.45)
\end{aligned}$$

We will now prove by induction that the lemma holds for all σ and β. We have proved this for the base cases in which $\sigma = 1 \vee \beta = 1$. For the induction hypothesis, assume the lemma holds for all σ and β such that $\sigma + \beta \leq k$ for some k. Then for any σ and β such that $\sigma > 1 \wedge \beta > 1 \wedge \sigma + \beta = k + 1$, we have ${}_{\sigma}B_{\beta} = {}_{\sigma}B_{\beta-1} + {}_{\sigma-1}B_{\beta}$. Since $s_{{}_{\sigma}B_{\beta-1}} = \sigma = (\sigma - 1) + 1 = s_{{}_{\sigma-1}B_{\beta}} + 1$ by

the induction hypothesis, we can apply Corollaries 7.19 and 7.20, and obtain

$$\begin{aligned}
s_{\sigma B_{\beta-1}+\sigma-1 B_\beta} &= s_{\sigma-1 B_\beta} + 1 \\
&= \sigma && (7.46)\\
N_{\sigma B_{\beta-1}+\sigma-1 B_\beta} &= N_{\sigma-1 B_\beta} \| \{ n_{\sigma B_{\beta-1}} + n_{\sigma B_{\beta-1}} \}) \\
&= \left\{ \binom{\beta+i}{\beta} \right\}_{i=1}^{\sigma-1} \left\| \left\{ \binom{\beta-1+\sigma}{\beta-1} + \binom{\beta+\sigma-1}{\beta} \right\} \right. \\
&= \left\{ \binom{\beta+i}{\beta} \right\}_{i=1}^{\sigma} && (7.47)\\
w_{\sigma B_{\beta-1}+\sigma-1 B_\beta} &= w_{\sigma B_{\beta-1}} + w_{\sigma-1 B_\beta} + n_{\sigma B_{\beta-1}} \\
&= \sigma \binom{\beta-1+\sigma}{\beta-1-1} + (\sigma-1)\binom{\beta+\sigma-1}{\beta-1} + \binom{\beta-1+\sigma}{\beta-1} \\
&= \sigma \binom{\beta+\sigma}{\beta-1} && (7.48)\\
R_{\sigma B_{\beta-1}+\sigma-1 B_\beta} &= R_{\sigma B_{\beta-1}} \\
&= \{i\}_{i=1}^{\sigma} && (7.49)\\
D_{\sigma B_{\beta-1}+\sigma-1 B_\beta} &= D_{\sigma B_{\beta-1}} \\
&= \{0\}^{\sigma} && (7.50)\\
b_{\sigma B_{\beta-1}} + {}_{\sigma-1}B_\beta &= \max \left(\{ b_{\sigma B_{\beta-1}}, b_{\sigma-1 B_\beta} \} \left\| \frac{N_{\sigma B_{\beta-1}} - 1 + \{0\} \| D_{\sigma-1 B_\beta}}{1 + \{0\} \| R_{\sigma-1 B_\beta}} \right. \right) \\
&= \max \left(\left\{ \frac{1}{\sigma} \left(\binom{\beta-2+\sigma}{\beta-2} - 1 \right), \right. \right. \\
&\qquad \left. \left. \frac{1}{\sigma-1} \left(\binom{\beta-2+\sigma}{\beta-1} - 1 \right) \right\} \right\| \\
&\qquad \frac{\left\{ \binom{\beta-1+i}{\beta-1} \right\}_{i=1}^{\sigma} - 1 + \{0\} \| \{0\}^{\sigma-1}}{1 + \{0\} \| \{i\}_{i=1}^{\sigma-1}} \\
&= \max \left(\left\{ \frac{1}{i} \left(\binom{\beta-1+i}{\beta-1} - 1 \right) \right\}_{i=1}^{\sigma} \right) \\
&= \frac{1}{\sigma} \left(\binom{\beta-1+\sigma}{\beta-1} - 1 \right). && (7.51)
\end{aligned}$$

□

Proof of Lemma 7.31. The proof is by induction on k. In the base cases, when $k \leq 1$, the lemma clearly holds, because

$$g_0 = f_0 f_0 = 0 \tag{7.52}$$
$$g_1 = f_0 f_1 + f_1 f_0 = 0 \tag{7.53}$$

For the induction hypothesis, assume the lemma holds for some $k \geq 0$ and $k + 1 \geq 1$. Then

$$\begin{aligned} g_{k+2} &= \sum_{i=0}^{k+2} f_i f_{k+2-i} \\ &= \sum_{i=0}^{k} f_i (f_{k+1-i} + f_{k-i}) + f_{k+1} f_1 + f_{k+2} f_0 \\ &= \sum_{i=0}^{k+1} f_i f_{k+1-i} - f_{k+1} f_0 + \sum_{i=0}^{k} f_i f_{k-i} + f_{k+1} \\ &= g_{k+1} + g_k + f_{k+1} \end{aligned} \tag{7.54}$$

□

Proof of Lemma 7.32. The proof is by induction on k. In the base cases, when $k \leq 1$, the lemma clearly holds, since $g_0 = g_1 = 0$ by Lemma 7.31. For the induction hypothesis, assume the lemma holds for some $k \geq 0$ and $k + 1 \geq 1$. Then

$$\begin{aligned} g_{k+2} &= g_{k+1} + g_k + f_{k+1} \\ &= \frac{(2k+2) f_{k+2} - (k+2) f_{k+1}}{5} + \frac{2k f_{k+1} - (k+1) f_k}{5} + f_{k+1} \\ &= \frac{(2k+2) f_{k+2} + (k+3) f_{k+1} - (k+1) f_k}{5} \\ &= \frac{(k+1) f_{k+2} + (2k+4) f_{k+1}}{5} \\ &= \frac{(2k+4) f_{k+3} - (k+3) f_{k+2}}{5} \end{aligned} \tag{7.55}$$

□

Proof of Lemma 7.37. The proof is by induction on k. In the base case, $k = 1$, we have ${}^{Y}_{X}F_{2k} = Y$ and ${}^{Y}_{X}F_{2k+1} = Y + X$. Therefore

$$\begin{aligned}
s_Y &= s_Y + k - 1 && (7.56)\\
w_Y &= f_{2k-2}w_X + f_{2k-1}w_Y + g_{2k-2}n_X + g_{2k-1}n_Y && (7.57)\\
N_Y &= N_Y \| \{f_{2i}n_X + f_{2i+1}n_Y\}_{i=1}^{k-1} && (7.58)\\
R_Y &= R_Y \| \{(f_{2i-1} - 1)n_X + f_{2i}n_Y + r_X\}_{i=1}^{k-1} && (7.59)\\
D_Y &= D_Y \| \{(f_{2i-1} - 1)w_X + f_{2i}w_Y + \\
&\qquad (g_{2i-1} - f_{2i-1} + 1)n_X + (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k-1} && (7.60)
\end{aligned}$$

and

$$\begin{aligned}
s_{Y+X} &= s_X + 1\\
&= s_X + k && (7.61)\\
w_{Y+X} &= w_Y + w_X + n_Y\\
&= f_{2k-1}w_X + f_{2k}w_Y + g_{2k-1}n_X + g_{2k}n_Y && (7.62)\\
N_{Y+X} &= N_X \| \{n_X + n_Y\}\\
&= N_X \| \{f_{2i-1}n_X + f_{2i}n_Y\}_{i=1}^{k} && (7.63)\\
R_{Y+X} &= R_Y \| \{n_Y + r_X\}\\
&= R_Y \| \{(f_{2i-1} - 1)n_X + f_{2i}n_Y + r_X\}_{i=1}^{k} && (7.64)\\
D_{Y+X} &= D_Y \| \{w_Y + d_X\}\\
&= D_Y \| \{(f_{2i-1} - 1)w_X + f_{2i}w_Y + \\
&\qquad (g_{2i-1} - f_{2i-1} + 1)n_X + (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k} && (7.65)
\end{aligned}$$

For the induction hypothesis, assume the lemma holds for some k. Then for

${}^{Y}_{X}F_{2(k+1)} = {}^{Y}_{X}F_{2k+1} + {}^{Y}_{X}F_{2k}$ we apply Corollary 7.20 and obtain

$$s_{{}^{Y}_{X}F_{2k+1}+{}^{Y}_{X}F_{2k}} = s_Y + k \tag{7.66}$$

$$\begin{aligned} w_{{}^{Y}_{X}F_{2k+1}+{}^{Y}_{X}F_{2k}} &= w_{{}^{Y}_{X}F_{2k+1}} + w_{{}^{Y}_{X}F_{2k}} + n_{{}^{Y}_{X}F_{2k+1}} \\ &= f_{2k-1}w_X + f_{2k}w_Y + g_{2k}n_X + g_{2k+1}n_Y + \\ &\quad f_{2k-2}w_X + f_{2k-1}w_Y + g_{2k-1}n_X + g_{2k}n_Y + \\ &\quad f_{2k-1}n_X + f_{2k}n_Y \\ &= f_{2k}w_X + f_{2k+1}w_Y + g_{2k+1}n_X + g_{2k+2}n_Y \end{aligned} \tag{7.67}$$

$$\begin{aligned} N_{{}^{Y}_{X}F_{2k+1}+{}^{Y}_{X}F_{2k}} &= N_{{}^{Y}_{X}F_{2k}} \| \{n_{{}^{Y}_{X}F_{2k+1}} + n_{{}^{Y}_{X}F_{2k}}\} \\ &= N_Y \| \{f_{2i}n_X + f_{2i+1}n_Y\}_{i=1}^{k-1} \| \\ &\quad \{f_{2k-1}n_X + f_{2k}n_Y + f_{2k-2}n_X + f_{2k-1}n_Y\} \\ &= N_Y \| \{f_{2i}n_X + f_{2i+1}n_Y\}_{i=0}^{k} \end{aligned} \tag{7.68}$$

$$\begin{aligned} R_{{}^{Y}_{X}F_{2k+1}+{}^{Y}_{X}F_{2k}} &= R_{{}^{Y}_{X}F_{2k+1}} \\ &= R_Y \| \{(f_{2i-1} - 1)n_X + f_{2i}n_Y + r_X\}_{i=1}^{k} \end{aligned} \tag{7.69}$$

$$\begin{aligned} D_{{}^{Y}_{X}F_{2k+1}+{}^{Y}_{X}F_{2k}} &= D_{{}^{Y}_{X}F_{2k+1}} \\ &= D_Y \| \{(f_{2i-1} - 1)w_X + f_{2i}w_Y + (g_{2i-1} - f_{2i-1} + 1)n_X + \\ &\qquad (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k} \end{aligned} \tag{7.70}$$

Combining the results for ${}^{Y}_{X}F_{2(k+1)}$ with the induction hypothesis, we can

apply Corollary 7.21 to ${}^{Y}_{X}F_{2(k+1)+1} = {}^{Y}_{X}F_{2(k+1)} + {}^{Y}_{X}F_{2k+1}$ and obtain

$$s_{{}^{Y}_{X}F_{2k+2}+{}^{Y}_{X}F_{2k+1}} = s_X + k + 1 \tag{7.71}$$

$$\begin{aligned} w_{{}^{Y}_{X}F_{2k+2}+{}^{Y}_{X}F_{2k+1}} &= w_{{}^{Y}_{X}F_{2k+2}} + w_{{}^{Y}_{X}F_{2k+1}} + n_{{}^{Y}_{X}F_{2k+2}} \\ &= f_{2k}w_X + f_{2k+1}w_Y + g_{2k+1}n_X + g_{2k+2}n_Y + \\ &\quad f_{2k-1}w_X + f_{2k}w_Y + g_{2k}n_X + g_{2k+1}n_Y + \\ &\quad f_{2k}n_X + f_{2k+1}n_Y \\ &= f_{2(k+1)-1}w_X + f_{2(k+1)}w_Y + g_{2(k+1)}n_X + \\ &\quad g_{2(k+1)+1}n_Y \end{aligned} \tag{7.72}$$

$$\begin{aligned} N_{{}^{Y}_{X}F_{2k+2}+{}^{Y}_{X}F_{2k+1}} &= N_{{}^{Y}_{X}F_{2k+1}} \| \{n_{{}^{Y}_{X}F_{2k+2}} + n_{{}^{Y}_{X}F_{2k+1}}\} \\ &= N_X \| \{f_{2i-1}n_X + f_{2i}n_Y\}_{i=1}^{k} \| \\ &\quad \{f_{2k}n_X + f_{2k+1}n_Y + f_{2k-1}n_X + f_{2k}n_Y\} \\ &= N_X \| \{f_{2i-1}n_X + f_{2i}n_Y\}_{i=0}^{k+1} \end{aligned} \tag{7.73}$$

$$\begin{aligned} R_{{}^{Y}_{X}F_{2k+2}+{}^{Y}_{X}F_{2k+1}} &= R_{{}^{Y}_{X}F_{2k+2}} \| \{n_{{}^{Y}_{X}F_{2k+2}} + r_{{}^{Y}_{X}F_{2k+1}}\} \\ &= R_Y \| \{(f_{2i-1}-1)n_X + f_{2i}n_Y + r_X\}_{i=1}^{k} \| \\ &\quad \{f_{2k}n_X + f_{2k+1}n_Y + \\ &\quad\ (f_{2k-1}-1)n_X + f_{2k}n_Y + r_X\} \\ &= R_Y \| \{(f_{2i-1}-1)n_X + f_{2i}n_Y + r_X\}_{i=1}^{k+1} \end{aligned} \tag{7.74}$$

$$\begin{aligned} D_{{}^{Y}_{X}F_{2k+2}+{}^{Y}_{X}F_{2k+1}} &= D_{{}^{Y}_{X}F_{2k+2}} \| \{w_{{}^{Y}_{X}F_{2k+2}} + d_{{}^{Y}_{X}F_{2k+1}}\} \\ &= D_Y \| \{(f_{2i-1}-1)w_X + f_{2i}w_Y + (g_{2i-1} - f_{2i-1} + 1)n_X + \\ &\qquad (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k} \| \\ &\quad \{f_{2k}w_X + f_{2k+1}w_Y + g_{2k}n_X + g_{2k+1}n_Y + \\ &\quad\ (f_{2k-1}-1)w_X + f_{2k}w_Y + (g_{2k-1} - f_{2k-1} + 1)n_X + \\ &\quad\ (g_{2k} - f_{2k})n_Y + d_X\} \\ &= D_Y \| \{(f_{2i-1}-1)w_X + f_{2i}w_Y + (g_{2i-1} - f_{2i-1} + 1)n_X + \\ &\qquad (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k} \| \\ &\quad \{(f_{2k+1}-1)w_X + f_{2k+2}w_Y + (g_{2k+1} - f_{2k+1} + 1)n_X + \\ &\quad\ (g_{2k+2} - f_{2k+2})n_Y + d_X\} \\ &= D_Y \| \{(f_{2i-1}-1)w_X + f_{2i}w_Y + (g_{2i-1} - f_{2i-1} + 1)n_X + \\ &\qquad (g_{2i} - f_{2i})n_Y + d_X\}_{i=1}^{k+1} \end{aligned} \tag{7.75}$$

□

Proof of Lemma 7.44. The proof is by induction. In the base case, $X = E$, the defining sequences of X are all empty and $s_X = w_X = 0$ and $n_X = 1$. The lemma clearly holds in this case. For the induction hypothesis, assume the lemma holds for some pebblers V and W. Then, for $X = V + W$, we have

$$\begin{aligned}
s_{VY+WY} &= \max(s_{VY}, s_{WY} + 1) \\
&= \max(s_V + s_Y, s_W + s_Y + 1) \\
&= \max(s_V, s_W + 1) + s_Y \\
&= s_X + s_Y && (7.76) \\
N_{VY+WY} &= N_{WY} \| \{n_{VY} + n_{WY}\} \\
&= N_Y \| (n_Y N_W) \| \{n_Y n_V + n_Y n_W\} \\
&= N_Y \| (n_Y (N_W \| \{n_V + n_W\})) \\
&= N_Y \| (n_Y N_X) && (7.77) \\
w_{VY+WY} &= w_{VY} + w_{WY} + n_{VY} \\
&= n_Y w_V + n_V w_Y + n_Y w_W + n_W w_Y + n_Y n_V \\
&= n_Y (w_V + w_W + n_V) + (n_V + n_W) w_Y \\
&= n_Y w_X + n_X w_Y && (7.78) \\
R_{VY+WY} &= R_{VY} \swarrow ((\{0\} \| R_{WY}) + n_{VY}) \\
&= (R_Y \| (n_Y R_V + r_Y)) \swarrow \\
&\quad ((\{0\} \| R_Y \| (n_Y R_W + r_Y)) + n_Y n_V) \\
&= R_Y \| (n_Y (R_V \swarrow ((\{0\} \| R_W) + n_V)) + r_Y) \\
&= R_Y \| (n_Y R_X + r_Y) && (7.79) \\
D_{VY+WY} &= D_{VY} \swarrow ((\{0\} \| D_{WY}) + w_{VY}) \\
&= (D_Y \| (n_Y D_V + w_Y R_V + d_Y)) \swarrow \\
&\quad ((\{0\} \| D_Y \| (n_Y D_W + w_Y R_W + d_Y)) + n_Y w_V + n_V w_Y) \\
&= D_Y \| (n_Y (D_V \swarrow ((\{0\} \| D_W) + w_V)) + \\
&\qquad w_Y (R_V \swarrow ((\{0\} \| R_W) + n_V)) + d_Y) \\
&= D_Y \| (n_Y D_X + w_Y R_X + d_Y) && (7.80)
\end{aligned}$$

□

Proof of Lemma 7.46. The proof is by induction on. In the base case, $X = E$, the lemma holds since $b_{XY} = b_Y \leq b_{Y+Y} \leq h_{XY}$.

For the induction hypothesis, assume $b_{VY} \leq h_{VY}$ and $b_{WY} \leq h_{WY}$ for some pebblers V and W. Then, for $X = V + W$, we have

$$\begin{aligned} b_{VY+WY} &= \max\left(\left(\frac{N_{VY} - 1 + (\{0\} \| D_{WY})}{1 + \{0\} \| R_{WY})} \swarrow \frac{N_{VY} + w_{WY} - n_{WY}}{n_{WY}}\right) \middle\| \right. \\ &\qquad \left. \{b_{VY}, b_{WY}\}\right) \\ &\leq \max\left(\frac{N_{VY} - 1 + (\{0\} \| D_{WY})}{1 + R_{WY}} \middle\| \left\{\frac{w_{WY}}{n_{WY}}, b_{VY}, b_{WY}\right\}\right). \end{aligned} \tag{7.81}$$

Since $n_V < n_X$ and $n_W < n_X$ we have $h_{VY} \leq h_{XY}$ and $h_{WY} \leq h_{XY}$. By the induction hypothesis, we therefore have

$$\begin{aligned} \max(\{b_{VY}, b_{WY}\}) &\leq \max(\{h_{VY}, h_{WY}\}) \\ &\leq h_{XY}. \end{aligned} \tag{7.82}$$

For $\frac{w_{WY}}{n_{WY}}$, we have

$$\begin{aligned} \frac{w_{WY}}{n_{WY}} &= \frac{n_Y w_W + n_W w_Y}{n_Y n_W} \\ &= \frac{w_W}{n_W} + \frac{w_Y}{n_Y} \\ &< \frac{n_W}{2} + \frac{w_Y}{n_Y} \\ &\leq h_{XY}, \end{aligned} \tag{7.83}$$

where the first inequality follows from Corollary 7.11.

What remains to show is that h_{XY} is an upper bound to

$$\begin{aligned} \frac{N_{VY} - 1 + (\{0\} \| D_{WY})}{1 + \{0\} \| R_{WY}} \leq \underbrace{\frac{N_Y - 1 + \{0\} \| D_Y}{1 + \{0\} \| R_Y}}_{\leq b_{Y+Y} \leq h_{XY}} \middle\| \underbrace{\left\{\frac{n_Y n_X - 1 + d_Y}{1 + r_Y}\right\}}_{\leq h_{XY}} \middle\| \\ \frac{n_Y n_X + n_Y D_W + w_Y R_W + d_Y}{1 + n_Y R_W + r_Y}. \end{aligned} \tag{7.84}$$

The first two subsequences are trivially bounded by h_{XY}. For the third, we

have

$$\begin{aligned}\frac{n_Y n_X + n_Y D_W + w_Y R_W + d_Y}{1 + n_Y R_W + r_Y} &\leq \frac{n_Y(n_X + D_W)}{n_Y R_W} + \frac{w_Y R_W + d_Y}{n_Y R_W + r_Y} \\ &< \frac{n_X}{R_W} + \frac{n_W}{2} + \max\left(\frac{w_Y}{n_Y}, \frac{d_Y}{r_Y}\right) \\ &< \frac{3}{2} n_X + \max\left(\frac{w_Y}{n_Y}, \frac{d_Y}{r_Y}\right) \\ &\leq h_{XY} \end{aligned} \tag{7.85}$$

Here, we use $r_W \geq 1$ and apply Corollary 7.14 to $\frac{D_W}{R_W}$. □

Proof of Lemma 7.47. The proof is by induction on σ. In the base case, $\sigma = 1$, the lemma holds, because ${}_1B_1Y = Y + Y$ for all pebblers Y. For the induction hypothesis, assume the lemma holds for all pebblers Y and some σ. Then

$$b_{{}_{\sigma+1}B_1Y} = b_{Y +_\sigma B_1 Y} \tag{7.86}$$

$$= \max\left(\{b_Y, b_{{}_\sigma B_1 Y}\} \middle\| \frac{N_Y - 1 + (\{0\} \| D_{{}_\sigma B_1 Y})}{1 + (\{0\} \| R_{{}_\sigma B_1 Y})}\right) \tag{7.87}$$

$$= \max\left(\{b_Y, b_{Y+Y}\} \middle\| \frac{N_Y - 1 + (\{0\} \| D_Y)}{1 + (\{0\} \| R_Y)}\right) \tag{7.88}$$

$$= b_{Y+Y}, \tag{7.89}$$

where the third equality holds due the induction hypothesis and Lemma 7.44 by discarding the tail of the R and D terms, and the last equality because $b_{Y+Y} \geq b_Y$. □

Proof of Lemma 7.49. The proof is by induction on. In the base case, $X = E$, the lemma holds because

$$(EY)Z = YZ = E(YZ) \tag{7.90}$$

by definition. For the induction hypothesis, suppose there exist pebblers V and W such that $(VY)Z = V(YZ)$ and $(WY)Z = W(YZ)$ for all pebblers Y and Z. Then, for $X = V + W$, we have

$$\begin{aligned}((V + W)Y)Z &= (VY + WY)Z \\ &= (VY)Z + (WY)Z \\ &= V(YZ) + W(YZ) \\ &= (V + W)(YZ). \end{aligned} \tag{7.91}$$

□

Proof of Lemma 7.50. We have

$$s_{XY} = s_X + s_Y = s_{YX} \tag{7.92}$$
$$n_{XY} = n_X n_Y = n_{YX} \tag{7.93}$$
$$w_{XY} = n_Y w_X + n_X w_Y = w_{YX}. \tag{7.94}$$

Therefore

$$\begin{aligned} \pi_{XY} &= \frac{s_{XY} w_{XY}}{n_{XY}(\log_2 n_{XY})^2} \\ &= \frac{s_{YX} w_{YX}}{n_{YX}(\log_2 n_{YX})^2} \\ &= \pi_{YX} \end{aligned} \tag{7.95}$$

□

Proof of Lemma 7.54. The proof is by induction on k. In the base case, $k = 0$, the defining sequences are all empty and the lemma clearly holds. For the induction hypothesis, assume the lemma holds for some k. Then for $X^{k+1} = XX^k$

$$\begin{aligned} s_{XX^k} &= s_X + s_{X^k} \\ &= s_X + k s_X \\ &= (k+1) s_X \end{aligned} \tag{7.96}$$
$$\begin{aligned} w_{XX^k} &= n_{X^k} w_X + n_X w_{X^k} \\ &= n_X^k w_X + n_X k n_X^{k-1} w_X \\ &= (k+1) n_X^k w_X \end{aligned} \tag{7.97}$$

$$
\begin{aligned}
N_{XX^k} &= N_{X^k} \| (n_{X^k} N_X) \\
&= \left(\Big\|_{i=0}^{k-1} \left(n_X^i N_X \right) \right) \| (n_X^k N_X) \\
&= \Big\|_{i=0}^{k} (n_X^i N_X) \qquad (7.98) \\
R_{XX^k} &= R_{X^k} \| (n_{X^k} R_X + r_{X^k}) \\
&= \left(\Big\|_{i=0}^{k-1} \left(n_X^i R_X + \frac{n_X^i - 1}{n_X - 1} r_X \right) \right) \Big\| \left(n_X^k R_X + \frac{n_X^k - 1}{n_X - 1} r_X \right) \\
&= \Big\|_{i=0}^{k} \left(n_X^i R_X + \frac{n_X^i - 1}{n_X - 1} r_X \right) \qquad (7.99) \\
D_{XX^k} &= \left(\Big\|_{i=0}^{k-1} \left(n_X^i D_X + \frac{n_X^i - 1}{n_X - 1} d_X + i n_X^{i-1} w_X R_X + \right. \right. \\
&\qquad \left. \left. \left(\frac{i n_X^{i-1}}{n_X - 1} - \frac{n_X^i - 1}{(n_X - 1)^2} \right) w_X r_X \right) \right) \Big\| \\
&\quad \left(n_X^k D_X + k n_X^{k-1} w_X R_X + \frac{n_X^k - 1}{n_X - 1} d_X + \right. \\
&\quad \left. \left(\frac{k n_X^{k-1}}{n_X - 1} - \frac{n_X^k - 1}{(n_X - 1)^2} \right) w_X r_X \right) \\
&= \Big\|_{i=0}^{k-1} \left(n_X^i D_X + \frac{n_X^i - 1}{n_X - 1} d_X + \right. \\
&\qquad \left. i n_X^{i-1} w_X R_X + \left(\frac{i n_X^{i-1}}{n_X - 1} - \frac{n_X^i - 1}{(n_X - 1)^2} \right) w_X r_X \right) \qquad (7.100)
\end{aligned}
$$

□

Proof of Lemma 7.56.

$$\begin{aligned}\frac{d_{X^k}}{r_{X^k}} &= \frac{d_X}{r_X} - \frac{w_X}{n_X - 1} + k\frac{w_X}{n_X(1 - n_X^{-k})} \\ &= \frac{d_X}{r_X} + \frac{w_X}{n_X}\left(\frac{kn_X^k}{n_X^k - 1} - \frac{n_X}{n_X - 1}\right) \\ &= \frac{d_X}{r_X} + \frac{w_X}{n_X}\left(k + \frac{k}{n_X^k - 1} - \frac{n_X}{n_X - 1}\right) \\ &< \frac{d_X}{r_X} + k\frac{w_X}{n_X} + \frac{w_X}{n_X}\frac{k + 1 - n_X^k}{n_X^k - 1} \\ &\leq \frac{d_X}{r_X} + k\frac{w_X}{n_X}, \end{aligned} \tag{7.101}$$

where the last inequality is due to the fact that $k \geq 1$ and $n_X \geq 2$. □

Proof of Lemma 7.57. The proof is by induction on k. The lemma holds for any $k \leq 2$, because

$$\begin{aligned} n_X^3 + \frac{d_X}{r_X} + (k-2)\frac{w_X}{n_X} &\geq n_X^3 - 2\frac{w_X}{n_X} \\ &> n_X^3 - n_X \\ &> n_X^2 \end{aligned} \tag{7.102}$$

and $b_{X^k} < n_X^k$. For the induction hypothesis, suppose $b_{X^j} \leq n_X^3 + \frac{d_X}{r_X} + (j - 2)\frac{w_X}{n_X}$ for all $0 \leq j \leq k$ for some $k \geq 2$. We can take $X^{k+1} = X^2X^{k-1}$ due to associativity of nested composition. Then, by Lemma 7.46, we have

$$\begin{aligned} b_{X^2X^{k-1}} \leq \max\Bigg(&b_{X^{k-1}+X^{k-1}}, \frac{n_{X^{k-1}}n_{X^2} - 1 + d_{X^{k-1}}}{1 + r_{X^{k-1}}}, \\ &\frac{3}{2}n_X + \max\left(\frac{w_{X^{k-1}}}{n_{X^{k-1}}}, \frac{d_{X^{k-1}}}{r_{X^{k-1}}}\right)\Bigg). \end{aligned} \tag{7.103}$$

We need to show that none of these terms is greater than $n_X^3 + \frac{d_X}{r_X} + (k-1)\frac{w_X}{n_X}$.

For the first term, we have

$$\begin{aligned} b_{X^{k-1}+X^{k-1}} &\leq b_{X^k} \\ &\leq n_X^3 + \frac{d_X}{r_X} + (k-2)\frac{w_X}{n_X} \\ &\leq n_X^3 + \frac{d_X}{r_X} + (k-1)\frac{w_X}{n_X} \end{aligned} \tag{7.104}$$

by the induction hypothesis.

Because $X \neq E$ we have $s_{X^{k-1}} > s_{X^{k-2}}$ and we can therefore conclude that $r_{X^{k-1}} > n_{X^{k-2}} = n_X^{k-2}$. Combining this with Lemma 7.56 gives

$$\frac{n_{X^{k-1}} n_{X^2} - 1 + d_{X^{k-1}}}{1 + r_{X^{k-1}}} \leq \frac{n_X^{k+1}}{n_X^{k-2}} + \frac{d_X}{r_X} + (k-1)\frac{w_X}{n_X} \tag{7.105}$$

for the second term.

Finally, for the third term, we have

$$\frac{3}{2} n_X + \max\left(\frac{w_{X^{k-1}}}{n_{X^{k-1}}}, \frac{d_{X^{k-1}}}{r_{X^{k-1}}}\right) \leq n_X^3 + \frac{d_X}{r_X} + (k-1)\frac{w_X}{n_X}. \tag{7.106}$$

by Lemma 7.56 and the fact that $n_X \geq 2$. □

Proof of Lemma 7.58. The proof is by induction on k. In the base case, $k = 2$, we apply Lemma 7.47:

$$\begin{aligned} b_{{}_\sigma B_1^2} &= b_{{}_\sigma B_1 + {}_\sigma B_1} \\ &= \max\left(\{b_{{}_\sigma B_1}\} \middle\| \frac{\{j+1\}_{j=1}^{\sigma} - 1 + (\{0\} \| \{0\}^\sigma)}{1 + (\{0\} \| \{j\}_{j=1}^{\sigma}}\right) \\ &= 1 \\ &= \frac{(k-1)\sigma + 1}{\sigma + 1}. \end{aligned} \tag{7.107}$$

For ease of notation, let $\gamma = \sigma + 1$. By Lemma 7.54 we have

$$N_{{}_\sigma B_1^k} = \left\| \prod_{i=0}^{k-1} \gamma^i \{j+1\}_{j=1}^{\sigma} \right. \tag{7.108}$$

$$R_{{}_\sigma B_1^k} = \left\| \prod_{i=0}^{k-1} \left(\gamma^i \{j+1\}_{j=1}^{\sigma} - 1\right) \right. \tag{7.109}$$

$$D_{{}_\sigma B_2^k} = \left\| \prod_{i=0}^{k-1} \left(i\gamma^{i-1}\sigma \{j+1\}_{j=1}^{\sigma} - \gamma^i + 1\right) \right. . \tag{7.110}$$

For the induction hypothesis, assume the lemma holds for some k. Then, for $b_{\sigma B_1^{k+1}}$, we have

$$\begin{aligned} b_{\sigma B_1^{k+1}} &= b_{\sigma B_1^k + \sigma B_1^k} \\ &= \max\left(\left\{ b_{\sigma B_1^k}, \frac{2\gamma^{k-1} - 1 + (k-2)\gamma^{k-2}\sigma - \gamma^{k-1} + 1}{\gamma^{k-1}} \right\} \middle\| \right. \\ &\qquad \left. \frac{\gamma^{k-1}\{j+2\}_{j=1}^{\sigma-1} - 1 + (k-1)\gamma^{k-2}\sigma\{j+1\}_{j=1}^{\sigma} - \gamma^{k-1} + 1}{} \right) \\ &= \max\left(\frac{(k-1)\sigma + 1}{\sigma + 1}, \frac{k\sigma + 1}{\sigma + 1} \right) \\ &= \frac{k\sigma + 1}{\sigma + 1}. \end{aligned} \tag{7.111}$$

□

Bibliography

[Abe99] Masayuki Abe. Mix-networks on permutation networks. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings*, volume 1716 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 1999.

[ABSdV19] Mark Abspoel, Niek J. Bouman, Berry Schoenmakers, and Niels de Vreede. Fast secure comparison for medium-sized integers and its application in binarized neural networks. In Mitsuru Matsui, editor, *Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings*, volume 11405 of *Lecture Notes in Computer Science*, pages 453–472. Springer, 2019.

[ACG+14] Prabhanjan Ananth, Nishanth Chandran, Vipul Goyal, Bhavana Kanukurthi, and Rafail Ostrovsky. Achieving privacy in verifiable computation with multiple servers - without FHE and without pre-processing. In Hugo Krawczyk, editor, *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, volume 8383 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2014.

[AJCC15] James Alderman, Christian Janson, Carlos Cid, and Jason Crampton. Access control in publicly verifiable outsourced computation. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information,*

Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015, pages 657–662. ACM, 2015.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c \log n$ parallel sets. *Comb.*, 3(1):1–19, 1983.

[Alb38] Abraham A. Albert. Symmetric and alternate matrices in an arbitrary field, I. *Transactions of the American Mathematical Society*, 43(3):386–436, 1938.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 201–209. ACM, 1989.

[BBSdV19] Frank Blom, Niek J. Bouman, Berry Schoenmakers, and Niels de Vreede. Efficient secure ridge regression from randomized Gaussian elimination. *IACR Cryptol. ePrint Arch.*, 2019:773, 2019.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2014.

[BdV18] Niek J. Bouman and Niels de Vreede. New protocols for secure linear algebra: Pivoting-free elimination and fast block-recursive matrix decomposition. *IACR Cryptol. ePrint Arch.*, 2018:703, 2018.

[BdV19] Niek J. Bouman and Niels de Vreede. A practical approach to the secure computation of the moore-penrose pseudoinverse over the rationals. *IACR Cryptol. ePrint Arch.*, 2019:470, 2019.

[BdV20] Niek J. Bouman and Niels de Vreede. A practical approach to the secure computation of the moore-penrose pseudoinverse over the rationals. In *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020. Proceedings*, 2020.

[Ben92] Adi Ben-Israel. A volume associated with $m \times n$ matrices. *Linear Algebra and its Applications*, 167:87–111, 1992.

[BG03] Adi Ben-Israel and Thomas N.E. Greville. *Generalized Inverses - Theory and Applications*. CMS Books in Mathematics. Springer-Verlag, New York, 2003.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BKLS18] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: A tool for cryptographically secure statistical analysis. *IEEE Trans. Dependable Secur. Comput.*, 15(3):481–495, 2018.

[BLT14] Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, volume 8788 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2014.

[BO71] Thomas L. Boullion and Patrick L. Odell. *Generalized Inverse Matrices*. Wiley, New York, 1971.

[BP08] Joan Boyar and René Peralta. Tight bounds for the multiplicative complexity of symmetric functions. *Theor. Comput. Sci.*, 396(1-3):223–246, 2008.

[BRP90] R.B. Bapat, K.P.S. Bhaskara Rao, and K. Manjunatha Prasad. Generalized inverses over integral domains. *Linear Algebra and its Applications*, 140:181–196, 1990.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[CD01] Ronald Cramer and Ivan Damgård. Secure distributed linear algebra in a constant number of rounds. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure multiparty computation and secret sharing: An information theoretic approach.* Cambridge University Press, 2015.

[CEK+02] Li Chen, Wayne Eberly, Erich Kaltofen, B. David Saunders, William J. Turner, and Gilles Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra Its Appl.*, 343–344:119–146, 2002.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 253–270. IEEE Computer Society, 2015.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[CJ02] Don Coppersmith and Markus Jakobsson. Almost optimal hash sequence traversal. In Matt Blaze, editor, *Financial Cryptography, 6th International Conference, FC 2002, Southampton, Bermuda, March 11-14, 2002, Revised Papers*, volume 2357 of *Lecture Notes in Computer Science*, pages 102–119. Springer, 2002.

[CKKC13] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. Multi-client non-interactive verifiable computation. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 499–518. Springer, 2013.

[CKP07] Ronald Cramer, Eike Kiltz, and Carles Padró. A note on secure computation of the moore-penrose pseudoinverse and its application to secure linear algebra. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 613–630. Springer, 2007.

[CLT14] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr, editors, *Proceedings of the 30th Annual Computer Security*

Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014, pages 266–275. ACM, 2014.

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster computing in zero knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 371–403. Springer, 2015.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multiparty computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[dH12] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming.* PhD thesis, Eindhoven University of Technology, 2012.

[dHSV16] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, volume 9646 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2016.

[DN02] Ivan Damgård and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2002.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[EK97] Wayne Eberly and Erich Kaltofen. On randomized lanczos algorithms. In Bruce W. Char, Paul S. Wang, and Wolfgang Küchlin, editors, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, ISSAC '97, Maui, Hawaii, USA, July 21-23, 1997*, pages 176–183. ACM, 1997.

[FGP14] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 844–855. ACM, 2014.

[FS01] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer, 2001.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic*

Techniques, Athens, Greece, May 26-30, 2013. Proceedings, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GKL+15] S. Dov Gordon, Jonathan Katz, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Multi-client verifiable computation with stronger security guarantees. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, volume 9015 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2015.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Goo11] Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, 2011.

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 684–693. ACM, 2014.

[GPRS96] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C.H. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation Techniques, Applications, and Tools*, pages 95–106, Philadelphia, 1996. SIAM.

[Gre66] Thomas N.E. Greville. Note on the generalized inverse of a matrix product. *SIAM Review*, 8(4):518–521, 1966.

[Gri92] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111. ACM, 1998.

[GSB+17] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *PoPETs*, 2017(4):345–364, 2017.

[Har86] Robert E. Hartwig. The reverse order law revisited. *Linear Algebra and its Applications*, 76:241–246, 1986.

[HICT14] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. *IACR Cryptol. ePrint Arch.*, 2014:121, 2014.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.

[IR01] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354. Springer, 2001.

[Jak02] Markus Jakobsson. Fractal hash sequence representation and traversal. In *Proc. IEEE International Symposium on Information Theory (ISIT '02)*, page 437. IEEE, 2002. Full version eprint.iacr.org/2002/001.

[JNO14] Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A framework for outsourcing of secure computation. In Gail-Joon Ahn, Alina Oprea, and Reihaneh Safavi-Naini, editors, *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 81–92. ACM, 2014.

[Kim03] Sung-Ryul Kim. Improved scalable hash chain traversal. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings*, volume 2846 of *Lecture Notes in Computer Science*, pages 86–95. Springer, 2003.

[KL96] Erich Kaltofen and Austin Lobo. On rank properties of Toeplitz matrices over finite fields. In Erwin Engeler, B. F. Caviness, and Yagati N. Lakshman, editors, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, ISSAC '96, Zurich, Switzerland, July 24-26, 1996*, pages 241–249. ACM, 1996.

[KMR12] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 797–808. ACM, 2012.

[KMWF07] Eike Kiltz, Payman Mohassel, Enav Weinreb, and Matthew K. Franklin. Secure linear algebra using linearly recurrent sequences. In Salil P. Vadhan, editor, *Theory of Cryptography, 4th Theory*

of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings, volume 4392 of *Lecture Notes in Computer Science*, pages 291–310. Springer, 2007.

[KS91] Erich Kaltofen and B. David Saunders. On Wiedemann's method of solving sparse linear systems. In Harold F. Mattson, Teo Mora, and T. R. N. Rao, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 9th International Symposium, AAECC-9, New Orleans, LA, USA, October 7-11, 1991, Proceedings*, volume 539 of *Lecture Notes in Computer Science*, pages 29–38. Springer, 1991.

[Lam81] Leslie Lamport. Password authentification with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[LP98] Frank Thomson Leighton and C. Greg Plaxton. Hypercubic sorting networks. *SIAM J. Comput.*, 27(1):1–47, 1998.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2011.

[Mal10] Gennadi I. Malaschonok. Fast generalized bruhat decomposition. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 12th International Workshop, CASC 2010, Tsakhkadzor, Armenia, September 6-12, 2010. Proceedings*, volume 6244 of *Lecture Notes in Computer Science*, pages 194–202. Springer, 2010.

[MF06a] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[MF06b] Payman Mohassel and Matthew K. Franklin. Efficient polynomial operations in the shared-coefficients setting. In Moti Yung,

Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2006.

[MS74a] George Marsaglia and George P.H. Styan. Equalities and inequalities for ranks of matrices. *Linear and Multilinear Algebra*, 2(3):269–292, 1974.

[MS74b] George Marsaglia and George P.H. Styan. Rank conditions for generalized inverses of partitioned matrices. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 437–442, 1974.

[Mul87] Ketan Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Comb.*, 7(1):101–104, 1987.

[MW08] Payman Mohassel and Enav Weinreb. Efficient secure linear algebra in the presence of covert or computationally unbounded adversaries. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 2008.

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001*, pages 116–125. ACM, 2001.

[NO07] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.

[NW06] Kobbi Nissim and Enav Weinreb. Communication efficient secure linear algebra. In Shai Halevi and Tal Rabin, editors, *Theory*

of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings, volume 3876 of *Lecture Notes in Computer Science*, pages 522–541. Springer, 2006.

[NWI+13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 334–348. IEEE Computer Society, 2013.

[Pan66] Victor Ya. Pan. Methods of computing values of polynomials. *Russian Mathematical Surveys*, 21(1):105, 1966.

[Pea68] Martin H. Pearl. Generalized inverses of matrices with entries taken from an arbitrary field. *Linear Algebra and its Applications*, 1(4):571–587, 1968.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252. IEEE Computer Society, 2013.

[PS73] Mike Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[PTK13] Andreas Peter, Erik Tews, and Stefan Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Trans. Inf. Forensics Secur.*, 8(12):2046–2058, 2013.

[Rao73] C. Radhakrishna Rao. *Linear Statistical Inference and its Applications*. Wiley, New York, 1973.

[RM71] C. Radhakrishna Rao and Sujit Kumar Mitra. *Generalized inverse of matrices and its applications*. Wiley, New York, 1971.

[Roh65] Charles A. Rohde. Generalized inverses of partitioned matrices. *Journal of the Society for Industrial and Applied Mathematics*, 13(4):1033–1035, 1965.

[Sch88] Claus-Peter Schnorr. The multiplicative complexity of Boolean functions. In Teo Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 6th International Conference, AAECC-6, Rome, Italy, July 4-8, 1988, Proceedings*, volume 357 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 1988.

[Sch16] Berry Schoenmakers. Explicit optimal binary pebbling for one-way hash chain reversal. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*, volume 9603 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2016.

[Sch17] Berry Schoenmakers. Binary pebbling algorithms for in-place reversal of one-way hash chains. *Nieuw Archief voor Wiskunde serie 5*, 18(3):199–204, 2017.

[Sel03] Yaron Sella. On the computation-storage trade-offs of hash chain traversal. In Rebecca N. Wright, editor, *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*, volume 2742 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2003.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SK95] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme - A practical solution to the implementation of a voting booth. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *Advances in Cryptology - EUROCRYPT '95, International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 21-25, 1995, Proceeding*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 1995.

[Spr83] J. Springer. Die exakte Berechnung der Moore–Penrose-Inversen einer Matrix durch Residuenarithmetik. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 63(3):203–210, 1983.

[ST06] Berry Schoenmakers and Pim Tuyls. Efficient binary conversion for paillier encrypted values. In Serge Vaudenay, editor, *Advances*

in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings, volume 4004 of *Lecture Notes in Computer Science*, pages 522–537. Springer, 2006.

[SV15] Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[SVdV15] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-friendly outsourcing by distributed verifiable computation. *IACR Cryptol. ePrint Arch.*, 2015:480, 2015.

[SVdV16] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve A. Schneider, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, volume 9696 of *Lecture Notes in Computer Science*, pages 346–366. Springer, 2016.

[vzGG03] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra (2. ed.)*. Cambridge University Press, 2003.

[Wan81] Paul S. Wang. A p-adic algorithm for univariate partial fractions. In Paul S. Wang, editor, *SYMSAC 1981, Proceedings of the Symposium on Symbolic and Algebraic Manipulation, Snowbird, Utah, USA, August 5-7, 1981*, pages 212–217. ACM, 1981.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations*

of Computer Science, Toronto, Canada, 27-29 October 1986, pages 162–167. IEEE Computer Society, 1986.

[YSEL09] Dae Hyun Yum, Jae Woo Seo, Sungwook Eom, and Pil Joong Lee. Single-layer fractal hash chain traversal with almost optimal complexity. In Marc Fischlin, editor, *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, volume 5473 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2009.

[YZ10] Andrew Chi-Chih Yao and Yunlei Zhao. Deniable internet key exchange. In Jianying Zhou and Moti Yung, editors, *Applied Cryptography and Network Security, 8th International Conference, ACNS 2010, Beijing, China, June 22-25, 2010. Proceedings*, volume 6123 of *Lecture Notes in Computer Science*, pages 329–348, 2010.

Summary

Assorted Algorithms and Protocols for Secure Computation

Secure multiparty computation allows multiple mutually distrusting parties holding private inputs to securely compute a function on their joint inputs without revealing any private information. This thesis covers a variety of topics related to secure protocols with a particular focus on protocols involving secure multiparty computation.

Chapter 2 presents a study of the multiplicative complexity of polynomial evaluation and its generalization to the d-linear complexity. From these results, lower bounds for the secure evaluation of symmetric functions using secure arithmetic are derived. The results can also be applied directly to efficiently implement secure functions with small domains.

Chapter 3 introduces novel protocols for securely sampling random permutations and obliviously manipulating permutations or applying permutations to input lists of secret-shared values. Such protocols are a useful building block for more complex protocols for secure computation. One particular example application combining techniques from Chapters 2 and 3, oblivious array indexing, is presented in detail. The final section of Chapter 3 is dedicated to using the secure random permutation protocol to deal and play out online card games and illustrates how the protocols described in this chapter can be modified to provide additional security properties, or optimized under certain assumptions.

Chapter 4 is dedicated to secure sorting, which is another application made possible by the secure random permutation protocol. This chapter introduces a multi-pivot variant of quicksort, which is particularly suited for practically efficient application in secure computation when used in conjunction with the aforementioned protocols for oblivious permutation.

Chapter 5 introduces novel secure protocols for the computation of the

Moore-Penrose pseudoinverse. In particular, the protocols are more practically efficient than existing protocols. Furthermore, the chapter deals with the conditions under which the pseudoinverse computed over a finite field, as is typical for secure computation, corresponds to the pseudoinverse over the rational numbers, which is required for most real-world applications.

Chapter 6 presents novel secure protocols for outsourced computation, in which computational tasks can be outsourced to computation providers in such a way that both data privacy and correctness of computation are provided. This is achieved using a combination of secure multiparty computation techniques and results in verifiable computation.

Chapter 7 introduces a novel framework for hash chain reversal. Hash chains find practical application in authentication protocols for resource-constrained devices. The generality of the framework is demonstrated by providing a uniform description of previously known results. Furthermore, using the framework a long-standing open question regarding achievability of a lower bound is answered in the affirmative.

Curriculum Vitæ

Niels de Vreede was born on 17 August 1986 in Helmond, the Netherlands. After finishing his VWO degree in 2004 at the St.-Willibrord Gymnasium in Deurne, the Netherlands, he started his studies in Technische Natuurkunde and Toegepaste Wiskunde at the Technische Universiteit Eindhoven, from which he obtained a Bachelor's degree in Technische Natuurkunde in 2007. Thereafter, he pursued a Master's degree in Computer Science and Engineering, graduating cum laude in 2013 within the Security group of the Department of Mathematics and Computer Science at Technische Universiteit Eindhoven on Privacy-preserving Biometric Databases: from Authentication to Efficient Identification Systems. In the same year, he started a PhD project at Technische Universiteit Eindhoven under the supervision of dr. ir. Berry Schoenmakers, of which the results are presented in this dissertation.