Cryptographic Protocols: Theory and Implementation

Martin Geisler

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Cryptographic Protocols: Theory and Implementation

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Martin Geisler
February 26, 2010

Abstract

The art of keeping messages secret is ancient. It must have been invented only shortly after the invention of the messages themselves. Merchants and generals have always had a need to exchange critical messages while keeping them secret from the prying eyes of competitors or the enemy. Classical cryptography was thus concerned with message confidentiality and integrity. Modern cryptography cover a much wider range of subjects including the area of secure multiparty computation, which will be the main topic of this dissertation.

Our first contribution is a new protocol for secure comparison, presented in Chapter 2. Comparisons play a key role in many systems such as online auctions and benchmarks — it is not unreasonable to say that when parties come together for a multiparty computation, it is because they want to make decisions that depend on private information. Decisions depend on comparisons. We have implemented the comparison protocol in Java and benchmarks show that is it highly competitive and practical.

The biggest contribution of this dissertation is a general framework for secure multiparty computation. Instead of making new ad hoc implementations for each protocol, we want a single and extensible framework. We call this framework VIFF, short for *Virtual Ideal Functionality Framework*.

VIFF implements a UC functionality for general multiparty computation on asynchronous networks. We give a formal definition of the functionality in Chapter 3. There we also describe how we implemented the functionality with a variant of the classic BGW protocol. The protocol is secure against a semi-honest adversary.

In Chapter 4 we describe a new protocol for VIFF that is secure against malicious adversaries. The protocol guarantees termination if the adversary allows a preprocessing phase to terminate, in which no information is released. The communication complexity of this protocol is the same as that of a passively secure solution up to a constant factor. It is secure against an adaptive and active adversary corrupting less than $n/3$ players.

Following the presentation of VIFF, we turn to a more theoretical subject. Chapter 5 investigates the notion of a covert adversary — an adversary type that intuitively lies in between semi-honest and malicious adversaries. The main idea is that we accept that a cheating adversary may succeed with a given probability, which need *not* be negligible. The reasoning is that in many real-world cases, a large probability of being caught is sufficient to prevent the adversary from trying to cheat. We show how to compile a passively secure protocol for honest majority into a protocol that is secure against covert attacks, again for honest majority. The transformed protocol catches cheating with probability $\frac{1}{4}$. Though we present no implementation of this compiler, we believe it will be very efficient and practical to implement using, say, VIFF. The cost of the modified protocol is essentially twice that of the original plus an overhead that only depends on the number of inputs.

We round off this dissertation with Chapter 6. There we return to the practical side of things and consider how users of online collaboration tools and network storage services place considerable trust in their providers. We presents a novel approach for protecting data integrity in revision control systems hosted by an untrusted provider. It guarantees atomic read and write operations on the shared data when the service is correct and preserves fork-linearizability when the service is faulty. A prototype has been implemented on top of the Subversion revision control system; benchmarks show that the approach is practical.

Acknowledgments

I would like to thank Ivan Damgård for being my advisor during these four years. Thank you for the support and encouragement and for letting me try out my own ideas.

I am very grateful to all my colleagues at DAIMI: Rikke Bendlin, Dan Lund Christensen, Matthias Fitzi, Jakob Funder, Thomas Jakobsen, Marcel Keller, Mikkel Krøigård, Jonas Kölker, Carolin Lunemann, Sigurd Meldgaard, Gert Læssøe Mikkelsen, Thomas Mølhave, Janus Dam Nielsen, Jesper Buus Nielsen, Claudio Orlandi, Jakob Pagter, Michael Pedersen, Tord Reistad, Louis Salvail, Christian Schaffner, Michael I. Schwartzbach, Miroslava Sotakova, Asgeir Steine, Rune Thorbek, Tomas Toft, Nikos Triandopoulos, and Sarah Zakarias. I will miss you guys a lot! Special thanks to the MADALGO research group for always having a cold bottle of Coca Cola in their fridge.

Christian Cachin was an excellent host for my stay abroad at the IBM Zurich Research Laboratory. Thanks also to the rest of the security group at IBM, who made sure I had a very pleasant stay.

I am thankful to Lars Geisler, Marcel Keller, Mikkel Krøigård, Gert Læssøe Mikkelsen, Thomas Mølhave, Rune Thorbek and Tomas Toft for their many good comments during proof reading and to Henrik Stuart for sharing his expert knowledge about LaTeX.

Finally and most importantly, I want to thank my wife, Stephanie, for all her love and encouragement over the years. I also want to thank my parents, Marianne and Lars, and my brother, Kristoffer, for their help and support.

Martin Geisler,
Sabro, February 26, 2010.

Contents

Chapter 1

Introduction

The topic of this dissertation is cryptographic protocols, in particular protocols for secure multiparty computation (MPC). The field of multiparty computation was introduced by the following little story:

> *Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?*

This question was put forth by Yao [100] in 1982. He continued to describe a cryptographic protocol that would allow the two millionaires to settle the question. At the end of the decade, secure multiparty computation was an established branch of cryptography and the first completeness papers [10, 24] had shown how any computable function could be computed using secure multiparty computation.

The realization that all functions can be computed without leaking private information is astounding. When the concept of public-key cryptography was introduced in 1976 by Diffie and Hellman [41], they wrote:

> *We stand today on the brink of a revolution in cryptography.*

Although the well-known RSA cryptosystem [88] was introduced two years later, we needed much more infrastructure to make the revolution a reality. It should take some twenty years before the use of public-key cryptography became truly common with the rise of the Internet. Today, many people order books, travels, movie tickets and countless other things in online shops. As part of the transaction, their computers engage in a little protocol based on public-key cryptography to establish a secure key with the online retailer. The use of public-key cryptography is mostly invisible to the common user, which is why it has been able to become so successful.

Establishing a secure key for private communication is an example of a very specific cryptographic protocol. Secure multiparty computation is a much wider research area. It has an even greater potential for changing the way our computers share information when they work together. Multiparty computation is also considerably harder and will take even longer than public-key cryptography to gain a widespread impact. However, compared to

1976, we now do have the necessary infrastructure to run secure multiparty computation for real. This has been demonstrated recently in an online auction [16], which used software written for this dissertation. With the great advances made on both the theoretical and technological side, we believe that we will witness a new cryptographic evolution in the future.

This PhD dissertation documents some of the most recent advances in the design and implementation of these protocols. Its main contribution is the combination of new protocols with implementations of these protocols. As is normal practice in computer science, the first years of secure multiparty computation can best be described as a mathematical exercise. The first completeness papers [10, 24] show how any function computable by a Turing machine can be computed using secure multiparty computation. However, they did not back this with an actual implementation. The software presented in this dissertation tries to turn some of the theory into practice.

We will begin by describing the structure of this dissertation and followed by a review of the basic notions used in secure multiparty computation. We will assume the reader has a basic knowledge about cryptography and cryptographic protocols. We will nevertheless give a basic introduction of the security model used below.

1.1 Outline

The exploration of secure multiparty computation begin with the design and implementation of a new protocol for secure comparison of integers. This work is presented in Chapter 2. The chapter is based on the following papers:

[33] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *ACISP*, volume 4586 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2007

[34] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1 (1):22–31, 2008

[36] I. Damgård, M. Geisler, and M. Krøigaard. A correction to ‘Efficient and Secure Comparison for On-Line Auctions’. *IJACT*, 1(4):323–324, 2009

Following that, we introduce the secure multiparty computation framework called VIFF. This is Chapter 3. The material in that chapter is partly new and party from these sources:

[47] M. Geisler. VIFF: Virtual ideal functionality framework, 2007. Homepage: `http://viff.dk/`

[48] M. Geisler. *Implementing Asynchronous Multi-Party Computation.* PhD progress report, University of Aarhus, Denmark, Jan. 2008

[50] M. Geisler, I. Damgård, and B. Pinkas. MPC virtual machine specification. Technical Report D4.3, CACE: Computer Aided Cryptography Engineering, 2009

Using VIFF, we were able to implement and benchmark a new protocol for asynchronous multiparty computation. This the content of Chapter 4, which was also presented as the paper:

[37] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009

Chapter 5 investigates how one can obtain highly efficient and practical protocols, if one is willing to settle for security against covert adversaries. The chapter was published as:

[39] I. Damgård, M. Geisler, and J. B. Nielsen. From passive to covert security at low cost. In D. Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010

Returning to the issue about implementing cryptographic protocols, Chapter 6 describes an implementation of a revision control system with extra integrity guarantees. This was previously published as:

[19] C. Cachin and M. Geisler. Integrity protection for revision control. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 382–399, 2009

Finally, Chapter 7 summarizes the findings of this dissertation. In addition to the papers mentioned above, the author of this dissertation has made a minor contribution to the paper:

[49] M. Geisler and N. P. Smart. Distributing the key distribution centre in Sakai-Kasahara based systems. In M. G. Parker, editor, *IMA International Conference*, volume 5921 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 2009

The paper describes how MPC can be used to de-centralize a key distribution center. Using VIFF, we were able to quickly benchmark the core operation performed by the key distribution center. The benchmark results confirmed the practicality of the suggested protocol.

Before diving into protocol design in the next chapter, we will introduce the notation used in this dissertation and the security framework we will work with in the majority of the chapters.

1.2 Notation

We will denote the participants in our protocols by $P_1, \dots, P_n$ for a total of n parties. We let t denote the corruption threshold up to which the protocol remains secure. The adversary, who tries to violate the privacy or correctness of the protocol, is denoted by $\mathcal{A}$.

The protocol inputs are elements from some field $\mathbb{F}$. Shamir secret sharing [92] of $x \in \mathbb{F}$ with threshold/degree t results in a set of shares denoted by $[x]_t$ or simply $[x]$ when the threshold is clear from the context. We use the notation $[x] + a[y]$ where a is a public constant to denote the set of shares obtained by locally adding the share of x to the share of y times a. Since Shamir sharing is linear, we have $[x] + a[y] = [x + ay]$.

1.3 The Universally Composable Security Framework

Proving that a cryptographic protocol is secure is a hard problem. First we must define rigorously what "secure" means, and then we must prove that the protocol lives up to the definition. The Universally Composable (UC) security framework by Canetti [23] helps solve these problems. We will now give a brief overview of the UC framework we will be using repeatedly in the rest of this dissertation.

1.3.1 Security Definition

We begin with an informal statement of what it means to be secure:

Definition 1.1 A protocol is *secure* if an outside observer cannot distinguish between an execution of the real protocol and a known-secure replacement protocol. ♦

Even though this definition is very short and slightly recursive, it turns out to be useful. It gives a testable condition of when a protocol is secure: you take a protocol which is known to be secure and prove that the new protocol is indistinguishable from the secure protocol. At the hearth of this definition lies an assumption that security is transitive — if a protocol π looks very much like a protocol ρ, and if ρ is (somehow) known to be secure, then π is taken to be secure as well.

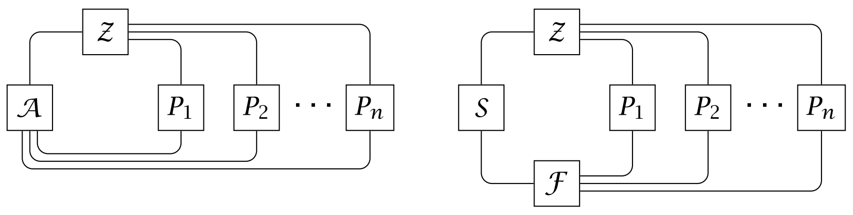
To bootstrap this process we use a trick and compare our protocol with a protocol which we *define* to be secure. This ideal protocol uses an *ideal functionality* to do the computation. This is an interactive Turing machine, $\mathcal{F}$, which cannot be corrupted and always calculates the correct result. The ideal protocol is simple: everybody starts by handing their inputs over to $\mathcal{F}$, which spends some time calculating and finally gives everybody their correct result. Clearly no information is leaked and it makes good sense to define this as "secure". Since the parties do nothing in the ideal protocol we will call them *dummy parties*. To provide maximum generality, we will allow the observer to specify the inputs to the dummy parties. We will call the observer the *environment* and denote it by $\mathcal{Z}$ from now on to match the standard UC terminology.

The real protocol is not so simple and contains actual parties which follow some protocol π without having an ideal functionality to help them.

An additional entity is the *adversary*, $\mathcal{A}$. By default the adversary listens to all communication between the parties and observes the internal state of corrupted players (a *passive* or *semi-honest* adversary) but may also be allowed to take full control over a party and change messages arbitrarily (an *active* or *malicious* adversary). The adversary can talk with the environment both to provide details of what it sees in the protocol, but also to receive instructions on what to do next. Figure 1.1a shows this situation with no corruptions. Note that the parties cannot communicate directly with each other — all communication between parties must pass through the adversary. Unless encryption is used, $\mathcal{A}$ can thus read all messages.

In a synchronous network, the adversary is forbidden from interfering with the network traffic. This means that $\mathcal{A}$ must deliver all messages received in round i at the beginning of round $i+1$. To model an asynchronous network, $\mathcal{A}$ is allowed to delay and reorder the delivery of the messages sent between the parties in the protocol arbitrarily, even when semi-honest. We only require that messages are eventually delivered and that they are delivered unmodified. This models a typical asynchronous network with a reliable transport layer.

The goal of $\mathcal{Z}$ is to distinguish the situation in Figure 1.1a from the ideal protocol execution. Since $\mathcal{Z}$ expects to talk to the adversary, we include an extra party called a simulator, $\mathcal{S}$, in the ideal protocol experiment shown in Figure 1.1b. The job of the simulator is to pretend to be the adversary towards $\mathcal{Z}$. To do this the simulator gets help from $\mathcal{F}$ — the precise amount of help allowed is part of the description of $\mathcal{F}$. Since the behavior of $\mathcal{F}$ is our definition of security, we do not want the help to leak private information. This means that $\mathcal{F}$ will typically send messages of the form $(x := ?)$ which only tells $\mathcal{S}$ that x now has a value, but does not tell it which value.



(a) Real world.

(b) Ideal world.

Figure 1.1: The two central protocol experiments in the UC framework. Note how the two figures looks exactly the same from the perspective of $\mathcal{Z}$ who, in both cases, interacts with $n+1$ parties over a network.

Having described the two protocol experiments and their participants, we now define a random variable $\textsc{exec}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ to be the output of $\mathcal{Z}$ in the ideal world execution with the ideal functionality $\mathcal{F}$ and simulator $\mathcal{S}$. Define $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}$ to be the output of $\mathcal{Z}$ in the real world execution of π under attack by $\mathcal{A}$. Let k be the *security parameter*. We can then refine Definition 1.1 slightly as follows:

Definition 1.2 (UC Security) A protocol π is *secure* with regard to an ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{Z}$ the statistical difference between $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ is negligible in k. ♦

We say that π UC-realizes $\mathcal{F}$.

A more formal definition can be found in the full UC paper [23], but this definition should capture the gist of what it means to be UC-secure. Let us just here remark that if the environment has unbounded computational power, one talks about *perfect* security when the two distributions are equal, and *statistical* security otherwise. If $\mathcal{Z}$ is limited to polynomial computations one obtains *computational* security.

1.3.2 Simulation

To prove that a protocol is secure, we must describe how the simulator fools the environment. It must produce output distributed in the same way as $\mathcal{A}$ would in the real world, otherwise $\mathcal{Z}$ would notice and the protocol would not be secure. Figure 1.2 shows how the simulator will typically run a copy of $\mathcal{A}$ inside and feed it input as it gets messages from $\mathcal{F}$. Based on how $\mathcal{A}$ reacts to the inputs, the simulator can decide what to send to $\mathcal{Z}$ and $\mathcal{F}$.

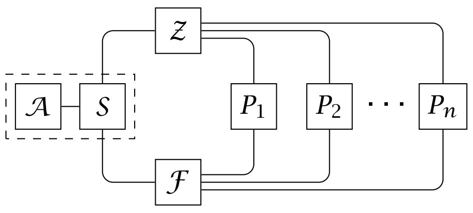


Figure 1.2: The ideal world with the adversary run by the simulator. Giving $\mathcal{A}$ inputs distributed as in the real world ensures that the output of $\mathcal{A}$ is correctly distributed in the ideal world.

1.3.3 Composability

An important benefit of the UC framework is *composability*: UC-secure protocols can be composed in arbitrary ways to build larger protocols, which remain UC-secure.

Protocols are composed via the definition of a *hybrid world*. This protocol experiment is like the real world, but the protocol participants can additionally make calls to an ideal functionality. Let ρ be a protocol that UC-realizes a functionality $\mathcal{F}$ and let π be a hybrid protocol that makes calls to $\mathcal{F}$. We say that π is a $\mathcal{F}$-hybrid protocol. Let π^{ρ} denote the real world protocol where we have replaced each invocation of $\mathcal{F}$ in π with an execution of ρ. The composition theorem in [23] says that π^{ρ} realizes $\mathcal{G}$ in the real world when π realizes $\mathcal{G}$ in the hybrid world.

UC functionalities have been defined for standard tasks such as secure message transmission, public-key cryptography, secret sharing, etc.

Chapter 2

Homomorphic Encryption and Secure Comparison

We begin our treatment of cryptographic protocols by looking at a simple problem in where two parties wish to conduct an online auction. Auctions requires comparison of bids to determine the winner — we propose an efficient and practical protocol for comparison of private integers. This chapter also introduces a homomorphic cryptosystem needed by the protocol. The system is designed to be more efficient than previous solutions and can also be used as the basis of efficient and general secure multiparty computation.

The underlying theme of this thesis is the investigating of protocol design and protocol implementations. We have therefore implemented the comparison protocol in Java, in order to demonstrate its practicality. For comparison of 16-bit numbers with security based on 1,024-bit RSA (executed by two parties), our implementation takes 0.28 seconds including all computation and communication. Using precomputation, we estimate that one can save a factor of roughly 10.

The content of this chapter was first published in a preliminary form at the ACISP 2007 conference [33]. A full version appeared in IJACT [34] the following year. Unfortunately, the cryptosystem was flawed. The correction [36] is included in the following.

2.1 Introduction

Consider the following setting: Two or more players are given integers n_A and n_B, where one or both are private, i.e., not known to all players. We then want to decide whether $n_A \geq n_B$, while making sure that the comparison result is the only new information that becomes known. Many variants of this problem exist, depending on whether n_A, n_B are known to particular players, or unknown to everyone. It may even be the case that the comparison result is not supposed to be public, but is to be produced in encrypted form, for instance.

Secure comparison is a special case of general secure computation where all players hold private inputs and want to compute some agreed function of these inputs. Comparison protocols are very important ingredients in many potential applications of secure computation. Examples of this include auctions, benchmarking, and secure extraction of statistical data from databases.

As a concrete example to illustrate the application of the results from this chapter, we take a closer look at online auctions: many online auction systems offer as a service to their customers that one can submit a maximum bid to the system. It is then not necessary to be online permanently. The system will instead automatically bid for you, until you win the auction or your specified maximum is exceeded. We assume in the following what we believe is a realistic scenario, namely that the auction system needs to handle bidders that bid online manually, as well as others that use the option of submitting a maximum bid.

Clearly, such a maximum bid is confidential information. Both the auction company and other participants in the auction have an interest in knowing such maximum bids in advance, and could exploit such knowledge to their advantage. The auction company could force higher prices (what is known as "shill bidding") and thereby increase its income and other bidders might learn how valuable a given item is to others and change their strategy accordingly.

In a situation where anyone can place a bid by just connecting to a web site, the security one can obtain by storing the maximum bids with a single trusted party is questionable, in particular if that trusted party is the auction company. Indeed, there are cases known from real auctions where an auction company has been accused of misusing its knowledge of maximum bids [89].

An obvious solution is to share the responsibility of storing the critical data among several parties, and do the required operations via secure computation. One can then make sure that, unless all parties are corrupted, every time the bid goes up, it will become known whether a given player is still in the game because his maximum is larger then the current price, but the actual value of the maximum will remain secret. To keep the communication pattern simple and to minimize problems with maintenance and other logistical problems, it seems better to keep the number of involved players small. We therefore consider the following model:

An input client C supplies an ℓ-bit integer m as private input to the computation, which is done by players A and B. Because of our motivating scenario, we require that the input is supplied by sending one message to A, respectively to B, and no further interaction with C is necessary. One may, for instance, think of A as the auction house and B as an accounting company. We will also refer to these as the *server* and *assisting server*.

An integer x (which we think of as the currently highest bid) is public input. As public output, we want to compute one bit that is 1 if $m > x$ and 0 otherwise, i.e., the output tells us if C is still in the game and wants to raise the bid, say by some fixed amount agreed in advance. Of course, we want to do the computation securely so that neither A nor B learns any information on m other than the comparison result.

We will assume that players are honest but curious. We believe this is quite a reasonable assumption in our scenario: C may submit incorrectly formed input, but since the protocol handles even malformed input deterministically, he cannot gain anything from this: any malformed bid will determine a number x_0 such that when the current price reaches x_0, the protocol output will cause C to leave the game. So this is equivalent to submitting x_0 in correct format. Moreover, the actions of A and B can be checked after the auction is over — if C notices that incorrect decisions were taken, he can prove that his bid was not correctly handled. Such "public disgrace" is likely to be enough to discourage cheating in our scenario. Nevertheless, we sketch later in the chapter how to obtain active security at moderate extra cost.

2.1.1 Our Contribution

We first propose a new homomorphic cryptosystem that is well suited for our application, this is the topic of Section 2.2. The cryptosystem is much more efficient than, e.g., the encryption scheme by Paillier [81] in terms of en- and decryption time. The efficiency is obtained partly by using a variant of Groth's idea of exploiting subgroups of $\mathbb{Z}_n^*$ for an RSA modulus n [57], and partly by aiming for a rather small plaintext space, of size $\theta(\ell)$.

Later in the chapter, we show how the size of the plaintext space can be extended to allow for more general applications, how to do threshold decryption, and how to apply these observations to do secure multiparty computation.

In Section 2.3 we propose a comparison protocol in our model described above, based on additive secret sharing and homomorphic encryption. The protocol is a new variant of an idea originating in a paper by Blake and Kolesnikov [11]. Their original idea was also based on homomorphic encryption but required a plaintext space of size exponential in ℓ. Here, we present a new technique allowing us to make do with a smaller plaintext space. This means that the exponentiations we do will be with smaller exponents and this improves efficiency. Also, we save computing time by using additive secret sharing as much as possible instead of homomorphic encryption.

As mentioned, our encryption is based on a k-bit RSA modulus. In addition there is an "information theoretic" security parameter t involved which is approximately the logarithm of the size of the subgroup of $\mathbb{Z}_n^*$ we use. Here, t needs to be large enough so that exhaustive search for the order of the subgroup and other generic attacks are not feasible. Section 2.4 contains more information about the security of the protocol.

In the protocol, C sends a single message to A and another to B, both of size $\mathcal{O}(\ell \log \ell + k)$ bits. To do the comparison, there is one message from A to B and one from B to A. The size of each of these messages is $\mathcal{O}(\ell k)$ bits. As for computational complexity, both A and B need to do $\mathcal{O}(\ell(t + \log \ell))$ multiplications mod n. Realistic values of the parameters might be $k = 1024$, $t = 160$, and $\ell = 16$. In this case, counting the actual number of multiplications works out to roughly 7 full scale exponentiations mod n, and takes 0.28 seconds in our implementation, including all computation and communication time. Moreover, most of the work can be done as preprocessing. Using this possibility in the concrete case above, the online work for B is about 0.6 exponentiations for A and 0.06 for B, so that we can expect to save a factor of at least 10 compared to the basic version. It is clear that the online performance of such a protocol is extremely important: auctions often run up a certain deadline, and bidders in practice sometimes play a strategy where they suddenly submit a much larger bid just before the deadline in the hope of taking other bidders by surprise. In such a scenario, one cannot wait a long time for a comparison protocol to finish.

We emphasize that, while it may seem easier to do secure comparison when one of the input numbers is public, we do this variant only because it comes up naturally in our example scenario. In fact, it is straightforward to modify our protocol to fit related scenarios. For instance, the case where A has a private integer a, B has a private integer b and we want to compare a and b, can be handled with essentially the same cost as in our model. Moreover, at the expense of a factor about 2 in the round, communication and computational complexities, our protocol generalizes to handle comparison of two integers that are shared between A and B, i.e., are unknown to both of them. It is also possible to keep the comparison result secret, i.e., produce it in encrypted form. More details on this are given in Section 2.5.

Finally, in Section 2.6 we describe our implementation and the results of a benchmark between our proposed protocol and the one from [44].

2.1.2 Related Work

There is a very large amount of work on secure auctions, which we do not attempt to survey here, as our main concern is secure protocols for comparison, and the online auction is mainly a motivating scenario. One may of course do secure comparison of integers using generic multiparty computation techniques. For the two-party case, the most efficient generic solution is based on Yao-garbled circuits, which were proposed for use in auctions by Naor et al. [78]. Such methods are typically less efficient than ad hoc methods for comparison — although the difference is not very large when considering passive security. For instance, the Yao garbled circuit method requires — in addition to garbling the circuit — that we do an oblivious transfer of a secret key for every bit position of the numbers to compare. This last part is already comparable to the cost of the best known ad hoc methods.

There are several existing ad hoc techniques for comparison. We already mentioned the one from [11] above, a later variant appeared in [12], allowing comparison of two numbers that are unknown to the parties. A completely different technique was proposed earlier by Fischlin [44].

It should be noted that previous protocols typically are for the model where A has a private number a, B has a number b, and we want to compare a and b. Our model is a bit different, as we have one public number that is to be compared to a number that should be known to neither party, and so has to be shared between them. However, the distinction is not very important: previous protocols can quite easily be transformed to our model, and as mentioned above, our protocol can also handle the other models at marginal extra cost. Therefore the comparison of our solution to previous work can safely ignore the choice of model.

Fischlin's protocol is based on the well-known idea of encrypting bits as quadratic residues and non-residues modulo an RSA modulus, and essentially simulates a Boolean formula that computes the result of the comparison. Compared to [11, 12], this saves computing time, since creating such an encryption is much faster than creating a Paillier encryption. However, in order to handle the nonlinear operations required in the formula, Fischlin extends the encryption of each bit into a sequence of λ numbers, where λ is a parameter controlling the probability that the protocol returns an incorrect answer. Since these encryptions have to be communicated, we get a communication complexity of $\Omega(\lambda \ell k)$ bits. The parameter λ should be chosen such that $5\ell \cdot 2^{-\lambda}$ is an acceptable (small enough) error probability, so this makes the communication complexity significantly larger than the $\mathcal{O}(\ell k)$ bits one gets in our protocol and the one from [12].

The computational complexity for Fischlin's protocol is $\mathcal{O}(\ell\lambda)$ modular multiplications, which for typical parameter values is much smaller than that of [11, 12], namely $\mathcal{O}(\ell k)$ multiplications.[1] Fischlin's result is not directly comparable to ours, since our parameter t is of a different nature than Fischlin's λ: t controls the probability that the best known generic attack breaks our encryption scheme, while λ controls the probability that the protocol gives incorrect results. However, if we assume that parameters are chosen to make the two probabilities be roughly equal, then the two computational complexities are asymptotically the same.

Thus, in a nutshell, [11, 12] has small communication and large computational complexity while [44] is the other way around. In comparison, our contribution allows us to get "the best of both worlds". In Section 2.6.3 we give results of a comparison between implementations of our own and Fischlin's protocols. Finally, note that our protocol always computes the correct result, whereas Fischlin's has a small error probability.

In concurrent independent work, Garay et al. [46] propose protocols for comparison based on homomorphic encryption that are somewhat related to ours, although they focus on the model where the comparison result is to remain secret. They present a logarithmic round protocol based on emulating a new Boolean circuit for comparison, and they also have a constant round solution. In comparison, we do not consider the possibility of saving computation and communication in return for a larger number of rounds. On the other hand, their constant round solution is based directly on Blake and Kolesnikov's method, i.e., they do not have our optimization that allows us to make do with a smaller plaintext space for the encryption scheme, which means that our constant round protocol is more efficient.

[1]In [11, 12] the emphasis is on using the comparison to transfer a piece of data, conditioned on the result of the comparison. For this application, their solution has advantages over Fischlin's, even though the comparison itself is slower.

2.2 Homomorphic Encryption

For our protocol we need a semantically secure and additively homomorphic cryptosystem which we will now describe.

2.2.1 Key Generation

To generate keys, we take as input parameters k, t, and ℓ, where $k > t > \ell$. We first generate a k-bit RSA modulus $n = pq$ for primes p, q. This should be done in such a way that there exists three primes u, v_p, and v_q, where $v_p \mid (p-1)$ and $v_q \mid (q-1)$. We will later be doing additions of small numbers in $\mathbb{Z}_u$ where we want to avoid reductions modulo u, but for efficiency we want u to be as small as possible. For these reasons we choose u as the minimal prime greater than $\ell + 2$. The only condition on v_p and v_q is that they are random t-bit primes. We set $v = v_p v_q$ in the following.

Finally, we choose random elements $g, h \in \mathbb{Z}_n^*$ such that the multiplicative order of h is v modulo p and q, and g has order uv. The public key is now $pk = (n, g, h, u)$ and the secret key is $sk = (p, q, v_p, v_q)$. The plaintext space is $\mathbb{Z}_u$, while the ciphertext space is $\mathbb{Z}_n^*$.

2.2.2 Encryption and Decryption

To encrypt $m \in \mathbb{Z}_u$, we choose r as a random $2t$-bit integer, and let the ciphertext be

$$E_{pk}(m, r) = g^m h^r \bmod n.$$

For decryption of a ciphertext c, it turns out that for our main protocol, we will only need to decide whether c encrypts 0 or not. This is easy, since $c^v \bmod n = 1$ if and only if c encrypts 0. This follows from the fact that v is the order of h, uv is the order of g, and $m < u$. If the party doing the decryption has also stored the factors of n, one can optimize this by instead checking whether $c^v \bmod p = 1$, which will save a factor of 3–4 in practice.

It is also possible to do a “real” decryption by noting that

$$E_{pk}(m, r)^v = (g^m h^r)^v = (g^v)^m \bmod n.$$

Clearly, g^v has order u, so there is a 1–1 correspondence between values of m from $\mathbb{Z}_u$ and values of $(g^v)^m \bmod n$. Since u is very small, one can simply build a table containing values of $(g^v)^m \bmod n$ and corresponding values of m.

2.2.3 Security Evaluation

To evaluate the security, there are various attacks to consider. Factoring n will be sufficient to break the scheme, so we must assume factoring is hard. Also note that it does not seem easy to compute elements with orders such as g, h unless you know the factors of n, so we implicitly assume here that knowledge of g, h does not help to factor. Note that it is very important that g, h both have the same order modulo both p and q. If g had order uv modulo p but was 1 modulo q, then g would have the correct order modulo n, but $\gcd(g-1, n)$ would immediately give a factor of n.

One may also search for the secret key v, and so t needs to be large enough so that exhaustive search for v is not feasible. A more efficient generic attack (which is the best we know of) is to compute $h^R \bmod n$ for many large and random values of R. By the "birthday paradox", we are likely to find values R, R' where $h^R = h^{R'} \bmod n$ after about $2^{t/2}$ attempts. In this case v divides $R - R'$, so generating a few of these differences and computing the greatest common divisor will produce v. Thus, we need to choose t such that $2^{t/2}$ exponentiations is infeasible.

To prove the cryptosystem semantically secure, we must ensure that h^r is indistinguishable from a uniformly random element in the group generated by h. Recall that h has order v. In the original version of the cryptosystem [33, 34], v was a t-bit prime and the randomizer r was chosen as a $2t$-bit integer. Consequently, h^r were statistically indistinguishable from a uniformly random element in the group generated by h. Following the correction in [36], v is now a $2t$-bit composite. The natural choice for r would therefore be an integer larger than v, say a $3t$-bit integer. However, as mentioned above, we will make the assumption that raising h to a $2t$-bit integer is sufficient to make h^r indistinguishable from a uniformly random element.

To justify this assumption, we can look at a recent result by Goldreich and Rosen [53], who recently investigated indistinguishability in a setting similar to ours. Let n be a random k-bit RSA modulus, and let h be a random element of $\mathbb{Z}_n^*$. They showed that $h^r \bmod n$ is computationally indistinguishable from $h^R \bmod n$, where $R \in_R \{0, \ldots, \mathrm{ord}_n(h) - 1\}$, and $r \in_R \{0, \ldots, 2^{\lceil k/2 \rceil} - 1\}$. In other words, R is a random exponent of "full length" and r is a random exponent of "half length". We cannot apply their result directly to our case since our h is not random. However, their result suggests that using a $2t$-bit randomizer can be enough to hide h^r.

2.2.4 Subgroup Indistinguishability Conjecture

To say something more precise about the required assumption, let $G = \langle g \rangle$ be the group generated by g, and $H = \langle h \rangle$ the group generated by h. We have $H \leq G$ and that a random encryption is simply a uniformly random element in G. The assumption underlying security is now

Conjecture 2.1 *For any constant ℓ and for appropriate choice of t as a function of the security parameter k, the tuple (n, g, h, u, x) is computationally indistinguishable from (n, g, h, u, y), where n, g, h, u are generated by the key generation algorithm sketched above, x is uniform in G and y is uniform in H.* ◊

Proposition 2.2 *Under the above conjecture, the cryptosystem is semantically secure.* □

Proof: Consider any polynomial time adversary who sees the public key, chooses a message m and gets an encryption of m, which is of the form

$g^m h^r \bmod n$, where g has order uv and h has order v modulo p and q. The conjecture now states that even given the public key, the adversary cannot distinguish between a uniformly random element from H and one from G. But h^r was already indistinguishable from a random element in H, and so it must also be computationally indistinguishable from a random element in G. But this means that the adversary cannot distinguish the entire encryption from a random element of G, and this is equivalent to semantic security — recall that one of the equivalent definitions of semantic security requires that encryptions of m be computationally indistinguishable from random encryptions. ■

The only reason we set t to be a function of k is that the standard definition of semantic security talks about what happens asymptotically when a *single* security parameter goes to infinity. From the known attacks sketched above, we can choose t to be much smaller than k. Realistic values might be $k = 1024, t = 160$.

Having seen the security proof, we can now explain why the cryptosystem had to be changed. As mentioned above, the original system used a single t-bit prime v in place of the two t-bit primes, v_p, v_q. The prime v was chosen to divide both $p - 1$ and $q - 1$ and used in place of $v_p v_q$. However, such a v also divides the (public) $n - 1$. Now compute $a = (n-1)/u^j$ where u^j is the maximal power of u that divides $n - 1$ and consider $x \in G$. If $\mathrm{ord}(x) = v$, then $x \in H$ and $x^a = 1$ since a must contain a v factor. If the order of x is u or uv, then $x \in G \setminus H$ and $x^a \neq 1$ since a contains no u factors. The conjecture no longer holds.

Note that while it may seem natural to "solve" the problem by having v divide only $p - 1$, say, this would not be secure either: if h still has to have order v, this would force h to be 1 modulo q, and so one could factor n just by computing $\gcd(n, h - 1)$.

Using two primes, v_p, v_q ensures that there no longer is an easy connection between $n - 1$ and the secret key, rendering the above attack useless.

A central property of the encryption scheme is that it is homomorphic over u, i.e.,

$$E_{pk}(m,r) \cdot E_{pk}(m',r') \bmod n = E_{pk}(m + m' \bmod u, r + r').$$

The cryptosystem is related to that of [57], in fact ciphertexts in his system also have the form $g^m h^r \bmod n$. The difference lies in the way n, g and h are chosen. In particular, our idea of letting h, g have the same order modulo p and q allows us to improve efficiency by using subgroups of Z_n^* that are even smaller than those from [57].

2.3 Comparison Protocol

For the protocol, we assume that A has generated a key pair $sk = (p, q, v)$ and $pk = (n, u, g, h)$ for the homomorphic cryptosystem we described previously. The protocol proceeds in two phases: an input sharing phase in which the client must be online, and a computation phase where the server and assisting server determine the result while the client is offline. See Figure 2.1 on the following page for an overview.

In the input sharing phase C secret shares his input m between A and B:

- Let the binary representation of m be $m_\ell \cdots m_1$, where m_1 is the least significant bit. C chooses, for $i = 1, \dots, \ell$, random pairs $a_i, b_i \in \mathbb{Z}_u$ subject to $m_i = a_i + b_i \bmod u$.
- C sends privately $a_1, \dots, a_\ell$ to A and $b_1, \dots, b_\ell$. This can be done using any secure public-key cryptosystem with security parameter k, and requires communicating $\mathcal{O}(\ell \log \ell + k)$ bits.[2] In practice, a standard SSL connection would probably be used.

In the second phase we wish to determine the result $m > x$ where x is the current public price (with binary representation $x_\ell \cdots x_1$).

Assuming a value $y \in \mathbb{Z}_u$ has been shared additively between A and B, as C did it in the first phase, we write $[y]$ for the pairs of shares involved, so $[y]$ stands for "a sharing of" y. Since the secret sharing scheme is linear over $\mathbb{Z}_u$, A and B can compute from $[y]$, $[w]$ and a publically known value α a sharing $[y + \alpha w \bmod u]$. Note that this does not require interaction but merely local computation. The protocol proceeds as follows:

- A and B compute, for $i = 1, \dots, \ell$ sharings $[w_i]$ where
$$w_i = m_i + x_i - 2x_i m_i = m_i \oplus x_i.$$
- A and B now compute, for $i = 1, \dots, \ell$ sharings $[c_i]$ where
$$c_i = x_i - m_i + 1 + \sum_{j=i+1}^{\ell} w_j.$$
Note that if $m > x$, then there is exactly one position i where $c_i = 0$, otherwise no such position exists. Note also, that by the choice of u, it can be seen that no reductions modulo u take place in the above computations.
- Let α_i and β_i be the shares of c_i that A and B have now locally computed. A computes encryptions $E_{pk}(\alpha_i, r_i)$ and sends them all to B.
- B chooses at random $s_i \in \mathbb{Z}_u^*$ and s_i' as a $2t$-bit integer and computes a random encryption of the form
$$y_i = (E_{pk}(\alpha_i, r_i) \cdot g^{\beta_i})^{s_i} \cdot h^{s_i'} \bmod n.$$
Note that, if $c_i = 0$, this will be an essentially random encryption of 0, otherwise it is an essentially random encryption of a random nonzero value.

  B sends these encryptions to A in randomly permuted order.

[2]We need to send $\ell \log \ell$ bits, and public-key systems typically have $\theta(k)$-bit plaintexts and ciphertexts.

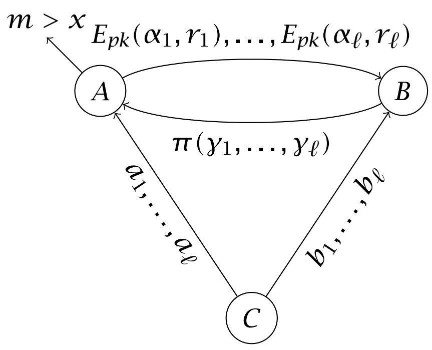

$m > x$
$E_{pk}(\alpha_1, r_1), \ldots, E_{pk}(\alpha_\ell, r_\ell)$
A
B
$\pi(\gamma_1, \ldots, \gamma_\ell)$
$a_1, \ldots, a_\ell$
$b_1, \ldots, b_\ell$
C

Figure 2.1: Our proposed protocol with both phases illustrated. In the first phase C sends shares to A and B. The second phase consists of a message from A to B and a reply, which A can decrypt to learn the result of the computation.

- A uses his secret key to decide, as described in the previous section, whether any of the received encryptions contain 0. If this is the case, he outputs "$m > x$", otherwise he outputs "$m \leq x$".

A note on preprocessing: One can observe that the protocol frequently instructs players to compute a number of form $h^r \bmod n$ where r is randomly chosen in some range, typically $\{0, \ldots, 2^{2t} - 1\}$. Since these numbers do not depend on the input, they can be precomputed and stored. As mentioned in the Introduction, this has a major effect on performance because all other exponentiations are done with very small exponents.

2.4 Security

In this section the protocol is proven secure against an honest but curious adversary corrupting a single player at the start of the protocol.

The client C has as input its maximum bid m and all players have as input the public bid x. The output given to A is the evaluation of $m > x$, and B and C get no output.

In the following we argue correctness and we argue privacy using a simulation argument. This immediately implies that our protocol is secure in Canetti's model for secure function evaluation [22] against a static and passive adversary.

2.4.1 Correctness

The protocol must terminate with the correct result: $m > x \iff \exists i : c_i = 0$. This follows easily by noting that both $x_i - m_i + 1$ and w_i is nonnegative so

$$\begin{aligned} c_i = 0 &\iff x_i - m_i + 1 + \sum_{j=i+1}^{\ell} w_j = 0 \\ &\iff x_i - m_i + 1 = 0 \wedge \sum_{j=i+1}^{\ell} w_j = 0. \end{aligned}$$

We can now conclude correctness of the protocol since $x_i - m_i + 1 = 0 \iff m_i > x_i$ and $\sum_{j=i+1}^{\ell} w_j = 0 \iff \forall j > i : m_j = x_j$, which together imply $m > x$. Note that since the sum of the w_j is positive after the first position in which $x_i \neq m_i$, there can be at most one zero among the c_i.

2.4.2 Privacy

Privacy in our setting means that A learns only the result of the comparison, and B learns nothing new. We can ignore the client as it has the only secret input and already knows the result based on its input.

First assume that A is corrupt, i.e, that A tries to deduce information about the maximum bid based on the messages it sees. From the client, A sees both his own shares $a_1, \ldots, a_\ell$, and the ones for B encrypted under some semantically secure cryptosystem, e.g., SSL. From B, A sees the message:

$$(E_{pk}(\alpha_i, r_i) \cdot g^{\beta_i})^{s_i} \cdot h^{s'_i} \bmod n.$$

By the homomorphic properties of our cryptosystem this can be rewritten:

$$E_{pk}(s_i \cdot \alpha_i, s_i \cdot r_i) \cdot E_{pk}(s_i \cdot \beta_i, s'_i) = E_{pk}(s_i(\alpha_i + \beta_i), s_i \cdot r_i + s'_i).$$

In order to prove that A learns no additional information, we can show that A could — given knowledge of the result, the publically known number and nothing else — simulate the messages it would receive in a real run of the protocol.

The message received and seen from the client can trivially be simulated as it consists simply of ℓ random numbers modulo u and ℓ encrypted shares. The cryptosystem used for these messages is semantically secure, so the encrypted shares for B can be simulated with encryptions of random numbers.

To simulate the messages received from B, we use our knowledge of the result of the comparison. If the result is "$m > x$", we can construct the second message as $\ell - 1$ encryptions of a nonzero element of $\mathbb{Z}_u^*$ and one encryption of zero in a random place in the sequence. If the result is "$m \leq x$", we instead construct ℓ encryptions of nonzero elements in $\mathbb{Z}_u^*$.

If we look at the encryptions that B would send in a real run of the protocol, we see that the plaintexts are of form $(\alpha_i + \beta_i)s_i \bmod u$. Since s_i is uniformly chosen, these values are random in $\mathbb{Z}_u$ if $\alpha_i + \beta_i \neq 0$ and 0 otherwise. Thus the plaintexts are distributed identically to what was simulated above. Furthermore, the ciphertexts are formed by multiplying $g^{(\alpha_i+\beta_i)s_i}$ by

$$h^{s_i r_i + s'_i} = h^{s_i r_i} h^{s'_i}.$$

Here h has order v which is $2t$ bits long, and as in Section 2.2.3, we will assume that taking h to the power of the random $2t$-bit number s'_i will produce something which is computationally indistinguishable from the uniform distribution on the subgroup generated by h. Since $h^{s_i r_i} \in \langle h \rangle$, the product will be indistinguishable from the uniform distribution on $\langle h \rangle$. So the s_i' effectively mask out $s_i r_i$ and makes the distribution of the encryption computationally indistinguishable from a random encryption of $(\alpha_i + \beta_i)s_i$. Therefore, the simulation is computationally indistinguishable from the real protocol messages.

The analysis for the case where B is corrupt is similar. Again we will prove that we can simulate the messages of the protocol. The shares received from the client and the encryptions seen are again simply ℓ random numbers modulo u and ℓ random encryptions and are therefore easy to simulate. Also, B receives the following from A:

$$E_{pk}(\alpha_i, r_i).$$

But since the cryptosystem is semantically secure, we can make our own random encryptions instead and their distribution will be computationally indistinguishable from the one we would get by running the protocol normally.

2.5 Extensions

Although the protocol and underlying cryptosystem presented in this chapter are specialized to one kind of comparison, both may be extended. In this section we will first consider how the protocol can be modified to handle more general comparisons where one input is not publically known, and we will also sketch how active security can be achieved. We also consider application of the cryptosystem to general multiparty computation.

2.5.1 Both Inputs are Private

Our protocol extends in a straightforward way to the case where A and B have private inputs a, b and we want to compare them. In this case, A can send to B encryptions of the individual bits of a, using his own public key. Since the cryptosystem is homomorphic over u, B can now do the linear operations on the bits of a and b that in the original protocol were done on the additive shares of the bits. Note that B has his own input in cleartext, so the encryptions of the exclusive-or of bits in a and b can be computed without interaction, using the formula $x \oplus y = x + y - 2xy$ which is linear if one of x, y is a known constant. B can therefore produce, just as before, a set of encryptions of either random values or a set that contains a single 0. These are sent to A for decryption. The only extra cost of this protocol compared to the basic variant above is that B must do $\mathcal{O}(l)$ extra modular multiplications, and this is negligible compared to the rest of the protocol.

2.5.2 Both Inputs are Shared, Secret Output

The case where both numbers a, b to compare are unknown to A and B can also be handled. Assume both numbers are shared between A and B using additive shares. The only difficulty compared to the original case is the computation of shares in the exclusive-or of bits in a and b. When all bits are unknown to both players, this is no longer a linear operation. But from the formula $x \oplus y = x + y - 2xy$, it follows that it is sufficient to compute the product of two shared bits securely. Let x, y be bits that are shared so $x = x_a + x_b \bmod u$ and $y = y_a + y_b \bmod u$, where A knows x_a, y_a and B knows x_b, y_b. Now, $xy = x_a y_a + x_b y_b + x_b y_a + x_a y_b$. The two first summands can be computed locally, and for, e.g., $x_a y_b$, A can send to B an encryption $E_{pk}(x_a)$. B chooses $r \in Z_u$ at random and computes an encryption $E_{pk}(x_a y_b - r \bmod u)$ using the homomorphic property. This is sent to A, and after decryption $(x_a y_b - r \bmod u, r)$ forms a random sharing of $x_a y_b$. This allows us to compute a sharing of xy, and hence of $x \oplus y$. Putting this method for computing exclusive-ors together with the original protocol, we can do the comparison at cost roughly twice that of the original protocol.

It follows from an observation in [95] that a protocol comparing shared inputs that gives a public result can always be easily transformed to one that gives the result in shared form so it is unknown to both parties. The basic idea is to first generate a shared random bit $[B]$ where B is unknown to both parties. Then from (bit-wise) shared numbers a, b, we compute two new bit-wise shared numbers $c = a + (b - a)B$ and $d = b + (a - b)B$. This just requires a linear number of multiplications. Note that $(c, d) = (a, b)$ if $B = 0$ and $(c, d) = (b, a)$ otherwise. Finally, we compare c, d and get a public result B'. The actual result can then be computed in shared form as $[B \oplus B']$.

2.5.3 Active Security

Finally, we sketch how one could make the protocol secure against malicious cheating. For this, we equip both A and B with private/public key pairs (sk_A, pk_A) and (sk_B, pk_B) for our cryptosystem. It is important that both key pairs are constructed with the *same* value for u. The client C will now share its input as before, but will in addition send to both players encryptions of all of A's shares under pk_A and all of B's shares under pk_B. Both players are now committed to their shares, and can therefore prove in zero-knowledge during the protocol that they perform correctly. Since the cryptosystem is homomorphic and the secret is stored in the exponent, one can use standard protocols for proving relations among discrete logs, see for instance [18, 45, 91]. Note that since the two public keys use the same value of u, it is possible to prove relations involving both public keys, for instance, given $E_{pk_A}(x)$ and $E_{pk_B}(y)$, that $x = y$. In the final stage, B must show that a set of values encrypted under pk_A is a permutation of a set of committed values. This is known as the shuffle problem and many efficient solutions for this are known — see, e.g., [56]. Overall, the cost of adding active security will therefore be quite moderate, but the computing the exact cost requires further work: The type of protocol we would use to check players' behavior typically have error probability 1 divided by the smallest prime factor in the order of the group used. This would be $1/u$ in our case, and the protocols will have to be repeated if $1/u$ is not sufficiently small. This results in a trade-off: we want a small u to make the original passively secure protocol more efficient, but a larger value of u makes the protocols we use for checking players' behavior more efficient.

2.5.4 Using the Cryptosystem for Multiparty Computation

In [28] it is described how any homomorphic cryptosystem with certain properties may be used for general multiparty computation (MPC), where several parties want to compute an arbitrary function of their inputs. The cryptosystem described in this chapter was already shown to be homomorphic, but in order for us to use the results from [28], we need to establish some other properties of the cryptosystem.

The basic ideas in the construction from [28] are as follows: A public key *pk* is assumed to be set up initially, where the secret key *sk* has been secret shared among the players. We assume an adversary that can corrupt (only) certain sets of players and *sk* is shared such that no corruptible set has information on it, whereas the set of players following the protocol could reconstruct it from their shares. What is needed is now a protocol for so called threshold decryption, where the players (using their shares of *sk* as private input) can decrypt a ciphertext securely, i.e., compute the plaintext while revealing no side information.

The cryptosystem is assumed to be additively homomorphic over some ring $\mathcal{R}$, in our case $\mathcal{R} = \mathbb{Z}_u$. If we want to compute the value $f(x_1, \dots, x_n)$ where the x_i's are private inputs and f is a given function, we think of the inputs and outputs as elements in $\mathcal{R}$, and we assume that f is given as an arithmetic circuit over $\mathcal{R}$, i.e., computing f can be done by additions and multiplications in $\mathcal{R}$. This is without loss of generality as any Boolean circuit can be simulated with these operations. But if we can have $\mathcal{R} = \mathbb{Z}_u$ for sufficiently large u, we may be able to compute f by a circuit over $\mathbb{Z}_u$ that is much smaller than a Boolean circuit. In particular, we can simulate integer addition and multiplication by single operations in $\mathbb{Z}_u$ if u is large enough compared to the inputs.

To compute f securely, players submit their inputs encrypted under *pk* and we then simulate the circuit computing f gate by gate. If we have encryptions $E_{pk}(a), E_{pk}(b)$ of the inputs to an addition gate, the homomorphic property immediately implies that we get an encryption of the output $a + b$ as $E_{pk}(a) \cdot E_{pk}(b) = E_{pk}(a + b)$. For multiplication, a simple protocol is given in [28] for computing securely an encryption $E_{pk}(ab)$ from $E_{pk}(a)$ and $E_{pk}(b)$. It requires a single threshold decryption and that each player sends a ciphertext to all other players. At end of the day, we have encryptions of the output values, which we then decrypt using threshold decryption.

To make this work against a passive adversary, all you need is the homomorphic property and secure threshold decryption. For active security, one needs in addition protocols by which players can show that they follow the rules during the multiplication operations.

We now look at how our encryption scheme can be made to fit this model. In our comparison protocol, we have assumed that the input was given as sharings or encryptions of single bits of the inputs. But as explained above, for general-purpose MPC, it is desirable to be able to supply an entire input number by sending a single ciphertext. We therefore need to consider the size of the our plaintext space. In the cryptosystem as described above, the plaintext space must be small in order to decrypt efficiently. Therefore, to have a larger plaintext space without loosing efficiency, we propose the following modification to the cryptosystem.

Instead of u being a prime, we make u a power of 2. After raising a ciphertext $g^m h^r \bmod n$ to the power of v, we get the number $(g^v)^m \bmod n$, where $g^v \bmod n$ is of order 2^c for some integer c. This means that to get m, we need to solve the discrete log problem where the base is 2-smooth. We can apply the Pohlig-Hellman algorithm [84] and solve the problem using $\mathcal{O}(c)$ modular multiplications.

It is now possible to use this cryptosystem for MPC with protection against semi-honest adversaries. It follows from the above that all we need is to show that secure threshold decryption is possible. So instead of having a single private key, we will instead secret-share it between the N parties. One way to do this is additive integer secret sharing, that is, the private share of player i, for $i = 1, \dots, N$, is a random $3t$-bit number v_i, and we give to all players $v_0 = v - \sum_{i=1}^{N} v_i$. Note that even if all shares except one, say v_i, are known, the only information related to v that can be computed is $v - v_i$, which is statistically indistinguishable from a random (negative) $3t$-bit number.

Now the parties can work together to decrypt a ciphertext $c = g^m h^r \bmod n$. Each player simply broadcasts $c^{v_i} \bmod n$, and then everyone can compute

$$\prod_{i=0}^{N} c^{v_i} \bmod n = (g^v)^m \bmod n.$$

If we include $g^v \bmod n$ in the public key, m can now be found as in the single user case above. Note that we now need to make a stronger assumption, namely that the cryptosystem remains semantically secure, even when $g^v \bmod n$ is given.

It is straightforward to show:

Lemma 2.3 *The above threshold decryption protocol is secure against a passive and static adversary, corrupting up to $N - 1$ players.* □

Proof: To prove the security of the decryption protocol, we will show that given the public information, i.e., public key as well as the ciphertext and plaintext, we can efficiently simulate everything else the adversary sees, namely the shares of corrupted players and the messages received from the honest players in the protocol.

Assume without loss of generality that player N is the only uncorrupted player. Now, generate shares $v_0, \dots, v_N$ as described above using secret value 0, and give $v_0, \dots, v_{N-1}$ to the adversary, to play the role of shares of corrupted players. As argued above, since the adversary doesn't see v_N, this has distribution statistically close to what would be the case if the real v-value had been used. We simulate the message sent by player N when decrypting c as

$$x = c^v \Big(\prod_{i=0}^{N-1} c^{v_i} \Big)^{-1}.$$

In the real world, the v_i are chosen at random such that $\sum_{i=0}^{N} v_i = v$. It follows that x has the same distribution as player N's contribution in the real world. Note that c^v needs to be computed as well, but since we are given g^v along with the public key, we may compute c^v as $(g^v)^m = c^v$.

The above argument shows that the decryption protocol can be simulated without knowledge of the key shares for the honest player, and therefore that the adversary gains no information about v by participating in the protocol. ■

It follows from the above that we may use our cryptosystem, with the modification for the plaintext space, to achieve general MPC that is secure against a passive and static adversary corrupting all but one of the players.

It is possible to have security against malicious adversaries as well. According to [28] this only requires the existence of honest verifier zero-knowledge proofs (Σ-protocols) for proving certain statements on ciphertexts. It follows from [45] that such Σ-protocols can be constructed using standard methods (even though the orders of g and h are hidden). Unfortunately, soundness of the most efficient versions of the protocols from [45] require that the order of g, h have only large prime factors. This is not true in our case. The problem can be solved at some cost in efficiency by using the basic protocols with 1-bit challenges and running several instances in parallel.

2.6 Complexity and Performance

In this section we measure the performance of our solution through practical tests. The protocol by Fischlin [44] provides a general solution to comparing two secret integers using fewer multiplications than the other known general solutions. We show that in the special case where one integer is publically known and the other is additively shared between two parties, our solution provides for faster comparisons than our adaptation of [44].

2.6.1 Setup and Parameters

As described above, our special case consists of a server, an assisting server and a client. The client must be able to send his value and go offline, whereafter the other two parties should be able to do the computations together. In our protocol the client simply sends additive shares to each of the servers and goes offline. However, the protocol by Fischlin needs to be adapted to this scenario before we can make any reasonable comparisons. A very simple way to mimic the additive sharing is for the client to simply send his secret key used for the encoding of his value to the server while sending

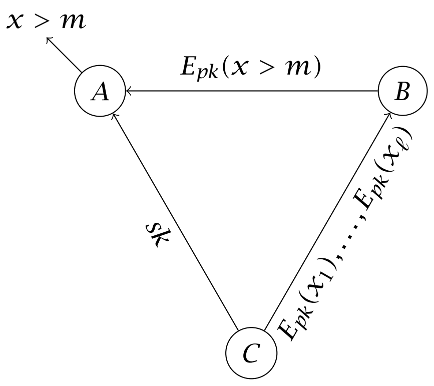


Figure 2.2: The modified Fischlin protocol. The client C can go off-line after having sent the key to A and the encryptions to B. From that point the protocol proceeds as in [44].

the actual encoding to the assisting server. Clearly the computations can now be done by the server and assisting server alone, where the server plays the role of the client. The modified protocol is shown in Figure 2.2.

Together, the key and encoding determine the client's secret value, but the key or the encoding alone do not. The key of course reveals no information about the value. Because of semantic security, the encryption alone does not reveal the secret to a computationally bounded adversary.

Another issue is to how to compare the two protocols in a fair way. Naturally, we want to choose the parameters such that the two protocols offer the same security level, but it is not clear what this should mean as some of the parameters in the protocols control events of very different nature. Below, we describe the choices we have made and the consequences of making different choices.

Both protocols use an RSA modulus for their encryption schemes, and it is certainly reasonable to use the same bit length of the modulus in both cases, say 1,024 bits. Our encryption scheme also needs a parameter t which we propose to choose as $t = 160$. This is because the best known attack tries to have random results of exponentiations collide in the subgroup with about 2^{160} elements. Assuming the adversary cannot do much more than 2^{40} exponentiations, the collision probability is roughly $2^{2\cdot 40}/2^{160} = 2^{-80}$.

We do not have this kind of attack against Fischlin, but we do have an error probability of $5\ell \cdot 2^{-\lambda}$ per comparison. If we choose the rationale that the probability of "something going wrong" should be the same in both protocols, we should choose λ such that Fischlin's protocol has an error probability of 2^{-80}. An easy computation shows that for $\ell = 16$, $\lambda = 86$ gives us the desired error probability, and it follows that $\lambda = 87$ works for $\ell = 32$.

We have chosen the parameter values as described above for our implementation, but it is also possible to argue for different choices. One could argue, for instance, that breaking our scheme should be as hard as factoring the 1,024-bit modulus using the best known algorithm, even when the generic attack is used. Based on this, t should probably be around 200. One could also argue that having one comparison fail is not as devastating as having the cryptosystem broken, so that one could perhaps live with a smaller value of λ than what we chose. Fischlin mentions an error probability of 2^{-40} as being acceptable. These questions are very subjective, but fortunately, the complexities of the protocols are linear in t and λ, so it is easy to predict how modified values would affect the performance data we give below. Since we find that our protocol is about 10 times faster, it remains competitive even with $t = 200, \lambda = 40$.

2.6.2 Implementation

To evaluate the performance of our proposed protocol we implemented it along with the modified version of the protocol by Fischlin [44] described above. The implementation was done in Java 1.5 using the standard BigInteger class for the algebraic calculations and Socket and ServerSocket classes for TCP communication. The result is two sets of programs, each containing a server, an assisting server, and a client. Both implementations weigh in at about 1,300 lines of code. We have naturally tried our best to give equal attention to optimizations in the two implementations.

We tested the implementations using keys of different sizes (k in the range of 512–2,048 bits) and different parameters for the plaintext space ($\ell = 16$ and $\ell = 32$). We fixed the security parameters to $t = 160$ and $\lambda = 86$ which, as noted above, should give a comparable level of security.

The tests were conducted on six otherwise idle machines, each equipped with two 3 GHz Intel Xeon CPUs and 1 GiB of RAM. The machines were connected by a high-speed LAN. In a real application the parties would not be located on the same LAN: for credibility the server and assisting server would have to be placed in different locations and under the control of different organizations (e.g., the auction house and the accountant), and the client would connect via a typical Internet connection with a limited upstream bandwidth. Since the client is only involved in the initial sharing of his input, this should not pose a big problem — the majority of network traffic and computations are done between the server and assisting server, who, presumably, have better Internet connections and considerable computing power.

The time complexity is linear in ℓ, so using 16-bit numbers instead of 32-bit numbers cuts the times in half. In many scenarios one will find 16 bits to be enough, considering that most auctions have a minimum required increment for each bid, meaning that the entire range is never used. As an example, eBay require a minimum increment which grows with the size of the maximum bid meaning that there can only be about 450 different bids on items selling for less than \$5,000 [42]. The eBay system solves ties by extra small increments, but even when one accounts for them one sees that the 65,536 different prices offered by a 16-bit integer would be enough for the vast majority of cases.

Table 2.1: Benchmark results. The first column denotes the key size k, the following columns have the average time to a comparison. The average was taken over 500 rounds, after an initial warm-up phase of 10 rounds. The abbreviation "DGK" refers to our protocol and "F" refers to the modified Fischlin protocol. The subscripts refer to the ℓ parameter used in the timings.

k	DGK_{16}	F_{16}	DGK_{32}	F_{32}
512	82 ms	844 ms	193 ms	1,743 ms
768	168 ms	1,563 ms	331 ms	3,113 ms
1,024	280 ms	2,535 ms	544 ms	5,032 ms
1,536	564 ms	4,978 ms	1,134 ms	10,135 ms
2,048	969 ms	8,238 ms	1,977 ms	16,500 ms

2.6.3 Benchmark Results

The results of the benchmarks can be found in Table 2.1 with a graph in Figure 2.3 on the following page. From the table and the graph it is clear to see that our protocol has performed significantly faster in the tests than the modified Fischlin protocol. The results also substantiate our claim that the time taken by an operation is proportional to the size of ℓ and that we do indeed roughly halve the time taken by reducing the size of ℓ from 32 to 16 bits.

We should note that these results are from a fairly straight-forward implementation of both protocols. Further optimizations can likely be found, in both protocols.

2.7 Conclusion

This chapter has demonstrated a new protocol for comparing a public and a secret integer using only two parties, which among other things has applications in online auctions. Our benchmarks suggest that our new protocol is highly competitive and reaches an acceptably low time per comparison for real-world application.

We have also shown how to extend the protocol to the more general case where we have two secret integers and to the active security case. However, further work is needed to evaluate the competitiveness of the extended protocols.

Acknowledgments. We thank Claudio Orlandi for helping us realize the problem with the original scheme. The authors would also like to thank Tomas Toft, Rune Thorbek, and Thomas Mølhave for their comments and suggestions.

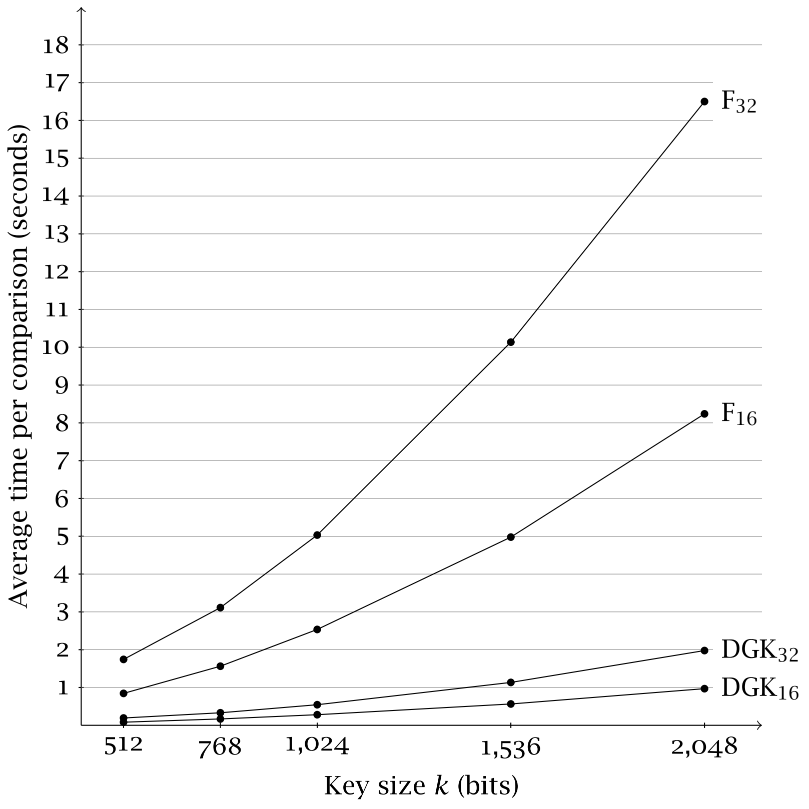


Figure 2.3: Graph of the data from Table 2.1 on the preceding page.

Chapter 3

Virtual Ideal Functionality Framework

The benchmark results in Chapter 2 where obtained using an ad hoc implementation made especially for those benchmarks. Re-inventing everything from scratch for each new protocol is a waste of useful resources, so in this chapter we present a general framework for doing secure multiparty computation. This is the Virtual Ideal Functionality Framework, or VIFF for short. Simply put, VIFF aims at providing an efficient and high-level basis on which practical applications using MPC can be built. It is also our hope that the framework offered by VIFF will facilitate rapid prototyping of new protocols by researchers and so lead to more protocols with practical applications. The source code and documentation is therefore freely available from the VIFF homepage [47].

The material presented in this chapter is partly from the VIFF homepage [47], partly from the author's progress report [48], and partly from a report on VIFF for the CACE project [50]. Much of it is also new: Section 3.2 on related work, Section 3.5 on the implementation of VIFF, and Section 3.8 which describes applications that use VIFF.

We will start by describing the setting in which VIFF came to be. We then describe the overall goals of VIFF and elaborate on how we believe to have reached some of these goals.

3.1 Introduction

This section serves to give a quick summary of the historical background for the VIFF system and an overview of its features.

3.1.1 History

VIFF was started by the author of this dissertation in March 2007 as a spin-off project from the Secure Information Management And Processing (SIMAP) project, which in turn is a successor to the Secure Computing Economy and

Trust (SCET) project, both at the University of Aarhus. The SCET project set out to implement a platform for multiparty computations with a focus on economic applications such as auctions and benchmarking. The project implemented a prototype of a secure double auction [15]. SIMAP had a more general aim and designed a dedicated programming language in which high-level protocol descriptions can be specified [79]. In January 2008 the SIMAP project ran the first ever large-scale MPC application in which Danish farmers traded sugar beet contracts using a secure double auction [16].

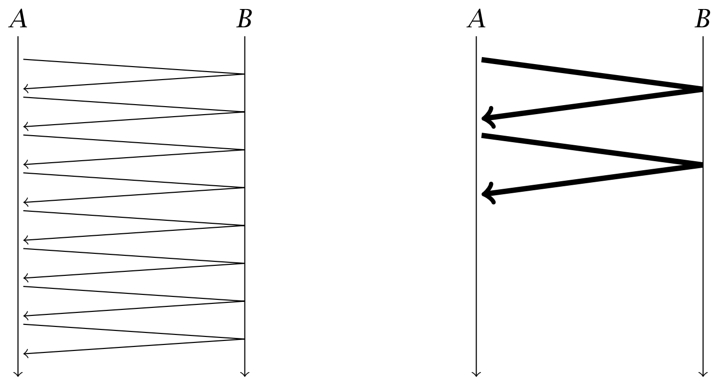
The overall goal for the SCET and SIMAP projects is the development of efficient and practical tools for building MPC applications to solve real-world problems. In SIMAP, this is achieved with a combination of efficient cryptographic protocols coupled with a domain-specific language. This dissertation will focus on the underlying protocols used by the domain-specific language. That is not to say that the language is not important — it is in fact quite important in order to enable "normal" programmers to use the cryptographic protocols in an easy and *secure* manner without being security experts themselves.

The sugar beet auction mentioned above [16] was built on a system written in Java. The implementation had some 130 classes and interfaces and about 8,500 lines of code. While implementing the comparison protocol it had become clear that there were a number of problems with its design. This lead the author of this dissertation to begin experimenting with a new design in Python.

We knew from the beginning that it was important to execute operations in batches to use the bandwidth in the most efficient way. This is necessary since networks have significant round trip times. Figure 3.1 illustrates this difference. So the SIMAP system had a concept of "batch jobs". These were used to group parallel operations together. The system would then execute either one normal operation at a time or a single batch job consisting of multiple operations. An example of a batch job could be the many secure multiplications needed to compute the inner product of two vectors with secret shared components.

However, the batch job architecture implemented in the SIMAP project turned out to be difficult to work with. The primary problem was that special code is needed for dealing with combinations of different batch jobs. So when implementing a new operation, such as a comparison protocol, one had to define how a batch job would look like for the new operation and how the batch job would be combined with all other types of batch jobs. This severely limited the modularity since every new piece of code needed to know about every old piece. The second problem with using batch jobs is that they require the programmer to determine the optimal grouping. This might be possible in small programs or for particular functions such as our inner product example from before, but requiring such a "bird's eye view" of the system will again limit the modularity.

A second problem with the SIMAP system was that although the high level interface manipulated secret shared values as first class objects, the lower level runtime system did not. This meant that the runtime system could not be extended in terms of itself, akin to how the earliest compilers



(a) Normal execution. (b) Batched execution.

Figure 3.1: Normal and batched execution. In a normal execution each operation triggers its own little message, resulting in a significant waste due to the round trip time. The waste can be reduced dramatically by grouping data for several operations into a single batch.

had to be written in assembler instead of a higher level language. The low level kept secret values in a `HashMap` and referenced them using integers. Apart from being awkward to program with, this design meant that we had to do our own memory management by keeping track of when values could be deleted from the `HashMap` and when they should be retained. In a language like C, this might have been acceptable. However, given that Java already has a garbage collector, it is a shame not to utilize it. We will have more to say about memory management in Section 7.1.3.

3.1.2 Feature Overview

VIFF provides the following notable features:

Asynchronous execution: As described in further detail in Section 3.3.2, modern networks are all asynchronous by nature. VIFF is designed to be used on such networks.

Automatic parallel scheduling: Network latencies will typically dominate the execution time. This makes it important to execute many operations in parallel in order to lower the average waiting time. Also, the automatic parallelism can potentially yield a faster execution since it will adapt better to changing network conditions: With a static schedule based on rounds, the execution stalls if a round takes longer than expected. VIFF would begin executing the next available operation immediately.

Easy composability: VIFF was designed with a simple core on which more complex protocols can be built. Combining smaller protocols into larger protocols is an essential feature. Protocols written for VIFF can automatically be run in parallel with other protocols. This applies to both new primitive operations and complex protocols.

3.2 Related Work

Though the field of secure multiparty computation is almost three decades old, the number of practical implementations is small. We have already briefly described the SCET and SIMAP projects from the University of Aarhus. In this section we will describe the other two frameworks known to the author: Fairplay and SHAREMIND.

3.2.1 Fairplay

The first general-purpose system for secure two-party function evaluation was the Fairplay project. It was started at the Hebrew University of Jerusalem, and presented by Malkhi et al. [69] in 2004. Fairplay version 1.0 implements the original two-party computation protocol by Yao [100] with security against an active adversary.

Programs for Fairplay are written in a dedicated language called SFDL (Secure Function Definition Language). Yao's original millionaires problem is implemented by the program in Figure 3.2.

```
program Millionaires {
    type int = Int<20>; // 20-bit integer
    type AliceInput = int;
    type BobInput = int;
    type AliceOutput = Boolean;
    type BobOutput = Boolean;
    type Output = struct {
        AliceOutput alice,
        BobOutput bob
    };
    type Input = struct {
        AliceInput alice,
        BobInput bob
    };

    function Output output(Input input) {
        output.alice = (input.alice > input.bob);
        output.bob = (input.bob > input.alice);
    }
}
```

Figure 3.2: Yao's millionaires problem.

The high-level SFDL program is compiled into a low-level Boolean circuit. In doing so, the compiler will inline functions (direct or indirect recursion is forbidden), unroll loops (the number of loop iterations must be a compile-time constant), and turn all operations into a series of primitive "hardware" operations that operate on single bits only. Finally, the circuit is run through a peephole optimizer followed by duplicate and dead code removal. The optimization step is quite important in order to cut down on the number of gates in the final circuit — it is reported in [69] that this often reduces the size of the circuit by an order of magnitude.

The two players — Alice and Bob — must now evaluate the optimized circuit on their private inputs. To do this, the circuit is turned into a Yao-garbled circuit by Bob which is transferred to Alice. The garbled circuit consists of permuted and encrypted truth tables and we will now sketch how they are constructed.

First, Bob assigns two random keys for each wire in the circuit. The first key is used when the wire carries a 0, the second when it carries a 1. To illustrate this, consider a truth table for computing $z = x \wedge y$, i.e., a table for AND. Bob assigns random keys x_0 and x_1 to the first input wire and random keys y_0, y_1 to the second input wire. The output wire is assigned a random z_0 for a result of 0 and a random z_1 for a result of 1. He then uses the keys x_i and y_j to encrypt $z_{i \wedge j}$, please see Figure 3.3. The final truth table consists of the permuted encryptions only, in order to completely hide the nature of the gate [67].

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(a) Normal.

x	y	$x \wedge y$
x_0	y_0	$E_{x_0 \| y_0}(z_0)$
x_0	y_1	$E_{x_0 \| y_1}(z_0)$
x_1	y_0	$E_{x_1 \| y_0}(z_0)$
x_1	y_1	$E_{x_1 \| y_1}(z_1)$

(b) Garbled.

—
$E_{x_1 \| y_1}(z_1)$
$E_{x_0 \| y_1}(z_0)$
$E_{x_1 \| y_0}(z_0)$
$E_{x_0 \| y_0}(z_0)$

(c) Permuted.

Figure 3.3: The main steps in the transformation of a Boolean circuit into a Yao-garbled circuit. The normal truth table is garbled (encrypted) using random keys assigned to each wire. Finally, the keys identifying the wires are thrown away, the encryptions are permuted randomly, and the table header is removed.

Having received the garbled circuit, Alice needs the keys for the input wires. Bob can simply send the keys corresponding to his own input bits since the keys be themselves reveal nothing. Alice cannot just ask for the keys corresponding to her input bits — she must instead obtain them by oblivious transfer (OT). They engage in an OT for each input wire. Bob holds input keys i_0 and i_1 and Alice retrieves i_σ corresponding to her input bit σ.

Alice can now decrypt the first layer of gates to reveal the next set of keys. Continuing like this, she will eventually decrypt the final layer. The keys for the output wires can be chosen as 0 and 1 so that Alice obtains the result directly. If some of the bits should be revealed to Bob only, then Bob can encrypt the bits as usual by assigning random keys to that output wire.[1]

[1]It is unavoidable that one of the parties learn the output before the other.

This description leaves out some details from [69], such as the precise construction of the truth tables and how Alice tries to catch a malicious Bob by verifying $m - 1$ out of m garbled circuits.

FairplayMP is an updated version with support for more than two parties [9]. It combines a protocol by Beaver et al. [6] with the classic BGW [10] protocol to obtain an constant-round protocol (their implementation use eight rounds). Whereas Fairplay is secure against an active adversary, FairplayMP is secure against a passive adversary only.

The secure function definition language has also been updated to version 2.0 [43]. The new language is similar to SFDL 1.0, but has the necessary syntactic support for describing multiparty protocols. More information is available at the Fairplay homepage: `http://fairplayproject.net/`.

3.2.2 Sharemind

The Sharemind project by Bogdanov et al. [14] provides a system for general multiparty computation on large data sets. They focus on speed and massive scalability in a setting with three players and a passive adversary.

The focus on speed is evident in the choice of C++ as the implementation language. Fairplay is implemented in Java because of its good support for cross-platform usage [9] and VIFF is implemented in Python because of its flexibility and the availability of the Twisted network library. Both Java and Python operate on a much higher level than C++.

They have also chosen a somewhat unusual domain for their input numbers. Whereas almost all cryptographic protocols focus on (prime) fields such as $\mathbb{Z}_p$, Sharemind works with integers from the ring $\mathbb{Z}_{2^{32}}$. This gives them a substantial speed advantage since they — as the only system out of the three under consideration — can use plain machine words as their basic type for arithmetic operations. Put differently, the CPU will do useful work in all its clock cycles, instead of doing modulo reductions much of the time. The choice of $\mathbb{Z}_{2^{32}}$ as the basic input domain did force the Sharemind developers to design their own protocol for multiplication [14]. The problem is the lack of multiplicative inverses, which renders Shamir secret sharing impossible.

The Sharemind architecture consist of miners who hold the data and clients who manipulate the data. Whereas a system like Fairplay expects the protocol participants to change dynamically, the miners are meant to be running for an extended period of time. A typical example would be privacy preserving data mining where the three miners share a large database among them. The three miners stay online, and clients can connect to them at different times to query the database. Please see Figure 3.4 on the next page.

The operations supported are secure addition, multiplication, comparison, and bit extraction of 32-bit integers. Unlike VIFF and Fairplay, but similar to SIMAP, Sharemind has vectorized versions of all its operations. In SIMAP, multiple operations can be bundled together in a batch job as described in Section 3.1.1. In Sharemind, all the basic operations such as addition, multiplication, and comparison work with vectors by default.

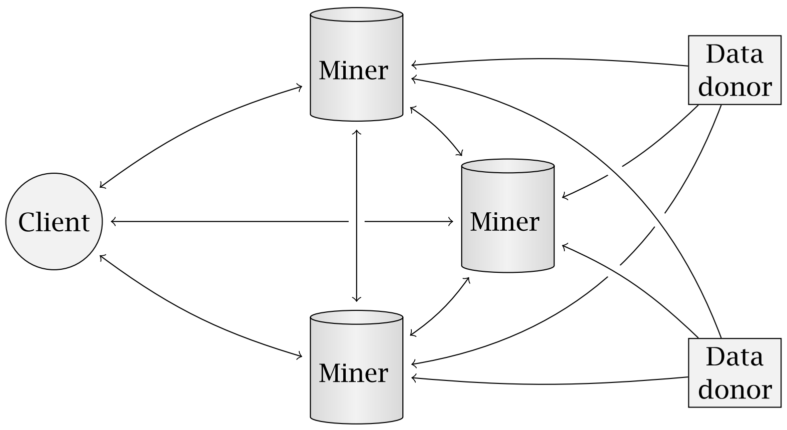


Figure 3.4: The SHAREMIND architecture. Three miner nodes store an additive secret sharing of the data, which is provided by several data donors. One or more clients connect to the miner nodes and instruct them to do privacy preserving computations on the data.

```
public count (private int needle, private int[] haystack) {
    public int n; n = getRowCount(haystack);

    // An indicator vector of ones and zeroes
    private bool[n] indicator;
    indicator = (haystack == needle);

    private int sum; sum = vecSum(indicator);
    public int count; count = declassify(sum);
    return count;
}
```

Figure 3.5: Finding the proverbial needle in a haystack [63].

This is very important for scalability when performing data mining on large data sets. It is reported in [14] that the use of vectorized operations give a factor 27 increase in throughput with vectors containing 100,000 elements.

Like Fairplay, SHAREMIND offers a high-level language in which programmers can implement the privacy preserving algorithms. The language is called SecreC (pronounced “secrecy”) and is currently under development. The language has a C-like syntax and allows the programmer to declare variables as public or private. Private variables are kept in their additively secret shared form throughout the computation. A simple SecreC example is given in Figure 3.5.

Instead of looping through the vector one element at a time, the code does a single vectorized comparison. As described above, using as many vectorized functions as possible greatly improves the performance.

Programs written in SecreC will be compiled down to an assembly language developed for SHAREMIND. As an example of SHAREMIND assembly, consider the program in Figure 3.6 on the facing page.[2]

The program finds the minimum element on the stack using a logarithmic number of rounds. Lines 5–9 push data on the stack — here the values are public, but the language has instructions for pushing database columns onto the stack too. Such a column can contain private data, i.e., data secret shared among the miners. The program then enters a loop in which it repeatedly halves the number of elements on the stack by a vectorized comparison. The **pushlte** command on line 22 treats the stack as $[\dots, \mathbf{x}, \mathbf{y}]$ where $\mathbf{x}$ and $\mathbf{y}$ contain div elements. The vectors are compared pair-wise and the smallest element is retained from each pair, see Figure 3.7.

Using the assembly language, the set of miner nodes can be seen as a virtual machine. The assembly language is the instruction set for this machine. This is similar to how we see VIFF as an arithmetic black-box defined by the functionality $\mathcal{F}_{\text{VIFF}}$.

SHAREMIND can be downloaded from `http://sharemind.cs.ut.ee/`.

3.3 Security Model

VIFF is designed to accommodate different security models — it was important for us to make a decoupled design in which the core of VIFF is completely oblivious to the specific security model in use.

3.3.1 Adversary Types

We have focused on the most classic and simple setting first and implemented protocols with security against threshold adversary structures. A threshold adversary is allowed to corrupt any t out of the n players. The standard BGW protocol [10, 51] is secure as long as an honest but curious adversary corrupt no more than $n/2$ of the parties. The protocol proposed in Chapter 4 can tolerate $n/3$ maliciously corrupted parties.

The parties are assumed to be connected by pair-wise secure and authenticated channels. With ideal secure channels, our protocols would be secure against an adaptive adversary who corrupts players during the protocol execution itself. We live in the real world, where ideal secure channels are hard to come by, so in practice we are protecting the channels using standard cryptographic tools, e.g., SSL. Relying on cryptographic tools means that we can only prove the protocols secure against a static adversary. However, it also allows us to delegate the job of building an efficient secure channel to someone else by using standard libraries. Furthermore, it is possible that the parties already posses the required SSL certificates, issued by a trusted certificate authority. As an example, the Danish nation-wide public key infrastructure [62] could provide the necessary digital signatures.

The adversary will have full control over the scheduling of the network. This means that he can delay and reorder packets in arbitrary ways. We assume secure and reliable channels, so he cannot inspect or change the content of the packets, and he can also not remove packets.

[2]The example is from the file `bin/assembly/MinimumLog.txt` in SHAREMIND 1.91.

```
public int size
public int mod
public int div

push 3
push 1
push 10
push 7
push 8

loop:
    stacksize size
    cmp size == 1 // Single element on the stack => we are done.
        jt :end

    mov mod, size % 2
    cmp mod == 0 // Adjust for odd number of elements.
        jf :pad

    compare: // Perform vectorized comparison.
        mov div, size / 2
        pushlte div
        jmp :loop

    pad:
        dec size
        jmp :compare
end:

public int result
pop result // Declassify top element on stack.
println result
```

Figure 3.6: SHAREMIND assembly for finding the minimum stack element.

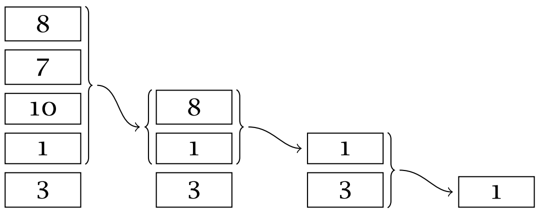


Figure 3.7: Vectorized comparisons.

3.3.2 Network Types

The demands we put on synchronicity of the network play a key role in the protocol design. There is a clear trade-off between using a simple model and using a more complex model, as we will see next.

Synchronous Networks

The classic results on secure multiparty computation [24, 54] assumed a synchronous setting. This is a quite strong assumption, but it makes the model much simpler. In a synchronous network, everything follows the ticks made by a global clock. Each tick indicates the start of a new network round. A round begins with the delivery of all messages sent in the previous round. Each party is then asked to specify a number of new messages which will be delivered at the beginning of the next round.

In practice, synchronous networks are only found in the optical fiber networks used by the telecommunications industry. These backbone networks use specialized equipment operating at fixed bandwidths. The synchronous model does not match the consumer-grade communication networks or computers used for normal Internet traffic. Modern networks are better thought of as being *asynchronous*.

Asynchronous Networks

Every time a packet is sent over the Internet, the appropriately named Internet Protocol (IP) [85] is used. IP has the responsibility of getting the packets to the correct destination. But IP gives very few guarantees: intermediate routers might drop packets at any time (due to problems like congestion and transmission errors) and packets may be reordered or duplicated. In particular, the IP gives no guarantees about the delivery time (if the packet reaches the destination at all). IP packets have a checksum field in their header and this is used for verifying data integrity. In other words, IP will only ensure that if a packet is delivered, then it will arrive unchanged and at the correct destination.

The Transmission Control Protocol (TCP) [86] is used to create a virtual connection on top of the connection-less IP network. Because packets can be lost on the IP level, TCP must be prepared to ask for retransmission of data. This means that the delivery can be delayed further. Two parties communicating over TCP are reading and writing a *stream* of bytes — there are no messages at the TCP level. This is again a contrast to the classical way networks are modeled. As bytes are written to the stream, TCP will take care of buffering and will send out IP packets as they are filled or when it has been too long since the last packet was sent. Such buffering introduces further unpredictable delays in the protocol.

With regard to the security, then it should be noted that asynchronous networks makes it impossible for a protocol to determine if a party is just being slow, or if the party is behaving maliciously. Protocols that want to tolerate a malicious adversary will therefore have to be designed in such a way that they can always continue, even if up to t of the parties have not been heard from. In the worst case, this can lead to *input deprivation* where some honest parties do not get to provide input to the computation. We will how giving up on guaranteed termination can help prevent this in Chapter 4.

No Global Clocks

Another difference lies in the assumptions about timekeeping. The nodes in a synchronous network must be carefully synchronized, preferably to atomic clocks which act as a global clock. Normal computers do not have access to a globally correct clock. Computers are typically built with an on-board oscillator used to keep track of the time. Even if initially synchronized, clocks will drift away from each other since frequencies of oscillators vary with temperature. The Network Time Protocol (NTP) is widely used to keep computers synchronized to a standard time [76]. Roughly speaking, this is done by exchanging packets containing timestamps, from which the network delay can be estimated and the local clock adjusted accordingly. But the NTP server is a trusted third party and we will not allow our protocols to rely on such a service.

3.4 VIFF Ideal Functionality

When using VIFF, the developer is making library calls to the VIFF API. We will now formalize this interface as an ideal functionality within the UC framework as described in Section 1.3. We call the functionality $\mathcal{F}_{\text{VIFF}}$ and this functionality covers the core of VIFF. There are several implementations of this ideal functionality, each with a different security guarantee. For security against a semi-honest adversary, we have an implementation for two players based on the Paillier [81] cryptosystem and a much faster implementation for more players based on the BGW [10] protocol. We also have an implementation secure against a malicious adversary, this is the focus of Chapter 4. We call each such implementation a *runtime*. The runtimes implement the basic functionality, but will normally also provide extra functions. We will discuss these extra functions in Section 3.6.2 on page 52.

The VIFF ideal functionality will of course be designed in such as way that it can be implemented efficiently on modern communication networks as described in Section 3.3.2. We begin by defining a number of primitive operations which are necessary and sufficient to do efficient MPC.

We want the functionality to allow several primitive operations to be started at once due to the inherit delays of network traffic, which makes it possible to send out several packets before getting a reply to the first. The programming language defined by the available commands resemble a simple imperative programming language of the same flavor as Pascal and C. The language is *straight-line*, meaning that there are no looping or branching constructs. The meta-program run by the environment can still include branching and looping. By this we mean that the environment is free to decide on the next command based on output from the functionality.

The ideal functionality reacts on input from the environment $\mathcal{Z}$ (sent through the dummy parties) and processes one command at a time — if $\mathcal{Z}$ sends commands too fast they are buffered. Commands are discarded if they are not *valid*, meaning that they must consist of a recognized instruction as well as a *program counter.* A program counter is an opaque tag that must be unique for a given protocol run. The program counters are used by $\mathcal{F}_{\text{VIFF}}$ to associate each result with a specific command — this is necessary because multiple concurrent commands may be executing at the same time.

Each valid command may have some conditions attached to it. If the conditions are not fulfilled, $\mathcal{F}_{\text{VIFF}}$ buffers the command. If more than one command is eligible for execution, $\mathcal{F}_{\text{VIFF}}$ will choose the next command to execute at random. The functionality has a memory M, in which it stores a mapping between variables names and values. The values are elements from some field $\mathbb{F}$, picked to be suitably large for the computation at hand.

The functionality leaks information to the adversary about all commands it receives from the dummy parties, including the sending party. Private inputs in the commands are blanked (replaced by ?). This models that the values used in the computation are secret but the computation trace itself is public and known to the adversary. As each command is executed, $\mathcal{Z}$ will expect to see acknowledgments with the correct program counters. We let the adversary $\mathcal{A}$ send those through $\mathcal{F}_{\text{VIFF}}$, who will simply pass on any input it get from $\mathcal{A}$.

The protocol execution terminates when $\mathcal{Z}$ outputs a bit. It is up to $\mathcal{Z}$ to decide (depending on the output it receives from the parties and the adversary $\mathcal{A}$) when it terminates the execution. Please see Figure 3.8 for the definition of $\mathcal{F}_{\text{VIFF}}$.

3.4.1 Real World Protocol

We will now describe the protocols which will realize the ideal world protocol just described by $\mathcal{F}_{\text{VIFF}}$ in Figure 3.8. The protocol presented here is the basic protocol that is secure against a passive adversary only. It closely resembles the protocol by Ben-Or et al. [10] (with the simplified multiplication protocol of Gennaro et al. [51]) but it has been adapted to work in a fully asynchronous setting. Please see Chapter 4 for a protocol secure against active adversaries.

Each of the n parties $P_1, \dots, P_n$ has a memory M_i that maps variable names to values. We assume that the parties communicate over secure and authenticated pair-wise channels using a semantically secure public-key cryptosystem and that the public keys have been distributed securely in advance. In practice, each party could obtain a SSL certificate from a known certificate authority (CA) and announce this certificate to the others before the computation starts. When keys have been distributed the different commands are implemented as follows:

Input: When $\mathcal{Z}$ sends

$$(x := v, pc),$$

to P_i, P_i secret shares the value v into shares $v_1, \dots, v_n$ using a Shamir secret sharing [92] with threshold t. The shares are sent securely to

The ideal functionality $\mathcal{F}_{\text{VIFF}}$ accepts the following commands:

Input: The parties provide input to the computation by assigning values to variables. Variables spring into life upon their first assignment and remain defined for the rest of the protocol execution. A variable is defined when $\mathcal{F}_{\text{VIFF}}$ receives a valid command of the form

$$(x := v, pc)$$

from some party P_i. The functionality reacts by storing $x \mapsto v$ in its memory M. The adversary is simply told that P_i sent $(x := ?, pc)$.

Output: Variables can be output to reveal their value to a particular party. If x is a previously defined variable and $\mathcal{F}_{\text{VIFF}}$ receives

$$(\texttt{output}, x, P_i, pc)$$

from at least $t + 1$ parties, then $\mathcal{F}_{\text{VIFF}}$ waits for S to send $(\texttt{ok}, pc)$ before it sends $(\texttt{ok}, M(x), pc)$ to P_i.

Linear combination: To store a linear combination of previously defined variables $x_1, \dots, x_j$ with publicly known constants $c_1, \dots, c_j$ in x, all P_i send

$$(x := c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_j \cdot x_j, pc)$$

to $\mathcal{F}_{\text{VIFF}}$. This makes $\mathcal{F}_{\text{VIFF}}$ store the assignment $x \mapsto \sum_{i=1}^{j} c_i \cdot M(x_i)$ in its memory. Please note that this command covers simple addition when all $c_i = 1$.

Multiplication: When $\mathcal{F}_{\text{VIFF}}$ has received

$$(x := y \cdot z, pc)$$

from all P_i where y and z are already defined, $\mathcal{F}_{\text{VIFF}}$ stores the assignment $x \mapsto M(y) \cdot M(z)$ in its memory.

Synchronization: By sending

$$(\texttt{synchronize}, pc)$$

to all P_i, the environment ask the parties to synchronize. Synchronization is a tool for $\mathcal{Z}$ to structure its meta-program, and $\mathcal{F}_{\text{VIFF}}$ needs to do nothing here — it is handled entirely by $\mathcal{A}$, as we will see later.

Figure 3.8: Core VIFF ideal functionality called $\mathcal{F}_{\text{VIFF}}$.

the other parties, i.e., P_i sends $(E_{pk_j}(v_j), pc)$ to each P_j where $j \neq i$. When all shares has been sent, P_i stores $x \mapsto v_i$ in M_i and outputs $(\mathtt{ok}, pc)$ to $\mathcal{Z}$.

The other parties P_j store $x \mapsto v_j$ in M_j and output $(\mathtt{ok}, pc)$ to $\mathcal{Z}$ when they have received the share from P_i.

Output: To open x to P_i the environment sends

$$(\mathtt{output}, x, P_i, pc),$$

to at least $t + 1$ parties. Party P_j sends its share of x securely to P_i and output $(\mathtt{ok}, pc)$ to $\mathcal{Z}$. When receiving $t + 1$ shares, P_i will reconstruct the value v stored in x and output $(\mathtt{ok}, v, pc)$ to $\mathcal{Z}$.

Linear combination: Receiving

$$(x := c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_j \cdot x_j, pc)$$

where all x_k are defined will make P_i store $x \mapsto \sum_{k=1}^{j} c_k \cdot M_i(x_k)$ in its memory and output $(\mathtt{ok}, pc)$ to $\mathcal{Z}$.

Multiplication: The environment sends

$$(x := y \cdot z, pc),$$

to all parties. When receiving the command and when both shares are defined, each party runs the multiplication protocol by Gennaro et al. [51]. In this protocol, each party starts by locally multiplying the two shares y_i and z_i to get a temporary share x'. The share x' correspond to a Shamir sharing of $y \cdot z$ with threshold $2t$.

This share is reshared by having each P_i run the same commands as if it had received the command $(x_i := x', pc_i)$ where pc_i is a fresh program counter derived from pc and i. Each party P_i recombines its shares $x_1, \ldots, x_n$ into its share of x using a threshold of $2t$. When a party has reconstructed its share of x, it outputs $(\mathtt{ok}, pc)$ to $\mathcal{Z}$.

Synchronization: When receiving

$$(\mathtt{synchronize}, pc),$$

P_i will send $(\mathtt{ready}, pc)$ to all other parties. When P_i has received such ready messages from all parties, it outputs $(\mathtt{ok}, pc)$ to $\mathcal{Z}$.

This should be seen as “half” a Byzantine agreement. When P_i outputs $(\mathtt{ok}, pc)$, it knows that everybody else is ready too. But P_i cannot be certain that the other parties have received the $(\mathtt{ready}, pc)$ sent by itself — the mutual agreement is missing. However, for our purposes this will suffice.

3.4.2 Simulator

Having described how we realize $\mathcal{F}_{\text{VIFF}}$, we must now show how we can simulate the protocol. This done on a case by case basis by describe how the simulator reacts to messages from the ideal functionality in order to pose as the adversary towards the environment. In each case S will simulate the messages seen by $\mathcal{A}$ in the real world execution, and simply pass on the output from $\mathcal{A}$ (if any) to $\mathcal{Z}$.

Input: When S receives $(x := ?, pc)$ from P_i it picks random values $r_1, \dots, r_n$ and encrypts each value $c_j = E_{pk_j}(r_j)$. The simulator gives these ciphertexts to $\mathcal{A}$ with P_i as the sender and P_j as the receiver. If $\mathcal{A}$ produces an output, this is sent to $\mathcal{Z}$. The simulator sends $(\texttt{ok}, pc)$ back to P_i (through $\mathcal{F}_{\text{VIFF}}$).

When $\mathcal{A}$ delivers c_j to P_j, S sends $(\texttt{ok}, pc)$ to P_j (again through $\mathcal{F}_{\text{VIFF}}$).

Assuming a semantically secure cryptosystem, the output produced by $\mathcal{A}$ in the ideal world will match the output produced in the real world and $\mathcal{Z}$ is thus unable to distinguish between the two cases.

Output: We will start by considering the case where x is opened to an honest party P_i. The simulator is told by $\mathcal{F}_{\text{VIFF}}$ when each P_j receives the $(\texttt{output}, x, P_i, pc)$ message. The simulator invents a share r_j at random and encrypts it to get $c_j = E_{pk_i}(r_j)$. The simulator gives c_j to $\mathcal{A}$ on behalf of P_j for delivery to P_i. Any output by $\mathcal{A}$ is sent to $\mathcal{Z}$.

The encryptions received by $\mathcal{A}$ in the real world contain Shamir shares, which (when looking at up to t shares) are uniformly random numbers. The simulator can therefore simulate this perfectly towards the adversary by encrypting random numbers. When S sees that $\mathcal{A}$ has delivered all shares to P_i, it sends $(\texttt{ok}, pc)$ to $\mathcal{F}_{\text{VIFF}}$ in order to release the pending $(\texttt{ok}, v, pc)$ message to P_i.

If the party P_i is corrupt, S must work a little harder. It starts by sending $(\texttt{ok}, pc)$ to $\mathcal{F}_{\text{VIFF}}$ in order to obtain v from $\mathcal{F}_{\text{VIFF}}$ (S receives messages from $\mathcal{F}_{\text{VIFF}}$ sent to the corrupt P_i). The adversary knows (by previous simulation) the share v_i belonging to P_i. This share was chosen at random by S without knowing v. But because $\mathcal{A}$ can only corrupt up to t players, there will be at least one share which is unknown to $\mathcal{A}$ but needed for reconstructing v.

With the knowledge of v and the shares possessed by $\mathcal{A}$, S chooses consistent shares for the honest parties and sends them securely to $\mathcal{A}$. By sending the shares in the same order as the $(\texttt{output}, x, P_i, pc)$ messages arrive, the simulator ensures that $\mathcal{A}$ sees the same arrival order as in the real world and which makes the view of the adversary indistinguishable from the real world view.

Linear combination: In the real world no communication is done, and the adversary sees no messages from the parties. The simulator should therefore do nothing in the ideal world.

Multiplication: In the real world $\mathcal{A}$ sees the communication produced by the resharing. Those messages are produced by the result of the parties executing the same steps as if they had received $(x_i := x', pc_i)$ and it can be simulated in the same way as a normal assignment.

Synchronization: When synchronizing, the only communication produced is the $(\texttt{ready}, pc)$ messages sent by each party to the other parties and the final $(\texttt{ok}, pc)$ message sent to $\mathcal{Z}$ when a party hears that all other parties are ready.

To simulate the messages sent in response to a $(\texttt{synchronize}, pc)$, the simulator sends a $(\texttt{ready}, pc)$ message to $\mathcal{A}$ from P_i when it learns that P_i has received $(\texttt{synchronize}, pc)$ from $\mathcal{Z}$.

When $\mathcal{A}$ has delivered $(\texttt{ready}, pc)$ message from all other parties to P_i, then S sends $(\texttt{ok}, pc)$ to P_i. This is a perfect simulation of the real world.

The correctness of this simulation was argued above and it is also clear that the simulator is efficient. The simulator uses $\mathcal{A}$ as a black-box and thus works for any $\mathcal{A}$. It produces a view for $\mathcal{Z}$ that is identical in the two worlds, except for the possibility that the semantically secure cryptosystem breaks. We conclude that the real world protocol is secure with regard to $\mathcal{F}_{\text{VIFF}}$.

Having a mathematical model for a secure cryptographic protocol that can be used for multiparty computations is only the first step towards making useful computations. Implementing the model is the next step and will be described in the following.

3.5 VIFF Implementation

VIFF is implemented in the Python programming language developed by van Rossum et al. [96] in the early 1990s. Python is multi-paradigm language with an equal focus on procedural and object-oriented programming. Support for anonymous and higher-order functions make some functional programming possible too.

The choice of Python was largely driven by the desire to have a flexible language for rapid prototyping and by the need for a good library for asynchronous network communication. Like many languages, Python comes with a standard library that gives access to sockets for doing network communication. Twisted [65] is a Python framework that abstracts the low-level socket communication away and allows the programmer to easily build efficient network applications with asynchronous communication.

3.5.1 Twisted Network Library

It turns out that the features provided by Twisted for asynchronous network communication fit exceedingly well for our purposes. We will now explain exactly what we mean by “asynchronous” since this will be a central theme in the coming.

When program read and write from files and network sockets (or perform any other kind of I/O), they can do so in can do so in one of two modes: blocking (synchronous) or non-blocking (asynchronous). This refers to the behavior of the system calls, which can either block the execution of the program until all data has been read or written; or they can avoid blocking and let the program continue. Write calls are often non-blocking because the program can continue after passing the data onto the operating system. The OS will buffer the data and make sure that it is eventually written to disk or sent it over the network.

When reading data, most programs block while the operating system copies the available data into a memory buffer allocated by the program. This is the natural way to structure things for the vast majority of programs, which operate in a sequential fashion: read some files, processing the data, write it to a database, read some more, etc. When a program, e.g., a webserver needs to read from multiple network sockets concurrently it can do so by spawning a thread for each socket. Threads carry a non-trivial overhead and accesses to shared state require careful synchronization. Synchronization introduce additional overheads. An alternative is to use non-blocking reads and this is the approach taken by Twisted.

In a world of blocking I/O, one could imagine a function called getPage, which retrieves a webpage and returns the HTML code of the page. We can use it like this:

```
html = getPage("http://example.net/")
```

The program blocks until getPage returns and html is now a string with the HTML code retrieved. Fetching multiple pages by calling getPage multiple times becomes a sequential operation. In Twisted — in the world of non-blocking I/O — getPage does not block the rest of the program. Instead it returns a *deferred* result immediately.

Deferred values are a key concept in the Twisted framework. The concept is also known as *promises* or a *futures* in other asynchronous libraries. A deferred is a placeholder for a future value, it is a promise of something that has yet to happen. The only way to use a deferred is to say: "when you are ready, please call this function." Functions attached to deferreds in this way are known as *callbacks*. In practice you use a deferred like this:

```
def print_contents(contents):
    print "The Deferred has called us with:"
    print contents

yahoo = getPage("http://yahoo.com/")
google = getPage("http://google.com/")

yahoo.addCallback(print_contents)
google.addCallback(print_contents)
```

The getPage function returns a Deferred which will eventually hold the HTML of the fetched page. As mentioned above, we are working with non-blocking

I/O, so getPage returns immediately and gives us a Deferred. When the HTML arrives, the Deferred will invoke its callbacks in sequence. Here we retrieve the front pages of two well-known search engines and attach a callback to each Deferred. Depending on which page arrives first, the HTML code from either Yahoo! or Google will be printed first.

As can be seen, callbacks are simply function pointers. If several callbacks are added to the same Deferred, then they will be invoked one by one when the Deferred receives its value. The callbacks function as a pipeline where the return value from one callback becomes the first argument of the next. If the concept of deferreds still seem foreign, then note that the same programming style is well-known from graphical user interfaces where the programmer also attaches callbacks to certain events (button click, mouse over, etc.) in the user interface. Interactive websites that make use of AJAX (*Asynchronous* JavaScript And XML) are also using callbacks extensively.

3.5.2 Deferreds in VIFF

VIFF aims to be usable by parties connected by real world networks, the kinds we described in Section 3.3.2. Such networks are all asynchronous by nature, which means that no upper bound can be given on the message delivery time. A well-known example is the Internet where the communication from A to B must go through many hops, each of which introduces an unpredictable delay. Targeting networks with this kind of worst-case behavior from the beginning means that VIFF works well in all environments, including local area networks which typically behave in a much more synchronous manner.

To cope with the asynchronous setting the VIFF runtime system tries to avoid waiting unless it is explicitly asked to do so. In a synchronous setting all parties wait for each other at the end of each round, but VIFF has no notion of "rounds". What determines the order of execution is solely the inherent dependencies in a given program. If two parts of a program have no mutual dependencies, then their relative ordering in the execution is unpredictable. This assumes that the calculations remain secure when executed out-of-order. Protocols written for asynchronous networks naturally enjoy this property since the adversary can delay packets arbitrarily, which makes the reordering done by VIFF a special case.

The Twisted framework and the Deferred class in particular allow VIFF to handle the asynchronous setting nicely. The Deferred class is used extensively via a subclass called Share. Using suitable operator overloading we are able to allow the programmer to do arithmetic with Share objects and so treat them like normal numbers. Key to the implementation of VIFF is a function gather_shares which takes a list of Share objects as input and returns a new Share. This Share will call its callbacks when all Share objects in the list have called their callbacks. We use this to make Share objects that *wait* on other Share objects.

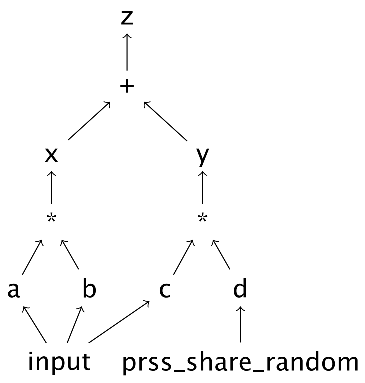
As an example, consider the simple program in Figure 3.9a for three players. It starts by defining the field $\mathbb{Z}_{1031}$ where the toy-calculation will take place and a list with the IDs of all players. The user is then prompted for input (an integer). All three players then take part in a Shamir sharing of

```
# (Standard setup not shown.)
Zp = GF(1031)

input = int(raw_input("Your input: "))
a, b, c = rt.input([1, 2, 3], Zp, input)
d = rt.prss_share_random(Zp)

x = a * b
y = c * d
z = x + y
```

(a) VIFF program.



(b) Expression tree.

Figure 3.9: A small toy-example written for VIFF and the corresponding expression tree. Each of the three player provides a private input and a forth number is shared with PRSS. They continue by multiplying and adding the secret shared numbers.

their respective inputs, this results in three Share objects being defined. A fourth Share object is generated using pseudorandom secret sharing [29].

Here all variables represent secret-shared values. The execution of the above calculation is best understood as the evaluation of a tree, please see Figure 3.9b. Arrows denote dependencies between the expressions that result in the calculation of the variable z.

The two variables x and y are mutually independent, and so one cannot reliably say which will be calculated first. But more importantly: we may compute x and y in *parallel*. It is in fact very important for efficiency reasons that we calculate x and y in parallel. The execution time of a multiparty computation is limited by the speed of the CPUs engaged in the local computations and by the delays in the network. Network latencies can easily reach a hundred milliseconds or more, and will typically dominate the running time.

When we say *parallel* we mean that when the calculation of x is blocked and waits on network communication from other parties, then it is important that the calculation of y gets a chance to begin its network communication. This puts maximum load on both the CPU and the network.

3.5.3 Automatic Parallelism

The big advantage of this system is that it automatically runs the operations in parallel: The calculations implicitly create the tree shown in Figure 3.9b, and this tree is destroyed as fast as possible when operands become ready. There is no predefined execution order in the tree — it is deconstructed from the leaves inwards at the same speed as the needed operands arrive.

Please note that this simple system for parallel scheduling automatically extends to all levels in the program, i.e., from primitives like addition and multiplication up to compound protocols like comparisons or even entire auctions. This is a very important consequence since it allows us to build a larger protocol π by combining smaller protocols and still be sure that several instances of π can be executed in parallel. It is the uniform interface enforced by always working on deferred values which enables this kind of modularity. At runtime, a new protocol π will simply correspond to a subtree in Figure 3.9b and its leaf nodes will be executed in parallel with all other leaf nodes in the tree.

This mode of executing changes the semantics of a program using VIFF from that of a normal Python program. Each statement is no longer executed when it is encountered, it is merely scheduled for execution and then executed later when the operands are available. The semantics of a program using VIFF is thus more like that of a declarative programming language where you declare your intentions but where the compiler takes care of scheduling the calculations in the optimal order.

3.5.4 Tracking Asynchronous Operations

The above description of the abstract mathematical protocols has not concerned itself with how the program counters are selected. It merely stipulated that program counters should be unique for a given protocol run. This is in fact the typical level of detail devoted to such "implementation details" in most protocol descriptions.

However, selecting unique and consistent program counters is crucial for the correct operation of the protocols and we had to design a general system for doing so when implementing VIFF. We will now describe the technique used to pick program counters in VIFF. We believe this technique can be used in other concurrent settings as well.

Since VIFF programs are also normal Python programs, we seek a solution that will integrate seamlessly with Python and in particular work with the dynamic nature of Python programs. Had we used a dedicated (and suitably restricted) programming language for writing our programs, we could probably have used static analysis to assign program counters to each statement. The Python programs that use VIFF as a library are, however, completely general and do not lend themselves very well to static analysis. Our solution is therefore based on decisions made at runtime.

The basic assumption is that the parties are executing the same program. Any disagreement on the protocol to execute must be resolved via an external mechanism. The program makes calls to send and receive data from the network, and it is those calls that we must pair up. The calls are asynchronous, meaning that we do not wait to receive an answer before we proceed with the program.

The Problem

Consider the following program fragment, where we assume that the shares a, b, c, and d have already been correctly defined:

```
x = a * b
y = c * d
```

The overloaded multiplication operator leads to a call to the mul method of the Runtime instance used. The mul method is roughly defined as follows:[3]

```
def mul(self, share_a, share_b):
    result = gather_shares([share_a, share_b])
    result.addCallback(lambda (a, b): a * b)
    result.addCallback(share_recombine)
    return result
```

The share_recombine function takes care of re-sharing the product and recombining the shares received from the other parties. We must therefore ensure that the two calls to share_recombine match up: when P_1 executes share_recombine in response to the arrival of a and b, then it should use the same program counter as P_2 and P_3 does when they receive a and b.

A simple solution is to keep a single counter per party. The parties increment it per multiplication:

```
def mul(self, share_a, share_b):
    self.program_counter += 1
    result = gather_shares([share_a, share_b])
    result.addCallback(lambda (a, b): a * b)
    result.addCallback(share_recombine, self.program_counter)
    return result
```

Here we increment the program counter at the beginning of the method, and pass the updated value as an extra argument to the share_recombine function. It would not have worked to simply let share_recombine access self.program_counter since we cannot predict when share_recombine will be called — the program counter will almost certainly have been updated when share_recombine is finally called.

Though this technique solves the problem for this particular case, it is not very general. The big problem is that it is impossible to maintain a system using this solution:

- Updating the program counter manually is tedious and error prone. The programmer ends up weaving the program counter state into all functions, which smells of bad design.

- What happens if share_recombine needs more than one program counter? The mul method would presumably have to reserve them up front.

- What happens if we change the implementation of a primitive protocol? All callers would presumably have to recompute how many program counters they need to reserve.

[3]We have removed the code that handles local multiplication by constants and the code for share_recombine is not shown.

The Solution

We can solve the first program by changing how addCallback works, or rather, by providing a substitute function which will take care of incrementing the program counter:

```
def schedule_callback(self, deferred, func, *args, **kwargs):
    self.program_counter += 1
    saved_pc = self.program_counter

    def callback_wrapper(*args, **kwargs):
        try:
            current_pc = self.program_counter
            self.program_counter = saved_pc
            return func(*args, **kwargs)
        finally:
            self.program_counter = current_pc

    deferred.addCallback(callback_wrapper, *args, **kwargs)
```

This is a method on Runtime. The schedule_callback method takes a Deferred and a callback function as arguments. It increments the current program counter and stores it away in its local saved_pc variable. It then defines a local function. The function wraps the call to func in code that will temporarily set the program counter to saved_pc. It is this inner function which is added as a callback to deferred.

The schedule_callback method relieves the rest of the code from having to update the program counter and to pass it as an explicit argument to callback functions. We can use it like this:

```
def cb(ignored):
    print "callback:", rt.program_counter

d1 = Deferred()
d2 = Deferred()

print "main:", rt.program_counter
rt.schedule_callback(d1, cb)
print "main:", rt.program_counter
rt.schedule_callback(d2, cb)
print "main:", rt.program_counter

d1.callback(None)
d2.callback(None)
```

When executed, the code will give the following output:

```
main: 0
main: 1
main: 2
```

```
callback: 1
callback: 2
```

Notice how the callback functions now simply use the global program counter on the Runtime object instead of managing it explicitly. The exact details of how the program counter is updated have also been hidden — the program counter is now simply an opaque object to the callback function.

This is only half a solution, however, since each callback only receives a single program counter. We will solve this problem next by extending the above code slightly.

We want to give each callback function its own isolated space of program counters. We do this by using a *list* of numbers. A program counter is now a vector of numbers, such as [1, 0, 2]. The list defines a simple hierarchy: we can derive a sub-program counter by appending a digit to get [1, 0, 2, 0], which we can pass it to another function. This function must now keep the first three digits fixed, but will otherwise have control over the "name space" offered by the [1, 0, 2] prefix. Changing the implementation of a function will no longer affect the callers of the function. They just need to pass the function a freshly derived sub-program counter.

The code for schedule_callback changes only slightly:

```
def schedule_callback(self, deferred, func, *args, **kwargs):
    self.program_counter[−1] += 1
    saved_pc = self.program_counter[:]

    def callback_wrapper(*args, **kwargs):
        try:
            current_pc = self.program_counter[:]
            self.program_counter[:] = saved_pc
            self.program_counter.append(0)
            return func(*args, **kwargs)
        finally:
            self.program_counter[:] = current_pc

    return deferred.addCallback(callback_wrapper, *args, **kwargs)
```

Incrementing the program counter has been turned into an increment of the last digit in the program counter and the func is now executed with its own sub-program counter, allocated by appending a zero to the saved program counter. The strange program_counter[:] syntax indicates list copying and in-place assignment.

The program counter starts with the value [0] and our example code from before now outputs:

```
main: [0]
main: [1]
main: [2]
callback: [0, 0]
callback: [1, 0]
```

$$\begin{tikzcd}
{[1, 2, 3]} \arrow[d] \\
{[1, 2, 4]}
\end{tikzcd}$$

(a) Simple increment.

$$\begin{tikzcd}
{[1, 2, 3]} \arrow[d] \arrow[dr] & \\
{[1, 2, 4]} & {[1, 2, 3, 0]}
\end{tikzcd}$$

(b) Spawning a sub-program counter.

Figure 3.10: Updating the program counter. The final digit is always available for update and we can get a fresh program counter by incrementing it. Creating a sub-program counter both increments the original program counter (to ensure that allocating multiple sub-program counters yield unique program counters) and adds an extra digit to the program counter (to give the callback its own name space).

The first callback function is free to allocate new program counters beneath the [0] name space and the second can allocate new counters below [1].

A final detail: Some functions (such as those related to PRSS) depend on a unique program counter on each invocation, even though they are run synchronously (not via a callback). These functions simply increase the last digit in the current program counter:

```
rt.program_counter[−1] += 1
```

Figure 3.10 illustrates the two different ways of incrementing the program counter.

Malicious Adversaries

The above description could make it sound as if we are dependent on an honest but curious adversary, who will follow the protocol and pick program counters faithfully. This is not the case. If a malicious adversary picks incorrect program counters, the honest parties will simply store the incoming data. This can lead to excessive memory usage, but is otherwise harmless. We have some suggestions for dealing with this in Section 7.2.3.

3.6 Mapping the Protocol to VIFF Methods

VIFF offers a set of commands closely resembling the MPC commands described in Section 3.4.1. For the description of the VIFF commands, let rt be an instance of the Runtime class and that x, y, and z are instances of the Share class. We will describe the commands with three parties for concreteness, but this is not a limit of the VIFF runtime.

3.6.1 Standard Commands

We begin by describing the commands that map directly to commands from Section 3.4.1.

Input: The Runtime.input method is used for providing private input to the computation. In the PassiveRuntime and ActiveRuntime classes it is implemented by Shamir sharing values over $\mathbb{Z}_p$. The two player PaillierRuntime uses additive secret sharing instead.

The method command is often used in a symmetric way where all parties wish to supply their input to the computation. For this reason, the method takes a parameter that indicates the *list* of parties who give input and return a *tuple* containing a share for each input given.

As an example, consider three parties P_1, P_2, and P_3 who all wish to give input. They must all execute:

```
x, y, z = rt.input([1, 2, 3], Zp, $v_i$)
```

Each party uses its own private input value in place of v_i. The result is a triple, which we can assign directly to three different variables.

This corresponds directly to the environment sending

$$(x := v_1, pc_1), \qquad (y := v_2, pc_2), \qquad (z := v_3, pc_3)$$

to P_1, P_2, and P_3, respectively.

All players must call the method, even if they do not provide any input. This is because they still need the shares sent by the players who do provide input. If only P_1 gives an input, the code looks like this:

```
if rt.id == 1:
    x = rt.input([1], Zp, $v_1$)
else:
    x = rt.input([1], Zp)
```

Output: Executing the command

```
open_x = rt.output(x)
```

will open x to all parties. This corresponds to the $\mathcal{Z}$ sending

$$(\texttt{output}, x, P_i, pc_i)$$

to all parties $P_1, \ldots, P_n$ and for all i — all parties broadcast their share to all other parties.

As for input, output can be sent to a subset of the parties. This is done by specifying an optional third argument. If only P_2 and P_3 is to receive the output, then all parties execute:

```
open_x = rt.output(x, [2, 3])
```

For P_1, open_x will have the value None.

Linear combination: Forming a linear combination of shares x, y, and z using coefficients a, b, and c can be done by executing

```
w = a * x + b * y + c * z
```

and this maps directly to $\mathcal{Z}$ sending the command

$$(w := a \cdot x + b \cdot y + c \cdot z, pc)$$

to all parties. This involves no communication between the parties.

Multiplication: Shares can be multiplied by executing

```
z = x * y
```

and this maps directly to $\mathcal{Z}$ sending the command

$$(z := x \cdot y, pc)$$

to all parties. As in the real world model, this involves a resharing.

Synchronization: Executing a function f after synchronizing is done by

```
sync = rt.synchronize()
sync.addCallback(f)
```

This corresponds to the environment sending

$$(\texttt{synchronize}, pc)$$

to all parties and then executing f when all parties are ready.

3.6.2 Additional Commands

In addition to these primitive commands, VIFF provides a number of higher-level commands. They use the primitives described above and are thus secure since the primitives themselves are secure. The commands are:

Exclusive-or: If x and y are bit values, the exclusive-or can be calculated by

```
z = x ^ y
```

This simply calculates $z = x + y - 2 \cdot x \cdot y$ when x and y are shares from a field $\mathbb{Z}_p$. When the shares represent values from $\mathbf{GF}(2^8)$, in which exclusive-or is simply addition, the call is made with no communication at all.

Exponentiation: A share x can be raised to an integer y with a simple

```
z = x ** y
```

The power y must be a public integer and the exponentiation is done using square-and-multiply.

Pseudorandom secret sharing: The runtime can create a secret sharing of a uniformly pseudorandom number using no communication by the technique of pseudorandom secret sharing described by Cramer et al. [29]. The parties simply execute

```
rand = rt.prss_share_random(field)
```

The field variable indicates the field, either $\mathbb{Z}_p$ or $\mathbf{GF}(2^8)$. The parties can share a particular value by doing

```
x, y, z = rt.prss_share(v_i)
```

each with their own value for v_i. The sharing costs a broadcast which needs not be encrypted. Pseudorandom secret sharing has no corresponding command in the real world model.

3.6.3 Commands from Mixin Classes

Wait, there is even more! We provide a number of *mixin classes*, which can be used to extend the basic VIFF runtime classes.

A mixin class provides only a few methods of its own and cannot be used in isolation. To use it, one defines a new class that combines the mixin functionality with basic functionality via multiple inheritance. We could have made the mixin classes "full" classes by letting them inherit directly from, say, PassiveRuntime. However, we would also like to see the functionality used with the BasicActiveRuntime class. This suggests that the final inheritance hierarchy should be determined by the application programmer, not by us. A design with Mixin classes give this flexibility.

Equality: The ProbabilisticEqualityMixin class can be mixed with a runtime to endow it with an equal method:

```
bit = rt.equal(x, y)
```

This makes bit a secret shared Boolean. The protocol used is by Nishide and Ohta [80] and was implemented by Sigurd Meldgaard.

Comparison: Secure comparison is provides via operator overloading:

```
bit = x > y
```

This makes bit a secret shared Boolean. VIFF provides two comparison protocols, one by Toft [94] which returns a secret shared $\mathbf{GF}(2^8)$ element, and another which returns a bit secret shared in $\mathbb{Z}_p$ (unpublished work by Tomas Toft).

The programmer selects the right comparison protocol by mixing his runtime with either ComparisonToft05Mixin for the older protocol returning a $\mathbf{GF}(2^8)$ element or with ComparisonToft07Mixin for the newer protocol returning a $\mathbb{Z}_p$ element.

3.7 Multiplication in VIFF

As an example of real VIFF code, we have included the implementation of the standard BGW multiplication protocol [10] with the optimization by Gennaro et al. [51]. The protocol is secure is secure against a semi-honest adversary corrupting up to $t = n/2$ of the parties. Please see Figure 3.11.

```
1   def mul(self, share_a, share_b):
        assert isinstance(share_a, Share) or isinstance(share_b, Share), \
3           "Either share_a or share_b must be a Share."

5       if not isinstance(share_a, Share):
            # Then share_b must be a Share => local multiplication.
7           # We clone first to avoid changing share_b.
            result = share_b.clone()
9           result.addCallback(lambda b: share_a * b)
            return result
11      if not isinstance(share_b, Share):
            # Likewise when share_b is a constant.
13          result = share_a.clone()
            result.addCallback(lambda a: a * share_b)
15          return result

17      # At this point both share_a and share_b must be Share objects.
        # We wait on them, multiply, reshare, and recombine.
19      result = gather_shares([share_a, share_b])
        result.addCallback(lambda (a, b): a * b)
21      self.schedule_callback(result, self._shamir_share)
        self.schedule_callback(result, self._recombine, 2*self.threshold)
23      return result
```

Figure 3.11: The standard multiplication protocol for passive adversaries.

The code handles both local multiplication and multiplication involving network traffic. First, if either share_a or share_b is a not a Share object, i.e., one of them is a constant integer or a FieldElement, then we do a quick local multiplication. Assume that share_a is the constant and share_b is the Share (lines 5–10). We cannot simply multiply share_a and share_b since share_b is a Deferred and might not have a value yet. The solution is to clone share_b and add a callback to it. This callback is simply a lambda expression (an anonymous function) that takes care of the correct multiplication when share_b eventually gets a value (line 9). The opposite case is handled in the same way (lines 11–15). If it is established that both share_a and share_b are Share objects we create a new Share which waits on both of them (line 19). We then add several callbacks: first we multiply, then we reshare, and finally we recombine. These three operations will be executed in sequence when both share_a and share_b have received their values due to incoming network traffic. The last two callbacks involve network traffic, and must be added using a more expensive mechanism to ensure agreement on the labels put on the data as it is sent over the network.

In all three cases the mul method returns result to the caller (lines 10, 15, or 23). Note that result probably does not have a value at this point, but result is a Share that we have prepared in such a way that it *will* receive the correct value at some point in the future. All VIFF methods follow this pattern.

3.8 VIFF Applications

VIFF has been used for several small and some larger applications. The largest applications are listed below.

3.8.1 Nordic Sugar

In Denmark, the production of sugar beet is managed by sugar beet contracts issued between farmers and Danisco, the only sugar beet processor on the Danish market. A sugar beet contract determines the quantity of sugar beet that a farmer is allowed to produce. Traditionally, sugar beet contracts have been traded between individual pairs of farmers. This has been done in spite of the fact that trading in a central market was known to increase the overall profit. A central market has, however, not been possible due to conflicting interests and lack of trust between the parties.

In January 2008 the first large scale secure multiparty computation was carried out, effectively solving this problem. This was done by the SIMAP research project [16]. In the summer of 2009 the same computation was successfully repeated, this time using VIFF.

The computation was a double auction in which the production rights for several thousand tons of sugar beets were traded. During the first weeks of the auction, several hundred Danish sugar beet farmers submitted their encrypted bids to a central database. Then the actual computation took place between three players:

- Nordic Sugar, the Danish sugar beet processor
- DKS, the consolidation of Danish sugar beet farmers
- Partisia, a Danish company specialized in secure multiparty solutions

The computation took about 15 minutes using three laptops connected by a LAN. Most of the computation time was spent converting the encrypted bids back into secret shares. The actual multiparty computation took only a couple of minutes. As a result, the sugar beet contracts could be traded at an optimal price without any sensitive information being disclosed.

Using secure multiparty computation, trading sugar beets using this kind of auction was possible without finding and paying a trusted third party to manage the auction. Such a trusted party would — if it could be found at all — probably have been quite expensive.

3.8.2 Distributed RSA

VIFF was also used by Mauland [71] to implemented a protocol for distributed RSA key generation [17]. The private key from an RSA key pair must be kept in a highly secure location (to prevent unauthorized persons from stealing it) but because we want to use the key, we cannot just write it on a piece of paper and store that in a safe.

This tension between high availability and high security makes a distributed solution attractive. The protocol by Boneh and Franklin [17] generates an RSA keys in a distributed fashion and ensures that the private key is never available in the clear to any given party. An attacker must break into all machines to learn the private key. Meanwhile, the parties can use their shares of the private key to securely decrypt messages encrypted under the public key.

Mauland [71] report that generating a 1,024-bit RSA key took about 30 minutes on average (the time varied between 7 seconds and 2.5 hours). We believe that these results can be improved by using the GMPY [70] library more aggressively.

3.8.3 Distributed AES

The Advanced Encryption Standard (Rijndael) block cipher turns out to have nice arithmetic properties which makes its computation by arithmetic circuits relatively fast. Damgård and Keller [30] have implemented a multiparty version of AES for VIFF.

The code is distributed with VIFF and lets users securely compute a secret shared AES encrypted ciphertext of a (possibly) secret shared plaintext with a (possibly) secret shared key. Encrypting a 128-bit block using a 128-bit secret shared AES key takes about 2 seconds using three machines equipped with 2.4 GHz Dual-Core AMD Opteron™ processors.

3.8.4 Secure Voting

Typical Internet voting systems store all votes in a single location. Vegge [97] used VIFF to implement a distributed voting system. The system removes the single point of failure by storing the votes in secret shared form among three servers. The voters will do the secret sharing on their own machine, encrypt the shares, and send the encrypted shares to a database. Each share is encrypted under the public key belonging to the computation server that will do the actual multiparty computation.

This project shows how VIFF can be integrated seamlessly with many other technologies. The user creates a poll on a website programmed in PHP and the votes are cast using a Java applet. The applet has the responsibility of encrypting the votes for the computation servers. When all voters have cast their vote, an XML-RPC message is sent to the Python program running on the servers. That program decrypts the shares and uses VIFF to compute the result of the poll.

3.9 Conclusion

We have presented a general, practical framework for secure multiparty computation. In this chapter, we focused on the core runtime, especially its formulation as a UC functionality and we described how this functionality is implemented as a Python library. We presented the asynchronous network library called Twisted, and explained how it plays a key role in making VIFF able to automatically run operations in parallel. The novel concept of using a vector to keep track of the parallel operations was also explained.

The VIFF architecture is flexible yet efficient and we will present benchmark results to support this in Chapter 4. We will also present a protocol secure against malicious adversaries in that chapter.

VIFF can be freely downloaded from `http://viff.dk/` and we hope others will expand it with more other protocols.

Chapter 4

Active Adversaries in VIFF

The previous chapter presented VIFF, the new framework for specifying secure multiparty computation. The focus was on passive adversaries. We will now present a protocol which offers security against active adversaries. The protocol was implemented and benchmarked using VIFF.

The protocol and implementation described here was presented at the PKC 2009 conference [37]. Section 4.4 is new and Section 4.5 has been updated significantly with newly made benchmark results.

4.1 Introduction

A general multiparty computation protocol is an extremely powerful tool that allows n parties to compute any agreed function $f(x_1, \dots, x_n)$, where each input x_i is privately held by the i'th player P_i, and where only the intended result becomes known to the players. The function is often represented by an arithmetic circuit C over some suitable finite field $\mathbb{F}$. It is required that privacy of the inputs and correctness of the result is ensured even in the presence of an adversary who may corrupt some number t of the players.

From the basic feasibility results of the late 80-ties [10, 24], it follows that any efficiently computable function may be computed securely in the model where players have secure point-to-point channels, if and only if $t < n/3$. In case the adversary is semi-honest, i.e., corrupted players follow the protocol, the bound is $t < n/2$. Under computational assumptions, $t < n/2$ corruptions can be tolerated even if the adversary is malicious, i.e., corrupt players behave arbitrarily [54].

The solution from [10] with passive security can lead to quite practical solutions, when combined with later techniques for optimizing the efficiency of particular primitives, such as integer comparison — even to the point where large-scale practical applications can be handled [16].

On the other hand, this type of solution is not satisfactory in all cases. It is of course desirable to provide security against active cheating. However, this usually incurs a large cost in terms of efficiency. Techniques have been proposed to reduce this cost [60], but they — like most previous protocols — are designed for synchronous communication. Common ways to communicate, such as the Internet, are arguably better modeled as asynchronous networks, where there is no guarantee that a message is delivered before a certain time. Note that the way we model the network can have dramatic consequences for the practical efficiency of a protocol. If we run a synchronous protocol on top of any real network, we are forced to make every round last enough time so that we can be sure that all messages from honest players have been delivered. Otherwise, we may conclude that an honest player is corrupt because he did not send the message he was supposed to, and take action accordingly. Now, of course, the protocol is no longer secure. It follows that, for instance, on a network that usually delivers messages fast, but occasionally takes much longer time, a synchronous protocol may be much slower in practice than an asynchronous one, where every player may continue as soon as he has enough information to do so.

Our goal was therefore to develop and implement a practical general MPC protocol, with security against malicious adversaries on an asynchronous network. Compared to the usual model for asynchronous MPC, we make two modifications, both of which we believe are well motivated:

- We allow our protocol to have one synchronization point. More precisely, the assumption is that we can set a certain time-out, and all messages sent by honest players before the deadline will also be delivered before the deadline.

- We do not guarantee that the protocol always terminates and gives output to all honest players. Instead we require the following: The preprocessing phase of the protocol, up to the synchronization point, never releases any new information to the adversary. The adversary may cause the preprocessing to fail, but if it terminates successfully, the entire protocol is guaranteed to terminate with output to all honest parties.

A discussion of this model: Without the first assumption, i.e., if the protocol is fully asynchronous, one cannot guarantee that all honest players will be able to contribute input since the protocol cannot distinguish between t corrupt players that have not sent anything, and t honest players whose messages have been delayed. We believe that in most practical applications, this is not acceptable, and this is why we introduce the synchronization point, it is a minimal assumption allowing all honest players to provide input. We stress that the protocol is *asynchronous both before and after the synchronization point*. In other words, a protocol in this model is free to harvest the efficiency gain that follows from players being able to proceed as soon as possible. The only constraint we put is that honest players must reach the deadline on time, so we can have agreement on whether the preprocessing succeeded.

On the second point, although we do give the adversary extra power to stop the protocol, this is arguably of no use in practice: if the corrupted players can only stop the protocol at a point where they have learned nothing new, they have very little incentive to do so.

In this model, assuming secure point-to-point channels and that Byzantine agreement is available, we present a protocol that is perfectly secure against an adaptive and active adversary corrupting less than $n/3$ of the players. The communication and computational complexities (total communication and work done) are $\mathcal{O}(n^2|C|k)$ where $|C|$ is the size of the arithmetic circuit being computed and k is the bit length of elements in the field used. It is noteworthy that a straightforward implementation with only passive security would have the same asymptotic complexity, all other things being equal.

As for any protocol in the point-to-point model, the exact security properties of an actual implementation of our protocol depend on how the point-to-point channels and — in our case — the Byzantine agreement are implemented. The choice of implementation does not, however, affect the complexities since the Byzantine agreement is only used once. In a typical implementation where one goes for efficiency — such as the one we present below — one would use standard encryption tools to implement the channels and do the Byzantine agreement based on public-key signatures. This gives a protocol with computational security against a static adversary (also, such an implementation is not known to be insecure against an adaptive adversary).

In recent concurrent work, Hirt et al. [61] construct an asynchronous protocol of similar asymptotic complexity as ours. This protocol is fully asynchronous, so it does not guarantee that all honest parties can provide inputs, and it is computationally secure against a static adversary. In another recent work Beerliová-Trubíniová et al. [8] present a protocol with a single synchronization point like we have. This protocol guarantees termination, has a better security threshold ($n/2$), but is only computationally secure against a static adversary, and has larger asymptotic complexity than our protocol.

Regarding efficiency in practice, we stress that although our implementation is only computationally secure, it is an advantage, also from a practical point of view, that the basic protocol is information theoretic, because the tools used (secret sharing etc.) are much more efficient than computationally secure alternatives, such as homomorphic public-key encryption. Such techniques are used in both [8] and [61], making them much less efficient in practice than our construction.

Thus, our result is incomparable to previous work, and we believe it provides a new trade-off between security properties that is attractive in practice. We later give more exact numeric evidence of the efficiency.

Our protocol is based on Beaver's well known circuit randomization techniques, where one creates in a preprocessing phase shared random values a, b, c with $ab = c$. We show two techniques for generating these triples efficiently. One is a variant of the protocol from [7], the other is based on pseudorandom secret sharing [29], it is much faster for a small number of players, but only gives computational security. Both protocols are actually synchronous, but we handle this via a new technique that may be of independent interest, namely a general method by which — if one accepts that the protocol may abort — a synchronous protocol can be executed in an asynchronous fashion, using a single synchronization point to decide if the protocol succeeded.

A crucial observation we make is that if the protocol is based on Shamir secret sharing with threshold less than $n/3$, then the computation phase can be done asynchronously and still guarantee termination, if the preprocessing succeeded.

A final contribution of this chapter is an implementation of our protocol in VIFF. The protocol was designed with an eye towards an efficient implementation in a system like VIFF, which is basically asynchronous and operates on the principle that players proceed whenever possible (but can handle synchronization points when asked to do so).

4.2 Overview and Security Model

The goal of the protocol is to securely compute $(y_1, \dots, y_n) = f(x_1, \dots, x_n)$. For notational convenience we assume that all inputs and outputs are single field elements. In addition each y_i can assume the value $y_i = \perp$, which indicates to P_i that the computation failed.

When in the following, we say that x is *publicly reconstructed* from $[x]_t$, where at most $t < n/3$ players are actively corrupted, this simply means that each player sends his share to all other players. This allows all honest players to reconstruct x using standard decoding techniques since $t < n/3$. We may also *privately open* x to player P_i by sending shares only to him.

4.2.1 Overview of the Protocol

Our protocol consists of two phases, the *preprocess and input phase* and the *computation phase*.

Preprocessing and input phase.

In the preprocessing phase, we can make use of any protocol that can generate a given number of multiplication triples, i.e., random secret-shared values $[a]$, $[b]$, $[c]$ where $ab = c$. In addition, for each player P_i, it should construct a secret sharing $[r_i]$ where r_i is random and reveal r_i privately to P_i. The protocol ends by outputting "success" or "failure" to all players, depending on whether the required values were successfully constructed or not. The purpose of $[r_i]$ is to allow P_i to contribute his input x_i securely by broadcasting $r_i + x_i$.

Instead of attempting to build such a protocol directly for the asynchronous model, it is much easier to design a protocol for the synchronous model with broadcast, we give two examples of this in Section 4.3. We then show below a special way to run any such protocol in an asynchronous way, i.e., we can avoid the use of timeouts after each communication round and we avoid having to implement broadcast. The price we pay for this is that the adversary can force the preprocessing to fail.

The basic idea is that in each round all parties just wait for messages from all other parties and progress to the next round immediately if and when they all arrived. Some extra technicalities are needed to make sure there is agreement at the end on whether the preprocessing succeeded, and to make sure that no information on the inputs is revealed prematurely.

To emulate a synchronous protocol with R rounds, each P_j proceeds as follows:

1. Wait for an input (`begin preprocess`). Let $r = 1$ and for each P_i compute the message $m_{j,i,1}$ to be sent to P_i in the first round of the synchronous protocol. Also compute the message $m_{j,1}$ to be broadcast in the first round.

2. Send $(m_{j,i,1}, m_{j,1})$ to P_i.

3. While $r \leq R$:

   (a) Wait until a message $(m_{i,j,r}, m_{i,r})$ arrived from all P_i.

   (b) From the incoming messages
   $$((m_{1,j,r}, m_{1,r}), \dots, (m_{n,j,r}, m_{n,r}))$$
   compute the messages
   $$(m_{j,1,r+1}, \dots, m_{j,n,r+1})$$
   that the preprocessing protocol wants to send in the next round, and the message $m_{j,r+1}$ to be broadcast.

   (c) $r := r + 1$.

4. Let $g_j \in \{$`preprocess success`$,$`preprocess failure`$\}$ denote the output of the preprocessing protocol and let M_j consist of the broadcast messages $m_{i,r}$ for $i = 1, \dots, n$ and $r = 1, \dots, R$.

   Send $($`check`$, g_j, M_j)$ to all parties.

5. Wait until all $n-1$ other parties P_i send $($`check`$, g_i, M_i)$. If all P_i sent $g_i =$ `preprocess success` and $M_i = M_j$, then send $s_j = x_j + r_j$ to all parties.

6. Wait to receive s_i from all other parties, let $S_j = (s_1, \dots, s_n)$ and send S_j to all parties.

7. If all $n-1$ other parties P_i sent some S_i before the timeout and all $S_i = S_j$, then let $q_i =$ `success`. Otherwise, let $q_i =$ `failure`.

8. Run a Byzantine agreement (BA) on the q_i to agree on a common value $q \in \{$`failure`$,$`success`$\}$. Being a BA this protocol ensures that if $q_i =$ `success` for all honest parties, then $q =$ `success`, and if $q_i =$ `failure` for all honest parties, then $q =$ `failure`.

We assume that the preprocessing phase is started enough in advance of the time-out to guarantee that it will terminate successfully on time when there is no cheating. However, as mentioned in the introduction, the adversary can stop the preprocessing, in particular if a corrupted party does not send a message the preprocessing dead-locks.

Note that if just one honest party outputs $q_i = \texttt{success}$, then the preprocessing protocol terminated successfully before the timeout and all the values s_i were consistently distributed. In particular, if $q = \texttt{success}$, then $q_i = \texttt{success}$ for at least one honest P_i, and therefore the preprocessing and inputting were successful.

As for security, if after each communication round in Step 3 the parties compared the messages $m_{i,r}$ and terminated if there was disagreement, then it is clear that a secure synchronous protocol[1] run asynchronously this way is again secure. The only loss is that the adversary can now deprive some parties of their input. The reason why it is secure to postpone the check of consistency of the broadcasted message until Step 5 is that the inputs x_i do not enter the computation until Step 6 and that there are no other secrets to be leaked, like secret keys. Sending inconsistent broadcast messages before Step 6 will therefore yield no information leakage. After Step 5 it is known that the preprocessing was an emulation of a consistent synchronous execution, at which point it becomes secure to use the result r_i to mask x_i.

This way to emulate a synchronous protocol in an asynchronous environment is generic and does not just apply to our protocols here.

Computation phase.

If $q = \texttt{failure}$, then all parties output $y_i = \perp$. If $q = \texttt{success}$, then the parties compute $[x_i] = s_i - [r_i]$ for all P_i and run the asynchronous protocol described below which compute sharings $[y_i]$ of the outputs from the sharings $[x_i]$, making use of the multiplication triples from the preprocessing. Finally the shares of $[y_i]$ are sent privately to P_i which computes y_i.

We may assume that for each multiplication we have to do, a triple $[a], [b], [c]$ as described above is constructed in the preprocessing. To handle any arithmetic circuit describing the desired function, we then only need to describe how to deal with linear combinations and multiplications of shared values.

Linear Combinations: Shamir sharing is linear, and any linear function of shared values can therefore be computed locally by applying the same linear function to the shares.

Multiplication: Consider a multiplication gate in the circuit and let $[a]$, $[b]$, $[c]$ be the triple constructed for this gate. Assume we have computed sharings of the two input values $[x]$ and $[y]$, so we now wish to

[1]The synchronous security should be against a rushing adversary.

compute $[xy]$. Note that

$$\begin{aligned} xy &= ((x-a)+a)((y-b)+b) \\ &= de + db + ae + ab, \end{aligned}$$

where $d = x - a$ and $e = y - b$. We may now publicly reconstruct d and e, since they are just random values in $\mathbb{F}$. The product can then be computed locally as

$$[xy] = de + d[b] + [a]e + [c].$$

The overall cost of this multiplication is the cost of two public reconstructions and a constant number of local arithmetic operations.

A crucial observation is that this protocol (assuming the triples are given) can be executed in a completely asynchronous fashion, and is guaranteed to terminate. At each multiplication gate, each player simply waits until he has received enough shares of d and e and then reconstructs them. More precisely, we need that at least $n - t$ shares of each value have arrived, and that at least $n - t$ of them are consistent with some polynomial. Since there are $n - t$ honest players, $n - t$ consistent shares will eventually arrive. Moreover, if $n - t$ shares are found to be consistent, since $t < n/3$, these must include at least $t + 1$ shares from honest players, and so the correct value is always reconstructed. One can test if the conditions are satisfied using standard error correction techniques.

4.2.2 Security Model

The security of our protocol can be phrased in the UC framework [23]. For the protocol we assume the standard asynchronous communication model of the UC model, except that we let the timeout of P_i be called by the adversary by inputting (`timeout`) to that party, and that we assume secure point-to-point channels where the adversary can decide when a message sent is delivered. Our protocols are secure and terminate no matter when the timeouts are called. They provide outputs, $\neq \perp$, if all parties behave honestly in the preprocessing and the timeouts are called after the preprocessing succeeded at all honest parties. We formalize that by implementing an ideal functionality.

For a function $f\colon \mathbb{F}^n \to \mathbb{F}^n$, let $\mathcal{F}_{\mathrm{FSFE}}^{f}$ be the following ideal functionality for fair secure function evaluation.

1. On input (`begin preprocess`) from P_i, inform the adversary by sending it (P_i, `begin preprocess`).
2. On input (x_i) from P_i, output (P_i, `gave input`) to the adversary.
3. If the adversary inputs (`early timeout`), then output $(y_i = \perp)$ to all P_i, and terminate.
4. If all P_i have input both (`begin preprocess`) and x_i and the adversary then inputs (`late timeout`), then compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and output (y_i) to all P_i, and terminate.

Note that the adversary can always make the evaluation fail, but must do so in a fair way: either no party learns anything, or all parties learn a correct output. Our protocol securely implements this ideal functionality when $t < n/3$ parties are corrupted. If the BA is modeled as an ideal functionality, then our implementation is perfectly secure. We will not give the full simulation proofs below, as they follow more or less straightforwardly using known techniques.

On a high level, however, the simulation proceeds as follows: First the simulator simulates the first 4 steps while emulating the algorithms of honest players as specified in the protocol. This is possible as the secret inputs of honest players are not used in these steps. We write $\bar{P}_i$ for the simulator's "copy" of honest P_i.

If some honest $\bar{P}_j$ computed $g_j =$ `preprocess failure`, then the simulator inputs (`early timeout`) to $\mathcal{F}^f_{\text{FSFE}}$, which will make it output $y_i = \perp$ to all players. Clearly the same happens in the real execution since P_j sends $g_j =$ `preprocess failure` to all honest parties.

If all honest $\bar{P}_j$ compute $g_j =$ `preprocess success`, then the preprocessing was secure. This ensures that the sharings $[r_i]$ are consistent, and since the simulator knows the shares of all $\bar{P}_j$, it can compute all r_i. From the s_i broadcast by the corrupted parties in the simulation it computes $x_i = s_i - r_i$ and inputs these to $\mathcal{F}^f_{\text{FSFE}}$ on behalf of the corrupted parties. It broadcasts random s_i's on behalf of honest players.

Then the simulator finishes the execution of the preprocess and input phase. If during this the adversary cheats or calls the timeouts at a time which makes the BA terminate with $q =$ `failure`, then the simulator inputs (`early timeout`) to $\mathcal{F}^f_{\text{FSFE}}$, which will make it output $y_i = \perp$ to all P_i. Clearly the same happens in the real execution.

If $q =$ `success` in the simulation, the simulator inputs (`late timeout`) to $\mathcal{F}^f_{\text{FSFE}}$, and learns the output for corrupted parties. It can now simulate the computation phase using standard techniques until all parties have computed their outputs.[2] Namely, since the computation phase is a sequence of public reconstructions, the simulator for each reconstruction selects the value to be opened, either a random value or a result y_i, as appropriate. It then computes shares to send on behalf of the honest players such that they are consistent with the opened value and the shares held by corrupted players.

4.3 Protocol for Preprocessing

In this section, we describe the techniques used in the preprocessing phase. One version of the preprocessing is obtained by simplifying in a straightforward way the protocols from Beerliová-Trubíniová and Hirt [7], where *hyperinvertible* matrices are used to generate multiplication triples. Another version is based on pseudorandom secret-sharing [29].

[2]In this process, the simulator may need to control the time at which results are delivered to honest parties, depending on when the adversary chooses to deliver the messages in the simulated execution.

4.3.1 Preprocessing based on Hyperinvertible Matrices

In this subsection we will show how the preprocessing and input phase works. This amounts to showing how to generate the multiplication triples.

The key element in the way we generate triples is that while in [7], a player elimination step is run whenever a fault occurs, we accept the possibility that our protocol will not terminate. Therefore we can simplify and speed up the protocols considerably by cutting away the player elimination and simply aborting if a fault is detected. For completeness and readability, we will describe the most important protocols here, but refer to [7] for security proofs and some of the tools.

In order for us to be able to generate multiplication triples, we first need to be able to generate double sharings of random element — that is, two Shamir sharings of the same random element, possibly with different thresholds. In other words we wish to generate for a random $r \in \mathbb{F}$ sharings $[r]_d$ and $[r]_{d'}$, where d and d' are the degrees or thresholds. A more compact notation for the double sharing is $[r]_{d,d'}$.

We will need some facts from [7] on reconstructing shared values, namely how to reconstruct a value robustly to one player using $\mathcal{O}(nk)$ bits of communication and how to reconstruct up to $T = n - 2t$ values publicly using $\mathcal{O}(n^2k)$ bits, where k is the size of a field element.

The following is based on the concept of hyperinvertible matrices. We define "hyperinvertible" as in [7], where a straightforward way to construct such a matrix is also presented:

Definition 4.1 An $m \times n$ matrix M is *hyperinvertible* if for any selection $R \subseteq \{1, \dots, m\}$ of rows and $C \subseteq \{1, \dots, n\}$ of columns such that $|R| = |C| > 0$, the square matrix M_C^R consisting of the intersections between rows in R and columns in C is invertible. ♦

The protocol for generating $T = n - 2t$ double sharings now works as follows (it assumes the existence of an publicly known $n \times n$ hyperinvertible matrix M):

1. Each player P_i Shamir shares a random value s_i to the others using both d and d' as degrees. Every P_i now knows shares of $[s_1]_{d,d'}, \dots, [s_n]_{d,d'}$, but shares from corrupted players may be incorrect.

2. The players locally compute

$$([r_1]_{d,d'}, \dots, [r_n]_{d,d'}) = M([s_1]_{d,d'}, \dots, [s_n]_{d,d'}).$$

   Note that there are actually two vectors here, and the matrix is applied to both, creating two new vectors.

3. All sharings $[s_i]_{d,d'}$ are verified for $i = T + 1, \dots, n$. They are verified by having each P_j send his share of $[s_i]_{d,d'}$ to P_i. Each P_i that is given shares must then check whether they are consistent and that both parts of the double sharing represent the same value. If not, P_i sets an *unhappy* flag to indicate the fault.

4. The double sharings $[r_1]_{d,d'}, \dots, [r_T]_{d,d'}$ are the output.

The double sharing protocol is guaranteed to either output $T = n - 2t$ correct and random double sharings that are unknown to the adversary or make at least one honest player unhappy. This is proved in [7], along with the fact that the communication complexity is $\mathcal{O}(n^2k)$ bits. In our case, if an honest player becomes unhappy at any point, all other players are informed and the honest players will abort, as described in the Section 4.2. That is, we skip the player elimination used in [7].

If we only wanted to generate a set of T single Shamir sharings, it is easy to see that we can use the protocol above but considering only sharings using degree d for each step. The complexity of this is half that of creating double sharings. This is used for generating the sharings $[r_i]$ of a random r_i for each player P_i, that we promised in the Section 4.2.

Generating Multiplication Triples.

Given sharings

$$[a_1]_t, \dots, [a_T]_t, [b_1]_t, \dots, [b_T]_t$$

and

$$[r_1]_{t,2t}, \dots, [r_T]_{t,2t}$$

of random and independent numbers $a_i, b_i, r_i \in \mathbb{F}$, we can generate T multiplication triples as follows:

1. The players compute $[a_i]_t[b_i]_t - [r_i]_{2t} = [a_ib_i - r_i]_{2t}$ for $i = 1, \dots, T$.[3] They then attempt to publicly reconstruct all of the $a_ib_i - r_i$. If the reconstruction of any of the values fails, an honest player becomes unhappy and we abort.

2. The players locally compute $[a_ib_i]_t = a_ib_i - r_i + [r_i]_t$. All honest players now own shares of the $[a_ib_i]_t$, the $[a_i]_t$ and the $[b_i]_t$ for $i = 1, \dots, T$.

This protocol is clearly secure, assuming that the sharings we start from have been securely constructed. The simulator would choose random values s_i to play the role of $a_ib_i - r_i$, it would then expand the set of shares known by corrupt players of $[a_ib_i - r_i]$ to a complete set consistent with s_i and use these shares as those sent by honest players. Please see [7] for more details.

The communication complexity is $\mathcal{O}(n^2k)$ bits for the reconstructions and therefore a total of $\mathcal{O}(n^2k)$ bits including the generation of the double sharings. That is, we can reconstruct $T = n - 2t = \Theta(n)$ shares with a communication complexity of $\mathcal{O}(n^2k)$, where k is the bit length of the field elements.

[3]The notation $[a_i]_t[b_i]_t$ means that each player locally multiplies its shares $[a_i]_t$ and $[b_i]_t$. This gives a $2t$ sharing of a_ib_i.

4.3.2 Preprocessing based on Pseudorandom Secret-Sharing

We show here how to do the preprocessing based on pseudorandom secret-sharing. The techniques used are described in detail in [29], but we present here an overview for completeness.

Pseudorandom Secret-Sharing.

Let A be a set of players of size $n - t$. We can create a random, shared secret by defining for each set A a random value r_A and give it to all players in A. The secret is then given by

$$s = \sum_A r_A.$$

Now every maximal unqualified set $\{1, \ldots, n\} \setminus A$ misses exactly one value, namely r_A.

Keeping the above in mind, pseudorandom secret-sharing (PRSS) is then based on the observation that we can create many random shared secrets by distributing once and for all one set of r_A values.

The trick is to use a pseudorandom function ψ_{r_A} with r_A as its key. If the parties agree on some publicly known value a, they can generate the random values they need as $\psi_{r_A}(a)$. So the secret is now

$$s = \sum_A \psi_{r_A}(a).$$

What we actually want, however, is a Shamir sharing. This can be fixed as follows. Define a degree at most t polynomial f_A by $f_A(0) = 1$ and $f_A(i) = 0$ for all $i \in \{1, \ldots, n\} \setminus A$. Now each player P_i computes its share

$$s_i = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A| = n-t, i \in A}} \psi_{r_A}(a) f_A(i).$$

This is in fact a Shamir sharing of s, since it defines the polynomial

$$f(\mathsf{x}) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A| = n-t}} \psi_{r_A}(a) f_A(\mathsf{x}).$$

It is easy to see that this polynomial has degree at most t and that

$$f(0) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A| = n-t}} \psi_{r_A}(a) = s,$$

which means that it shares the right secret. It is also clear that $s_i = f(i)$, which means that our sharing is a correct Shamir sharing.

Pseudorandom Zero-Sharing.

We will need one more tool to be able to generate multiplication triples, namely what is defined in [29] as pseudorandom zero-sharing (PRZS).

Like PRSS, it creates a Shamir sharing using only local computations, but in this case it is a sharing of 0. We will need a sharing of degree $2t$ in the following, but the approach works just as well with other thresholds. First, for a set $A \subseteq \{1, \ldots, n\}$ of size $n - t$ we define the set

$$G_A = \left\{ g \in \mathbb{Z}_p[x] \;\middle|\; \deg(g) \leq 2t \wedge g(0) = 0 \wedge (j \notin A \Rightarrow g(j) = 0) \right\}.$$

This is a subspace of the vector space of polynomials of degree at most $2t$. Because every polynomial in the set has $t + 1$ zeros, the subspace must have dimension $2t + 1 - (t + 1) = t$. The construction from [29] needs a basis for this subspace, but no explicit construction was given there. We suggest to use the following:

$$(g_A^1, \ldots, g_A^i, \ldots, g_A^t) = (x f_A, \ldots, x^i f_A, \ldots, x^t f_A),$$

where the f_A is defined as above. It is a basis because it has t elements of G_A which are all of different degrees and therefore linearly independent. Exactly as for PRSS, we assume that we have values r_A known (only) by players in A. Now we define the share at player j as

$$s_j = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A| = n-t, j \in A}} \Big(\sum_{i=1}^{t} \psi_{r_A}(a, i) g_A^i(j) \Big).$$

Note here that the inner sum is a pseudorandom choice of a polynomial from G_A, evaluated in the point j. Now we want to show that this leads to a Shamir sharing of 0, so we define the corresponding polynomial as

$$g_0(\mathsf{x}) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A| = n-t}} \Big(\sum_{i=1}^{t} \psi_{r_A}(a, i) g_A^i(\mathsf{x}) \Big).$$

The degree of g_0 is clearly at most $2t$, and it is also easy to see that it is consistent with the shares above and that $g_0(0) = 0$.

Making triples using PRSS and PRZS.

In order to make multiplication triples, we already know that it is enough if we can build random sharings $[a]_t, [b]_t,$ and a double sharing $[r]_{t,2t}$.

Using PRSS, it is easy to construct the random degree t sharings. A double sharing can be constructed as follows: Create using PRSS a random sharing $[r]_t$ and use PRZS to create a sharing of zero $[0]_{2t}$. Now

$$[r]_{2t} = [r]_t + [0]_{2t}$$

is clearly a sharing of r of degree $2t$. We can therefore use pseudorandom secret sharing and pseudorandom zero sharing to locally compute all the values needed to make multiplication triples. The only interaction needed is one public opening for each triple as described in Section 4.3.1.

This is faster than using hyperinvertible matrices for a small number of players, but does not scale well: since $n - t = \Theta(n)$, the local computation is exponential in n, as clearly seen from the benchmark results in Section 4.5. The break-even point between PRSS and hyperinvertible matrices depends both on local computing power and on the cost of communication.

Security of the PRSS approach.

We claim that the overall protocol is secure against a computationally bounded and static adversary, when based on PRSS.

To argue this, consider some adversary who corrupts t players, and let A be the set of $n - t$ honest players. Now let π_{random} be the protocol that runs as described above, but where the function ψ_{r_A} is replaced with a truly random function.[4]

When we execute PRSS or PRZS in π_{random}, all secrets and sets of shares held by the honest players are uniformly random, with the only restriction that they are consistent with the shares held by corrupt players. We can therefore use the proof outlined in Section 4.2.2 to show that π_{random} implements $\mathcal{F}_{\text{FSFE}}^{f}$ (with perfect security).

For the rest of the argument, we refer to the protocol using the pseudorandom function as π_{pseudo}. We claim that real-world executions of π_{random} and π_{pseudo} are computationally indistinguishable. Assume for contradiction that there exists some computationally bounded environment $\mathcal{Z}$ that can distinguish between the two with a non-negligible advantage.

From $\mathcal{Z}$ we can now build a new machine $\mathcal{M}$, which gets oracle access to some function f and outputs its guess of whether the function is pseudorandom or truly random.

$\mathcal{M}$ simply runs the protocol with f inserted in the place of ψ_{r_A} (i.e., it runs either π_{random} or π_{pseudo}) for $\mathcal{Z}$. If $\mathcal{Z}$ outputs "π_{random}", M outputs "truly random", otherwise it outputs "pseudorandom". Clearly, M can distinguish between a pseudorandom function and a truly random function with a non-negligible advantage, breaking the security of our PRF.

Combining this with the fact that π_{random} securely realizes $\mathcal{F}$, we see that the same holds for π_{pseudo} (with computational security): the simulator that works for π_{random} also works for π_{pseudo}.

4.4 Integration into VIFF

The above protocol has been implemented in VIFF and we present benchmark results in Section 4.5. Using the plug-in architecture of VIFF described in Section 3.6.3, an application developer can use the protocol by creating a runtime class mixed with the `BasicActiveRuntime`. This class contains the multiplication protocol itself.

Multiplication triples are needed too and a common interface has been defined for producing them. The two preprocessing techniques are implemented in `TriplesHyperinvertibleMatricesMixin` and `TriplesPRSSMixin`. These mixin classes work together with `BasicActiveRuntime` to make a full, usable runtime. Such a runtime support all the operations of the passively secure runtime, but with active security. The actively secure runtime can be used as a drop-in replacement for a passively secure runtime. User code remain unchanged and the programmer still writes

[4]This can be formalized by assuming an ideal functionality that gives oracle access to the function for the honest players as soon as the adversary has corrupted a set of players initially.

```
z = x * y
```

to make a secure multiplication. Likewise for comparisons and other compound protocols, which inherit the security of the multiplication protocol and thus automatically become actively secure.

4.5 Benchmark Results

An old proverb goes, *the proof of the pudding is in the eating!* We have therefore put VIFF to the test in order to evaluate its performance. It is worth pointing out that this is one of the major contributions of VIFF — it is now feasible to actually implement a protocol found in the literature and compare its performance to that of other protocols. The asymptotic nature of the big-$\mathcal{O}$ notation hides constant factors. In some cases, these constant factors are merely a distraction and here the big-$\mathcal{O}$ notation shines. However, one should never forget that a real system is of a finite size, and so a big constant factor can become very important. We will see an excellent example of this behavior below, where a protocol with quadratic complexity is beaten by a protocol with exponential complexity, albeit only for a small number of players. Many protocols exhibit the same asymptotic round and communication complexities, so we need real benchmarks on realistic data to compare them.

4.5.1 Test Setup

We have tested VIFF on a number of machines located at the Department of Computer Science, Aarhus University. The machines were connected by a standard gigabit local area network. On the software side, we used Red Hat Enterprise Linux Client 5.4, Python 2.4.3, and VIFF 1.0.

We group the machines into classes by their CPU type. The classes are shown in Table 4.1, along with the number of machines in each class. At the beginning of each test run, a set of n idle machines are selected. Machines are picked from the most powerful class first (the top row), followed by less powerful machines, until n machines have been found. Machines were disqualified from selection if they had less than 512 MiB of free main memory or if their *load average* over the past 1, 5, and 15 minutes measured over 0.3. The system load is a classical performance metric on Unix-like systems. It is defined as the average number of jobs waiting to be executed on the CPU. The machines generally had loads of 0.0–0.1 when the benchmarks were executed.

We ran run benchmarks with $n = 4, 7, \ldots, 31$ corresponding to thresholds $t = 1, 2, \ldots, 10$, respectively. In each test we secret-shared 2,000 random numbers and multiplied or compared the 1,000 pairs in parallel. We used number in $\mathbb{Z}_p$ where p is a 32-bit prime for multiplication and a 65-bit prime for comparisons. The time needed for preparation of the input numbers is not counted, we only report the time it took to compute the secret shared result. We had each party output the total time used and we report the

Table 4.1: Machine classes used for benchmarks. For each class, the last column gives the number of machines available with that configuration. All machines were equipped with dual core CPUs. The data was read from `/proc/cpuinfo` and `/proc/meminfo`.

Machine Class	Speed	Memory	Count
Intel® Core™ 2 Duo CPU E8500	3.16 GHz	4 GiB	18
Intel® Core™ 2 Duo CPU E8400	3.00 GHz	4 GiB	1
Intel® Core™ 2 Duo CPU E6750	2.66 GHz	2 GiB	10
Intel® Core™ 2 Duo CPU E6550	2.33 GHz	2 GiB	5
Intel® Core™ 2 CPU 6600	2.40 GHz	2 GiB	3
Intel® Core™ 2 CPU 6400	2.13 GHz	2 GiB	1
Intel® Xeon™ CPU	3.00 GHz	1 GiB	24

minimum time per operation. The minimum value was chosen in order to best compare the protocols against one another. We assume the local clocks have a negligible skew, and so the minimum time gives the purest performance indication — values above the minimum must contain errors stemming from fluctuations in system load or congestion on the network. That being said, we kept and eye on the median value and the average values. The minimum value were generally within 0.1 ms from the median value, and the average value were up to 0.5 ms further away from the minimum for large values of n.

4.5.2 Multiplications

Multiplication and reconstruction of secret shared values are arguably the most important protocols in VIFF. Many larger protocols are built upon these primitives. The basic BGW protocol for multiplication [51] has every party communicate with every other party in addition to performing a small amount of local computation. Reconstructing (opening) a secret shared value has an identical communication pattern, albeit with a little less local computation. Because of their similarities, multiplications and openings are often counted together and used as a measure of protocol performance. As we will see below, they can only be lumped together for small values of n.

We tested how multiplications scale as the number of players grow. These results are in Table 4.2 on the following page. Columns 3 and 4 give the online time when using PassiveRuntime and ActiveRuntime, respectively. The final two columns give the offline time spent per multiplication triple.

Passively Secure Protocol

The average online time per multiplication are shown in Figure 4.1 on the next page. Looking at the graph, it may at first appear to grow linearly. However, it turns out that the running time for the passively secure protocol is better described by a quadratic function. The function $f(x) = 0.799 - 0.006x + 0.009x^2$ fits the data best, yielding an R^2 value of 0.986. This function has been draw on the plot in Figure 4.1.

Table 4.2: Multiplication of 32-bit numbers.

n	t	Passive	Active	Hyper	PRSS
		—— Online ——		——— Offline ———	
4	1	0.7 ms	1.3 ms	5.7 ms	1.2 ms
7	2	1.2 ms	2.0 ms	10.0 ms	2.7 ms
10	3	1.7 ms	2.7 ms	13.9 ms	10.8 ms
13	4	2.4 ms	3.5 ms	20.0 ms	65.3 ms
16	5	3.1 ms	4.2 ms	26.3 ms	456.0 ms
19	6	3.9 ms	5.0 ms	34.9 ms	—
22	7	4.8 ms	6.0 ms	46.7 ms	—
25	8	5.6 ms	6.9 ms	65.9 ms	—
28	9	6.9 ms	7.7 ms	73.4 ms	—
31	10	9.5 ms	8.7 ms	102.0 ms	—

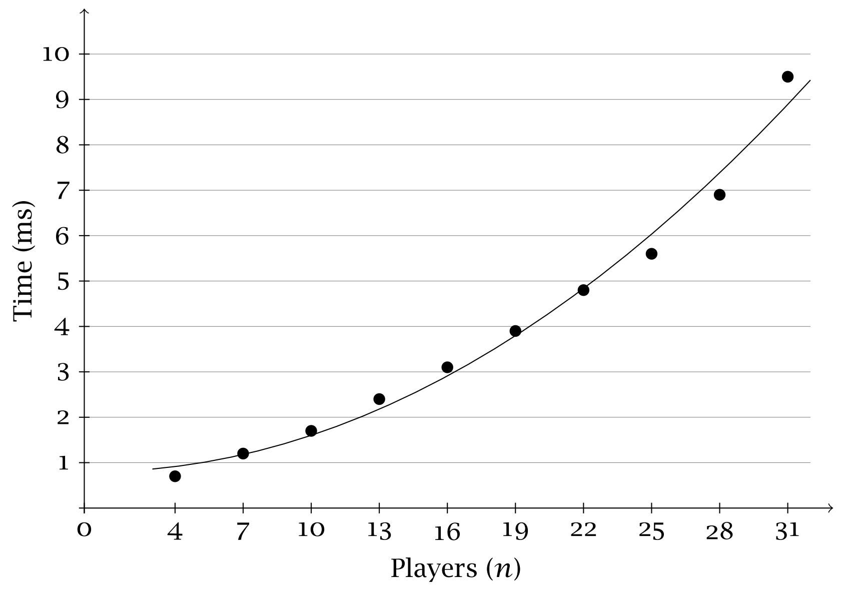


Figure 4.1: Online time for multiplication of 32-bit numbers when using the passively secure protocol. The bullets show the measured running times, the curved line is a best fit quadratic function.

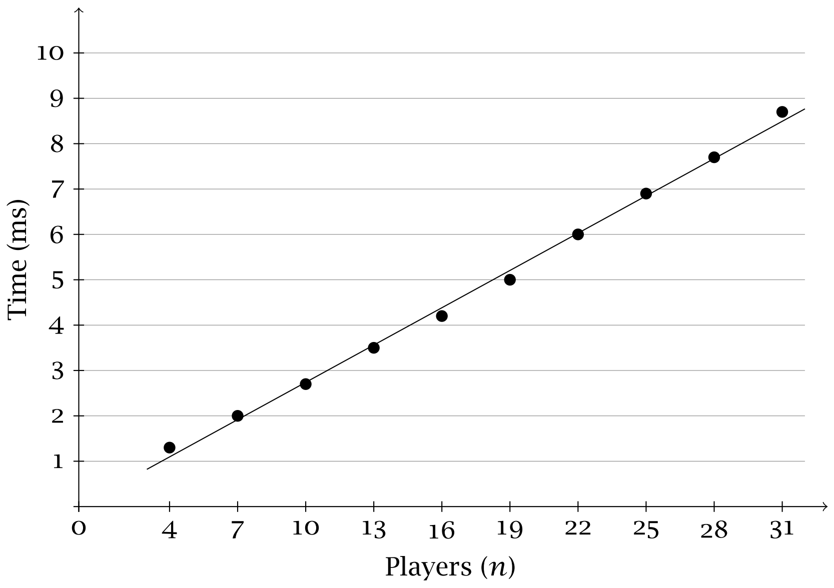


Figure 4.2: Scatter plot of the online time for multiplication of 32-bit numbers using the protocol secure against malicious adversaries. The line is a best fit linear function.

The BGW protocol [10, 51] has linear communication complexity, but *quadratic* computational complexity. The quadratic term enters when each party Shamir shares the local product. To Shamir secret sharing a value, the party selects a random polynomial of degree t and evaluates it in $n - 1$ points. Each evaluation takes time $\mathcal{O}(t)$, and with t fixed as a constant fraction of n, the Shamir sharing becomes a quadratic operation. As the number of parties grow, this factor becomes increasingly important. For 31 parties, the online time even exceeds the time for actively secure protocol.

Actively Secure Protocol

We have depicted the benchmark results for the actively secure protocol in Figure 4.2. Here we see a strongly linear growth, which is also what the theory predicts. The figure shown the function $f(x) = -0.001 + 0.274x$ along with the data. The R^2 value was 0.997.

The actively secure protocol consist of two public reconstructions and a constant number of local arithmetic operations. To reconstruct a Shamir shared value we need compute

$$f(0) = \sum_{(x_i,s_i)\in S} \left(s_i \prod_{\substack{(x_j,s_j)\in S,\\ j\neq i}} \frac{x_j}{x_j - x_i} \right)$$

for a set $S = \{(x_{i_1}, s_{i_1}), \ldots, (x_{i_{t+1}}, s_{i_{t+1}})\}$ of $t + 1$ shares where $s_i = f(x_i)$. We can cache the $\lambda_i = x_j/(x_j - x_i)$ terms and this simplifies the expression

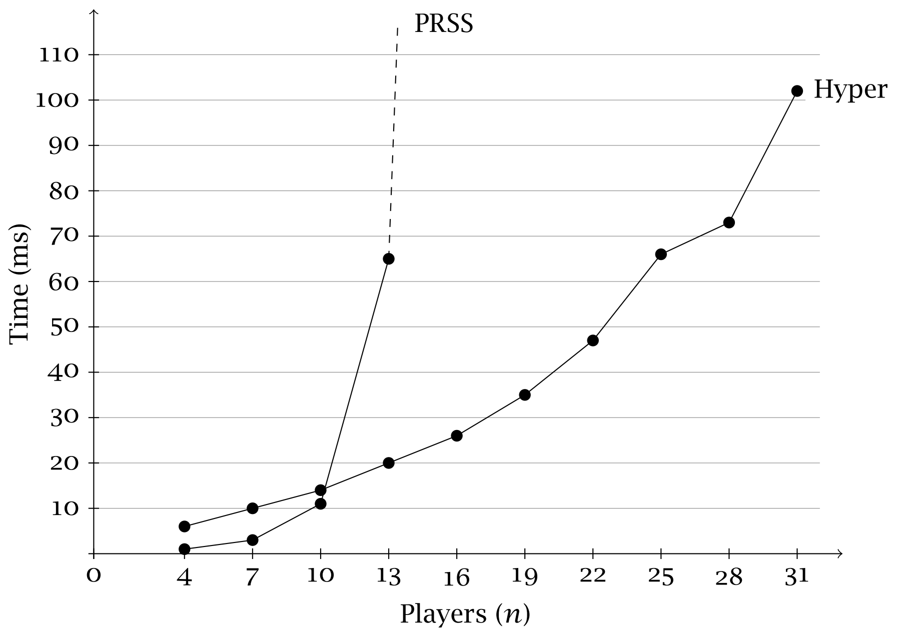


Figure 4.3: Offline (preprocessing) time needed to generate a single 32-bit multiplication triple.

for $f(0)$ to

$$f(0) = \sum_{(s_i,x_i)\in S} s_i\lambda_i.$$

This can be computed in time $\mathcal{O}(n)$. The λ_i clearly depend on the x_i, but we only need to compute each λ_i once.

4.5.3 Multiplication Triples

The key to the excellent online performance of the actively secure protocol described in this chapter, is a supply of ready-made multiplication triples. We generate those in advance using either a technique based on hyperinvertible matrices or a technique based on pseudorandom secret sharing. Columns 4 and 5 of Table 4.2 present the results and Figure 4.3 visualize them.

As expected, the PRSS based preprocessing is much faster for a small number of players but does not scale well. We had to abandon it for $n > 16$. The amount of work per player depends on the number of subsets of size $n - t$ and with t as a fixed fraction of n, $\binom{n}{n-t}$ grows exponentially. Still, for 4 or 7 parties, the PRSS based method is much superior to using hyperinvertible matrices.

The preprocessing time per multiplication triple produced via hyperinvertible matrices shows a more moderate growth, though it is still a quadratic growth. As before, the communication complexity is linear, but

Table 4.3: Comparison of 32-bit numbers using a 65-bit prime field.

n	t	Passive	Active
4	1	133 ms	175 ms
7	2	228 ms	284 ms
10	3	518 ms	587 ms
13	4	2,014 ms	2,593 ms

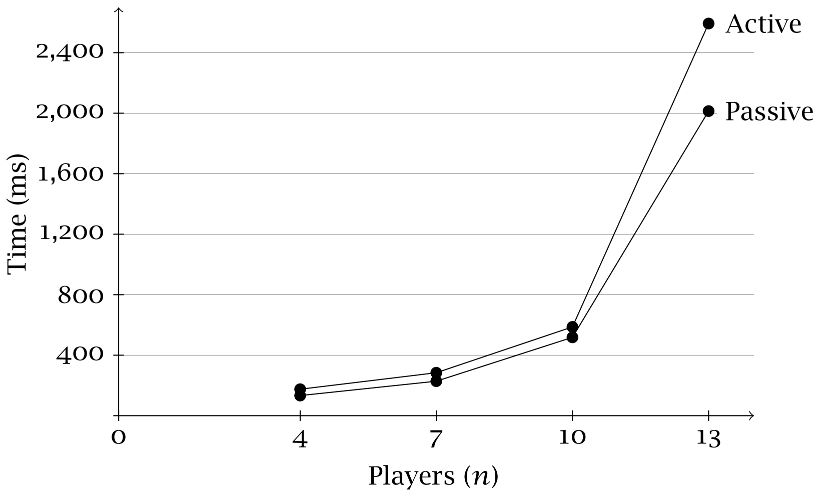


Figure 4.4: Online time for comparison of 32-bit numbers.

the computational complexity is quadratic. This due to the Shamir sharing and the matrix multiplication involved in producing the double sharings.

We tested the protocols on a gigabit LAN where the computational complexity becomes the dominant factor — with ping times below 1 ms, the parties can communicate faster than they can compute.

4.5.4 Comparisons

Secure comparison of secret shared values is another important protocol. Without being able to compare values, it is difficult to make decisions based on the inputs. VIFF provides two protocols for secure comparison. The earlier of the two, provided by the class ComparisonToft05Mixin, takes two inputs from $\mathbb{Z}_p$ and produces a secret shared bit in $\mathbf{GF}(2^8)$. A later version (with the imaginative name of ComparisonToft07Mixin) gives an output in $\mathbb{Z}_p$ and is thus a bit more straight-forward to use. We have benchmarked the latter protocol here, please see Table 4.3 and Figure 4.4.

Instead of the usual 1,000 parallel operations, we only made 100 parallel comparisons. The benchmarks demonstrate that secure comparisons are 2–3 orders of magnitude more expensive than multiplications in terms of online processing time.

Table 4.4: Bandwidth statistics. We have measured the data sent per pair-wise connection for a single operation when operating on 32-bit inputs. The total amount of bandwidth used in the network is thus $n(n-1)$ times higher since each of the n parties maintain $n-1$ connections.

Operation	Field Size	Passive	Active
Multiplication	32 bit	28 bytes	56 bytes
Comparison	65 bit	5,183 bytes	7,048 bytes

4.5.5 Bandwidth

A final parameter is the bandwidth usage of protocols. We have measured the number of bytes sent over each pair-wise connection when doing multiplications and when doing comparisons. The results are in Table 4.4. In contrast to the previous benchmarks, these results are independent of the number of parties since we give the numbers per pair-wise connection. The first thing to note is that, as expected, the actively secure multiplication protocol uses exactly twice as much bandwidth as the passively secure protocol. For comparisons, the gap between the two runs is smaller. We will attribute this to the fact that a comparison consist of more than just a number of multiplications — each secure comparison consumes 69 multiplication triples — it is only the multiplications that double in complexity when we have to protect against malicious adversaries. In particular, the protocol for pseudorandom secret sharing remains unchanged.

4.6 Conclusion

We have presented an efficient protocol for general multiparty computation secure against active and adaptive adversaries. The protocol provides a new trade-off between guaranteeing termination and efficiency which we believe is relevant in practice. To demonstrate this we have implemented the protocol in a framework for secure multiparty computation called VIFF.

This allowed us to show that achieving active security can be very efficient. In fact, in our test environment, the protocol is faster than the standard passively secure protocol for a large number of parties. For a smaller number of parties, the cost of security against a malicious adversary is only a fixed overhead of about 1 ms in online time. The protocol assumes one is willing to accept that the preprocessing step might fail without revealing any private data. We believe this to be very well-suited for practical applications where the parties typically have a much stronger incentive to participate in the computation than to halt it.

For a small number of parties, the cost of preprocessing is only slightly larger than the online cost. For instance, 7 parties can prepare 1,000 multiplications in about 2.7 seconds and execute them in using only 2 seconds. With 7 parties, this gives a throughput of 500 secure multiplications per second (770 multiplications per second for 4 parties), which we feel should cover many practical applications.

Chapter 5

From Passive to Covert Security at Low Cost

Chapter 3 presented a protocol secure against active adversaries along with an implementation of the protocol in VIFF. In this chapter, we will define a third kind of adversary — a covert adversary — and show how one can compile many passively secure protocols into protocols secure against a covert adversary.

Apart from Section 5.6, which is new, the material is taken from [39], which was presented at the TCC 2010 conference. Some technical details have been deferred to a full version of the conference paper, which can be found in the Cryptology ePrint Archive [38].

5.1 Introduction

When studying cryptographic protocols, the behavior of the adversary has traditionally been categorized as being either *semi-honest* (passive) or *malicious* (active). A semi-honest adversary will only listen in on the network communication and spy passively on the internal state of the corrupted protocol participants. At the other end of the spectrum, a malicious adversary can make corrupted parties behave arbitrarily and will try to actively disrupt the computation in order to gain extra information and/or cause incorrect results.

The notion of a semi-honest adversary was first defined by Goldreich et al. [54], though it was used implicitly in other early works (e.g., by Yao [100] in the describing of a solution to the millionaire's problem). While [54] also considered malicious adversaries this is also a classic notion in the distributed communication community, where such behavior is called *Byzantine* [64].

Aumann and Lindell [3] introduce a third type of adversary called a *covert* adversary. This is intuitively an adversary which is able to do an active attack, but will behave correctly if the risk of being caught is sufficiently large — even if that risk is not essentially 1. The argument for studying covert adversaries is that there are many real world situations where the consequences of being caught out-weights the benefit of cheating — even a small but non-negligible risk of being caught is a deterrent. An example could be companies that agree to conduct an auction using secure multiparty computation. If a company is found to be cheating it may be subject to fines and it will hurt its long-term relationships with customers and other companies.

In the standard simulation-based definition of secure multiparty computation a protocol is said to *securely evaluate a function f* if no attack against the protocol can do better than an attack on an ideal process where an ideal functionality evaluates f and hands the result to the parties. Aumann and Lindell [3] give three different models of what a covert adversary can do by defining two different ideal functionalities that may compute f as usual, but may also act differently, depending on what the adversary does. They also define what it means for a protocol to implement an ideal functionality securely, this is a fairly standard simulation-based definition for sequentially composable protocols.

Thus, the special ingredient in the model that allows to accommodate covert attacks is only in the definition of the functionalities, which correspond to different levels of security, which are called *Explicit Cheat Formulation* (ECF) and *Strong Explicit Cheat Formulation* (SECF).[1] The basic idea in both cases is that the adversary may decide to try to cheat and must inform the functionality about this. The functionality then decides if the cheating is detected which happens with probability ε, where ε is called the *deterrence factor*. In this case all parties are informed that some specific corrupt party cheated. Otherwise, with probability $1 - \varepsilon$, the cheating is undetected, and there is no security guarantee anymore: the functionality gives all inputs to the adversary and lets him decide the outputs. The difference between the two variants is that for ECF, the adversary gets the inputs of honest parties and decides their outputs immediately when he decides to cheat. For SECF, this only happens if the cheat is not detected.

Thus, with ECF, the adversary is caught with probability ε, but will learn the honest parties' inputs even if he is caught. With SECF, he must try to cheat *and succeed* to learn anything he was not supposed to.

5.1.1 Our Contribution

We propose a new construction that "compiles" a passively secure protocol into a new protocol with covert security. The approach is generic, but for concreteness we describe the idea starting from the classical BGW protocol [10] for evaluating arithmetic circuits.

We assume honest majority and synchronous communication with secure point-to-point channels. We also assume a poly-time adversary, as we use cryptographic tools.

The basic idea is to first use a protocol with full active security to do a small amount of computation. Here, we will prepare two sets of (secret-shared) inputs to the passively secure protocol. However, only one set of

[1]They also have a so called Failed Simulation definition which is weaker and which we do not use here.

sharings contains the actual inputs, while the other — the *dummy* shares — contain only zeros. Initially, it is unknown which set is the dummy one. Then we run the passively secure protocol on both sets of inputs until parties hold shares of the outputs, which they must commit to.

Now we reveal which sharings contained dummy values, and everything concerning the dummy execution can be then made available to check that no cheating occurred here. If no cheating was detected, we open the outputs of the real execution.

The intuition is that the adversary has to decide whether to cheat without knowing which execution is the dummy one, and therefore we can catch him with probability $\frac{1}{2}$ if he cheats at all, so one would expect this to give a deterrence factor of $\frac{1}{2}$.

However, while the intuition is straightforward, there are several non-trivial technicalities to take care of to make this work. We need parties to be able to prove that they really sent/received a given message earlier, and we have to do the final check without introducing too much overhead. After solving these problems, we obtain a protocol with deterrence factor $\frac{1}{4}$ whose complexity is essentially twice that of the passive protocol plus the overhead involved in preparing the inputs (which does not depend on the size of the computation).

It should be noted that there is an overhead involved in proving what messages were sent in the past. For this, players need to sign the messages they send. However, unless the arithmetic circuit we compute has very large depth and small breath, the cost of signing can be amortized over several operations requiring communication, and so is not significant. For the most advanced version of our construction, players also need to UC commit at the end to the set of messages they sent to each player. Our solution to this in the standard model is based on Paillier encryption and is quite elaborate, but for a practical implementation one can use the random oracle model, in which case commitment reduces essentially to hashing the messages, and is not a major cost.

We note that we focus on the complexity we get when there is no deviation from the protocol. In our construction, the adversary can slow things down by a factor linear in the number of parties by deviating, but the protocol is still secure, the adversary can only make it fail if he runs the risk of actually cheating and hence of being caught. Now, the spirit of covert security is that the adversary is to some extent rational, he does not cheat because it does not pay off to do so. It seems to us that there is little benefit in practice for the adversary in only slowing things down, while he cannot learn extra information or influence the result. We therefore believe that the complexity in practice can be expected to be what we get when there is no deviation.

We show our protocol is secure by showing that it implements Aumann and Lindell's functionality *in the UC model* [23], i.e., we do not use their simulation notion. The only difference this makes is that we get a stronger composition property for our protocol.

We show that the classical passively secure protocol by Ben-Or et al. [10]

can be compiled to give a protocol with SECF security. Our approach can be used in a more general way, to compile any passively secure protocols into a covert protocol, if the original protocol satisfies certain reasonable conditions. The conditions are essentially as follows. The protocol should be based on secret sharing and consist of a computation phase and a reconstruction phase.

Computation phase: The computation phase starts from sharings of the inputs and produces sharings of the outputs, where the view of $t < n/2$ passively corrupted parties is independent of the shared inputs being computed on.

Reconstruction phase: The reconstruction phase consists of a single message from each party to each other party — i.e., it is non-interactive.

Passive security: Suppose uniformly random sharings of the inputs are dealt by an ideal functionality. Consider the protocol that executes the computation phase on these sharings followed by the reconstruction phase. This protocol should be passively secure against $t < n/2$ statically corrupted parties.

The approach is again to run two executions of the computation phase in parallel, one on the real inputs and one on dummy inputs, and to commit the parties to their communication in both. Then it is reveal which execution is which and the dummy execution is checked for consistency before the reconstruction phase is run. Since the view of the corrupted parties in the computation phase is independent of the inputs being computed on as long as they follow the protocol, the corrupted parties have to decide on which execution to cheat in without knowing which is which and is hence caught with probability $\frac{1}{2}$. The reader interested in the details can refer to [38]. If the computation phase leaks no information, even under active attacks (as is the case for the BGW protocol), we get SECF security, otherwise ECF security is obtained.

5.1.2 Related Work and Discussion

Goyal et al. [55] improve Aumann and Lindell's 2-party protocol and also give a general multiparty computation protocol with covert security for the case of dishonest majority.

Our work focuses instead on honest majority. The skeptical reader may ask whether this is really interesting: The motivation for covert security is to settle for less than full robustness in return for more efficient protocols, and it may seem that we already know how to have great efficiency with honest majority and full active security. For instance, in [7, 32], it is shown that unconditionally secure evaluation of a circuit C for n parties and $t < n/3$ corruptions can be done in complexity $O(|C|n)$ plus an overhead that only depends on the depth of the circuit, and in [35], it is shown under a computational assumption that this can be reduced to $O(|C|)$ except for logarithmic factors plus an overhead that is independent of the circuit. Here, the security threshold can be arbitrarily close to $\frac{1}{2}$.

How could we hope to be better than that? There are two answers to this: First, the previous protocols are not as efficient as it may seem. The result from [35] only works asymptotically for a large number of parties and very large computations, it makes non black-box use of a pseudorandom function and is, in fact, very far from being practical. The protocols in [7, 32] use only cheap information theoretic primitives, but the security threshold in non-optimal and there is an overhead implying that deep circuits are expensive.

However, these protocols can all become much simpler and more practical if we assume the adversary is semi-honest. For instance, when the adversary is semi-honest the protocols from [7, 32] can tolerate $t < n/2$ and no longer have an overhead that depends on the circuit depth. Our compiler works for any "reasonable" protocol that is based on secret sharing, so we can use it on these simpler passively secure protocols and get a protocol with covert security, but with efficiency and security threshold similar to the passively secure solutions.

The second answer is that general circuit evaluation is not the only application. There are many special purpose protocols that are designed for a passive adversary but where obtaining active security comes at a significant cost. One example is the protocol by Algesheimer et al. [2] for distributed RSA key generation. Another is the auction application described in [16]. In both cases the protocols do not go via evaluation of a circuit for the desired function, but gets significant optimizations by taking other approaches. We can use our construction here to get covert security at a cost essentially a factor of two.

5.2 Preliminaries

Aumann and Lindell [3] present three successively stronger notions of security in the presence of covert adversaries, of which we consider the two strongest ones. There the adversary is forced to decide whether to cheat without knowledge of the honest parties' inputs. As mentioned, these are called ECF and SECF and are defined by specifying two (very similar) ideal functionalities.

For convenience, we give the ECF and SECF functionalities here. The only differences from [3] is that we do not include an option for the adversary to abort the protocol, and also, if no cheating is detected, the adversary cannot stop the functionality from giving outputs to the honest parties. This gives a stronger notion of security, and we can obtain it as we assume an honest majority.

Another difference is that we relax the requirements on the detection mechanism slightly. In [3] it is required that only one corrupted party is detected and that the honest parties agree on that party. We allow that several corrupted parties are detected and allow that different honest parties detect different sets of corrupted parties. The only requirement is that there is at least one corrupted party which is detected by all honest parties. In the presence of an honest majority, the stronger detection requirement in [3] can then be implemented using a Byzantine agreement at the end of the protocol on who should take the blame. We prefer to see this negotiation as external to the protocol and thus allow the more relaxed detection. See Figure 5.1.

Let f be a function with n inputs and n outputs, where n is the number of parties. The ECF functionality $\mathcal{F}_{\text{ECF}}^f$ for function f with deterrence factor ε works as follows:

Inputs: Any honest party P_i sends input x_i to $\mathcal{F}_{\text{ECF}}^f$, while the adversary $\mathcal{A}$ sends input on behalf of the corrupted parties.

Cheat detection: Let $C \subset \{1, \ldots, n\}$ denote the indices of the corrupted parties and let $H = \{1, \ldots, n\} \setminus C$ be the honest parties. The adversary can at any time instruct $\mathcal{F}_{\text{ECF}}^f$ to give outputs of the form $(\texttt{corrupt}, j)$ for $j \in C$ to P_i with $i \in H$. For $i \in H$, let $J_i \subset C$ be the set of j for which P_i output $(\texttt{corrupt}, j)$.

Attempted cheat: If the functionality $\mathcal{F}_{\text{ECF}}^f$ receives $(\texttt{cheat})$ from $\mathcal{A}$, it will send $(x_1, \ldots, x_n)$ to $\mathcal{A}$. It then decides randomly if the cheating was detected or not:

Undetected: With probability $1 - \varepsilon$, $\mathcal{F}_{\text{ECF}}^f$ sends $(\texttt{undetected})$ to the adversary. Then $\mathcal{A}$ specifies for each $i \in H$ an output y_i and $\mathcal{F}_{\text{ECF}}^f$ outputs y_i to P_i for $i \in H$.

Detected: With probability ε, $\mathcal{F}_{\text{ECF}}^f$ sends $(\texttt{detected})$ to $\mathcal{A}$. In this case $\mathcal{A}$ also gets to decide the output y_i for $i \in H$, but must ensure that $\cap_{i \in H} J_i \neq \emptyset$ at the end of the execution.

Output generation: If $\mathcal{A}$ did not attempt to cheat, $\mathcal{F}_{\text{ECF}}^f$ computes outputs $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and gives y_i to P_i.

Figure 5.1: Functionality $\mathcal{F}_{\text{ECF}}^f$.

The functionality $\mathcal{F}_{\text{SECF}}^f$ is defined exactly as $\mathcal{F}_{\text{ECF}}^f$, except that when the adversary sends a cheat message, the functionality does not send the inputs of honest parties to the adversary. This only happens if the cheating is undetected.

We can now define security:

Definition 5.1 Protocol π computes f with ε-ECF (SECF) security and threshold t if it implements $\mathcal{F}_{\text{ECF}}^f$ ($\mathcal{F}_{\text{SECF}}^f$) in the UC model, securely against poly-time adversaries corrupting at most t parties. ♦

This definition naturally extends to a hybrid UC model where certain functionalities are assumed to be available. By the UC composition theorem and given implementations of the auxiliary functionalities, a protocol follows that satisfy the above definition without auxiliary functionalities.

In the following, we will consider secure evaluation of an arithmetic circuit C over some finite field $\mathbb{F}$. We assume that each input and output of C is assigned to some party, whence C induces in a natural way a function

f_C of the form considered above. In the following, "computing C securely" will mean computing f_C securely in the sense of the above definition.

5.3 Auxiliary Functionalities

We define some ideal functionalities in order to make the presentation clearer. They represent sub-protocols which we show how to implement in Section 5.5.

Message Transmission Functionality

We will make use of a functionality $\mathcal{F}_{\text{TRANSMIT}}$ that is an enhancement of the standard model for secure point-to-point channels. It essentially allows to prove to third parties which messages one received during the protocol, and to further transfer such revealed messages. It does not commit the corrupted parties to what they sent to each other. See Figure 5.2 for details.

The ideal functionality $\mathcal{F}_{\text{TRANSMIT}}$ works with message identifiers *mid* encoding a sender $s(mid) \in \{1, \ldots, n\}$ and a receiver $r(mid) \in \{1, \ldots, n\}$. We assume that no *mid* is used twice. The functionality works as follows:

Secure transmit: On input $(\texttt{transmit}, mid, m)$ from $P_{s(mid)}$ and input $(\texttt{transmit}, mid)$ from all (other) honest parties, store (mid, m), mark it as *undelivered*, and output $(mid, |m|)$ to the adversary. If P_s does not input a $(\texttt{transmit}, mid, m)$ message, then output $(\texttt{corrupt}, s(mid))$ to all parties.

Synchronous delivery: At the end of each round, deliver each undelivered (mid, m) to $P_{r(mid)}$ and mark (mid, m) as delivered.

Reveal received message: On input $(\texttt{reveal}, mid, i)$ from a party P_j which at any point received the output (mid, m), output (mid, m) to P_i.

Do not commit corrupt to corrupt: If both P_j and P_s are corrupt, then the adversary can ask $\mathcal{F}_{\text{TRANSMIT}}$ to output (mid, m') to any honest P_i for any m' and any *mid* with $s(mid) = s$.

Figure 5.2: Ideal Functionality $\mathcal{F}_{\text{TRANSMIT}}$.

This functionality will be used for all private communication in the following, and provides a way to reliably show what was received at any earlier point in the protocol. This is used when the dummy execution is checked for consistency.

Input Functionality

For notational convenience we assume that each P_i has one input $x_i \in \mathbb{F}$. The input functionality is given in Figure 5.3 on the following page. Note that we let the adversary pick the dummy inputs, which is done simply not to decide at this abstract level on any specific set of dummy inputs. We also let the adversary pick the shares the functionality should produce for corrupt players. This is necessary to be able to implement the functionality with a real-life protocol.

The ideal functionality $\mathcal{F}_{\text{INPUT}}$ is parametrized by a secret sharing scheme, sss, and works as follows.

1. Receive an input x_i from each P_i and an input $(d_1, \dots, d_n)$ from the adversary. The adversary also inputs x_i for $i \in C$.
2. Flip a uniformly random bit $d \in_{\text{R}} \{0, 1\}$.
3. Let $e = 1 - d$. Let $x^{(i,d)} = d_i$ be the *dummy* inputs and let $x^{(i,e)} = x_i$ be the *enriched* inputs.
4. For every $x^{i,d}$ and $x^{i,e}$, the adversary inputs sets of shares $X^{i,d}$ and $X^{i,e}$. They each contain a share for every player in C, and we think of $X^{i,d}$ as the set of shares of $x^{i,d}$ that the adversary wants the functionality to produce for corrupt players.
5. For $j = 1, \dots, n$ and $c = 0, 1$, sample $[x^{(j,c)}] \leftarrow \text{sss}(x^{(j,c)} \mid X^{j,c})$, by which we mean that shares of $x^{i,c}$ are sampled, conditioned on players in C receiving shares $X^{i,c}$.
6. Output $(x_i^{(j,0)})_{j=1}^n$ and $(x_i^{(j,1)})_{j=1}^n$ to P_i.
7. On a later input $(\texttt{reveal}, i, k)$, output d and $(x_i^{(j,d)})_{j=1}^n$ to P_k.

Figure 5.3: Ideal Functionality $\mathcal{F}_{\text{INPUT}}$.

Commitment Functionality

We use a flavor of commitment where the committer cannot avoid that a commitment is revealed. The details are given in Figure 5.4.

The functionality $\mathcal{F}_{\text{COMMIT}}$ uses commitment identifiers encoding the sender $s(cid)$ of the commitment. It works as follows:

Commit: On input $(\texttt{commit}, cid, m)$ from $P_{s(cid)}$ and input $(\texttt{commit}, cid)$ from all (other) honest parties, $\mathcal{F}_{\text{COMMIT}}$ store (cid, m) and output $(\texttt{commit}, cid, |m|)$ to the adversary.

Reveal: On input $(\texttt{reveal}, cid, r)$ from all honest parties, where (cid, m) is stored, give (cid, m) to P_r.

Figure 5.4: Ideal Functionality $\mathcal{F}_{\text{COMMIT}}$.

Coin-Flip Functionality

We use the coin-flip functionality given in Figure 5.5 on the facing page.

The functionality $\mathcal{F}^{B}_{\text{FLIP}}$ is parametrized by a positive integer B and works as follows:

1. Sample a uniformly random $k \in_R \{0, \dots, B-1\}$.
2. When the first honest party inputs (`flip`), output k to the adversary.
3. If in the round where the first honest party inputs (`flip`) there is some party P_i which does not input (`flip`), then output (`corrupt`, i) to all parties.

Figure 5.5: Ideal Functionality $\mathcal{F}^{B}_{\text{FLIP}}$.

5.4 Protocol

Having defined the necessary ideal functionalities, we will now describe how we use them to compile the classical passively secure protocol by Ben-Or et al. [10] based on Shamir secret-sharing into one with covert security. This protocol computes an arithmetic circuit C with passive security. Assuming the inputs to the arithmetic circuit have been secret shared, the protocol does addition by having parties add their shares locally, and multiplication by local multiplication of shares followed by a re-sharing by each parties of the local products. Due to space constraints, we assume the details are known to the reader.

The protocols in this section use the auxiliary functionalities we defined. Thus the actual complexity of our construction depends on the implementation of those auxiliary functionalities. It turns out that the overhead incurred includes a contribution coming from the cryptographic primitives we use, this overhead does not depend on the communication complexity of the protocol we compile. In addition, the adversary can choose to slow down $\mathcal{F}_{\text{TRANSMIT}}$ by a factor of n, but since he cannot make it fail, a covert adversary is unlikely to make such a choice as discussed in the introduction.

We begin with a simple construction which has a rather poor computational complexity. Following that, we show how the simple protocol can be adapted to yield a better complexity.

Theorem 5.2 *The protocol in Figure 5.6 on the next page computes C with $\frac{1}{2}$-SECF security in the $(\mathcal{F}_{\text{TRANSMIT}}, \mathcal{F}_{\text{INPUT}}, \mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{FLIP}})$-hybrid world. This is against a static adversary and with threshold t, where $t < n/2$.* □

Proof: Initially S is given the inputs of the corrupt parties. It passes them on to $\mathcal{A}$ and simulates the protocol execution up until the point where the bit d is revealed and it is determined which of the two executions were the dummy execution. S does this by inventing random shares whenever $\mathcal{A}$ would expect to see a share from an honest party. $\mathcal{A}$ will always see only t shares and any subset of size t look completely random in the real protocol execution. S can therefore simulate them perfectly by giving $\mathcal{A}$ random values.

During the protocol, $\mathcal{A}$ is observed by S and it can thus be determined if $\mathcal{A}$ ever sends an incorrect intermediate result to one of the honest parties.

In general, if any of the ideal functionalities output $(\texttt{corrupt}, j)$ to P_i, then P_i also outputs $(\texttt{corrupt}, j)$. Not mentioning this further, the protocol proceeds in five steps:

1. All parties provide input to $\mathcal{F}_{\text{INPUT}}$. In return they obtain shares of secret sharings $[x^{(j,0)}]$ and $[x^{(j,1)}]$ for $j = 1, \ldots, n$. Nobody knows which sharings are dummy at this point.
2. Each party P_i generates random keys K_i^0 and K_i^1 and commit to them using $\mathcal{F}_{\text{COMMIT}}$ twice.
3. The passively secure protocol is run on both input sets $\{[x^{(j,0)}]\}_{j=1}^n$ and $\{[x^{(j,1)}]\}_{j=1}^n$. This evaluates the circuit C twice. The parties use $\mathcal{F}_{\text{COMMIT}}$ to commit to their shares of the output. All randomness used in the first and second protocol run come from pseudorandom generators seeded by K_i^0 and K_i^1, respectively.
4. The parties query $\mathcal{F}_{\text{INPUT}}$ for the random bit d and the shares of $\{[x^{(j,d)}]\}_{j=1}^n$. They then use $\mathcal{F}_{\text{COMMIT}}$ to reveal the key K_i^d used for the pseudorandom generator for all P_i. Knowing the initial inputs and the seed for the pseudorandom generator used, the entire message trace of all parties is fixed. The parties also open the commitments to the dummy output shares.
5. Each party locally simulates the entire dummy execution to determine if any cheating took place. This amounts to checking for each party whether his input shares of $[x^{(j,d)}]$ (revealed by $\mathcal{F}_{\text{INPUT}}$) and seed K_i^d (revealed by $\mathcal{F}_{\text{COMMIT}}$) together lead to the shares he claims to have obtained of the output (revealed by $\mathcal{F}_{\text{COMMIT}}$) if he follows the passively secure protocol on the messages that other parties would have sent if they followed the protocol on their shares and expanded randomness. If no discrepancies are found, the output shares of the real execution are opened.

   Otherwise, the honest parties must determine who cheated. Note that it is possible for a corrupt party to "frame" an honest party by sending him wrong intermediate results. The honest party cannot tell the difference and will produce incorrect output. $\mathcal{F}_{\text{TRANSMIT}}$ is there to safeguard honest parties against this form of attack. The parties call it to reveal all messages that were received in the dummy execution.

   The parties have already locally simulated the dummy execution so they know the correct message trace. It is therefore simple to match this against the actual message trace revealed by $\mathcal{F}_{\text{TRANSMIT}}$ and pinpoint the first deviation. If P_j made the first mistake, the honest parties output $(\texttt{corrupt}, j)$ and halt.

Figure 5.6: Simple version.

- If $\mathcal{A}$ did not cheat at all, or if $\mathcal{A}$ cheated in both executions, then S simply follows the protocol. In the first case $\mathcal{F}_{\text{SECF}}^{f_C}$ will give S the outputs for the corrupt parties, which S can pass along to $\mathcal{A}$ unchanged. In the second case, $\mathcal{A}$ will be caught with certainty before seeing anything which depend on the honest parties inputs. S can therefore simulate the protocol execution towards $\mathcal{A}$ using random shares only.

- If $\mathcal{A}$ cheats in execution d' (first or second execution), S will send (`cheat`) to $\mathcal{F}$. The functionality then determines if the cheat was successful:

  Detected: The simulator must now ensure that $\mathcal{A}$ believes he cheated in the dummy execution.

  $\mathcal{A}$ will want to query $\mathcal{F}_{\text{INPUT}}$ for the value of d and the shares of the dummy inputs. In response, S sends a response with $d = d'$, which means that $\mathcal{A}$ cheated in the dummy execution. S must also send back shares of the inputs $\{x^{(j,d)} = d_j\}_{j=1}^{n}$ consistent with the shares $\mathcal{A}$ has already seen. At this point $\mathcal{A}$ has only seen the shares it chose for the (*non-qualified*) subset of corrupt parties when $\mathcal{F}_{\text{INPUT}}$ was called initially. S can therefore choose polynomials that agree with these values and correspond to a sharing of the inputs d_j, and finally compute consistent shares of the honest parties using these polynomials.

  If P_j were the first corrupt party who send an incorrect message to an honest party, S will send (`corrupt`, j) to $\mathcal{F}_{\text{SECF}}^{f_C}$.

  Undetected: If the adversary $\mathcal{A}$ was not caught, the functionality responded with (`undetected`) together with the honest parties' inputs. The simulator must therefore make it look as if $\mathcal{A}$ cheated in the execution that was not opened, i.e., the real execution. As above, S can compute polynomials that will give a correct sharing of inputs based on what $\mathcal{A}$ already knows and with $d = 1 - d'$.

  Using these inputs together with the corrupt parties' inputs and outputs, S can now compute the consequence of $\mathcal{A}$'s cheating, i.e., the altered outputs of the honest parties. It passes these outputs to $\mathcal{F}_{\text{SECF}}^{f_C}$ as the honest parties' outputs.

It is clear that the above simulation matches the output of $\mathcal{A}$ in the hybrid world perfectly when $\mathcal{A}$ did not cheat and when $\mathcal{A}$ was foolish enough to cheat in both executions.

When $\mathcal{A}$ cheats in just one execution, S will make the honest parties output (`corrupt`, j) for some corrupt P_j (if $\mathcal{A}$ was detected) or output normal outputs (if $\mathcal{A}$ was undetected). Each of these two cases are picked with probability exactly $\frac{1}{2}$ by the random choice made by $\mathcal{F}_{\text{SECF}}^{f_C}$. We get the same probability distribution in the hybrid world where $\mathcal{F}_{\text{INPUT}}$ picks the bit d uniformly at random.

In total, we can now conclude that the protocol in Figure 5.6 on the facing page computes f_C with $\frac{1}{2}$-SECF security. ■

The above protocol has each party execute the passively secure protocol twice after which each party simulates the actions of all other parties in the dummy execution. In the standard BGW protocol [10], each party has a computational complexity of $\mathcal{O}(n)$ per gate. By asking every party to simulate every other party, we increase the computational complexity to $\mathcal{O}(n^2)$ per gate.

The communication complexity is doubled by running the passively secure protocol twice. In the normal case where the dummy execution is found to contain no errors, the communication complexity is increased no further. When errors *are* detected, every party is sent the messages communicated by every other party. This will again introduce a quadratic blowup, now in the communication complexity. We argued in the introduction that even a small fixed probability of catching misbehavior is enough to deter the parties. Because of that, we expect to find no discrepancies most of the time, and thus obtain the same *communication* complexity as the original protocol within a constant factor. We still have a quadratic blowup in the *computational* complexity. However, local computations are normally considered free compared to the communication, i.e., the network is expected to be the bottleneck. So for a moderate number of parties, this simple protocol can still be quite efficient.

Still, we would like to lower the complexity when errors are detected. Below we propose a slightly more complex protocol which has only a constant overhead in both computation and communication both when no errors are detected and when the parties are forced to do a more careful verification.

If no errors are detected, each party does two protocol executions followed by a check of the input/output behavior of one other party. This is clearly a constant factor overhead compared to the passively secure protocol. When a party is accused, all other parties must check this party. This adds only a linear overhead to the overall protocol, and thus the protocol in Figure 5.7 on the facing page has a linear overhead compared to the passively secure protocol.

It might seem as an overkill in the protocol in Figure 5.7 on the next page to use $\mathcal{F}_{\text{TRANSMIT}}$ for communication and then also have the parties commit to their communication using $\mathcal{F}_{\text{COMMIT}}$. The reason for the commitments is to commit the corrupted parties to what they sent among each other before it is revealed which parties check which parties. If we do not do that, they might decide on which of them was the deviator after the revelation of d and k and thus always pick the deviator to be one which is checked by a corrupted party. For an example of what can go wrong without the commitments the interested reader can refer to Section 5.4.1.

Theorem 5.3 *The protocol in Figure 5.7 on the facing page computes C with $\frac{1}{4}$-SECF security in the $(\mathcal{F}_{\text{TRANSMIT}}, \mathcal{F}_{\text{INPUT}}, \mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{FLIP}})$-hybrid world. This is with threshold t, where $t < n/2$.* □

Proof: The simulator for the protocol in Figure 5.7 on the next page runs like the simulator for the protocol in Figure 5.6 on page 88, except that it must now only output $(\texttt{corrupt}, i)$ to $\mathcal{F}$ if it determines that a message

This is a modification of the protocol in Figure 5.6 on page 88. After running Step 1–3 unchanged, it continues with:

1. All P_i use $\mathcal{F}_{\text{COMMIT}}$ to commit to their view of the protocol, i.e., all messages exchanged between P_i and P_j for all j. This results in commitments $\text{comm}^{(i,0)}_{\{i,j\}}$ for the first execution and $\text{comm}^{(i,1)}_{\{i,j\}}$ for the second, where $\text{comm}^{(m,c)}_{\{i,j\}}$ is the view of P_m of what was sent between P_i and P_j in execution number c.
2. The parties query $\mathcal{F}_{\text{INPUT}}$ for the random bit d and the shares of the dummy inputs. They then use $\mathcal{F}^{n-1}_{\text{FLIP}}$ to flip a uniformly random $k \in \{1,\dots,n-1\}$ that will be used when checking. $\mathcal{F}_{\text{COMMIT}}$ is used by all parties to reveal the key K^d_i used for the pseudorandom generator for all P_i. Finally, the commitments to shares in the output from the dummy execution are opened.
3. Each party P_i checks P_l, where $l = (i-1+k \bmod n)+1$, i.e., he checks P_{i+k} with wraparound from P_n back to P_1.

   The commitments $\text{comm}^{(j,d)}_{\{l,j\}}$ and $\text{comm}^{(l,d)}_{\{l,j\}}$ are opened to P_i, i.e., the committed views of P_l and P_j of what was exchanged between them. If there is a disagreement, then P_i broadcasts a complaint and P_l and P_j must decommit to all parties and use $\mathcal{F}_{\text{TRANSMIT}}$ to show which messages they received from the other. This will clearly detect at least one corrupt party among P_l and P_j if P_i was honest, or reveal P_i as corrupt if the commitments were equal after all, i.e., if P_i made a false accusation.

   If all committed views agree, then P_i simulates the local computations done by P_l and checks whether this leads to the shares of the dummy output opened by P_l and the messages sent according to $\text{comm}^{(l,d)}_{\{l,j\}}$. If a deviation is found, P_i broadcasts an accusation against P_l, and all parties check P_l as P_i did. If they verify the deviation they output $(\texttt{corrupt}, l)$, otherwise they output $(\texttt{corrupt}, i)$.
4. If no accusations were made, the output of the real protocol execution is opened.

Figure 5.7: Efficient version.

trace for a corrupt party P_i was checked by an honest party, and it must do while maintaining the same probability distribution as in the hybrid world.

As before, S will simulate $\mathcal{A}$ and observe the messages sent to honest parties. As soon as an incorrect message is observed in execution d' and all parties committed to their communication with the other parties, we know there exists an offset $k' \in \{1, \dots, n-1\}$ for which an honest P_i would catch a corrupt P_l, where $l = (i - 1 + k' \bmod n) + 1$ in execution d':

- If two parties P_l and P_j committed to $\mathrm{comm}_{\{l,j\}}^{(l,d')} \neq \mathrm{comm}_{\{l,j\}}^{(j,d')}$, then one of them is corrupted, P_l say, and we pick k' such that P_l is checked by an honest P_i.

- If $\mathrm{comm}_{\{l,j\}}^{(l,d')} = \mathrm{comm}_{\{l,j\}}^{(j,d')}$ for all pairs of parties, then the wrong message sent to an honest party in execution d' implies that some party P_l is committed to values which are not consistent with an execution of the protocol, and we pick k' to ensure that P_l is checked by an honest party.[2]

The simulator sends (`cheat`) to $\mathcal{F}_{\mathrm{SECF}}^{fc}$. We have two outcomes:

Detected: Set $d = d'$ and sample k at random such that P_l is checked by an honest party.

Undetected: Set $d = d'$ with probability $\frac{1}{3}$, and $d = 1 - d'$ otherwise. Sample $k \in \{1, \dots, n-1\}$ such that P_l is checked by an honest party with probability $\alpha = \frac{4}{3}((n-t)/(n-1) - \frac{1}{4})$.

If $\mathcal{A}$ did not cheat, S selects d and k as in the hybrid protocol. The simulation continues as in the hybrid world with these choices for d and k. The ideal world output clearly matches the hybrid world.

When $\mathcal{A}$ did cheat, we will show that d and k are picked with the correct distribution. First note that S pick $d = d'$ with probability $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{1}{3} = \frac{1}{2}$, as in the hybrid world.

For the selection of k, note that a cheating party will always have a unique distance to every honest party. These distances make up a subset of $\{1, \dots, n-1\}$ of size $n - t$. The cheater is caught exactly when the offset is picked within this subset. This happens with probability $(n-t)/(n-1)$ in the hybrid world. The simulator picks k among the indices of honest parties with the same probability: $\frac{1}{4} + \frac{3}{4}\alpha = (n-t)/(n-1)$. We conclude that S will simulate the hybrid world. ■

5.4.1 Chinese Whispers

In our protocol we have all parties commit to the messages they exchanged before the revelation of d and k. This might seem superfluous, as $\mathcal{F}_{\mathrm{TRANSMIT}}$ was already used to commit the parties to their messages. The difference

[2]Note that P_l need not be the one who sent the incorrect message to the honest party — P_l may have behaved locally consistent given its inputs — but S will be able to find a first deviator, and it will clearly not be one of the honest parties.

is that $\mathcal{F}_{\text{COMMIT}}$ also binds the corrupted parties to what they have exchanged among themselves, whereas $\mathcal{F}_{\text{TRANSMIT}}$ allows the corrupted parties to change their mind on what they exchanged among themselves. We give an example demonstrating that it makes a difference whether the corrupted parties are committed to was sent among them before we reveal who checks who.

Consider the following passively secure protocol CHINESE-WHISPERS for computing the identity function $(x_1, \dots, x_n) \mapsto (x_1, \dots, x_n)$.

1. In round 1 party P_1 should send 0 to P_2.
2. For $i = 2, \dots, n-1$, in round i party P_i should take the bit sent from P_{i-1} and send it to P_{i+1}.
3. In round n all parties $i = 2, \dots, n$ should behave as follows: If they received a 0 from P_{i-1} in round i, then they should output x_i. If they received a 1 then they should broadcast x_i and then output x_i.

CHINESE-WHISPERS is clearly passively secure. Consider now a setting where $P_1, \dots, P_t$ are corrupted and where P_t sends 1 to P_{t+1} in round t, a serious deviation which lets the corrupted parties learn all the inputs of the honest parties. If we now let the honest parties check a corrupted party at random, then with probability going to 1, as n grows, there will be a corrupted party P_i, $i \in \{2, \dots, t-1\}$, which is checked by a corrupted party. The adversary can then behave as if $P_1, \dots, P_{i-1}$ all received and sent 0, that P_i received 0 and sent 1, and $P_{i+1}, \dots, P_t$ all received and sent 1. Now all the parties except P_i followed the protocol, and since the local consistency of P_i is not checked this goes unnoticed. If we had forced the adversary to picked the party P_i having received 0 and sent 1 before the assignment of who checks who was made public, then P_i would have been checked by an honest party with probability at least $\frac{1}{2}$, which is what we need.

CHINESE-WHISPERS itself does not fit as a protocol we can transform, but can be embedded into any such protocol to render it vulnerable to the above attack.

5.5 Implementation of Sub-Protocols

In this section we sketch how to implement the sub-protocols described above. In all sub-protocols we will need a tool for stopping the protocol "gracefully" when corruption is detected This is done by all parties running the following rules in parallel.

1. If a party P_i sees that a party P_d deviates from the protocol, then P_i signs $(\texttt{corrupt}, d)$ to get signature y_i and sends the signature to all parties. Then P_i outputs $(\texttt{corrupt}, d)$.
2. If P_k received a signature y_i on $(\texttt{corrupt}, d)$ from $t+1$ distinct parties P_i, it considers these as a *proof* that P_d is corrupted, sends this proof to all parties, outputs $(\texttt{corrupt}, d)$, waits for one round and then terminates all protocols.

3. If P_k receives a proof that P_d is corrupt from any party, it relays this proof to all parties, outputs $(\texttt{corrupt}, d)$, waits for one round and then terminates all protocols.

If the signature scheme are unforgeable and only corrupted parties deviate from the protocol, then the protocol has the following two properties, except with negligible probability.

Detection soundness: If an honest party outputs $(\texttt{corrupt}, d)$, then P_d is corrupt.

Common detection: If an honest party terminates the protocol prematurely, then there exists P_d such that *all* honest parties have output $(\texttt{corrupt}, d)$.

The reason why the relayer P_r waits for one round before terminating is that P_r wants all other parties to have seen a proof that P_i is corrupt before it terminates itself. Otherwise the termination of P_r would be considered a deviation and an honest P_r could be falsely detected. In the following we do not always mention explicitly that the detection sub-protocol is run as part of all protocols.

Transmission Functionality

The transmission protocol can run in two modes. In *cheap mode* $\mathcal{F}_{\text{TRANSMIT}}$ is implemented as follows.

1. On input $(\texttt{transmit}, \mathit{mid}, m)$ party $P_{s(\mathit{mid})}$ signs (mid, m) to obtain signature σ_s and sends $(\mathit{mid}, m, \sigma_s)$ to $P_{r(\mathit{mid})}$.

2. On input $(\texttt{transmit}, \mathit{mid})$ party $P_{r(\mathit{mid})}$ waits for one round and then expects a message $(\mathit{mid}, m, \sigma_s)$ from $P_{s(\mathit{mid})}$, where σ_s is a valid signature from $P_{s(\mathit{mid})}$ on (mid, m). If it receives it, it outputs (mid, m).

3. On input $(\texttt{reveal}, \mathit{mid}, i)$ party P_j, if it has output (mid, m) at some point, sends $(\mathit{mid}, m, \sigma_s)$ to P_i, which outputs (mid, m) if σ_s is valid.

It is easy to check that this is a UC secure implementation under the following restrictions:

Synchronized input from honest parties: If some honest party receives input $(\texttt{transmit}, \mathit{mid})$, then all honest parties $P_i \neq P_{s(\mathit{mid})}$ receives input $(\texttt{transmit}, \mathit{mid})$ and if $P_{s(\mathit{mid})}$ is honest it gets input $(\texttt{transmit}, \mathit{mid}, m)$ for some m.

Signatures: Even corrupted P_s send along the signatures σ_s.

The restriction *Synchronized input from honest* can be enforced by the way the ideal functionality is used by an outer protocol, i.e., by ensuring that the honest parties agree on which message identifiers are used for which message in which rounds. This is the case for the way we use $\mathcal{F}_{\text{TRANSMIT}}$. The restriction *signatures* is unreasonable, and we show how to get rid of it below. We need the rule *Do not commit corrupt to corrupt* in $\mathcal{F}_{\text{TRANSMIT}}$ as we cannot prevent a corrupt P_s from providing a corrupt P_i with signatures on arbitrary messages, i.e., we cannot commit the corrupted parties to what they have sent among themselves.

As mentioned, the above implementation only works if all senders honestly send the needed signatures. If at some point some P_r does not receive a valid signature from P_s, it publicly accuses P_s of being corrupted and the parties switch to the below *expensive mode* for transmissions from P_s to P_r.

1. On input $(\texttt{transmit}, mid, m)$ party $P_{s(mid)}$ signs (mid, m) to obtain signature σ_s and sends (mid, m, σ_s) to all $P_i \neq P_{s(mid)}$.

2. On input $(\texttt{transmit}, mid)$ parties $P_i \neq P_{s(mid)}$ wait for one round and then expects a message (mid, m, σ_s) from $P_{r(mid)}$, where σ_s is a valid signature of $P_{s(mid)}$ on (mid, m). If P_i receives it, it sends (mid, m, σ_s) to $P_{r(mid)}$.

   Otherwise, it sends a signature γ_i on $(\texttt{corrupt}, i)$ to all parties.

3. On input $(\texttt{transmit}, mid)$ party $P_{r(mid)}$ waits for two rounds and then expects a message (mid, m, σ_s) from each P_i, where σ_s is a valid signature of $P_{s(mid)}$ on (mid, m). If it arrives from some P_i, then P_r outputs (mid, m).

Note that now each round of communication on $\mathcal{F}_{\text{TRANSMIT}}$ takes two rounds on the underlying network. Between two parties where there have been no accusations, messages are sent as before (Step 1 in the above protocol) and the extra round is used for silence — it is necessary that also non-accusing parties use two rounds to not lose synchronization.

If P_s sends a valid signature to just one honest party, then P_r gets its signature and can proceed as in optimistic mode. If P_s does not send a valid signature to any honest party, then all $n - t$ honest P_i send γ_i to all parties and hence all honest parties output $(\texttt{corrupt}, s)$ in the following round, meaning that P_s was detected. Using these observations it can easily be shown that the above protocol is a UC implementation of $\mathcal{F}_{\text{TRANSMIT}}$ against covert adversaries with deterrence factor 1. Note that it is not a problem that we send m in cleartext through all parties, as an accusation of P_s by P_r means that P_s or P_r is corrupt, and hence m need not be kept secret.

We skipped the details of how the accusations are handled. We could in principle handle accusations by using one round of broadcast after each round of communication to check if any party wants to make an accusation. After broadcasting the accusations, the appropriate parties can then switch to expensive model. To avoid using a Byzantine agreement primitive in each round, we use a slightly more involved, but much cheaper technique which communicates less than n^2 bits in each round and which only uses a BA primitive when there are actually some accusations to be dealt with. The full details are given in [38].

In cheap mode, using $\mathcal{F}_{\text{TRANSMIT}}$ adds an overhead $N\kappa$ bits compared to plain transmission, where κ is the length of a signature and N is the number of messages sent. In expensive mode this overhead is a factor n larger.

Commitment Functionality

The protocol uses a one-round UC commitment scheme with a constant overhead (commit to κ bits using $\mathcal{O}(\kappa)$ bits), which can be realized with static security in the PKI model [5] given any mixed commitment scheme [31] with a constant overhead. Concretely we can instantiate such a scheme under Paillier's DCR assumption. Note that opposed to Barak et al. [5] we do not need a setup assumption: we assume honest majority and can thus, once and for all, use an active secure MPC to generate the needed setup [54]. The protocol also uses an error-correcting code (ECC) for n parties which allows to compute the message from any $n - t$ correct shares.

If one is willing to use the random oracle model, UC commitment can instead be done by calling the oracle on input the message to commit to, followed by some randomness. In practice, this translates to a very efficient solution based on a hash function.

The protocol proceeds as follows.

1. On input $(\texttt{commit}, \mathit{cid}, m)$, $P_{s(\mathit{cid})}$ computes an ECC $(m_1, \dots, m_n)$ of m. The sender then computes $c_i \leftarrow \text{commit}_{pk_i}(m_i)$ and sends c_i to P_i via $\mathcal{F}_{\text{TRANSMIT}}$.
2. On input $(\texttt{reveal}, \mathit{cid}, r)$, P_i opens each c_i to P_i. The opening is sent via $\mathcal{F}_{\text{TRANSMIT}}$. If any P_i receives an invalid opening, it transfers c_i and m_i to all parties and P_s is detected as a cheater. Otherwise, P_i transfers c_i and the opening to P_r.
3. Then P_r collects validly opened c_i. Let I be the index of these and let m_i be the opening of c_i for $i \in C$. If $|I| < n - t$, then P_r waits for one round and terminates.[3] If $(m_i)_{i \in I}$ is not consistent with a codeword in the ECC, then P_r transfers $(c_i)_{i \in I}$ and the valid openings to the other parties which detect P_s as corrupted. Otherwise, P_r uses $(m_i)_{i \in I}$ to determine m and outputs (cid, m).

Assuming that a commitment to ℓ bits have bit-length $\mathcal{O}(\max(\kappa, \ell))$, where κ is the security parameter, the complexity of a commitment to ℓ bits followed by an opening is $\mathcal{O}(n \max(\kappa, \ell/n)) = \mathcal{O}(n(\kappa + \ell/n)) = \mathcal{O}(\ell + n\kappa)$. This is assuming that there are no active corruptions, such that $\mathcal{F}_{\text{TRANSMIT}}$ has constant overhead.

Flip Functionality

To implement $\mathcal{F}^{B}_{\text{FLIP}}$ the parties proceed as follows.

1. On input $(\texttt{flip})$, all P_i commit to a uniformly random $k_i \in \{0, \dots, B - 1\}$.
2. In the next round all P_i reveal k_i to all parties.
3. All parties output $k = \sum_{i=1}^{n} k_i \bmod B$.

[3]Since we assume that at most t parties are corrupted, we can assume that either P_s is detected or P_r receives $n - t$ commitments with corresponding valid decommitments.

Under the condition that the protocol is used by the honest parties in a way that guarantees that they input (`flip`) in the same round, the argument that the protocol implements the functionality against a covert adversary (with deterrence 1) is straight forward.

Input Functionality

The input functionality can be implemented using a VSS with a multiplication protocol active secure against $t < n/2$ corruptions. The VSS should have the property that it is possible to verifiable reconstruct the secret and the share of all parties given the shares of the honest parties — standard bivariate sharing has this property. We sketch the protocol.

1. Each P_i deals a VSS $[\![x_i]\!]$ of its input x_i.
2. The parties use standard techniques to compute a VSS $[\![d]\!]$ of a uniformly random $d \in_R \{0, 1\} \subset \mathbb{F}$.
3. For each input $[\![x_i]\!]$ the parties use an actively secure multiplication protocol to compute $[\![x^{(i,0)}]\!] = [\![d_i \cdot x_i]\!]$ and $[\![x^{(i,1)}]\!] = [\![(1 - d_i) \cdot x_i]\!]$. Each P_i takes its output to be $(x_i^{(j,0)})_{j=1}^n$ and $(x_i^{(j,1)})_{j=1}^n$, where $x_i^{(j,c)}$ is its point on the polynomial used by the sharing $[\![x^{(j,c)}]\!]$. The other values of the VSS are internal to the implementation of $\mathcal{F}_{\text{INPUT}}$ and only used for the below command.
4. On input $(\texttt{reveal}, i, k)$ the parties reconstruct $[\![d]\!]$ and all $[\![x^{i,d}]\!]$ towards P_k and P_k computes the points $x^{(j,d)}$ of P_j in all sharings and output $(x_i^{(j,d)})_{j=1}^n$.

5.6 Conclusion

Protection against an honest but curious adversary is often not enough. Most real-world scenarios will at least demand that cheating can be detected and that the guilty can be prosecuted using the normal legal system. A passively secure protocol does not offer any of these guarantees — if a corrupt party deviates from the protocol there will normally not be any mechanism in place to detect this. Additionally, if cheating can be detected, it might be the case that the honest parties cannot place the blame on any one party.

The notion of covert adversaries put these issues straight. For a protocol to securely realize a functionality in the presence of covert adversaries, the honest parties must be able to detect cheating, at least sometimes. When cheating is detected the protocol must ensure that it is done accurately.

We have shown how it is possible to compile a protocol secure against a semi-honest adversary into a protocol secure against a covert adversary. The cost is roughly a doubling in communication complexity and either a linear or quadratic overhead in computational complexity. Considering that a passively secure protocol (such as the well-known BGW protocol [10]) is already highly efficient, we find these overheads to be quite acceptable.

While we have not presented a implementation of our transformation, we believe that it is well suited for implementation in a framework like VIFF. The main obstacle is that we must have careful control over the randomness used. The program counters in VIFF would be the key tool to ensure that the honest parties agree on the randomness, i.e., the randomness for a given operation would be $K_i^0 \parallel pc$ and $K_i^1 \parallel pc$ in the first and second execution, respectively. Realizing this could be a nice project for a master's thesis.

Chapter 6

Integrity Protection for Revision Control

We started this dissertation by studying protocols secure against passive adversaries in Chapters 2 and 3. In Chapter 4, we described a protocol secure against active adversaries. These protocols have all been implemented. Chapter 5 described protocols under the more exotic notion of a covert adversary, but presented no implementation. In this chapter, we will yet again switch setting and investigate the possible consistency guarantees for revision control when clients use a malicious server. The material herein was published at the ACNS 2009 conference [19].

6.1 Introduction

Nowadays people from all continents and all time zones collaborate together in global companies and other organizations, formal or not. Prominent examples are open-source development projects, such as the GNU/Linux operating system. For exchanging documents and storing the output of their work, they typically rely on a remote provider that hosts a shared storage service. An important class of such storage services are *revision control systems* (RCS) that facilitate collaboration on a set of documents that belong together and exist in multiple versions.

Although the collaborators trust the storage provider to preserve their documents, there are good reasons to verify that the provider indeed behaves correctly. For example, there are reported cases of break-ins to popular open-source repositories, where security-critical operating system code may have been altered undetectedly [26]. In cooperations that span multiple organizations, the storage provider often is a third party with little interest in the resulting work. Generally, verification reduces trust in the storage provider. To protect against faulty or corrupted storage providers, cryptographic protection methods are needed.

In this chapter, we address *cryptographic integrity protection* for revision control systems. They represent the most important kind of multi-user storage and collaboration tools today, together with Wikis. We assume that clients are isolated and communicate directly with each other only under special circumstances; in fact, many clients may not even know each other. Our goal is to obtain a strong guarantee that a potentially faulty service provider has not altered the shared data.

The clients may use public-key signatures to authenticate their operations; this ensures that no unauthorized party can forge data in the repository. But in our model, replay attacks by a malicious storage server cannot be prevented, i.e., the server may return an outdated value to a reader, omitting a more recent update by another client. SUNDR [66] was the first storage system to address this problem by providing every client with a *fork-linearizable* view of the shared data. This notion ensures that all operations that a client does see are observed in the agreed linearization order, and if the server causes the views of two clients to differ in a single operation, they may never again see each others operations. This makes even subtle changes to the stored data easily detectable.

In this work, we describe the design and implementation of a consistent revision control system that preserves fork-linearizability. It relies on the fork-linearizable storage protocol of Cachin et al. [20] that reduces the communication overhead by an order of magnitude compared to the protocol of SUNDR. Our implementation extends the popular revision control system Subversion in a modular way.

The challenge in our work lies in the details of the integration of the fork-linearizable storage protocol with a revision control system. First, the abstract storage protocol uses only simple read and write operations on a file, whereas the revision control system implements transactions that usually read and update many files at once. Second, our goal is to be transparent to the server side of the underlying revision control system; therefore, we still rely on it to serialize concurrent updates. The implementation of our consistent revision control system merely extends this serialization order with the cryptographic consistency guarantees. Finally, the cryptographic operations must not be overly expensive; our hash-tree implementation exploits caching of the tree nodes and maintains them in the Subversion repository itself. This adds only little extra storage on top of the unchecked repository and requires few more operations.

6.1.1 Related Work

Protecting the integrity of stored data is an important question with a long history. But good solutions are needed today more than ever before [4], because personal and institutional data is stored and archived electronically. We describe here only a selection of the literature that uses the same model as our system, i.e., a remote untrusted bulk storage provider that offers read and write operations, accessed by one or more isolated clients with a small trusted memory.

Blum et al. [13] formalize the problem of memory checking and present the classical protection scheme based on hash trees [75]. With a memory consisting of n items and with random-access read and write operations to the memory, hash trees incur an overhead of $O(\log n)$ cryptographic operations. Several storage-system prototypes protect data integrity using hash trees; TDB [68] and SiRiUS [52] are two prominent examples. A similar approach has been proposed for protecting a CPU equipped with a trusted cache against unauthorized modifications to the main memory [25]. For database systems accessed through a query interface, Mykletun et al. [77] analyze the cost of integrity protection with cryptographic signatures that can be aggregated to reduce the space overhead. The recent work of Papamanthou et al. [82] shows how an array of data items can be authenticated with constant overhead for reading and sub-linear overhead for writing.

All systems mentioned so far consider either only one client or construct an abstraction of the trusted memory between clients (e.g., with digital signatures). The SUNDR system [66] is the only one protecting integrity for storage space shared by multiple clients that do not communicate among themselves. SUNDR guarantees linearizability when the storage service is correct and fork-linearizability when the service is faulty.

In distributed revision control, the two popular systems Git (`http://git.or.cz/`) and Mercurial (`http://mercurial.selenic.com/`) both employ hash values for identifying revisions. Without digital signatures, a corrupted server can trivially present modified changesets to a client (a changeset is the unit of an update between two revisions). The clients will no longer agree on the hashes identifying the revisions, but the server can keep passing content back and forth between the clients. Even if every client would sign all its updates, replay attacks would still be possible despite the use of hashes. Distributed revision control systems explicitly allow offline commits, and so the server can withhold changesets and claim that it has not seen them yet.

In practice, many open-source projects also publish digests or even cryptographic signatures on every release of their code. But since the cryptographic operations for authentication and verification are not transparently integrated with the storage mechanism, they require some manual intervention; hence, this method is not suitable for everyday collaboration.

6.1.2 Overview

The remainder of the chapter is organized as follows. Section 6.2 presents our system model and the design for our consistent revision control system. Section 6.3 describes our implementation. We evaluated our prototype system and present the results in Section 6.4. Section 6.5 discusses some limitations of our system and presents an outlook.

6.2 Design

This section presents the design of our consistent revision control system. In Section 6.2.1, we first describe the assumptions used by our system and the properties that it guarantees. We then introduce our abstract consistent storage service in Section 6.2.2, which provides a fork-linearizable storage space for small values, and review those properties of revision control systems that are relevant for our work in Section 6.2.3. In Section 6.2.4, we explain the design of the consistent revision control system.

6.2.1 Model

The system consists of an a priori unknown number of clients and a storage server. The server provides an abstraction of consistent shared storage to the clients, who access it using operations to read and write data, and with operations to control different revisions of the data. We assume that all clients are correct and follow the protocol. The server may be faulty or *corrupted* and deviate from the protocol in arbitrary ways, but not break any cryptographic primitives.

The clients never communicate with each other directly, they communicate only via the server. This model is convenient and realistic because the clients are not required to know each other, the network topology may prevent direct communication between them, and they can operate independently of each other. Revision control systems enable a convenient form of computer-supported cooperative work, because the collaborators can contribute at different times and from different locations.

We assume that each client is identified by a public key/private key pair, signed by a trusted certification authority (CA). Every client trusts one or more CAs, whose root keys it stores in a local directory in the form of self-signed X.509 certificates. Clients identify each other only by their public key; more precisely, clients accept every public key as the identity of another client when the key is accompanied by a certificate from a trusted CA. The system distributes the keys among clients as needed; a client only needs the trusted CA keys before it starts to interact with the storage service. Representing client identities by keys simplifies key distribution considerably [73].

Every client maintains a small trusted memory, whose size is independent of the size of the shared storage space. In order to prevent a corrupted server from introducing unauthorized modifications to the shared data, clients sign all their write operations and verify the integrity of the data they read using digital signatures. But since the clients do not communicate with each other, we cannot prevent that the server completes a write operation of one client, and still returns stale data to another client.

The notion of *fork-linearizability* provides the next-best notion of consistency in this model [20, 72]. It ensures that all operations in the view of every client are legal in the sense that data returned by a read operation has been written by the indicated client, and that when the server causes the views of two clients to differ, even in a single operation only, then these clients may not see any further operation of each other afterwards.

Our goal is to implement a storage service that provides read and write operations, which execute atomically and according to their specification whenever the server is correct; when the server is faulty, the storage service still provides fork-linearizability. We refer to the work of Cachin et al. [20] for a formal notion that captures this requirement under the name of a *fork-linearizable emulation* of a storage service on a potentially corrupted server. In the subsequent sections, we explain how fork-linearizable storage is implemented by our consistent storage service and by our consistent revision control service.

Naturally, a corrupted server may simply refuse to cooperate, and then the clients will have to reconstruct the shared data from their own records. But this attack cannot be prevented. There is no easy solution to this problem, except to choose a more trustworthy server.

On the other hand, a fork-linearizable emulation ensures that the server cannot violate the consistency of the storage service and hide this attack from clients that are suspicious. Even if the clients communicate out-of-band only occasionally, for example, by sending email to each other directly, or through a discussion forum on a project website, they are guaranteed to immediately discover any inconsistencies that were introduced ever by a faulty server.

A more subtle attack occurs when the server conspires with a client and violates the assumption that clients are correct. The current design does not prevent such behavior, but our system provides some means that help the correct clients to recover from such attacks. We discuss these in Section 6.5.

6.2.2 Consistent Storage Service

The *consistent storage service* (CSS) provides a simple interface for reading and writing short byte arrays and ensures fork-linearizability with an untrusted server. There is no hard limit on the size of the stored byte arrays, but the service is designed for sizes up to 10 or 100 KiB because all values are transiently kept in main memory.

CSS provides one storage location for every client, called a *register*. The client is the only one who may write to its register, but all clients may read from it, and there is an operation that reads all registers in a single step. Formally, CSS combines an array of single-writer/multi-reader registers [59] with an atomic snapshot object [1].

The service provides the following interface to clients, expressed as method invocations:

getkeys() returns a list of all client identities that are known to the server so far, represented by their public keys. The server learns the identity of a client as soon as the client invokes its first method. A client may use the output of the operation in subsequent queries.

write(*data*) stores *data* in the register of the client at the server, overwriting data previously stored there. In CSS, a client may only write to its own register.

read(*key*) reads the register identified by the public key *key* and returns the stored data. If no such data exists, the operation returns none.

readall() reads all registers in one step and returns a list of pairs (*key*, *data*), representing all registers stored by the server; every pair contains the corresponding client *key* and the stored *data*. This method is equivalent to invoking getkeys(), followed by invoking read(*key*) for all *key* values returned, all in one atomic step. Its purpose is to give a consistent view of all registers.

We use the *lock-step protocol* of Cachin et al. [20] to implement CSS. The protocol is noteworthy for using the server only as intermediary storage; in particular, the server does not perform any cryptographic operations. The protocol has been modified from using a fixed number of clients to handle an a priori unbounded number of clients that are identified only by a public key. Instead of using vectors, versions are represented by an associative array that maps every known client key to the corresponding timestamp. The clients maintain some state in their local memory and save it on persistent storage between operations. The protocol is shown in Figure 6.1.

The lock-step protocol has the drawback of not being wait-free [59] because when the server waits for the *commit* message from a client, no other client can proceed with an operation. Mazières and Shasha [72] and Cachin et al. [20] both present seemingly more efficient protocols that allow some client operations to proceed in parallel. However, it has been shown that in all fork-linearizable storage emulation protocols, a reader must wait for a concurrent writer [20].[1]

We therefore chose to implement CSS with the lock-step protocol for the following reasons: First, the addition of the readall operation introduces the above conflict between readall and *every* write operation. We know that our consistent revision control application (described in Section 6.2.4) will use only write and readall operations, and we expect that they occur about equally often. Hence, the potential for exploiting concurrency is reduced to concurrent read operations. Second, the protocol allowing for concurrent operations is considerably more involved than the lock-step protocol. The small potential gain did not merit the added implementation complexity.

6.2.3 Revision Control

A *revision control system* (RCS) provides operations for storing and retrieving multiple versions of the same set of documents. It facilitates collaboration among multiple users, who may work independently with the information. The RCS assigns revision numbers to the documents and maintains a history of all versions. The documents usually consist of a hierarchical set of files and directories in a file system. Revision control systems are an important collaboration tool, as can be seen from the large number of existing systems (Wikipedia's "List of revision control software" lists 64 systems as of September 2008).

For the purpose of designing our consistent RCS, we describe here the main features of a *generic* centralized RCS. A *centralized* RCS uses a dedicated server for controlling revisions and storing the history, in contrast to a *distributed* RCS, where this task is shared by all users. Our RCS is modeled after two popular RCS for source code, CVS (`http://www.nongnu.org/cvs/`)

[1]Weaker semantics than fork-linearizability can give rise to wait-free storage emulation protocols [21].

Preliminaries. CSS stores a register value $data_{key}$ for each client identified by *key*. Only the client identified by *key* may write to $data_{key}$, but every client may read from any register. Every client locally maintains a timestamp that it increments during every operation. We call an array of timestamps a *version*; a version is an associative array V that maps keys to timestamps, denoted by $V[key] = t$. We write $V[key] = \bot$ if $V[key]$ is not defined. Versions acts as a vector clock for ordering operations. Two versions V and W are ordered so that V is *smaller than or equal to* W whenever $V[key] \leq W[key]$ for all values *key* such that $V[key] \neq \bot$.

Client state. The client maintains a version T representing its last completed operation. Note that a client identified by *key* finds its own timestamp in $T[key]$. For simplicity of the description, we assume the client also keeps a copy of its own data value $data_{key}$ and writes it back during every read operation. (In the implementation, it only stores a collision-resistant hash of the data value and sends that in a read operation; in a write operation, it sends the data value.)

Server state. The server stores the register values in an associative array X, where $X[key] = (data_{key}, \sigma_{key})$, representing the register value and a digital signature issued under *key* on the string value $\|$ $data_{key} \| t$, where t is a timestamp equal to $T[key]$ when the client completed the operation that wrote $data_{key}$.

The server also keeps information from the last completed operation: the version V associated to it, the key *last* identifying the client performing the operation, and a digital signature ω under key *last* on commit $\| V$.

Operation. When a client identified by *key* invokes a write, read, or readall operation, it sends the request together with *key* in a *submit message* to the server. The server sends a *reply message*, containing the version V, the key *last*, and the accompanying signature ω from the last operation. In a read operation for register identified by *rkey*, the server also sends the register value $X[rkey] = (data_{rkey}, \sigma_{rkey})$. In a readall operation, the server adds all register values X. The server then waits for a *commit* message from this client and does not process any messages from other clients.

The client verifies that the reply message contains valid data: the version V must be at least as large as its own version T, the entry $V[key]$ must be equal to its own timestamp $T[key]$, and the signature ω on commit $\| V$ must be valid under key *last*. In a read or readall operation, the client also verifies that σ_{rkey} is a valid signature under *rkey* on the string value $\| data_{rkey} \| V[rkey]$, either for only one *rkey* in a single-register read or for all values *rkey* such that $X[rkey] \neq \bot$ in a readall operation. When the client detects any inconsistency in the reply, it considers the server to be faulty, generates an alarm, and aborts.

After the client has successfully verified the reply, it adopts the received version V as its own version T, increments its timestamp $T[key]$, and signs the new version T, resulting in a signature φ. It issues another signature σ on value $\| data_{key} \| T[key]$, binding its data value to the timestamp. Then it sends a *commit message* to the server, containing T, φ, $data_{key}$, and σ.

When receiving the commit message, the server stores T, *key*, and φ as its version V, client key *last*, and signature ω that represent the last operation. The server also updates $X[key]$ with the received value $data_{key}$ and σ.

Figure 6.1: CSS using the lock-step protocol (adapted from [20, 72]).

and Subversion (`http://subversion.tigris.org/`); they both allow users to update the same document concurrently.

We expect the client interface of an RCS to provide the following main operations:

Checkout: A `checkout` operation transfers all documents from the server repository to the client. It creates a copy of the files and directories on the client, called the *working copy*. All editing takes place there. The RCS also supports attributes attached to documents and version control for them.

Commit: After adding, modifying, or deleting some files in the working copy, the client wants to transfer the changes back to the central server, thereby making the changes visible to other clients. The client does this with a `commit` (or `checkin`) operation. Its effect is to create a new *revision* with a distinct identifier, called the *revision number*. We assume that revision numbers in a sequence of commits issued by multiple clients increase monotonically over time.

Update: An `update` operation transfers the most recent revision of all files from the server to the client and updates the working copy accordingly. The system also supports updating to a revision with a particular revision number.

When a client has modified some files locally and wants to commit the changes, it may have to perform an update, before the RCS allows a commit operation. This happens when some modifications of the client overlap and *conflict* with modifications committed by others. In this case, the commit operation will fail, the client is told to first update its working copy and to merge the concurrent changes, before the client may attempt another commit.

Typical RCS also support operations to populate the server repository with a set of documents initially, to rename repository contents, to create branches and merge them again, and to tag revisions with keywords. These operations may be present, but are not our main focus because they can be expressed as variations of the above three main operations.

We assume that all operations are transactional so that their changes either take effect in one atomic step on the server, or leave no trace in the repository in case of a failure.

6.2.4 Consistent Revision Control

Our *consistent revision control system* (CRCS) implements a revision control system that protects the integrity of the repository against a corrupted server. CRCS provides the same operations as an ordinary RCS and emulates a fork-linearizable storage service on the repository. We achieve fork-linearizability in terms of the checkout and update operations of CRCS, which implement a `read` operation on the repository, and in terms of the commit operation, which implements a `write` operation on the repository.

Fork-linearizability for a revision control system guarantees the following. Suppose a client A updates its working copy with CRCS to some revision number r. If A sees even a single file that was committed by another client B in revision r, then fork-linearizability implies that all files in client A's working copy have been cryptographically verified and are equal to those committed by B in revision r. Conversely, if there exists a more recent revision $s > r$ committed by a third client C, and the server hides revision s from A, then A can never again update to any revision committed by C or by anyone who updated to s. Because of this all-or-nothing implication of fork-linearizability, one can very easily detect even subtle modifications of a single file by a corrupted server.

We implement CRCS by combining our CSS with an unmodified RCS. CRCS computes a hash tree [75] over the set of documents in the repository and basically stores the root hash of the tree using CSS. This construction extends the integrity guaranteed by CSS from the root hash to the entire data. Suppose every client commits changes to CRCS by first committing its working copy using RCS, thereby obtaining a revision number r, computing the new root hash h, and then writing the tuple (r, h) to its register. This stores all information in CSS that another client needs for updating its working copy to the most recent revision and for verifying its integrity. But because CRCS also supports cryptographically verified update operations for previous revisions in the repository, the design is more complex.

Every client maintains a *revision log L* with information about every revision that it committed. The revision log is a list of tuples (r, h, c), denoting the revision number r, the root hash h, and a *revision commitment c*, sorted chronologically (i.e., according to r). Let H denote a collision-free cryptographic hash function. The revision commitment binds together all previous commit operations of the client in a hash chain; when committing revision r with hash h, the client computes c as $H(r \parallel h \parallel c')$, using the revision commitment c' from the last tuple in L (or $c' = \perp$ if L is empty). The same chaining scheme has been used in many other timestamping and data authentication algorithms [58].

For the description of the CRCS algorithm below, assume that every client stores its complete revision log in L. For increased efficiency, an implementation may actually maintain only the last tuple of L in CSS and keep the rest of L in untrusted shared storage; the collision resistance of H guarantees the uniqueness of every revision log given its last revision commitment.

The client proceeds now as follows to implement the main operations of CRCS. If one of the checks in the algorithm fails, the client generates an alarm and aborts.

Checkout: To check out the highest revision, the client invokes the readall() operation of CSS and determines the largest revision number r from the returned revision logs and the corresponding root hash h. After invoking checkout of RCS for revision r, the checkout algorithm recomputes the hash tree on the working copy and verifies that its root hash is equal to h.

Commit: The client first calls the update operation of CRCS (see below) to bring its working copy to the most recent revision according to CSS. Then it commits the working copy with RCS to obtain a new revision number r. If this fails, the operation aborts and the client is told to update and to try again. If all goes well, the client computes the root hash h of the hash tree on its working copy, extends the client's revision log L with r and h, and invokes write(L) from CSS.

Update: The update operation is very similar to checkout. The client performs readall() to obtain all revision logs, determines the largest revision number r with corresponding root hash h, calls update from RCS to bring its working copy to revision r, recomputes the changed paths in the hash tree, and verifies that the root hash matches h.

For updating to a particular revision r, the algorithm determines the client that committed r from all revision logs, locates the corresponding tuple $(r, h, \cdot)$ in some revision log L, and verifies L by following the hash chain from the tuple with r to the end of L. Then it proceeds as above, updating to revision r from RCS and verifying the working copy with respect to h.

When recomputing the hash tree for files that have changed in the repository, it is important that the client does that on a clean working copy, before the modifications from its working copy are applied. As the RCS merges the updates with the client's own changes, the update operation creates a working copy that differs from revision r in the repository.

Because the operations of CRCS verify that the working copy is consistent with the revision numbers and their root hashes maintained by CSS, the fork-linearizability of CSS implies the same property also for CRCS.

Note that the above algorithm introduces no new race conditions compared to RCS. As a consequence of synchronizing the client with CSS and RCS, it would be possible to create such problems. But the atomicity of the operations on CSS ensures that the more complex operations of CRCS are also atomic. In particular, whenever a client invokes checkout or update and retrieves some revision number from CSS, it always finds this revision in the repository of RCS. This holds because the commit operation of RCS precedes the writing of the corresponding revision number to CSS. Of course, there may already exist a more recent revision in the repository of RCS in the mean time, but this may also happen in the generic RCS, when another commit operation occurs immediately after an update.

6.3 Implementation

We have implemented our design in Python on Unix in two parts: first, the consistent storage service and, second, the consistent revision control system. The Python programming language encourages the kind of rapid prototyping we wanted and allowed a very natural transcription of the protocols. We chose *Subversion* (SVN) as the lower-level revision control system because it is widely used and because it fits our model of an RCS from Section 6.2.3. Hence, we refer to our implementation as *Consistent Subversion* (CSVN). Cryptographic operations are provided by OpenSSL via the M2Crypto Python interface to OpenSSL [93].

6.3.1 Consistent Storage Service

The implementation of CSS according to Section 6.2.2 stores arbitrary byte arrays. It is available as a library to clients. We wrote a simple interactive client application to read and write values entered by the user. The rich syntax of Python resulted in the server part of the algorithm in Figure 6.1 consisting of about 250 lines of code and the client part consisting of about 200 lines of code, including the operations for key management. The critical parts of our code, which implement Algorithm 1 and 2 from [20], is reproduced in Figure 6.2 for the client and Figure 6.3 for the server code. Python's support for (nested) tuple assignment makes this an almost literal translation of the original pseudo-code. Having a succinct implementation is important for maintainability, and especially important for security-relevant software.

CSS uses Python's object serialization over TCP connections for transport. The server implementation is single-threaded according to the lock-step protocol; it uses a time-out (with a default value of 300 sec) in order to tolerate a client that crashes between sending a submit message and sending the corresponding commit message. It would be prudent to integrate SSL support for increasing the security of the client-server connections; currently, network attacks appear to the clients as server faults.

6.3.2 Consistent Revision Control with Subversion

We implemented CSVN in the form of a library that interfaces to SVN and provides the three main revision control operations. The operations invoke our consistent storage service and the Python SVN Extension (`http://pysvn.tigris.org/`). The SVN server remains unchanged. We also created small wrapper scripts for a user to invoke the client operations. The CSVN library consists of about 170 lines of code, and the scripts of about 75 lines of code each. Hence, the code is very compact.

For the description below, let a *path* denote the unit of information managed by SVN; a path may be a directory containing other paths, a file, or a symbolic link.

Hash Trees

The protocol requires to compute a hash tree over the documents in a revision. Let us define a hash function $\mathcal{H}$ on paths maintained by SVN. The hash value of a path p that represents a file or a symbolic link is defined as

$$\mathcal{H}(p) = H\Big(H(p) \parallel H(C(p))\Big),$$

```
def read(self, cert):
    """Read from the register identified by *cert*."""
    x = self.x_bar
    self.send("SUBMIT", self.cert, "READ", cert)
    (V, l, phi_prime, (y, rho)) = self.recv("REPLY")
    self.validate_version(V, l, phi_prime)

    if not (V[cert] == 0 or self.verify(cert, rho, "VALUE", y, V[cert])):
        abort("Validation failed in read")

    self.T = V
    self.T[self.cert] += 1
    phi = self.sign("COMMIT", self.T)
    sigma = self.sign("VALUE", x, self.T[self.cert])
    self.send("COMMIT", self.T, phi, x, sigma)
    return y

def write(self, x):
    """Write *x* to my register."""
    self.x_bar = x
    self.send("SUBMIT", self.cert, "WRITE")
    (V, l, phi_prime) = self.recv("REPLY")
    self.validate_version(V, l, phi_prime)
    self.T = V
    self.T[self.cert] += 1
    phi = self.sign("COMMIT", self.T)
    sigma = self.sign("VALUE", x, self.T[self.cert])
    self.send("COMMIT", self.T, phi, x, sigma)
```

Figure 6.2: Main client code of the lock-step protocol, matching Algorithm 1 in [20].

```
def handle_read(self, cert, read_cert):
    """Handle a read operation"""
    self.send("REPLY", self.V, self.l, self.omega, self.X[read_cert])
    (T, phi, x, sigma) = self.recv("COMMIT")
    (self.V, self.l, self.omega) = (T, cert, phi)
    self.X[cert] = (x, sigma)

def handle_write(self, cert):
    """Handle a write operation."""
    self.send("REPLY", self.V, self.l, self.omega)
    (T, phi, x, sigma) = self.recv("COMMIT")
    (self.V, self.l, self.omega) = (T, cert, phi)
    self.X[cert] = (x, sigma)
```

Figure 6.3: The central code of the server, matching Algorithm 2 in [20].

where $C(p)$ is the content of p. The hash value of a path p representing a directory is

$$\mathcal{H}(p) = H(\mathcal{H}(p_1) \parallel \mathcal{H}(p_2) \parallel \cdots \parallel \mathcal{H}(p_n) \parallel H(p)),$$

where $p_1, \ldots, p_n$ is a sorted list of all paths in p. We denote the root hash of a repository by $\mathcal{H}(\text{“.”})$.

It would be prohibitively expensive to recompute the hash values of all paths in a large repository upon every change of a single file. Therefore, the client stores the hash value of every path as an SVN property of the path. During an update operation, CSVN recomputes the hash values of all changed files and of all directories along the path from the changed files to the root. For a repository with n files, this reduces the cost of updating m modified files from linear in n to $\mathcal{O}(m + d)$, where d is the maximum affected depth in the directory tree.

The hash values are stored on the SVN server because properties are revision-controlled in SVN. Note that storing them on untrusted storage is unproblematic. The hash values are not actually needed by a client who checks out the complete repository because the client recomputes the entire hash tree anyway during verification. But they are needed for partial checkouts, as explained below.

Integration with SVN

During checkout and update operations, CSVN installs a callback before invoking the SVN library, which collects all relevant events reported by SVN; such relevant events are the addition, update, and deletion of a path. Then CSVN invokes CSS to obtain a revision number r and retrieves revision r from SVN, as described in Section 6.2.4. To recompute the root hash of the working copy, CSVN traverses the working copy, but visits only paths for which a relevant event was collected during the SVN operation.

For a commit operation, CSVN first determines the modified paths which are going to be written to the repository. It does that with an SVN “info” operation that outputs a collection of changed paths. Then it traverses the working copy, visiting and recomputing hash values only for changed paths, and getting hash values for unchanged paths from their SVN properties. This yields the root hash $h = \mathcal{H}(\text{“.”})$. CSVN further invokes the “commit” operation of SVN to write the updates to the repository and to obtain the new revision number r. Finally, it retrieves the revision commitment c from the last tuple in L, appends $(r, h, H(r \parallel h \parallel c))$ to L, and writes L using CSS.

This completes the description of the main CSVN operations. Further SVN operations can be implemented easily using the CSVN library and the three main CSVN operations.

The description so far assumes that clients always check out and update the complete file set in the repository at once. But this is not required in SVN, where a client may check out only a subdirectory from a repository, or commit only a subset of its working copy. The revision number and the root hash stored in CSS are always global properties of the repository, though. Operations on the partial repository are supported by our design and rely on the hash values stored in the SVN properties. For example, to check out a subtree from a repository, CSVN also needs to read all files along the path from the subtree's root to the repository root before it can verify the root hash.

An important and nice feature of this implementation is that it does not add any additional SVN server operations; because they usually involve the network and contain a cryptographic authentication operation during login, they tend to be rather slow.

6.4 Evaluation

We report on benchmarks to measure the performance of CSVN client operations in comparison to an unmodified SVN client. Since every operation of CSVN also invokes the corresponding operation of SVN, we are primarily interested in the overhead of CSVN over SVN.

We report on two kinds of performance evaluations: an application benchmark using real-life file sets of different sizes and a synthetic benchmark with artificially made-up file sets. Each benchmark consists of a series of tests executed by two clients, called A and B, where each test uses different data. For each test, we run the unmodified SVN client and the CSVN client 20 times in succession and measure the average time taken by each step in the test. Every run starts with an empty repository and a freshly initialized CSS. Each test uses a pair of related file sets; we are interested in the time it takes to update a working copy and the repository from one file set to the other one.

Each run in a test consists of the following steps:

1. Client A initializes a new empty repository on the server. This step is the same for both systems, so we do not measure it.

2. *Create* — client A checks out revision 0, creating a working copy from the empty repository.

3. *Import* — client A copies the first file set into its working copy, adds it to the repository, and commits the changes; we measure the time for the commit operation only.

4. *Checkout all* — client B checks out the content of the repository into its own working copy; the working copies of A and B are now identical.

5. Client B modifies its working copy to reflect the second file set. This involves adding the files contained only in the second file set, deleting the files only present in the first set, and copying the changed files from the second set into the working copy. This step is identical for both systems and is not measured.

6. *Commit diff* — client B commits the changes in its working copy.

7. *Update diff* — client A, whose working copy still contains the first file set, updates it to the most recent revision in the repository, which

Table 6.1: The four tests of the application benchmark and the used Linux kernel version pairs. The third and fourth columns list the number of files in the first file set and the number of changed (added, modified, or deleted) files between the two file sets, respectively.

Test (version pair)	Size	Files	Changed
0.11 → 0.12	0.63 MiB	100	91
1.0 → 1.0.1	5.9 MiB	561	12
2.0.1 → 2.0.2	27 MiB	2,021	28
2.2.0 → 2.2.1	62 MiB	4,599	10

contains the second file set; the working copies of *A* and *B* are again identical.

This sequence of steps is designed to capture the overhead of committing and updating a large file set at once (in the *import* and *checkout all* steps) and of committing and updating smaller number of files in a larger file set (in the *commit diff* and *update diff* steps).

The benchmarks use two separate hosts, one for the server and one for both clients; they are connected by a gigabit LAN. The machine for the clients is an IBM x345 system with 2 GiB of RAM and two hyper-threaded Intel Xeon CPUs (3.06 GHz clock speed). The machine for the server is an IBM x335 system with 2 GiB of RAM and two hyper-threaded Intel Xeon CPUs (2.80 GHz clock speed). Both machines have a single IBM Ultra320 SCSI disk with 73.4 GB capacity and run Debian GNU/Linux 4.0 with kernel 2.6.18 and Subversion 1.5.2. The SVN server is accessed using SSH and all data is stored on the local filesystems. We use the SHA-1 hash function and 1,024-bit RSA for signatures.

6.4.1 Application Benchmark

The file sets in our application benchmark are different versions of the Linux kernel source tree, as reported in Table 6.1. All files can be downloaded from the Linux kernel archive (`http://kernel.org/`). We choose them since they represent a realistic directory structure and because the repository sizes range over several orders of magnitude, from 632 KiB to 62 MiB. We selected the four versions that make up the first file set in a test based on their relative size. For each test, we pick the subsequently released version of the Linux kernel and use it as the second file set. The results are shown in Table 6.2.

6.4.2 Synthetic Benchmark

In this benchmark, we wish to measure how the running time changes when we grow the directory structure in a repository from one directory to a large tree, but keep the number of files constant. To do this, we create four artificial file sets, each consisting of 256 files, each file of size 10 KiB, for a total data size of 2.5 MiB per file set. The files are filled with random pieces

Table 6.2: Results of the application benchmark. The numbers denote average elapsed time and standard deviation in seconds for SVN and CSVN in 20 runs, and the ratio of the two average times.

(a) Version 0.11 → 0.12.

Step	SVN	CSVN	Ratio
Create	1.18 ± 0.09	1.53 ± 0.33	1.30
Import	0.93 ± 0.02	1.94 ± 0.00	2.08
Checkout all	0.99 ± 0.00	1.10 ± 0.00	1.11
Commit diff	1.50 ± 0.02	2.30 ± 0.29	1.53
Update diff	0.94 ± 0.00	1.05 ± 0.01	1.11

(b) Version 1.0 → 1.0.1.

Step	SVN	CSVN	Ratio
Create	1.05 ± 0.05	1.45 ± 0.38	1.38
Import	3.70 ± 0.11	6.76 ± 0.00	1.83
Checkout all	1.98 ± 0.00	3.17 ± 0.48	1.60
Commit diff	1.24 ± 0.42	2.03 ± 0.01	1.63
Update diff	0.94 ± 0.01	1.11 ± 0.00	1.17

(c) Version 2.0.1 → 2.0.2.

Step	SVN	CSVN	Ratio
Create	1.46 ± 0.17	1.34 ± 0.25	0.92
Import	14.08 ± 0.71	28.84 ± 1.02	2.05
Checkout all	7.49 ± 2.38	12.31 ± 2.44	1.64
Commit diff	3.89 ± 1.59	5.15 ± 0.47	1.32
Update diff	0.94 ± 0.00	2.28 ± 0.02	2.42

(d) Version 2.2.0 → 2.2.1.

Step	SVN	CSVN	Ratio
Create	0.68 ± 0.06	1.18 ± 0.29	1.72
Import	36.35 ± 1.05	79.26 ± 1.29	2.18
Checkout all	13.38 ± 2.04	29.28 ± 2.28	2.19
Commit diff	10.20 ± 3.68	9.64 ± 2.53	0.95
Update diff	1.27 ± 1.53	1.69 ± 0.49	1.33

Table 6.3: Results of the synthetic benchmark. The numbers denote average elapsed time and standard deviation in seconds for SVN and CSVN in 20 runs, and the ratio of the two average times.

(a) Depth 0.

Step	SVN	CSVN	Ratio
Create	1.48 ± 0.12	1.35 ± 0.40	0.91
Import	2.71 ± 0.06	4.19 ± 0.50	1.55
Checkout all	1.99 ± 0.00	2.90 ± 0.01	1.46
Commit diff	1.87 ± 0.22	3.02 ± 0.01	1.61
Update diff	0.95 ± 0.00	1.73 ± 0.01	1.83

(b) Depth 2.

Step	SVN	CSVN	Ratio
Create	1.25 ± 0.06	1.25 ± 0.39	1.00
Import	2.01 ± 0.33	3.89 ± 0.01	1.93
Checkout all	1.99 ± 0.00	2.40 ± 0.01	1.21
Commit diff	0.93 ± 0.05	1.51 ± 0.01	1.63
Update diff	0.95 ± 0.00	1.01 ± 0.01	1.07

(c) Depth 4.

Step	SVN	CSVN	Ratio
Create	1.27 ± 0.01	1.46 ± 0.39	1.15
Import	1.95 ± 0.27	3.88 ± 0.01	1.99
Checkout all	1.99 ± 0.01	2.30 ± 0.02	1.16
Commit diff	0.92 ± 0.06	1.61 ± 0.03	1.75
Update diff	0.95 ± 0.00	0.94 ± 0.01	1.00

(d) Depth 8.

Step	SVN	CSVN	Ratio
Create	0.86 ± 0.23	1.33 ± 0.14	1.55
Import	4.33 ± 0.42	10.59 ± 0.89	2.44
Checkout all	8.75 ± 1.52	9.95 ± 1.14	1.14
Commit diff	0.88 ± 0.04	1.31 ± 0.24	1.50
Update diff	2.28 ± 1.15	1.90 ± 0.51	0.84

of C code taken from the Linux 2.2.1 kernel; this is to generate files looking like a real source tree. The files are stored in a directory structure of varying depth. We define a directory structure of depth d as a full binary tree of depth d and store $256/2^d$ files in each of the 2^d leaf directories.

Our file sets are four directory structures with depths 0 (all files in one directory), 2, 4, and 8 (every file in a separate directory). In each test, the second file set is identical to the first one, up to a random modification to one of the files in a leaf directory. The results are shown in Table 6.3.

6.4.3 Results

The results of both benchmarks show that CSVN adds an overhead of a factor that is generally less than 2 and usually also less than 1.5. In absolute terms, the *import* and the *checkout all* steps are the slowest operations because they involve all files. The *import* step generally incurs also the biggest overhead, usually around 2. But the overhead of the *checkout all* step is not noticeably different from the overhead of the remaining steps. Generally, CSVN adds only a moderate overhead to most operations compared to the normal SVN client.

Observe the bigger variation in the execution times of the tests with larger file sets. One reason for this effect may be that large data sets create more unexpected interactions with other programs due to swapping and disk operations than small data sets that fit in the kernel's buffer cache. Such variations also explain the few overhead ratios smaller than 1.

Among the results of the application benchmark in Table 6.2, the second largest overhead (after the *import* step) usually occurs for the *commit diff* step. The overhead on the large file sets is not bigger than that on the smaller file sets. This clearly shows the benefit of using a hash tree when only a small part of a large file set is updated.

In the results of the synthetic benchmark in Table 6.3, observe the overhead of the *commit diff* and the *update diff* steps. In both steps, only a single file is changed. The CSVN client must then read the hash values of all sibling files to compute the new hash values for the directory. With the increasing depth of the directory structure, the number of sibling files drops from 255 to 0, and this is reflected in the decreasing overhead.

In summary, although a 50%–100% larger execution time for SVN operations is clearly noticeable by the clients, we believe it is a reasonable price to pay for the added guarantee of cryptographically verified data integrity. These results should serve as a lower bound for the efficiency of our design, because they were carried out with our straight-forward layered prototype implementation in Python. If the CSVN operations would be integrated with the SVN client library, the directory tree in the working would have to be traversed only once instead of twice; moreover, hashing could be integrated with the traversal and performed concurrently with receiving or sending data to the server. With such an integrated design, the cryptographic overhead is likely to vanish, as shown in other benchmarks of cryptographic storage and file systems [99].

6.5 Conclusion

Protecting data integrity against unauthorized modifications is an important aspect of networked storage systems.

In this chapter, we have presented a novel approach to securing the integrity of data stored in revision control systems, and demonstrated its feasibility with our Consistent Subversion (CSVN) prototype. Our evaluation shows that the overhead is reasonable.

The biggest threat to our system are client failures. Protecting the system from malicious clients is also the area where future work is needed.

Our implementation already tolerates client crashes; one or more malicious clients alone cannot harm the integrity *if* the service provider is correct — measures to prevent such behavior can easily be added [66], but have not been described in this work. A corrupted client conspiring with a corrupted service provider, however, may undermine fork-linearizability.

A first barrier against such an attack is the CA that must authorize all clients before they access the service. It is therefore a good idea to make the CA is a separate entity from the storage service. If the threat of such a client-server conspiracy attack becomes too serious, one might adopt the complex cross-checking of versions signed by different clients introduced in SUNDR [72]. Unfortunately, the SUNDR protocol involves a much higher communication overhead in every operation. One should also develop an additional tool that helps the clients to recover from a server failure; it should automatically reconcile the state of the repository from the information held by the clients in their working copies and their local memories.

Acknowledgments. We are grateful to Idit Keidar, Alexander Shraer, and Marko Vukolić for many discussions and valuable comments.

This work was supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

Chapter 7

Conclusion and Future Work

In this final chapter, we will first sum up the experiences we have gained with the work on implementing secure multiparty computation. Following that, we will describe some possible future extensions to VIFF before we conclude this dissertation.

7.1 Lessons Learned

While working with VIFF, we have learned a number of lessons about real-world implementation of multiparty computation. We will discuss some of them in this section.

First, we will make a general observation on protocol design: simple, *information theoretically* secure protocols should be preferred. Intuitively, one might expect protocols secure against a weaker polynomial time adversary to be easier to implement and faster to execute than protocols secure against a much stronger adversary with unbounded computing power. This is not the case. Unconditionally secure protocols often employ simple arithmetic whereas cryptographically secure protocols end up using more expensive arithmetic such as exponentiations.

A cryptographically secure primitive must by definition rely on some hardness assumption. One must make a judgment about the amount of computing power the adversary is willing to invest in breaking the scheme, and choose a suitable margin. This safely margin ends up as a computational overhead in the protocol. Unconditionally secure protocols do not have such a safety margin.

7.1.1 Large-Scale Testing Matters

The traditional wisdom in protocol design is that local computation is free. The very first paper on multiparty computation by Yao [100] mention that the proposed solution becomes impractical when the size of the input numbers increase. The concern is the number of bits that must be transmitted, not the amount of local computation.

The underlying assumption is thus that network is the bottleneck. This was also our initial assumption. It is reflected in the section on benchmark results in [37], where we write that the quadratic growth in preprocessing time (when using hyperinvertible matrices) is unexpected.

The quadratic growth *is* unexpected if local computations are not considered since the communication complexity of the protocol (for a single party) is $\mathcal{O}(n|C|k)$ where $|C|$ is the size of the arithmetic circuit being computed and k is the bit length of elements in the field used.

Our testing showed (Section 4.5) that the local computations cannot be ignored on a fast network. With our benchmarks using a large number of parties, we saw a quadratic growth in the running time for both the classic BGW protocol [10] and for our preprocessing protocol using hyperinvertible matrices. This is not surprising when one inspects the code — there are several places which clearly depend in a quadratic way on the number of parties. The effect is slight and we would not have discovered this misconception if we had not tested on a large number of parties.

7.1.2 Choice of Language

One of the most important decisions in any software system is the implementation language. Among many other things, the choice of language effect the speed of development, the speed of the resulting implementation, and the supported platforms.

The SCET project used Microsoft's C# as the implementation language, along with a number of other Microsoft technologies. This limited the number of supported platforms to essentially one, namely Microsoft Windows. When the SIMAP project was started, it was decided that we wanted a broader platform coverage. All programmers on the project had experience with Java, which has better cross-platform support than C#. This made Java a natural choice for the project. In VIFF, we use Python as our implementation language, largely due to the Twisted network library, but also because Python is a true cross-platform language.

Python Critique

While Python is a wonderful language to program in — expressive, concise and flexible — it does have a number of shortcomings. The biggest problem is the lack of a static type system. Python is a strongly typed language, which means that all variables have a known type at runtime, but the type is not known statically and may even change from one execution to another.

This dynamic behavior makes it difficult to make any kind of static analysis of the code. This in turn makes it difficult to perform even the simplest classic optimization techniques such as function inlining, loop hoisting, or strength reduction.

The lack of static guarantees means that everything must be interpreted at runtime by the Python VM. Compared to compiled languages, this results in a quite substantial overhead for simple things such as function calls. As we shift our focus from designing VIFF to actually using it, we would like to seek ways to lower this overhead. Moreover, while we were initially satisfied with comparing protocols within VIFF, we would now like to compare our performance to other systems such as Fairplay and SHAREMIND. Their languages (Java and C++, respectively) offer many more opportunities for compilers to optimize the code.

Python Variants

There are some more or less experimental projects trying to lower or eliminate the overhead associated with Python. We shall mention only two project here. The most promising is the Unladen Swallow[1] project, which is proposed to be merged with the current C implementation of Python [98]. This work introduces a just-in-time compilation (JIT) backend to the Python virtual machine. The JIT compiler will emit optimized machine code. The performance improvements vary between benchmarks, but seem to lie around 40%. The initial goal for Unladen Swallow developers a five time improvement in performance. This was based on experience from JIT compilation of JavaScript code — another language which is notoriously hard to optimize due to its dynamic nature. The hope is still to reach this level of improvement with Python.

The PyPy project [87] has a more long-term goal of improving the speed of Python and other dynamic languages. For this purpose, they have written framework (in Python) that allows them to translate a language description (of Python, say) into a concrete virtual machine. Hence the name "PyPy" which hints to an implementation of Python in Python.

The idea is that it is easier to extend and change the target VM by changing the high-level Python translation process instead of changing the code of the VM itself, which may be written in a language like C. The PyPy project has also experimented with adding a JIT compiler to the Python VM. Their benchmarks show that it produces code that is 5–8 times faster than the standard Python. This was on benchmark programs doing lots of arithmetic. This might be interesting for VIFF, since our programs also contain a lot of arithmetic.

We have so far focused on the design and implementation of new protocols for VIFF. Optimizing for speed has been a secondary concern, and so we have not conducted benchmarks using either Unladen Swallow or PyPy.

Python Alternatives

Python has been great for rapid prototyping. The author of this dissertation feels that the succinct syntax and the flexibility of the language plays an important role here. Another language with a famously elegant syntax is Haskell [83]. Haskell is a lazily evaluated, statically typed, functional language. It supports type inference which greatly reduces the amount of boilerplate code needing to be written: local variables only need to be

[1]The Python programming language is named after *Monty Python*, not the species of snake. In *Monty Python and the Holy Grail*, the keeper of the Bridge of Doom asks King Arthur about the air-speed of an unladen swallow.

explicitly typed in rare circumstances when the compiler cannot determine their type by itself.

Being a statically typed language, the Haskell compiler has full access to all the known optimization techniques. The Haskell compilers are known to produce good machine code, comparable with that of modern C compilers.

The Haskell runtime has a feature which is of particular interest to us: ultra light-weight threads. A program can create thousands of these threads, which are then automatically multiplexed onto a handful of operating system threads. Using these threads, we should be able to obtain the very fine-grained parallelism present in VIFF, but without the use of callbacks.

The basic idea is to spawn a new light-weight thread for each operation. The thread will block until its inputs are ready. The Haskell concurrency libraries provide a type called MVar, which act as a kind of "box". The box can be either empty or full, and threads block when they try to grab the value from an already empty box. Using these boxes instead of Deferred instances, we can implement multiplication like this:

```
mul :: MVar Int -> MVar Int -> IO (MVar Int)
mul x y = do
  z <- newEmptyMVar
  forkIO $ do
             x_val <- takeMVar x
             y_val <- takeMVar y
             putMVar z (x_val * y_val)
  return z
```

The function takes two boxes (holding integers) as input and produces a new box holding an integer.[2] The function essentially creates an empty box and returns it to the caller. Meanwhile, a new light-weight thread has been spawned with forkIO. The thread performs the action in the inner do-block: it blocks on each input in turn, and stores the result in the box z.

The multiplication function is used as follows:

```
main = do
  x <- newMVar 10
  y <- newEmptyMVar
  z <- mul x y
  putMVar y 20
  z_val <- readMVar z
  print z_val
```

Notice how we multiply x and y, where x is a box containing 10 and y is an empty box. When we put a value into y, we can read the product out of z and finally print it on the terminal.

This example is of course far from complete, but it is encouraging to see how easy threads can be used in Haskell. The callback-based style of programming is more cumbersome than initially expected, so a return to a more normal control flow would be welcomed.

[2]The IO tag on the return type indicates that computation takes place in the so-called I/O monad. This is not important for our discussion.

7.1.3 Memory Management

It has turned out to be difficult to control the memory usage of programs that use VIFF. The problem is the "eager" evaluation used, in which it is easy to allocate a huge amount of work. Consider a simple loop:

```
for i in range(1000):
    x = x * x
```

With normal execution semantics, this would not require any additional memory given that x had already been allocated. However, when x is a Share object, the above two lines will not just overwrite the memory location of x in each iteration. What happens instead is that a large tree (shaped like a linked list) is formed in memory to hold the intermediate values.

This behavior is a double-edged sword: it is sometimes seen as a feature and sometimes seen as a bug. It is a feature when VIFF is able to automatically run things in parallel which the programmer might not even have considered. But when a program is slowed down by excessive memory allocations, then we would rather be without this "feature".

It is clear that the above example gains nothing from scheduling all 1,000 multiplications at once. Unfortunately, it is far from intuitive how to rewrite it to run in a constant amount of memory:

```
def sqpow(x, i):
    rt = x.runtime
    sq = x * x
    if i > 1:
        rt.schedule_callback(sq, lambda ignored: sqpow(sq, i − 1))
    return sq

x = sqpow(x, 1000)
```

Here we use a callback to suspend the remainder of the computation until each squaring is done. This kind of rewriting is only feasible for small examples like ours.

If we had used a system with light-weight threads as discussed in the previous section, we would have a simple, yet powerful, way to solve this problem: we could simply limit the number of active threads. The thread that computes the loop would block mid-way through the loop when it exhaust the thread pool. When the first multiplications finish, the thread is unblocked and can schedule further multiplications. This mechanism would give us a handle to control the amount of outstanding work and thus the amount of allocated memory.

7.2 Future Work

The protocols we have described in this dissertation can all extended to cover situations similar to the ones we have described them in. The secure comparison from Chapter 2 is a prime example, and we gave a number of examples of how to adapt it in Section 2.5.

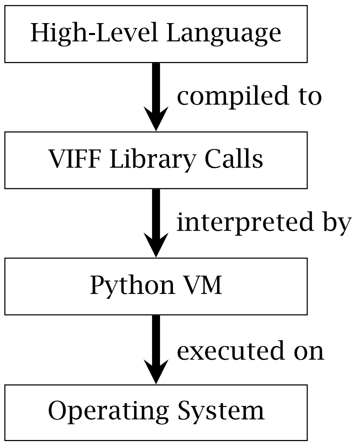


Figure 7.1: The language stack.

In this section, we will focus on future extensions for VIFF. When implementing any large software system, compromises will be made and design decisions that once seemed natural, can now be seen in a different light. VIFF is no different and has given rise to a number of ideas for improvements. We will describe some of them here.

7.2.1 VIFF as a Platform

Currently, VIFF is just a library for writing MPC protocols, and programs using VIFF are therefore normal Python programs. While VIFF already provides a rather high-level API for writing protocols, it would be interesting to develop an even higher-level language so that the programmer wont have to know Python at all. A compiler for this language would target the VIFF API as its platform. Figure 7.1 illustrates how VIFF can be used as an intermediate language.

Apart from additional convenience in the form of syntactic sugar, the high-level language can provide different kinds of static security analysis of the programs, something which is not possible for VIFF itself to do.

Meldgaard et al. [74] have already done some work in this direction as part of the CACE project. They have designed a language called PySMCL, which is a domain specific language for MPC embedded into Python. The language is a subset of Python, for which some static analysis can be performed. The analysis include tracking the data flow of secret shared values. This is used to warn the programmer of potential leaks when secret shared values are opened. The programmer must then acknowledge each leak by a special annotation to indicate that the programmer takes responsibility of the leak.

7.2.2 Recovering from Fail-Stop Crashes

Theoretically, the honest parties are supposed to follow the protocol exactly as it is laid out. However, as we all know, computers cannot be trusted to function perfectly at all times... This means that, despite our best intentions, the parties may crash in the middle of a protocol execution. This applies to both honest and corrupt parties. In the following we will assume a fail-stop model [90], that is, we expect (honest) parties to simply halt when they crash. Corrupt parties should also halt when they crash in case the adversary is semi-honest.

A program using VIFF will currently not handle failures in an elegant way, i.e., it will crash too when trying to use the half-closed TCP connection, or it will become stuck waiting for input that will never arrive. In the case of a protocol which only guarantees security against a semi-honest adversary, there are two ways to react:

- Do nothing like VIFF currently does. This is by definition the "right choice" since any protocol deviation should be treated as an active attack. In particular, there are no way to differentiate between a malicious party that tries to stall the computation by disconnecting and an unlucky party with a bad Internet connection.

  Whatever the cause, it is a correct choice for a passively secure protocol to abort when it detects that it is operating outside of the model in which it was proven secure.

- Attempt to recover — but do it securely. This is clearly more ambitious. It is clear that this can only work for benign errors where a party still wants to participate in the protocol.

A protocol secure against a malicious adversary will by definition be able to handle uncooperative parties and malformed input. However, even though the protocol by itself can deal with missing or bad input, VIFF also needs to handle failed connections and decoding exceptions robustly. The actively secure protocol implemented in ActiveRuntime will currently abort when it detects malformed input. If a sufficient number of parties crash or refuse to participate, the protocol becomes stuck.

Recovery Protocol

We will now sketch how VIFF can be extended to deal with crashing parties. Put simply, the parties maintain a log of their activities and use this to synchronize with each other after a crash.

As described in Section 3.5.4 on page 46, everything sent and received in VIFF is identified by a program counter. Each party keeps a log of all sent and received data. The log contains the program counter, the ID of the sender or receiver and the data itself. The log is stored on disk or some other durable medium (such as a transactional database). The size of the log file will be quite modest: instrumenting VIFF reveals that a single secure multiplication of 32-bit numbers using PassiveRuntime requires a player to send 28 bytes of data to each its peers (see Section 4.5.5). That number includes the program counters. A secure comparison of 32-bit numbers requires more, about 5 KiB in total. The double-auction implemented by the SIMAP project requires only 12 secure comparisons with a default price grid of 4,096 prices.

When a party P_i is rebooted after a crash, it first re-establishes connections with the other parties. It then asks each peer P_j for its list of expected data. This is a list of program counters for which P_j has not yet received any data from the crashed party. The log file is used to seed the list incoming_data dictionary for the crashed player and after that, P_i can quickly replay the computation.[3] Whenever P_i wants to send something to another party, it first consults the list of expected data for that party. That way, P_i can avoid sending data which P_j has already received.

When P_i has caught up, it may be the case that it now expects to receive some data from P_j, which P_j sent before the crash. Normally, P_i does not need to tell P_j when it expects data from it — that information is part of the protocol. However, when catching up, this inherent synchronization is lost. To ensure full synchronization, P_i will therefore explicitly inform each of its peers about what data is expects to receive from them after it has depleted its incoming_data dictionary of pre-seeded data.

Security

The above protocol for catching up leaks no private information. The crashed party will not send anything to the other parties which it has not already sent before. A party P_j might claim that is has not received anything at all from P_i, in which case P_i will replay the entire communication with P_j. This slows down P_i, but leaks no information. Note also, that when P_i sends out its request for data, the other parties may "change their mind" and send something else than they did before. This is of no consequence, since they base this decision on no more information than before.

All in all, when a player crashes and catches up, it is equivalent to having a (possible large) delay in the communication with that player.

The solution outlined above is quite practical and could be implemented in the current VIFF system. A more sophisticated solution would store the program state in addition to the network traffic. A crashed party could then restart its computation from where it crashed, instead of replaying it from the beginning. Using static analysis, it would be possible to prune the log file significantly. As an example, consider the following tiny program:

```
x = a * b
y = x * c
```

When we store x in the program state, we can throw away the data received in the multiplication protocol — it wont be needed again. Furthermore, if a and b are not used again, we can throw away the original shares that were recombined to give a and b. Additionally, P_i no longer needs to hold onto the share of x it sent to P_j when P_i receives a share of y from P_j. This is because P_j cannot send a share of y unless it already has all the shares it needs for x. Such static dependency analysis seems best suited for a dedicated compiler.

[3]With no network communication, P_i is able to quickly catch up.

7.2.3 Protection Against Active Adversaries

The previous section described how we could extend VIFF to better handle players that fail by accident, e.g., someone trips over the Internet connection and an otherwise honest party must now rejoin the computation. In this section we will look at how VIFF can be improved to better protect the honest parties from malicious behavior from the corrupt parties.

Handling Malformed Input with Twisted

As already mentioned, the ActiveRuntime will abort if the stringReceived method cannot unpack data received over the network. This is often not the optimal strategy considering that actively secure protocols are made to withstand bogus or missing input.

VIFF should instead raise an asynchronous exception. The Twisted library already provides an infrastructure for raising and handling exceptions asynchronously. This is done by attaching *errback* functions to Deferred instances. An errback is called if an exception is raised in a callback. Normally, control passes from callback to callback along the callback chain. However, when an exception is raised in a callback, control passes to the next errback. The errback can now attempt to correct the error. If the errback does not handle the failure, the next errback is called. Using this system, it is possible to simulate the normal try/except mechanism.

Consider a simple example where we parse a string as an integer check that the value is a legal value in our field. This is just a toy-example:

```
import sys
from twisted.internet.defer import Deferred

def parse_data(data):
    value = int(data)
    if value > 1031: # field modulus
        raise ValueError("%s is too large" % value)
    return value

def handle_error(failure):
    print "Correcting failure: %s" % failure.getErrorMessage()
    return 0

def output(value):
    print "Final value: %s" % value

a = Deferred()
a.addCallback(parse_data)
a.addErrback(handle_error)
a.addCallback(output)

# Trigger callback:
a.callback(sys.argv[1])
```

The handle_error errback is called if the parse_data function raises an exception. The error handler "corrects" the error by fixing the value to zero. Finally, the output callback prints the final value (either the integer parsed from the string or the zero coming from handle_error). Testing with different strings gives:

```
$ python errbacks.py 500
Final value: 500
$ python errbacks.py 5000
Correcting failure: 5000 is too large
Final value: 0
$ python errbacks.py bogus
Correcting failure: invalid literal for int() with base 10: 'bogus'
Final value: 0
```

In VIFF, we would convert an exception in the stringReceived method to a Failure instance instead of calling abort. This change would force the rest of the code to handle Failure objects too. Today, a Share object is assumed to (eventually) hold an element of a finite field, and as such, the code expects to be able to do arithmetic with Share objects. When a Share can also contain a Failure, this is no longer the case. A Share containing a Failure would probably have to simply short-circuit any arithmetic operation it participates in: if x is the faulty Share, then w = x + y * z would immediately make w a failed Share too. This is just a very rough sketch of how exception handling could work in VIFF.

Denial of Service Attacks

There is another area related to active adversaries where we could improve VIFF: denial of service attacks. If a corrupt party P_j sends 100 MB of data to an honest party P_i, the honest party will naively accept the data and insert it into its incoming_data dictionary. The data will sit around there for the duration of the computation unless P_i happens to expect data from P_j with a matching program counter. Working together, the corrupt parties can probably make P_i run out of memory. A first step to solve this would be to reject messages larger than, say, 100 KB. It is, however, not a complete solution since the corrupt parties can easily send many small messages.

Once again, the program counters (see Section 3.5.4 on page 46) will help us. We already saw how it is possible to predict the amount of preprocessed data needed in a protocol by making a dummy execution. We can do the same to obtain a trace with the valid program counters. When data is received, the program counter is stored after the first 5 bytes. This means that P_i can read the first part of the data to determine the program counter. It is then a simple matter to verify if the program counter is valid for the protocol.

Some protocols can deviate slightly from run to run. The protocol for secret sharing a bit from $\mathbb{Z}_p$ using pseudorandom secret sharing is the best example: The parties generate a sharing of a uniformly random element $[r]$

from $\mathbb{Z}_p$ and compute $[b] = [r]/\sqrt{r^2}$, which is either -1 or 1.[4] The division fails if $r^2 = 0$ and the parties must then restart the protocol, which leads to different program counters.

It is possible to handle those cases by reserving a small amount of extra program counters, depending on the expected number of restarts. The PRSS protocol just described is restarted with probability $1/p$. Here p could be a, say, 32-bit prime. Reserving program counters for only 4 extra restarts will thus bring the probability of falsely rejecting a good message down to under $1/2^{160}$. This lets the corrupt parties send very few "spam" messages while making sure that messages are not rejected incorrectly.

7.2.4 Black-Box Secret Sharing

When Shamir shared values are recombined in VIFF, the code used is completely general, i.e., it does a full Lagrange interpolation. Given a set $S = \{(x_{i_1}, s_{i_1}), \ldots, (x_{i_{t+1}}, s_{i_{t+1}})\}$ of $t + 1$ shares, where $s_i = f(x_i)$ for a random polynomial f of degree t, then f can be reconstructed as

$$f(x) = \sum_{(x_i,s_i)\in S} \Bigg(s_i \prod_{\substack{(x_j,s_j)\in S,\\ j\neq i}} \frac{x - x_j}{x_i - x_j} \Bigg).$$

Evaluating $f(0)$ allows us to recover the secret shared value. The computation is done over a finite field which leads to modular reductions after each multiplication and addition. Also, the fraction must be computed by first inverting the denominator followed by a multiplication in the field.

If we restrict ourselves to the common situation with three parties and at most one corrupt party, then things are simplified significantly. We can in fact do the computations directly over the *integers* instead of working inside the field. This will save a fair amount of local computation.

Shamir secret sharing a value s into s_1, s_2, and s_3 using $f(x) = \alpha x + s$ results in these shares for the parties:

$$P_1\colon s_1 = \alpha + s, \qquad P_2\colon s_2 = 2\alpha + s, \qquad P_3\colon s_3 = 3\alpha + s.$$

It is now clear that we can reconstruct the secret s again as:

$$s = 2s_1 - s_2 = 3s_2 - 2s_3.$$

This computation does not rely on any modulo reductions — it holds over the integers and is independent of the field used. A secret sharing scheme that work for an arbitrary Abelian group is called a black-box secret sharing scheme [27, 40].

Black-box schemes use only integer linear combinations for secret sharing and recombination. This makes it possible to use the scheme in settings where the group order is unknown. We have looked at the specific setting with $(n, t) = (3, 1)$, but in the literature the focus has been on finding

[4]Add one and divide by two to turn this into a true bit value. Note that r^2 can be opened safely since it does not reveal if r is positive or negative.

schemes for general (n, t). We will likely only need schemes for small values of n, so the above scheme may already be sufficient for our purposes.

We have made a quick implementation of the black-box reconstruction function described above. Benchmarks show that we can save about 15–20% on the time for a secure multiplication.

7.3 Conclusion

Nearly thirty years ago, Yao famously asked how the two millionaires can engage in a conversation in order to determine who is richer, but without disclosing *any* additional information. In the three decades that followed, a lot of effort has gone into answering Yao's question and into answering the broader question of how to conduct general secure multiparty computation.

We have given several answers of our own in this dissertation. First, in Chapter 2, we presented a highly practical protocol for secure comparison. Then, in Chapter 3 and 4 we presented protocols for general secure multiparty computation with security against both passive and active adversaries. Chapter 5 presented a compiler that will transform any protocol secure against a semi-honest adversary into a protocol secure against a covert adversary. Finally, Chapter 6 described how clients of online services can be better protected against a malicious server.

In addition to new protocols, a novel aspect of this dissertation is the focus on concrete implementations. Implementing the protocols allow us to compare protocols directly, instead of relying solely on the big-$\mathcal{O}$ notation. This is not to say that the asymptotic performance is unimportant, it is only a reminder that the hidden constants matter when looking at real-world performance. We have presented implementations and benchmarks of the protocols from all chapters except Chapter 5.

We hope this dissertation advance the state of the art in cryptographic protocol implementations. We have showed that it is now practical to implement and use the protocols developed during the last three decades. We hope many more will take up the challenge of turning protocols into running code.

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO*, pages 417–432, 2002.

[3] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In S. P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007.

[4] M. Baker, M. A. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. J. Giuli, and P. P. Bungale. A fresh look at the reliability of long-term digital storage. In Y. Berbers and W. Zwaenepoel, editors, *EuroSys*, pages 221–234. ACM, 2006.

[5] B. Barak, R. Canetti, J. B. Nielsen, and R. Pass. Universally composable protocols with relaxed set-up assumptions. In *FOCS*, pages 186–195. IEEE Computer Society, 2004.

[6] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513. ACM, 1990.

[7] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In R. Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.

[8] Z. Beerliová-Trubíniová, M. Hirt, and J. Nielsen. Almost-asynchronous multi-party computation with faulty minority. Manuscript, 2008.

[9] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.

[10] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10. ACM, 1988.

[11] I. F. Blake and V. Kolesnikov. Strong conditional oblivious transfer and computing on intervals. In P. J. Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 2004.

[12] I. F. Blake and V. Kolesnikov. Conditional encrypted mapping and comparing encrypted numbers. In G. Di Crescenzo and A. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2006.

[13] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[14] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[15] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In G. Di Crescenzo and A. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.

[16] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In R. Dingledine and P. Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[17] D. Boneh and M. K. Franklin. Efficient generation of shared rsa keys (extended abstract). In B. S. K. Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 1997.

[18] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In J. Stern, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 1997.

[19] C. Cachin and M. Geisler. Integrity protection for revision control. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 382–399, 2009.

[20] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In I. Gupta and R. Wattenhofer, editors, *PODC*, pages 129–138. ACM, 2007.

[21] C. Cachin, I. Keidar, and A. Shraer. Principles of untrusted storage: a new look at consistency conditions. In R. A. Bazzi and B. Patt-Shamir, editors, *PODC*, page 426. ACM, 2008.

[22] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[23] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.

[24] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19. ACM, 1988.

[25] D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society, 2005.

[26] CNET News. Red Hat, Fedora servers compromised, Aug. 2008. Available online: `http://news.cnet.com/8301-1009_3-10023565-83.html`.

[27] R. Cramer and S. Fehr. Optimal black-box secret sharing over arbitrary abelian groups. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2002.

[28] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In B. Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[29] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.

[30] I. Damgård and M. Keller. Secure multiparty AES. In *FC*, 2010 (to appear).

[31] I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2002.

[32] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[33] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *ACISP*, volume 4586 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2007.

[34] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

[35] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261, 2008.

[36] I. Damgård, M. Geisler, and M. Krøigaard. A correction to 'Efficient and Secure Comparison for On-Line Auctions'. *IJACT*, 1(4):323–324, 2009.

[37] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[38] I. Damgård, M. Geisler, and J. B. Nielsen. From passive to covert security at low cost. Cryptology ePrint Archive, Report 2009/592, 2009. `http://eprint.iacr.org/`.

[39] I. Damgård, M. Geisler, and J. B. Nielsen. From passive to covert security at low cost. In D. Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[40] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1989.

[41] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.

[42] eBay Inc. Bid increments, Oct. 2006. Available online: `http://pages.ebay.com/help/buy/bid-increments.html`.

[43] *SFDL Specification — Version 2.0.* Fairplay Project, Sept. 2008.

[44] M. Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In D. Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 457–472. Springer, 2001.

[45] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In B. S. K. Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997.

[46] J. Garay, B. Schoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. In T. Okamoto and X. Wang, editors, *PKC*, volume 4450 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2007.

[47] M. Geisler. VIFF: Virtual ideal functionality framework, 2007. Homepage: `http://viff.dk/`.

[48] M. Geisler. *Implementing Asynchronous Multi-Party Computation.* PhD progress report, University of Aarhus, Denmark, Jan. 2008.

[49] M. Geisler and N. P. Smart. Distributing the key distribution centre in Sakai-Kasahara based systems. In M. G. Parker, editor, *IMA International Conference*, volume 5921 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 2009.

[50] M. Geisler, I. Damgård, and B. Pinkas. MPC virtual machine specification. Technical Report D4.3, CACE: Computer Aided Cryptography Engineering, 2009.

[51] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[52] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*. The Internet Society, 2003.

[53] O. Goldreich and V. Rosen. On the security of modular exponentiation with application to the construction of pseudorandom generators. *Journal of Cryptology*, 16(2):71–93, 2003.

[54] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

[55] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT*, pages 289–306, 2008.

[56] J. Groth. A verifiable secret shuffle of homomorphic encryptions. In Y. Desmedt, editor, *PKC*, volume 2567 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2003.

[57] J. Groth. Cryptography in subgroups of $\mathbb{Z}_n^*$. In J. Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[58] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[59] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[60] M. Hirt and U. M. Maurer. Robustness for free in unconditional multi-party computation. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2001.

[61] M. Hirt, J. Nielsen, and B. Przydatek. Asynchronous multi-party computation with quadratic communication. In *ICALP (2)*, pages 473–485, 2008.

[62] IT- og Telestyrelsen. OCES — digital signatur, 2005. Homepage: `https://www.signatursekretariatet.dk/`.

[63] R. Jagomägis. Privacy-aware programming language SecreC. Technical report, University of Tartu, Institute of Computer Science, Nov. 2009.

[64] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[65] G. Lefkowitz, I. Shtull-Trauring, et al. Twisted. Release 2.5.0, Twisted Matrix Laboratories, Jan. 2007. Homepage: `http://twistedmatrix.com/`.

[66] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004.

[67] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[68] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, pages 135–150, 2000.

[69] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[70] A. Martelli et al. GMPY project, 2010. Homepage: `http://code.google.com/p/gmpy/`.

[71] A. Mauland. Realizing distributed RSA using secure multiparty computations. Master's thesis, Norwegian University of Science and Technology, July 2009.

[72] D. Mazières and D. Shasha. Building secure file systems out of byantine storage. In *PODC*, pages 108–117, 2002.

[73] D. Mazières, M. Kaminsky, F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedinigs of the 17th ACM Symposium on Operating System Principles (SOSP)*, 1999.

[74] S. Meldgaard, I. Damgård, J. D. Nielsen, and M. Schwartzbach. Domain specific language specifications with benchmark requirements. Technical Report D4.2, CACE: Computer Aided Cryptography Engineering, 2009.

[75] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[76] D. L. Mills. A brief history of NTP time: Memoirs of an Internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.

[77] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM Transactions on Storage*, 2(2):107–138, May 2006.

[78] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, New York, 1999. ACM Press.

[79] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, pages 21–30. ACM, 2007.

[80] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In T. Okamoto and X. Wang, editors, *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.

[81] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[82] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 437–448. ACM, 2008.

[83] S. L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[84] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over $\mathbf{GF}(p)$ and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[85] J. B. Postel, editor. *Internet Protocol*, RFC 791. Internet Engineering Task Force, Sept. 1981. Available online: `http://ietf.org/rfc/rfc791.txt`.

[86] J. B. Postel, editor. *Transmission Control Protocol*, RFC 793. Internet Engineering Task Force, Sept. 1981. Available online: `http://ietf.org/rfc/rfc0793.txt`.

[87] PyPy Development Team. PyPy project, 2004. Homepage: `http://codespeak.net/pypy/`.

[88] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[89] M. Rode. Debat: Reglerne er strammet op. *Morgenavisen Jyllands-Posten*, Oct. 2004.

[90] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[91] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[92] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11): 612–613, Nov. 1979.

[93] N. P. Siong and H. Toivonen. M2Crypto, 2008. Homepage: `http://chandlerproject.org/Projects/MeTooCrypto`.

[94] T. Toft. *Secure Integer Computation with Applications in Economics.* PhD progress report, University of Aarhus, Denmark, June 2005.

[95] T. Toft. *Primitives and Applications for Multi-party Computation*. PhD thesis, University of Aarhus, Denmark, Mar. 2007.

[96] G. van Rossum et al. Python. Release 2.5, Python Software Foundation, Sept. 2006. Homepage: `http://python.org/`.

[97] H. Vegge. Realizing secure multiparty computations. Master's thesis, Norwegian University of Science and Technology, June 2009.

[98] C. Winter, J. Yasskin, and R. Kleckner. Merging Unladen Swallow into CPython. PEP 3146, Python Foundation, 2010. Available online: `http://www.python.org/dev/peps/pep-3146/`.

[99] C. P. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don't know can hurt you. In *IEEE Security in Storage Workshop*, pages 47–61. IEEE Computer Society, 2003.

[100] A. C.-C. Yao. Protocols for secure computations. In *FOCS*, pages 160–164. IEEE, 1982.