D11.1

A Theoretical Evaluation of the Existing Secure Computation Solutions

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1 November, 2013
Duration:	36 months
Programme:	FP7/2007-2013
Deliverable Type:	Report
Reference Number:	ICT-609611 / D11.1 / 1.0
Activity and WP:	Activity 1 / WP11
Due Date:	October 2014 - M12
Actual Submission Date:	27th February, 2015
Responsible Organisation:	BIU
Editor:	Benny Pinkas
Dissemination Level:	Public
Revision:	1.1
Abstract:	The area of secure multi-party computation has experienced considerable progress in recent years, but research results were described in many separate and independent publications. This document summarizes an evaluation of existing state-of-the-art protocols for secure multi-party computation, and highlights the protocols that seem most relevant for actual deployment in the near term.
Keywords:	Secure multi-party computation, survey, analysis.

This project has received funding from the European Unions Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 609611.

Editor

Benny Pinkas (BIU)

Contributors (ordered according to beneficiary numbers)

Claudio Orlandi (AU)
Benny Pinkas (BIU)
Bogdan Warinschi (UNIVBRIS)
Dan Bogdanov (CYBER)
Thomas Schneider (TUDA)
Michael Zohner (TUDA)
Meilof Veeningen (TUE)
Niels de Vreede (TUE)

Executive Summary

This deliverable provides a theoretical evaluation of existing solutions to secure multi-party computation. This topic has received considerable research, particularly in the last decade, but the existing results were described in many separate publications, making it hard to obtain a complete picture of available technologies. The deliverable summarizes our analysis of existing techniques, and describes details of the techniques which were found to be most promising.
The first chapter of this deliverable describes the basic definitions and properties of secure multi-party computation. The second chapter describes a taxonomy of secure multi-party computation solutions, which is based on properties that are important for potential users of this technology (such as, e.g., the maturity of existing implementations), rather than on properties of the mathematical constructions.
The third chapter describes basic secure multi-party computation constructions that are secure against a weaker type of adversary, denoted as a semi-honest or passive adversary. The fourth chapter describes state-of-the-art solutions based on a technique called "cut and choose", with security against the strongest type of adversary, denoted as a malicious or active adversary. The fifth and sixth chapters describe state-of-the-art solutions that demonstrate extraordinary efficiency at the cost of running an initial preprocessing computation before the inputs become known. These solutions, too, are secure against malicious adversaries.
Whereas most of this document describes generic secure multi-party protocols that can be used for computing any functionality, the seventh chapter describes secure protocols for a specific task that we believe to be relevant to many applications: computing the intersection of two sets that are each known to a different party, in a way which reveals to the parties nothing but the intersection itself. This chapter describes solutions for this task that are much more efficient than using generic protocols for computing the intersection. The eighth chapter describes solutions that enable to verify the correctness of the computation by parties that were not involved in the computation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a distributed computing setting, several distinct parties wish to jointly compute some function of their respective inputs. *Secure multi-party computation* (or *MPC* in short) is a distributed computation that is run between several parties which have confidential inputs, and that must be secure even in the face of participants that are deliberately trying to "attack" the correct operation of the protocol. These participants might attempt, for example, to learn the inputs of other parties, or to affect the result of the computed function. To give a sense of the motivation for using secure computation and of settings where secure computation is called for, we briefly list here some exemplary scenarios. (A detailed analysis of settings in which secure multi-party computation can be useful appears in Chapter 2.)

- *Data analysis:* Two or more parties have large private data sets (say, the parties are hospitals and the data is medical records). They wish to run some data-mining algorithm on the union of their data sets, but must do that without revealing the data itself. That is, they need to learn the result of the data-mining algorithm, but do so without disclosing any other information about the private data sets.

- *Outsourced data:* One or more parties store their data encrypted on a remote server (or on several such servers). They then wish to compute some function of the data where the bulk of the computation is performed on the server side, but without disclosing the data itself to the servers.

- *Auctions:* A large number of bidders participate in a sealed-bid auction. Bids must be submitted by a specific deadline. We wish that the auctioneer, who decides the result of the auction, can do that without learning the values of the bids.

1.1 Security

Desired properties. Since the computation involves different parties that might have different motivations, it cannot be ruled out that some of these parties might behave maliciously in order to learn information about the private inputs of other parties, or in order to change the result of the computation. We must therefore assume that there might be an adversary that controls one or several of the parties participating in a secure computation protocol, and we must ensure that the protocol is secure even in this case. There are several properties that seem to need to be protected in order for the protocol to be secure.

- *Privacy:* No party should learn anything more than its prescribed output. That output might enable to learn something about other parties' inputs, but this seems inevitable.

For example, in an auction where the output contains the highest bid, it is possible to derive from this output that all other bids were lower than the winning bid. However, this should be the only information revealed about the losing bids.

- *Correctness:* The protocol must ensure that the correct function is computed. For example, in the case of the auction the winner must indeed be the highest bidder, and no corrupt behavior can make the protocol announce that another party is the winner.
- *Independence of Inputs:* Corrupt parties must be prevented from setting the value of the their inputs to depend on the inputs of other parties. Note that the fact that a protocol guarantees the privacy of the inputs does not necessarily guarantee the independence of the inputs (and therefore additional care must be taken to ensure the latter property). For example, a corrupt auction participant must not be able to behave in the protocol in a way that sets its input to be one Euro higher than the maximum of all other bids. Note that even if private is guaranteed, additional measures
- *Fairness:* The protocol must prevent a scenario where a corrupted party receives its output while honest parties do not receive their outputs. This property can be crucial, for example, for a protocol where a money transfer is exchanged for a receipt. There it must not occur that one party receives the transfer while the other party does not receive the receipt.
- *Verification:* It should be able to verify the correctness of a protocol run, even by parties different than the parties which executed the protocol.

It must be emphasized that not all protocols satisfy all these properties, but this list of properties does serve as a desired goal for secure multi-party protocols.

1.1.1 Security definitions

The standard definitions that are used today for defining security do not try to separately satisfy each of the aforementioned properties, but rather use the following "holistic" approach: Consider an *"ideal world"* in which an external trusted party is willing to help the parties carry out their computation. In such a world, the parties can simply send their inputs to the trusted party, who then computes the desired function and sends the outputs back to the parties. Since the only action carried out by a party is that of sending its input to the trusted party, the only freedom given to the adversary is in choosing the corrupted parties inputs. Notice that all of the above-described security properties hold in this ideal computation. For example, privacy holds because the only message ever received by a party is its output (and so it cannot learn any more than this). Likewise, correctness holds since the trusted party will always compute the function correctly.
Of course, in the *"real world"* there is no external party that can be trusted by all parties. Rather, the parties run some protocol among themselves without the help of any external trusted party. The goal of secure computation is to ensure that an adversary in this real world cannot cause more harm than in the ideal world (i.e., can only decide its input to the protocol, and try to learn something from the output that it receives). Therefore, the security definitions requires the following property: A real protocol that is run by the parties (in a world where no trusted party exists) is said to be secure if no adversary can do more harm in a real execution than in an execution that takes place in the ideal world. This can be formulated by saying that for any adversary carrying out a successful attack in the real world, there exists an adversary

that successfully carries out the same attack in the ideal world. However, successful adversarial attacks cannot be carried out in the ideal world and therefore no such attacks can occur in the real world.
Security was rigorously defined in the cryptorgaphic literature. Detailed definitions of security can be found, e.g., in [52, 58]. In these formal definitions the security of a protocol is established by comparing the result of a real protocol execution to the outcome of an ideal computation. That is, for any adversary attacking a real protocol execution, it must be shown that there exists an adversary attacking an ideal execution (one that is run with the help of a trusted party) such that the input/output distributions of the adversary and of the participating parties in the real and ideal executions are essentially the same. This formulation of security is called the ideal/real simulation paradigm. In order to motivate the usefulness of this definition, we describe why all the properties described above are implied.

- *Privacy* follows from the fact that the adversary's output is the same in the real and ideal executions. Since the adversary learns nothing beyond the corrupted party's outputs in an ideal execution, the same must be true for a real execution.
- *Correctness* follows from the fact that the honest parties outputs are the same in the real and ideal executions, and from the fact that in an ideal execution, the honest parties all receive correct outputs as computed by the trusted party.
- Regarding *independence of inputs*, notice that in an ideal execution, all inputs are sent to the trusted party before any output is received. Therefore, the corrupted parties know nothing of the honest parties inputs at the time that they send their inputs. In other words, the corrupted parties inputs are chosen independently of the honest parties inputs, as required.
- *Fairness* holds in the ideal world because the trusted party always returns all outputs. The fact that it also holds in the real world again follows from the fact that the honest parties outputs are the same in the real and ideal executions. (It should be noted that while the fairness property can be satisfied in principle, it is often quite inefficient to satisfy in protocols that aim to be practical.)

1.1.2 Adversarial power

Any security definition must also consider the power of the adversary that attacks a protocol execution. An adversary might control a subset of the parties participating in the protocol, and might operate in different ways. The analysis usually considers a worst case scenario where a single adversary controls multiple parties and can ensures that they collaborate in the adversary's favor (this situation is worse than that of multiple corrupt parties operating independently of each other).

Allowed adversarial behavior. There are different behaviors that an adversary might dictate to the parties that it controls:

- Adversaries that are denoted as *semi-honest* or *passive*, do not change the way in which the corrupted parties (that are controlled by them) participate in the protocol. That is, these parties correctly follow the protocol specification. However, the adversary obtains the internal state of all the corrupted parties (including the transcript of all the messages received), and attempts to use this to learn information that should remain private.

b
b

This is a rather weak adversarial model. However, there are some settings where it can realistically model the threats to the system. Semi-honest adversaries are also sometime called *honest-but-curious*.

The semi-honest model might model a scenario where protocol participants are honest during the protocol execution, but might become corrupted by adversary after the end of the protocol. At that stage the adversary gets to see the logs and audits of the protocol execution, and tries to learn information from these records. Moreover, it is not trivial to obtain security even against this weaker model of corruption, and therefore this problem is interesting as a first step before devising protocols that are secure against stronger adversarial behavior.

- *Malicious*, or *active* adversaries have full control over the corrupt parties, and these parties can arbitrarily deviate from the protocol specification, according to the adversary's instructions. In general, providing security in the presence of malicious adversaries is preferred, as it ensures that no adversarial attack can succeed. However, providing this level of security might be more difficult or costly compared to providing security against semi-honest adversaries.

- Security against *covert* adversaries defines a security level that often results in a reasonable tradeoff between security and efficiency. Covert adversaries, defined in [4], have full control over corrupt participants but also attempt to avoid being detected. A covert adversary with deterrence factor of ε will choose not to cheat if the probability of it being detected is greater than ε. In many settings where there are long lasting relationships between parties, the deterrence factor can be quite high, say 1% or 50%. (Whereas malicious adversaries might try to cheat even if their probability of success is much lower.) The overhead of protocols secure against covert adversaries becomes smaller as the deterrence factor grows, since the protocols can tolerate cheating to go unnoticed, as long as this happens with probability smaller than ε.

Complexity. The computational complexity of the adversary is an additional parameter of its power.

- *Polynomial-time* complexity is the common bound used in the theory of computer science for defining efficient computation. Therefore it is assumed that the adversary is allowed to run in (probabilistic) polynomial-time (that is, a run time that is a polynomial function of a security parameter, which is typically the length of the cryptographic keys used by the system). This means, for example, that the adversary cannot perform computations that take super-polynomial time (e.g., exponential time). For example, the adversary will not be able to break the security of common encryption algorithms, such as AES or RSA. (An adversary running in exponential time will be able to break these ciphers by simply enumerating over all possible encryption keys.)

- A stronger security model assumes that the adversary is *computationally unbounded*. In this model, the adversary has no computational limits; it can, for example, run arbitrarily long brute force attacks and break any encryption scheme that does not have information theoretic security. This model seems too powerful, but can be useful for ensuring that a protocol is secure even against future advances in cryptanalysis and in computational power. In some cases, it is indeed possible to design multi-party protocols that are information theoretically secure against computationally unbounded adversaries.

b
b

Corruption strategy. There are two main ways in which the adversary might corrupt the parties participating in the protocol.

- In the *static corruption* model the adversary decides in advance, before the start of the protocol, on a fixed set of parties whom it controls. Honest parties remain honest throughout the protocol, and corrupted parties remain corrupted.
- In the *adaptive corruption* model, rather than having a fixed set of corrupted parties, the adversary is able to dynamically corrupt parties during the computation. The choice of who to corrupt, and when, can be arbitrarily decided by the adversary and may depend on its view of the execution (for this reason it is called adaptive).

The static corruption case seems quite satisfactory for most scenarios, in particular for protocols that are not used as building blocks for higher level protocols. Achieving security against adaptive adversaries is typically quite costly in terms of performance.

1.2 Secure Multi-Party Protocols (MPC)

The definitions of secure computation seem rather ambitious, and therefore it is unclear (or at least it was not initially clear) whether achieving these security definitions is feasible. (By feasible we mean a protocol that satisfies the security definitions and runs in polynomial time. Feasibility results do not address any detailed performance analysis.) Feasibility research defines the boundaries of secure multi-party computation. Afterwards, much additional work is required in order to improve protocol performance. We therefore first describe basic feasibility results in secure multi-party computation, and then discuss actual secure multi-party protocols.

1.2.1 Feasibility of Secure Multi-Party Computation

Strong feasibility results have been established for different variants of secure multi-party computation. In order to describe these results, let us denote by n the number of parties participating in the protocol, and denote by t a bound on the number of parties that may be corrupted.

- For $t < n/3$, secure multi-party protocols with fairness can be achieved for any function in a point-to-point network and without any requirement for a trusted setup of the system.[1] This result can be achieved both in the setting where the adversary is computationally bounded (i.e., the computational setting) [53] (assuming the existence of enhanced trapdoor permutations), and with a computationally unbounded adversary (assuming that there are private channels between any two parties [16, 28].
- For $t < n/2$ (i.e., in the case of a guaranteed honest majority), secure multi-party protocols with fairness can be achieved for any function assuming that the parties have access to a broadcast channel. This can be achieved in the computational setting [53] (with the same assumptions as above), and in the information-theoretic setting (i.e., against adversaries that are computationally unbounded) [110].

[1] The fairness property is rather unique among the other security properties in that achieving the full notion of fairness is either impossible in some settings (although weaker notions of fairness are possible), or is rather expensive.

- For $n/2 \leq t < n$(i.e., when the corrupted parties can include all parties but one), secure multi-party protocols (without fairness) can be achieved assuming that the parties have access to a broadcast channel (and in addition assuming a stronger assumption about the existence of enhanced trapdoor permutations) [124, 53, 52]. These feasibility results hold only in the computational setting; analogous results for the information-theoretic setting cannot be obtained when $t \geq n/2$ [16].

This set of results shows that secure multi-party protocols exist for any distributed computing task. In the computational model, this holds for all possible numbers of corrupted parties. When, in addition, no honest majority exists, fairness is not obtained. All these results hold with respect to *malicious* adversaries, as well as for semi-honest adversaries. The status regarding adaptive versus static adversaries is more involved and is therefore omitted here.

1.2.2 Protocols

The initial work on secure multi-party computation was carried out in the late 80s and early 90s, and focused on feasibility results. The main research goal during that period was to establish which tasks can be computed securely and at which (asymptotic) cost. The typical research agenda of that time was influenced by complexity theory. Issues of concrete efficiency and implementation were not addressed at that time. In general, research at that time followed the following general lines:

- Each computation that can be efficiently computed by a program (i.e., computed in polynomial time), can also be computed by a Boolean or arithmetic circuit of polynomial time. It is therefore sufficient to focus on investigating secure computation for circuits of these types.
- Each Boolean circuit can be composed of AND and NOT gates alone. Similarly, each arithmetic circuit can be composed of gates computing addition and multiplication in some field. Therefore it is sufficient to focus on designing secure computation protocols for these types of gates, and composing them together.
- The important factor is the asymptotic overhead of the computation, rather than the constants. Therefore there is no need to optimize the protocols in order to reduce the constants that are incurred by the computation.
- There were attempts to understand and optimize some issues of the computation. These included the cryptographic assumptions on which security based based; the number of parties that can be corrupted by the adversary (the more corrupted parties the harder it is to ensure security); and the number of rounds of the computation.

Security against malicious adversaries

An example of the research agenda of the early days of research in secure multi-party computation is the way in which security against malicious adversaries was solved. Designing protocols that are secure against semi-honest adversaries was by itself a formidable task. Once such protocols were designed, it was shown by Goldreich, Micali and Widgets [53] how to "compile" any protocol secure against semi-honest adversaries to a protocol secure against malicious adversaries.
The transformation was achieved using zero-knowledge proofs. These proofs, initially formulated in [54], enable to prove the correctness of any NP statement in a way that reveals nothing

but the fact that the statement is correct. (For example, one could use these techniques to prove that it knows the factorization of a number without revealing any information about the factorization itself.) Now, recall that a semi-honest adversary is one which follows the protocol whereas a malicious adversary might behave arbitrarily. The generic transformation operates in the following way:

- Design a protocol secure against semi-honest adversaries.
- The protocol defines for each party P_i a program $Prog_i$ that P_i must run. The input to this program is the private input of this party, the randomness that it uses, and the messages it receives from other parties. Given these values the output of P_i is defined by $Prog_i$ in a deterministic way. This program $Prog_i$ is publicly known.
- Consider the following NP statement that can be given by P_i: "There exist a private input and internal randomness, such that the messages that I sent during the protocol execution are the output of $Prog_i$ given these values and the messages I received from other parties".
- Party P_i can therefore use zero-knowledge proofs to prove this statement without revealing any information about its inputs.
- That is, P_i proves that it behaves as a semi-honest adversary. The protocol that is run is secure against this type of behavior.

This approach is elegant and very powerful. It is also efficient in the sense that it runs in polynomial time. However, the actual runtime and space complexity of the proofs can be huge (albeit polynomial): The statement is usually translated to a statement in an NP-complete language, such as graph colorability or 3-SAT. This transformation requires to express the statement as a Turing machine program or as a Boolean circuit, and then transform the result to, e.g., a 3-SAT problem. The constants factors that are incurred by this approach are huge.

Recent work

More recently it was realized that the overhead of secure multi-party computation is actually quite feasible for many tasks that are of practical interest. As a result, more research has been invested in improving the overhead of secure computation protocols, resulting in huge improvements in their overhead.
One of the first results in this new line of work is a protocol for privacy preserving auctions [95] that hides the values of the bids even from the auctioneer itself. The protocol encoded the auction algorithm as a Boolean circuit (which is of a very reasonable size, since comparison operations can be implemented by sub-circuits whose size is linear in the length of the numbers that are compared). The protocol was implemented and run in a reasonable time.
A major boost for research on the efficiency and ease of development of secure multi-party protocols came with the Fairplay project [85]. That project developed a generic tool for secure two-party computation, consisting of (1) a high-level programming language for describing the function that needs to be computed; (2) a compiler translating programs written in this language to Boolean circuits; and (3) programs for the two parties to actually run Yao's secure two-party protocol on the resulting circuit. In doing so, the Fairplay project demonstrated that it is possible to transform the field of secure computation from a set of mathematical theorems to a set of tools that can be used by end-users who need not be familiar with the underlying cryptographic ideas and tools.

A strong indication of the feasibility and of the utility of secure multi-party computation was shown by work in Denmark on a secure protocol for auctions of sugar beets. That work resulted in a secure protocol that is run in practice until these days between farmers and the sugar beet processor, and sets the clearing price for sugar beets [24].
The initial work on implementations of multi-party protocols motivated new research directions in MPC technology:

- *Optimizing the constants.* The fact that secure multi-party protocols can be run in practice, motivated research into optimizing the actual run time of the protocols. In particular, it was obvious that it was important to optimize the constant coefficients of the overhead, rather than only care about the asymptotic overhead. For example, there are results on reducing, by a factor of $25\% - 50\%$ the size of the data that has to be communicated and the number of operations that have to be performed per gate [107, 76]. It was also shown how Yao's protocol for secure two-party computation can compute XOR gates essentially for free [75]. As a result, it makes sense to change the design of circuits so that they use a minimal number of gates that are different than XOR, even if this results in an additional number of XOR gates.

- *Security against malicious adversaries.* There were new research efforts to design efficient protocols that are secure against malicious adversaries. These efforts were motivated by the fact that security against malicious adversaries is required in many scenarios, and since the existing techniques, namely the GMW compiler based on generic zero-knowledge proofs, were extremely inefficient. Initial results, using the so called cut-and-choose methodology were shown in [81, 83, 82]. Many results followed.

- *Offloading work to a preprocessing phase.* Another approach for optimization is to offload the bulk of the overhead to a preprocessing phase that can be run before the parties learn their inputs. The rest of the computation (the online phase) can then become extremely efficient. This approach was first suggested by Beaver [12] and referred to as "Beaver triples"', is used by many state-of-the-art protocols for secure computation between many parties (in particular the protocols that we present in Chapters 5 and 6, namely the SPDZ protocol and its variants).

  Another very effective approach, denoted as "oblivious transfer extension", handles a cryptographic primitive known as "oblivious transfer", which was the performance bottleneck of many secure protocols. It was observed that precomputing a small number of oblivious transfers in a preprocessing phase enables to compute all future oblivious transfers very efficiently, using symmetric key cryptography alone [11]. This idea was later optimized [64, 3] and is used in many recent implementations of secure computation.

- *Implementation friendly protocols.* The performance of secure multi-party protocols can be greatly improved by taking into account systems issues, and changing the protocols to make the best use of the available resources. For example, the protocols can use the AES-NI instruction that is available in modern Intel chips. They can further optimized if there is no need to perform frequent AES key changes (since AES key scheduling slows down performance), and if pipelining can be used. Furthermore, the protocols can be designed to benefit from using multi-core architectures or GPUs. In addition, circuits that are very large might not naively fit in main memory. Efficient implementations should overcome this issue.

Chapter 2

A Taxonomy of Secure Computation Settings and Solutions

2.1 Introduction

Work on categorizing secure computation techniques has typically concentrated on the formal (e.g., cryptographic, complexity-theoretic) properties of the individual schemes. A good recent classification on the topic has been provided by Perry et al. [106]. In PRACTICE, we extend this systematization with categories that focus on the practical use of secure computation in real-world applications.

The PRACTICE project has described several application models that benefit from secure computation scenarios [57]. These models are based on both academic prototypes and real world deployments. However, not all secure computation techniques can support each application model. Therefore, we will define three general **usage models** that every known application fits into. Later, we will assign one or more categories to each secure computation technique. These assignments suggest, what kind of applications does the particular technology support best.

In this deliverable, we describe a wide range of cryptographic techniques for secure computation. Not all of them, however have been implemented or used in practice. Therefore, we will assign a **maturity of runtime** category to each technique. This category will indicate the engineering level of the best publicly known implementations of runtimes. Similarly, we will describe the kind of **application development tools** for each technique.

A developer considering the use of secure computation should also be aware of possible programming paradigm for a given tool. This information can greatly influence the choice as some techniques are significantly more efficient with certain types of problems. Therefore, we will list the **programming paradigm** for each secure computation technique.

The number of published secure computation techniques is large and growing, but not all of them have accompanying practical implementations in software or hardware. Schemes with great complexity-theoretic properties can be hard to implement in practice for various reasons. Once implemented, the performance of the system may not be suitable for practical applications. In this classification, we will assign the **performance level** category to secure computation techniques to denote the performance level of implementations available in practice.

2.2 Usage models

In deliverable D12.1 [57], we described secure computation applications according to the flow of private data among parties. We will now use the same model to describe three usage categories.

Note that these categories are non-exclusive as a secure computation paradigm can be suitable for many uses.

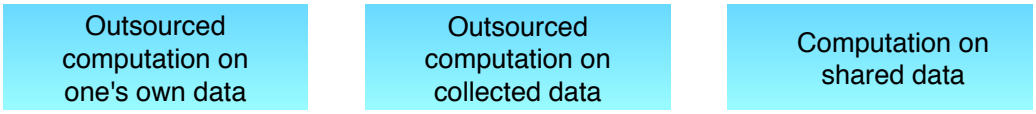
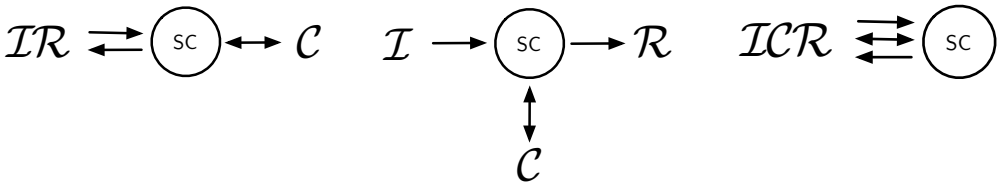


Figure 2.1: Categories of usage models



(a) Process own data (b) Process collected data (c) Process shared data

Figure 2.2: Abstract usage models of secure computing

Usage model 1: Outsourced computation on one's own data. In this category, we have use-cases where a data owner has data that needs to be processed and wants to outsource such processing to a service. This clearly represents the most common cloud computing scenarios where an individual or an organization outsources computation to a service provider. Secure computation is needed here to ensure that the service does not learn the confidential information or leak it to third parties.
The model is shown on Figure 2.2a. The input party encrypts its input data and sends it to the computing party (e.g., a cloud service provider) who processes the data without decrypting it during the process. Once the computation has been completed, the encrypted result is returned to the data owner who can decrypt it. Relevant secure computation techniques for this category include homomorphic encryption, property-preserving encryption and trusted hardware.

Usage model 2: Outsourced computation on collected data. The second category handles the situation where the party who needs to process the data does not have the data and needs to collect it. This scenario occurs in surveys, government statistics, social studies, medical research, auctions, and a wide range of corporate activities. Secure computation is needed to ensure that nobody except for the data owner has access to the data but, at the same time, data utility is retained.
The model in Figure 2.2b differs from the previous model in the way that the input parties $\mathcal{I}$ and the result parties $\mathcal{R}$ are separated from each other. This is to signify the fact that the $\mathcal{I}$ parties do not trust the $\mathcal{R}$ parties. Examples of suitable secure techniques for outsourced computation on collected data include property-preserving encryption and secure multi-party computation based on secret sharing or garbled circuits.

Usage model 3: Computation on shared data.
The final of the three categories describes applications where similar parties combine their information to jointly learn new things. These scenarios occur in industrial consortiums, research collaborations and joint activities between coalitions of nations. Secure computations allow partners to share data while ensuring control of this data during its processing. The latter

is achieved by having all parties participate in the actual computation (i.e., all parties being computing parties).
This model is shown on Figure 2.2c. Each party fulfils all roles by providing an input, contributing to the computation and benefiting from the results.

These models are abstract and can have modifications. E.g., in model of computation on shared data, not all parties may want to become computing parties, but will be happy with just providing inputs and learning outputs (and trust that the number of computing parties which is corrupt is limited, and therefore they are unable to breach the protocol). In all these settings, the computing parties (the $\mathcal{C}$ nodes) can be deployed on the cloud, because secure computation guarantees that the computing parties do not learn the private inputs.

2.3 Maturity of implementation

Secure computation implementations consist of various components, including the software - hardware for performing the cryptographic operations on data and the developer tools used to create applications. This category describes the readiness of a complete implementation for practical use. This evaluation is performed based on the capability to deploy the system in real applications. Figure 2.5 shows the progression of the levels towards the level of greater maturity.

Figure 2.3: Levels of implementation maturity

Maturity level 1: Theoretical construction.
Theoretical constructions are schemes that are described in research papers, but no reusable implementations are available. On this level, there are secure computation techniques for which the protocols are published, but no published implementation is known [1]. Maturity level 1 corresponds to Technological Readiness Levels 1 to 2[2]

Maturity level 2: Academic prototype.
Academic prototypes are developed by researchers to study a new secure computation technique, its costs and complexities. They often accompany research papers and provide preliminary performance metrics. These prototypes maybe short-lived, just to support a certain research paper. However, some such implementations persist and extend to provide support for many papers. Maturity level 2 corresponds to Technological Readiness Levels 3 to 4.
Notable examples of often reused research platforms for secure computation include Fairplay [47], SCAPI [45], Sharemind [116], TASTY [118] and VIFF [120].

Maturity level 3: Real-world applications.

[1] All decisions on whether an implementation is known, have been checked before the publication date of the deliverable. It is also possible that the authors may have missed a certain paper or published implementation. The authors will be thankful to learn of such omissions and to correct them.

[2] U.S. Department of Defense Technological Readiness Assessment (TRA) Guidance. `http://www.acq.osd.mil/chieftechnologist/publications/docs/TRA2011.pdf`

The development of real-world applications requires both technical and administrative excellence, as one has to meet end user acceptance and compliance goals. Until now, practical secure computation application have been mostly developed by research teams, using improved versions of academic prototypes. Still, secure computation applications with real-world stakeholder require significant effort over such prototypes. Maturity level 3 corresponds to Technological Readiness Levels 5 to 6.
The first practical real-world application of secure computing was the double auction for sugar beet producing contracts held in Denmark in 2008 [24]. Since then, more applications have been deployed, e.g. for financial reporting in an Estonian consortium of ICT companies [23].

Maturity level 4: Ready for industrial use.
Once an implementation has been used in several real-world applications, its developers can consider initiating a technology transfer process and make the tools available to the general public. As these levels are based on practical capability, it makes no difference whether the offering is based on a commercial or an open-source license. Maturity level 4 corresponds to Technological Readiness Levels 7 to 9.
At the time of writing this report, two companies are known to provide services based on cryptographic secure computation–Cybernetica[3] in Estonia and Partisia[4] in Denmark.

2.4 Programming paradigms

Secure computation systems have different programming approaches, based on the kind of protocols that are used. This information is useful for application developers as it can indicate how easy it would be to extend the system or to provide it with new features. For example, the development tools for Boolean circuits are fundamentally different than custom set intersection protocols. The different programming paradigms are described below and in Figure 2.4.

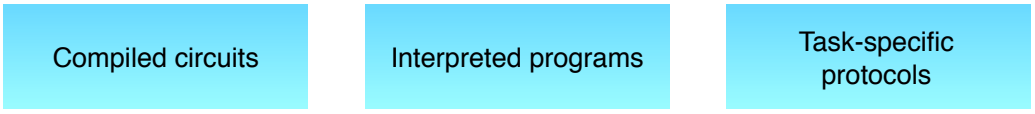


Figure 2.4: Categories for programming paradigm

Programming paradigm category 1: compiled circuits.
Boolean and arithmetic circuits are representations of a computing task inspired from electronics. Indeed, one can imagine a Boolean circuit as an electrical circuit in a processor. Boolean circuits operate on a bit level with logical operations and are, therefore, very compact representations of a computing task. Arithmetic circuits work on elements of a group, ring or field and use the arithmetic operations available. The common property of circuits is that they represent the complete secure computing task on a low level.

Programming paradigm category 2: interpreted programs.
With interpreted programs, the secure computing task is constructed from more complex secure primitives than arithmetic or logic. This is to reduce the amount of copying low-level circuits throughout the whole computation. This can make the application less efficient. However, it can also significantly simplify their development and execution, if the protocols with the necessary

[3]Cybernetica AS `http://www.cyber.ee`
[4]Partisia Market Design ApS `http://www.partisia.com`

security and composability properties are used. Such programs are typically evaluated on a virtual machine that executes secure computation primitives (e.g., Boolean circuits, algebraic protocols) on demand.

Programming paradigm category 3: task-specific protocols.
Tailored secure computation protocols exist for tasks like set intersection, frequent itemset mining etc. These protocols only perform the necessary task and are, therefore, harder to combine with other secure computation operations. However, their efficiency and compactness can easily justify their use. For example, it is more trivial to audit a protocol that does just one thing than a fully generic secure computation runtime.

2.5 Application development tools

When a secure computation system has been developed, it needs to be tailored for particular applications. With better tools, this tailoring becomes easier and requires less skilled labor and time. However, too much automation reduces flexibility and can reduce the usefulness of the tools. The following categories are not mutually exclusive as a secure computation technique could be programmable with several toolkits.

Figure 2.5: Categories of application development tools

Development tool category 1: hand-modified implementation.
In the simplest case, one directly modifies the secure computation runtime to integrate it with an application. This is the most time-consuming, but also most flexible approach. It is mostly used in academic projects.

Development tool category 2: embedded domain-specific language.
With an embedded DSL, some complexity of the secure computation paradigm can already be hidden from the application developer. Embedded DSLs allow the developer to describe the algorithm or business process and then translate it into the cryptographic operations underneath.

Development tool category 3: compiled language and interpreter.
A separate compilable language further separates the secure computation paradigm from the developer. An application written and compiled in such a language will later be interpreted by an interpreter or virtual machine that schedules the necessary secure computation primitives as required.

Development tool category 4: libraries of application-specific functionality.
Nearly all modern software is not developed from scratch with just the programming language. Instead, it is built on the foundation of other libraries. Once secure computation becomes easily programmable, the next step of the evolution is to create reusable libraries that speed up development. For example, these can be libraries in the programming language of the secure computation system.

2.6 Performance level

There is no established way for fairly comparing the performance of different secure computation systems. Each has its own bottlenecks and tasks that it can efficiently perform. Therefore, we have established our performance categories based on the best/most complex application or primitive that has been demonstrated in published literature. We also consider problems that are suitable for the particular secure computing system.

Figure 2.6: Levels of performance

Performance level 1: theoretical result.
This, simplest of categories, means that the result has not been implemented in practice and is, therefore, theoretical. Even if the research paper contains a good (e.g., constant overhead) complexity analysis, we cannot be sure of the performance until we have validated that constant in practice.

Performance level 2: single operations feasible.
The minimal implementation of a secure computing system typically picks a suitable primitive, implements and benchmarks it. Examples include secure arithmetic for systems based on homomorphic cryptography and a comparison or string algorithm for garbled circuits. At this level, we have systems that have published such a result and did not need extensive resources to complete. [5]

Performance level 3: algorithmic tasks feasible.
Belonging to the next level requires that the system can algorithmically combine secure operations to complete a non-trivial operation, e.g. sorting, clustering etc. We do not require a specific primitive, but some task that is suitable for the particular system. This category can be assigned to systems that have published results of experimentation with a non-trivial algorithm that uses branching and/or loops and composes several primitive operations. Again, we require that the resource use is not extensive.

Performance level 4: real-world applications feasible.
Real-world feasibility means that there is a use-case study or prototype application that demonstrates performance that is acceptable for end users. This is proven either by building the application and reporting successful real-world use, or showing a justified analysis showing that the resource usage is acceptable for an end user with realistic datasets.

[5] We interpret "extensive resources" (somewhat arbitrarily) as hardware and setup costing more than €100 000 by current prices.

Chapter 3

Generic Constructions Secure Against Semi-Honest Adversaries

This chapter outlines the main generic secure multi-party computation constructions that are secure against semi-honest (passive) adversaries: Yao's garbled circuits (Section 3.2) protocol for two-party secure computation [125]; the protocol of Goldreich, Micali and Wigderson, which is often referred to as the GMW protocol [53] (Section 3.3); and the protocol of Ben-Or, Goldwasser and Micali (the BGW protocol) [16]. The Yao and GMW protocols express the function to be computed as a Boolean circuit, and are secure based on standard assumptions in cryptography, similar to the hardness of computing discrete logarithms or factoring large numbers (these assumptions are also used to argue the security of standard public-key encryption schemes). The BGW protocol expresses the function that is computed as a arithmetic circuit, where each gate is an addition or a multiplication in a finite field. This protocol ensures unconditional security which does not depend on any cryptographic assumption. The Yao and GMW protocols use a simpler protocol, called oblivious transfer (OT), as an underlying primitive. We first describe this primitive in Section 3.1. Finally, we give a brief comparison between the protocols (Section 3.5). Some of our descriptions are based on [112] to which we refer for more details.

3.1 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic protocol executed between a sender S and a receiver R in which R obliviously selects one of the inputs provided by S.
More specifically, in the simplest case of a 1-out-of-2 oblivious transfer (which is also the variant used by most secure computation protocols), the parties have the following inputs:

- S has two inputs x_0, x_1. These inputs can be either bits or strings, depending on the variant of the protocol that is used.
- R has an input bit $b \in \{0, 1\}$.

At the end of the protocol R learns x_b but no other information about x_{1-b}, whereas S learns nothing.
In the general case, a 1-out-of-n OT_ℓ^m protocol is where S provides m n-tuples $(x_{11}, \ldots, x_{1n}), \ldots, (x_{m1}, \ldots, x_{mn})$ of ℓ-bit strings; R provides m selection numbers $r_1, \ldots, r_m$ with $1 \leq r_i \leq n$ and obtains x_{jr_j} $(1 \leq j \leq m)$ as output.

b
b

The widely used Naor-Pinkas OT protocol [94] is secure against semi-honest adversaries under the Decisional Diffie-Hellman (DDH) assumption in the random oracle model and requires both parties to perform $\mathcal{O}(m)$ modular exponentiations. Protocols also exist for computing OT with security against malicious (active) adversaries.
There are also additional powerful techniques that can substantially speed up the computation of OTs.

OT pre-computations The work in [11] showed how to pre-compute OTs on random inputs before the actual inputs are known, and later, in the online phase, use these pre-computed values as one-time pads to run OTs on the actual inputs.

OT extension In [64, 78] it was shown how to perform a large number, m, of OTs (namely OT_ℓ^m) using a small number of t base OTs on t-bit keys (OT_t^t), where t is a security parameter which can typically be set to $t = 128$. This approach is conceptually similar to the hybrid approach for encryption where instead of encrypting a large message using a public-key cipher such as RSA (which would be too expensive), a hybrid encryption scheme is used such that RSA is only used for encrypting a short symmetric key, and then the long message is encrypted using symmetric operations only.
The marginal cost for each additional OT in this approach is a small number of evaluations of a cryptographic hash function (modeled as random oracle) and of a pseudo-random function. These are very efficient symmetric key operations. More specifically, for each of the m OTs, S computes n hash evaluations and $t(1 + \log_2 n)$ pseudo-random bits, whereas R computes 1 hash evaluation and $t(1 + n)$ pseudo-random bits. The communication complexity is one message from R to S of size mnt bits and one in the opposite direction of size $mn\ell$ bits. Further optimizations for OT extension have been proposed in [73] and [3].

3.2 Yao's Garbled Circuits Protocol

The basic idea of Yao's garbled circuits protocol [125] is to let one party, called the *creator* or the circuit *constructor*, to encrypt the function to be computed. The function is represented as a Boolean circuit. The plain 0/1 values on wires are mapped to random-looking symmetric keys. Namely, for each wire two such random keys are chosen, one representing the 0 value and the other representing the 1 value. For each gate an encryption table is generated that allows to compute the gate's output key given its input keys. For instance, if one has the encryption table of, say, an AND table, and also has the keys that represent 1 values on each of the two input wires of the gate, then it is possible to compute the key representing a 1 value on the output wire of the gate (but no information about the key corresponding to 1 on this wire). For the same gate, given, say, the key corresponding to 0 on one input wire and the key corresponding to 1 on the other input wire, it is possible to compute the key corresponding to 0 on the output wire (but no information about the corresponding to 1 on this wire).
The creator transmits the encrypted circuit along with the keys corresponding to his own inputs wires to the other party, called the *evaluator*. The evaluator needs to first obtain the encrypted wire keys corresponding to his own inputs. He obtains them obliviously by running a 1-out-of-2 OT protocol with the creator. Once he has these keys, he uses them and the keys provided by the creator to evaluate the encrypted function gate by gate. Finally, the creator provides a mapping from the encrypted output to plain output. As the evaluation of Yao's garbled circuits is performed non-interactively, the resulting protocol has a constant number of rounds.
b
b

Yao's protocol has been extensively investigated. There are various extensions that enhance the speed of Yao's garbled circuits protocol: point-and-permute [85], free XOR [75], efficient encryption with a cryptographic hash function [83], garbled row reduction [95, 107], FleXOR citeKMR14, and pipelining [60]. Together, these techniques allow "free" evaluation of XOR gates (i.e., no communication and negligible computation); interweaving circuit generation and evaluation; and, per non-XOR gates, an overhead of performing 4 evaluations of a cryptographic hash function for the creator, transmission of an encrypted gate table with only 3 entries, and only a single evaluation of a cryptographic hash function for the evaluator. Another recently proposed optimization for Yao's protocol allows to use efficient hardware-based fixed-key AES for garbling [15].
Yao's protocol has also been extended to the multi-party setting, most notably in citeBMR90,BenDavidNP08.

3.3 The GMW Protocol

The GMW protocol [53, 52] can be used for secure computation between two or more parties (whereas Yao's protocol is specifically for two parties only). We describe here the two party version of the GMW protocol, run between two parties that can be denoted as P_1 and P_2. The two parties interactively compute a function using secret-shared values. For this, the value v of each input and intermediate wire is shared among the two parties in the following way: each party holds a random share v_i such that $v = v_1 \oplus v_2$. As XOR is an associative operation, XOR gates can be securely evaluated locally by XORing the shares. Namely, $v \oplus u = (v_1 \oplus v_2) \oplus (u_1 \oplus u_2) = (v_1 \oplus u_1) \oplus (v_2 \oplus u_2)$.
Secure evaluation of AND gates is more complicated. For that purpose the parties run an interactive protocol using one of the two techniques described below. Note that AND gates of the same layer in the circuit can be evaluated in parallel. Finally, the parties send their shares of the output wires to the party that should obtain the output. An implementation of the GMW protocol was given in [29] for multiple parties and in [112] for two parties.

Implementation based on oblivious transfer, To securely evaluate an AND gate on input shares v_1, v_2 and u_1, u_2, the two parties can run a 1-out-of-4 OT_1^1 protocol. Here, the chooser inputs its shares v_1, u_1 and the sender chooses a random output share z_2 and provides four inputs to the OT protocol such that the chooser obliviously obtains its output share $z_1 = z_2 \oplus ((v_1 \oplus v_2) \wedge (u_1 \oplus u_2))$. As described in Section 3.1, all OTs can be moved into a pre-processing phase such that the online phase is highly efficient (only two messages and inexpensive XOR operations).

Implementation based on multiplication triples. An alternative method to securely evaluate an AND gate on input shares v_1, v_2 and u_1, u_2 is to use *multiplication triples* [10]. Multiplication triples are random shares a_i, b_i, c_i satisfying $(c_1 \oplus c_2) = (a_1 \oplus a_2) \wedge (b_1 \oplus b_2)$. They can be generated in a setup phase, before the actual inputs are known, using a 1-out-of-4 OT_1^1 protocol in a similar way to the OT-based solution described above. In the online phase the parties use these pre-generated multiplication triples to mask the input shares of the AND gate, exchange $d_i = v_i \oplus a_i$ and $e_i = u_i \oplus b_i$, and compute $d = d_1 \oplus d_2$ and $e = e_1 \oplus e_2$. The output shares are computed as $z_1 = (d \wedge e) \oplus (b_1 \wedge d) \oplus (a_1 \wedge e) \oplus c_1$ and $z_2 = (b_2 \wedge d) \oplus (a_2 \wedge e) \oplus c_2$. The advantage of multiplication triples over the OT-based solution is that, per AND gate, each party needs to send only one message (independent of each other) and the size of the messages
b
b

is slightly smaller ($2+2$ bits instead of $2+4$ bits). As shown in [3], a multiplication triple can be generated efficiently using two random 1-out-of-2 OTs.

3.4 The BGW and CCD protocols

The protocols that are described in this section were designed for a set of n parties with possibly private inputs that wish to securely compute some function of their inputs in the presence of adversarial behavior. The BGW [16] and CCD [28] protocol showed that every functionality can be computed with *perfect security* in the presence of semi-honest adversaries corrupting a minority of parties, and in the presence of malicious adversaries corrupting up to a third of the parties. By perfect security we mean that there is zero probability for cheating by the adversary, and that security is not based on any cryptographic hardness assumption (such as the hardness of factoring numbers).
The BGW protocol builds on the GMW protocol. On a high level, the protocol works by having the parties compute a function (from n inputs) via an arithmetic circuit computing it. The parties first share their inputs to each other using the secret sharing scheme of Shamir [117]. For each gate, the parties compute shares of the output of a circuit gate given shares of the input wires of that gate. For addition gate this operation is simple, since the secret sharing scheme is linear: each party simply adds its shares of the two input wires to obtain a share of the output wire. For multiplication gates, the computation of the shares of the output wires requires all parties to run an interactive protocol between themselves, where each party sends a message to each other party. Finally, after all gates have been computed, the parties reconstruct the secrets from the shares on the output wires of the circuit in order to obtain output.

3.5 Comparison

In the following, we briefly compare the protocols that were described in this chapter, using the features of the taxonomy of Chapter 2.

Similarities All protocols provide cheap (essentially free, using efficient operations and no communication) computation of linear gates, being XOR gates in Boolean circuit and addition gates in arithmetic circuits. These computations require no communication. For AND gates, both the Yao and GMW protocols protocols use efficient symmetric cryptographic operations, whereas the BGW protocol requires modular addition and multiplication operations. Therefore, circuits should be optimized in the sense that the number of AND or multiplication gates must be minimized. Efficient circuits for standard functionalities are summarized for example in [74, 112].

Differences The main difference is that Yao's protocol has a *constant* number of rounds whereas the GMW, BGW and CCD protocols require the parties to send a message for each layer of AND gates. Therefore, Yao's protocol is preferable for networks with high latency. The main advantage of the GMW protocol is that it is a two-party protocol where *all symmetric cryptographic operations can be precomputed* independently of the function that is evaluated securely and in the online phase only efficient one-time pad operations are needed. Therefore, the GMW protocol is well-suited for computationally weak devices such as smart phones, cf. [43].

b
b

Arithmetic secret sharing (passive security)

Computation on shared data	Outsourced computation on collected data	Data sharing use-case demonstrated successfully in [70, 71] and others. Data collection and analysis demonstrated in [24, 23] and others.
Ready for industrial use		Companies are providing commercial services based on the technology (Cybernetica in Estonia, Partisia in Denmark).
Interpreted programs		Frameworks like FRESCO, SEPIA, Sharemind and VIFF use a program-based approach that can mix public and private computation.
Embedded DSL	Application libraries	SEPIA and VIFF have adopted an embedded DSL approach while Sharemind provides a compiler with a library of secure algorithms.
Real-world applications feasible		Several real-world uses have been documented [24, 23].

Table 3.1: Properties of protocols based on arithmetic secret sharing (passive security).

Yao's two-party protocol (passive and active security)

Computation on shared data	Outsourced computation on collected data	Data is either secret-shared to the two parties, or subsets of the data are known to each party.
Real-world applications		Initial commercial prototypes are being developed.
Compiled circuits		The computed function must be defined as a circuit.
Hand-modified implementation	Embedded DSL	Several academic compilers from a DSL to circuits exist.
Real-world applications feasible		Performance has been highly optimized.

Table 3.2: Properties of Yao's protocol and its variants (passive and active security).

The GMW protocol (passive security)

Computation on shared data	Outsourced computation on collected data	Data is either secret-shared to the two parties, or subsets of the data are known to each party.
Academic prototype		Some academic prototypes have been developed.
Compiled circuits		The computed function must be defined as a circuit.
Hand-modified implementation	Embedded DSL	Several academic compilers from a DSL to circuits exist.
Real-world applications feasible		Performance has been highly optimized (but to a lesser degree than Yao).

Table 3.3: Properties of the GMW protocol (passive security).

Fully homomorphic encryption (passive security)

Outsourced computation on one's own data	Computation on shared data	Outsourced computation on collected data	Data is either secret-shared to the two parties, or subsets of the data are known to each party.
Academic prototype			An academic prototype has been developed at IBM Research.
Compiled circuits			The computed function must be defined as a circuit.
Hand-modified implementation	Embedded DSL		Several academic compilers from a DSL to circuits exist.
Single operations feasible			Performance has been highly optimized (but to a lesser degree than Yao).

Table 3.4: Properties of protocols based on fully homomorphic encryption (passive security).

Chapter 4

The Cut-and-Choose Technique for Security Against Malicious Adversaries

The two main adversary models that have been considered in the literature are semi-honest and malicious (the covert adversary model has received considerably less attention). To recall, a semi-honest adversary follows the protocol specification but attempts to learn more than allowed by inspecting the transcript. In contrast, a malicious adversary can follow any arbitrary (probabilistic polynomial-time) strategy in an attempt to break the security guarantees of the protocol. On one hand, the security guarantees in the semi-honest case are rather weak, but there exist extraordinarily efficient protocols and implementations for this setting. On the other hand, security guarantees against malicious adversaries ensure much stronger security, but the protocols for that setting are considerably less efficient.
In this chapter we will focus on protocols for the two-party setting, where only two participants wish to run a secure computation between themselves.
There have been considerable research efforts in recent years on designing efficient protocols secure against malicious adversaries (where the efficiency can be translated to actual efficient implementations). We list below the main directions that were investigated by this research.

- One approach essentially performs an efficient zero-knowledge proof per gate of the circuit, proving that it was computed correctly [69, 100]. Each of these proofs requires the computation of several exponentiations, and the cost of computing these exponentiations renders this approach inefficient in practice.

- Another approach is based on the MPC-in-the-head technique, which has a very attractive asymptotic overhead of very few symmetric key operations per gate [68, 67, 80]. However, there are no implementations of this approach, and the actual constants affecting the overhead are unknown.

- It is possible to construct a protocol in the random-oracle model with an amortized overhead of $O(s/\log(|C|))$ symmetric-key operations per gate, where s is a security parameter and $|C|$ is the number of gates in the circuit [99, 48]. The number of rounds of this protocol depends on the depth of the circuit, unlike Yao's protocol which runs in a constant number of rounds.

- The protocols of [20, 40, 36] enable secure computation with any number of parties and are secure even if all but one of the parties are corrupt and collaborate with each other. Therefore, these protocols can also be applied in the two-party setting. We describe these protocols in detail in Chapters 5 and 6. The protocols are based on an extensive

preprocessing stage and on the usage of somewhat homomorphic encryption. Here too the number of rounds depends on the depth of the circuit, and therefore performance might be affected if the circuit is of large depth and the communication latency is high.

The cut-and-choose approach. A different approach, on which we will focus in this chapter, is to apply the *cut-and-choose* approach to Yao's protocol. Recall that in the basic Yao protocol one party prepares a garbled circuit, and the other party evaluates that circuit after obtaining, using oblivious transfer, the wire keys corresponding to its inputs. The cut-and-choose approach requires the circuit constructor to construct multiple copies of the circuit. The other party then chooses a random subset of these circuits and asks to verify that they were constructed correctly. If this test goes well, it proceeds to evaluate the remaining circuit. This basic approach must be extended to handle many other issues, such as making sure that checking circuits does not reveal any information about inputs, and, more importantly, verifying that the parties provide consistent inputs to all circuits.

Basics of the cut-and-choose approach. Assume that party P_1 is the circuit constructor, and the P_2 evaluates the circuit. Consider for a moment what happens if party P_1 is malicious. In such a case, it can definitely construct a garbled circuit that computes a function that is different than the one that P_1 and P_2 agreed to compute. To prevent this attack, according to the "cut-and-choose" technique, P_1 first constructs *many* garbled circuits and sends them to P_2. Then, P_2 asks P_1 to "open" half of them (namely, reveal the decryption keys corresponding to these circuits). P_1 opens the requested half, and P_2 checks that they were constructed correctly. If they were, then P_2 evaluates the rest of the circuits and derives the output from them. The idea behind this methodology is that if a malicious P_1 constructs the circuits incorrectly, then P_2 will detect this with high probability. Clearly, this solution solves the problem of P_1 constructing the circuit incorrectly. However, it does not suffice. First, it creates new problems within itself. Most outstandingly, once the parties now evaluate a number of circuits, some mechanism must be employed to make sure that they use the same input when evaluating each circuit (otherwise, as is shown below, an adversarial party could learn more information than allowed). Second, in order to present a proof of security (based on commonly accepted proof method of *simulation*, see [52]) there are additional requirements that are not dealt with by just employing cut-and-choose (e.g., input extraction). Third, the basic folklore description of cut-and-choose is very vague and there are a number of details that are crucial when implementing it. For example, if P_2 evaluates many circuits, then the protocol must specify what P_2 should do if it does not receive the same output in every circuit. If the protocol requires P_2 to abort in this case (because it detected cheating from P_1), then this behavior actually yields a concrete attack in which P_1 can always learn a specified bit of P_2's input. It can be shown that P_2 must take the majority output and proceed, even if it knows that P_1 has attempted to cheat. This is just one example of a subtlety that must be dealt with. Another example relates to the fact that P_1 may be able to construct a circuit that can be opened with two different sets of keys: the first set opens the circuit correctly and the second incorrectly. In such a case, an adversarial P_1 can pass the basic cut-and-choose test by opening the circuits to be checked correctly. However, it can also supply incorrect keys to the circuits to be computed and thus cause the output of the honest party to be incorrect.

4.1 The protocol of [81]

An initial cut-and-choose protocol based on Yao's protocol was presented by Mohassel and Franklin [92], but that protocol lacked a formal proof and was susceptible to a subtle attack (as described in [81]). The first formally proven protocol in this approach was given in [81], and later implemented in [83] (demonstrating a secure computation of the AES function). We describe here the protocol of [81].

4.1.1 The structure of the protocol

There are a number of issues that must be dealt with when attempting to make Yao's protocol secure against malicious adversaries rather than just semi-honest ones. First, the protocol must use an oblivious transfer subprotocol that is secure in the presence of malicious adversaries, as, for example, the protocol of [105]. In addition, the main protocol must be changed.
First and foremost, a malicious circuit constructor P_1 must be forced to construct the garbled circuit correctly so that it indeed computes the desired function. According to the cut-and-choose methodology, P_1 constructs many independent copies of the garbled circuit and sends them to P_2. Party P_2 then asks P_1 to open half of them (chosen randomly). After P_1 does so, and party P_2 checks that the opened circuits are correct, P_2 is convinced that most of the remaining (unopened) garbled circuits are also constructed correctly. (If there are many incorrectly constructed circuits, then with high probability, one of those circuits will be in the set that P_2 asks to open.) The parties can then evaluate the remaining unopened garbled circuits as in the original protocol for semi-honest adversaries, and take the *majority* output-value.

Why the protocol must output the majority output value. The reason for taking the majority value as the output is that the aforementioned test only reveals a single incorrectly constructed circuit with probability 1/2. Therefore, if P_1 generates a single or constant number of "bad" circuits, there is a reasonable chance that it will not be caught. In contrast, there is only an exponentially small probability that the test reveals no corrupt circuit *and* at the same time a majority of the circuits that are not checked are incorrect. Consequently, with overwhelming probability it holds that if the test succeeds and P_2 takes the majority result of the remaining circuits, the result is correct. We remark that the alternative of aborting in case not all the outputs are the same (namely, where cheating is detected) is not secure and actually yields a concrete attack. The attack works as follows. Assume that P_1 is corrupted and that it constructs all of the circuits correctly except for one. The "incorrect circuit" is constructed so that it computes the exclusive-or of the desired function f with the first bit of P_2's input. Now, if P_2 policy is to abort as soon as two outputs are not the same then P_1 knows that P_2 aborts if, and only if, the first bit of its input is 1.
The cut-and-choose technique described above indeed solves the problem of a malicious P_1 constructing incorrect circuits. However, it also generates new problems. The primary problem that arises is that since there are now many circuits being evaluated, we must make sure that both P_1 and P_2 use the same inputs in each circuit; we call these *consistency checks*. Consistency checks are important since if the parties were able to provide different inputs to different copies of the circuit, then they can learn information that is different from the desired output of the function. It is obvious that P_2 can do so, since it observes the outputs of all circuits, but in fact even P_1, who only gets to see the majority output, can learn additional information: Suppose, for example, that the protocol computes n invocations of a circuit computing the inner-product between n bit inputs. A malicious P_2 could provide the inputs $\langle 10\cdots 0\rangle$, $\langle 010\cdots 0\rangle$,...,$\langle 0\cdots 01\rangle$,
b
b

and learn all of P_1's input. If, on the other hand, P_1 is malicious, it could also provide the inputs $\langle 10\cdots 0\rangle$, $\langle 010\cdots 0\rangle$,...,$\langle 0\cdots 01\rangle$. In this case, P_2 sends it the value which is output by the majority of the circuits, and which is equal to the majority value of P_2's input bits.
Another problem that arises with regards to the security proof is that in the proof the simulator must be able to fool P_2 and give it incorrect circuits (even though P_2 runs a cut-and-choose test). This is solved using rather standard techniques, like choosing the circuits to be opened via a coin-tossing protocol (to our knowledge, this issue has gone unnoticed in all previous applications of cut-and-choose to Yao's protocol). Yet another problem is that P_1 might provide corrupt inputs to some of P_2's possible choices in the OT protocols. P_1 might then learn P_2's input based on whether or not P_2 aborts the protocol.
We present a high-level overview of the protocol, and the consistency checks that it performs. The full details of the protocol are in [81].
There are two security parameters. The parameter n is the security parameter for cryptographic schemes (used for bit commitment, encryption, and the oblivious transfer protocol). The parameter s is a statistical security parameter which specifies how many garbled circuits are used. The difference between these parameters is due to the fact that the value of n depends on computational assumptions, whereas the value of s reflects the error probability that is incurred by the cut-and-choose technique and as such is a "statistical" security parameter. Although it is possible to use a single parameter n, it may be possible to take s to be much smaller than n. Typically one can set $n = 128$ and set s so that the cheating probability will be around 2^{-40}. Recall that for simplicity, and in order to reduce the number of parameters, we denote the length of the input by n as well.

The protocol in detail

The protocol has the following structure. Parties P_1 and P_2 have respective inputs x and y, and wish to compute the output $f(x, y)$ for P_2. The parties first decide on a circuit computing f. They change the circuit by replacing each input wire of P_2 by the exclusive-or of s new input wires of P_2. Consequently, the number of input wires of P_2 increases by a factor of s. This is depicted in Figure 4.1. (It was shown in [81] how to reduce the number of new inputs from ns to $O(\max(n, s))$.)
Next, P_1 commits to s different garbled circuits computing f, where s is a statistical security parameter. P_1 also generates additional commitments to the garbled values (also referred to as "keys") corresponding to the input wires of the circuits. These commitments are constructed in a special way, different for P_1 and P_2's inputs, in order to enable consistency checks.
Then, for every input bit of P_2, the parties run a 1-out-of-2 oblivious transfer protocol in which P_2 learns the garbled values of input wires corresponding to its input. P_1 next sends to P_2 all the commitments it generated in the first step, and P_1 and P_2 run a coin-tossing protocol in order to choose a random string that defines which commitments and which garbled circuits will be opened. P_1 opens the garbled circuits and committed input values that were chosen in the previous step. P_2 verifies the correctness of the opened circuits and runs consistency checks based on the decommitted input values.
Finally, P_1 sends to P_2 the garbled values corresponding to P_1's input wires. P_2 runs consistency checks on these values as well, and assuming that all of the checks pass, P_2 evaluates the unopened circuits and takes the majority value as its output.
In more detail, the protocol operates in the following way:
Setup: Parties P_1 and P_2 have respective inputs x and y, and wish to compute the output $f(x, y)$ for P_2.
Protocol:
b
b

0. The parties decide on a circuit computing f. They then change the circuit by replacing each input wire of P_2 by a gate whose input consists of s new input wires of P_2 and whose output is the exclusive-or of these wires (such an s-bit exclusive-or gate can be implemented using $s-1$ two-bit exclusive-or gates). Consequently, the number of input wires of P_2 increases by a factor of s. This is depicted in Figure 4.1.

1. P_1 commits to s different garbled circuits computing f, where s is a statistical security parameter. P_1 also generates additional commitments to the garbled values corresponding to the input wires of the circuits. These commitments are constructed in a special way in order to enable consistency checks.

2. For every input bit of P_2, parties P_1 and P_2 run a 1-out-of-2 oblivious transfer protocol in which P_2 learns the garbled values of input wires corresponding to its input.

3. P_1 sends to P_2 all the commitments of Step 1.

4. P_1 and P_2 run a coin-tossing protocol in order to choose a random string that defines which commitments and which garbled circuits will be opened.

5. P_1 opens the garbled circuits and committed input values that were chosen in the previous step. P_2 verifies the correctness of the opened circuits and runs consistency checks based on the decommitted input values.

6. P_1 sends P_2 the garbled values corresponding to P_1's input wires in the unopened circuits. P_2 runs consistency checks on these values as well.

7. Assuming that all of the checks pass, P_2 evaluates the unopened circuits and takes the majority value as its output.

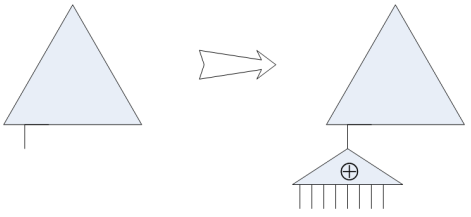


Figure 4.1: Transforming one of P_2's input wires (Step 0 of the protocol).

Checks for Correctness and Consistency

P_1 and P_2 run a number of checks, with the aim of forcing a potentially malicious P_1 to construct the circuits correctly and use the same inputs in (most of) the evaluated circuits. This section describes these checks.

Encoding P_2's input: As mentioned above, a malicious P_1 may provide corrupt input to one of P_2's possible inputs in an OT protocol. If P_2 chooses to learn this input it will not be able to decode the garbled tables which use this value, and it will therefore have to abort. If P_2 chooses to learn the other input associated with this wire then it will not notice that the first input is corrupt. P_1 can therefore learn P_2's input based on whether or not P_2 aborts. This

attack is often referred to as the selective abort attack. (Note that checking that the circuit is well-formed will not help in thwarting this attack, since the attack is based on changing P_1's input to the OT protocol.) The attack is prevented by the parties replacing each input bit of P_2 with s new input bits whose exclusive-or is used instead of the original input (this step was described as Step 0 of the protocol. P_2 therefore has 2^{s-1} ways to encode a 0 input, and 2^{s-1} ways to encode a 1, and given its input it chooses an encoding with uniform probability. The parties then execute the protocol with the new circuit, and P_2 uses oblivious transfer to learn the garbled values of its new inputs. As is shown in [81], if P_1 supplies incorrect values as garbled values that are associated with P_2's input, the probability of P_2 detecting this cheating is *almost independent* (up to a bias of 2^{-s+1}) of P_2's actual input. This is not true if P_2's inputs are not "split" in the way described above. The encoding presented here increases the number of P_2's input bits and, respectively, the number of OTs, from n to ns. In [81] it is demonstrated how to reduce the number of new inputs for P_2 (and thus OTs) to a total of only $O(\max(s, n))$.

An unsatisfactory method for proving consistency of P_1's input: Consider the following idea for forcing P_1 to provide the same input to all circuits. Let s be a security parameter and assume that there are s garbled copies of the circuit. Then, P_1 generates two ordered sets of commitments for every wire of the circuit. Each set contains s commitments: the "0 set" contains commitments to the garbled encodings of 0 for this wire in every circuit, and the "1 set" contains commitments to the garbled encodings of 1 for this wire in every circuit. P_2 receives these commitments from P_1 and then chooses a random subset of the circuits, which will be defined as check-circuits. These circuits will never be evaluated and are used only for checking correctness and consistency. Specifically, P_2 asks P_1 to de-garble all of the check-circuits and to open the values that correspond to the check-circuits in *both* commitment sets. (That is, if circuit i is a check-circuit, then P_1 decommits to both the 0 encoding and 1 encoding of all the input wires in circuit i.) Upon receiving the decommitments, P_2 verifies that all opened commitments from the "0 set" correspond to garbled values of 0, and that a similar property holds for commitments from the "1 set".
It now remains for P_2 to evaluate the remaining circuits. In order to do this, P_1 provides (for each of its input wires) the garbled values that are associated with the wire in all of the remaining circuits. Then, P_1 must prove that all of these values come from the same set, without revealing whether the set that they come from is the "0 set" or the "1 set" (otherwise, P_2 will know P_1's input). In this way, on the one hand, P_2 does not learn the input of P_1, and on the other hand, it is guaranteed that all of the values come from the same set, and so P_1 is forced into using the same input in all circuits. This proof can be carried out using, for example, the proofs of partial knowledge of [32]. However, this would require n proofs, each for s values, thereby incurring $O(ns)$ costly asymmetric operations which we want to avoid.

Proving consistency of P_1's input: P_1 can prove consistency of its inputs without using public-key operations. The proof is based on a cut-and-choose test for the consistency of the commitment sets, which is combined with the cut-and-choose test for the correctness of the circuits. (Note that in the previous proposal, there is only one cut-and-choose test, and it is for the correctness of the circuits.) We start by providing a high level description of the proof of consistency: The proof is based on P_1 constructing, *for each of its input wires*, s pairs of sets of commitments. One set in every pair contains commitments to the 0 values of this wire in *all* circuits, and the other set is the same with respect to 1. The protocol chooses a random subset of these pairs, and a random subset of the circuits, and checks that these sets provide consistent inputs for these circuits. Then the protocol evaluates the *remaining* circuits, and asks P_1 to

open, in each of the *remaining* pairs, and only in one set in every pair, its garbled values for all evaluated circuits. (In this way, P_2 does not learn whether these garbled values correspond to a 0 or to a 1.) In order for the committed sets and circuits to pass P_2's checks, there must be large subsets C and S, of the circuits and commitment sets, respectively, such that every choice of a circuit from C and a commitment set from S results in a circuit and garbled values which compute the desired function f. P_2 accepts the verification stage only if all the circuits and sets it chooses to check are from C and S, respectively. This means that if P_2 does not abort then circuits which are not from C are likely to be a minority of the evaluated circuits, and a similar argument holds for S. Therefore the majority result of the evaluation stage is correct. The exact construction is as follows:

STAGE 1 – COMMITMENTS: P_1 generates s garbled versions of the circuit. Furthermore, it generates commitments to the garbled values of the wires corresponding to P_2's input in each circuit. These commitments are generated in ordered pairs so that the first item in a pair corresponds to the 0 value and the second to the 1 value. The procedure regarding the input bits of P_1 is more complicated (see Figure 4.2 for a diagram explaining this construction). P_1 generates s pairs of sets of committed values for each of its input wires. Specifically, for every input wire i of P_1, it generates s sets of the form $\{W_{i,j}, W'_{i,j}\}_{j=1}^{s}$; we call these commitment sets. Before describing the content of these sets, denote by $k_{i,r}^{b}$ the garbled value that is assigned to the value $b \in \{0,1\}$ in wire i of circuit r. Then, the sets $W_{i,j}$ and $W'_{i,j}$ both contain $s+1$ commitments and are defined as follows. Let $b \in \{0,1\}$ be a random bit, chosen independently for every $\{W_{i,j}, W'_{i,j}\}$ pair. Define $W_{i,j}$ to contain a commitment to b, as well as commitments to the garbled value corresponding to b in wire i in all of the s circuits, and define $W'_{i,j}$ similarly, but with respect to 1–b. In other words, $W_{i,j} = \{\mathsf{com}(b), \mathsf{com}(k_{i,1}^{b}), \ldots, \mathsf{com}(k_{i,s}^{b})\}$ and $W'_{i,j} = \{\mathsf{com}(1\text{–}b), \mathsf{com}(k_{i,1}^{1-b}, \ldots, \mathsf{com}(k_{i,s}^{1-b})\}$. The fact that b is chosen randomly means that with probability 1/2 the set $W_{i,j}$ contains the commitments to values corresponding to 0, and with probability 1/2 it contains the commitments to values corresponding to 1. We stress that in each of the pairs $(W_{i,1}, W'_{i,1}), \ldots, (W_{i,s}, W'_{i,s})$, the values that are committed to are the same. The only difference is that independent randomness is used in each pair for choosing b and constructing the commitments. We denote the first bit committed to in a commitment set as the indicator bit.
After constructing these circuits and commitment sets, P_1 sends to P_2 all of the s garbled circuits (i.e., the garbled gate-tables and output-tables, but *not* the garbled values corresponding to the input wires), and all the commitment sets. Note that if P_1's input is of length n, then there are sn pairs of commitment *sets*; and a total of $sn(2s+2) = O(s^2 n)$ commitments.

STAGE 2 – CHALLENGE: Two random strings $\rho, \rho' \in \{0,1\}^s$ are chosen and sent to P_1 (in the actual protocol, these strings are determined via a simple coin-tossing protocol). The string ρ is a challenge indicating which garbled circuits to open, and the string ρ' is a challenge indicating which commitment sets to open. We call the opened circuits check-circuits and the unopened ones evaluation-circuits. Likewise, we call the opened sets check-sets and the unopened ones evaluation-sets. A circuit (resp., commitment set) is defined to be a check-circuit (resp., check-set) if the corresponding bit in ρ (resp., ρ') equals 1; otherwise, it is defined to be an evaluation-circuit (resp., evaluation-set).

STAGE 3 – OPENING: First, P_1 opens all the commitments corresponding to P_2's input wires in all of the check-circuits. Second, in all of the *check-sets* P_1 opens the commitments that correspond to *check-circuits*. That is, if circuit r is a check circuit, then P_1 decommits to all of

b
b

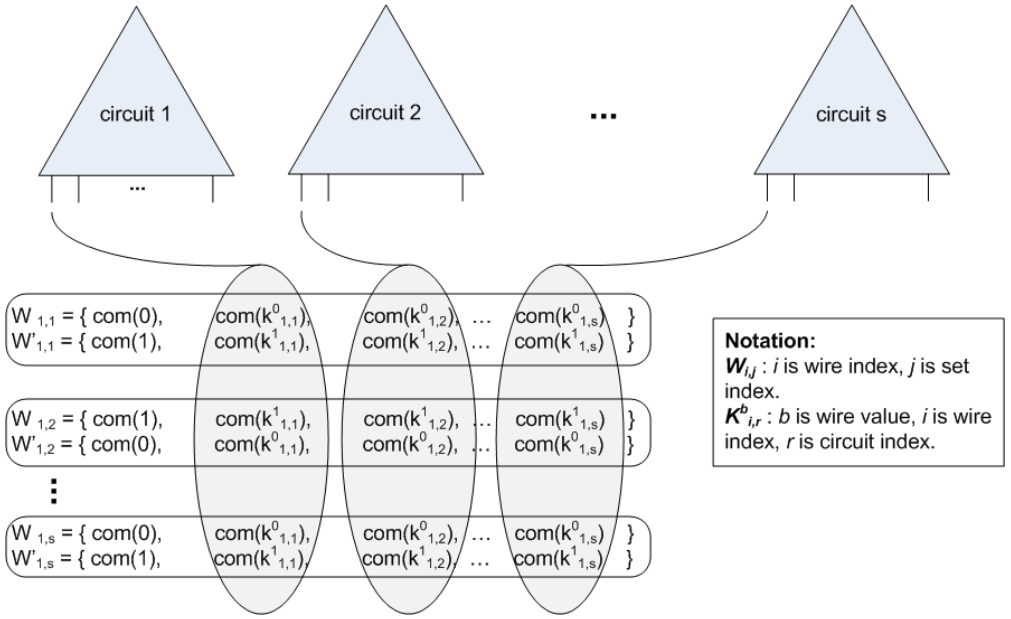


Figure 4.2: The commitment sets corresponding to P_1's **first** input wire.

the values $\mathsf{com}(k^0_{i,r}), \mathsf{com}(k^1_{i,r})$ in check-sets, where i is any of P_1's input bits. Finally, for every check-set, P_1 opens the commitment to the indicator bit, the initial value in each of the sets $W_{i,j}, W'_{i,j}$. See Figure 4.3 for a diagram in which the values which are opened are highlighted (the diagram refers to only one of P_1's input wires in the circuit).

Stage 4 – Verification: In this step, P_2 verifies that all of the check-circuits were correctly constructed. In addition, it verifies that, with regards to P_1's inputs, all the opened commitments in sets whose first item is a commitment to 0 are to garbled encodings of 0; likewise for 1. These checks are carried out as follows. First, in all of the check-circuits, P_2 receives the decommitments to the garbled values corresponding to its own input, and by the order of the commitments P_2 knows which value corresponds to 0 and which value corresponds to 1. Second, for every check-circuit, P_2 receives decommitments to the garbled input values of P_1 in all the check-sets, along with a bit indicating whether these garbled values correspond to 0 or to 1. It first checks that for every wire, the garbled values of 0 (resp., of 1) are all equal. Then, the above decommitments enable the complete opening of the garbled circuits (i.e., the decryption of all of the garbled tables). Once this has been carried out, it is possible to simply check that the check-circuits are all correctly constructed. Namely, that they agree with a specific and agreed-upon circuit computing f.

Stage 5 – Evaluation and Verification: P_1 reveals the garbled values corresponding to its input: If i is a wire that corresponds to a bit of P_1's input and r is an evaluation-circuit, then P_1 decommits to the commitments $k^b_{i,r}$ in all of the *evaluation-sets*, where b is the value of its input bit. This is depicted in Figure 4.4. Finally, P_2 verifies that (1) for every input wire, all the opened commitments that were opened in evaluation-sets contain the same garbled value, and (2) for every i, j P_1 opened commitments of evaluated circuits in exactly one of $W_{i,j}$ or $W'_{i,j}$. If these checks pass, it continues to evaluate the circuit.

b
b

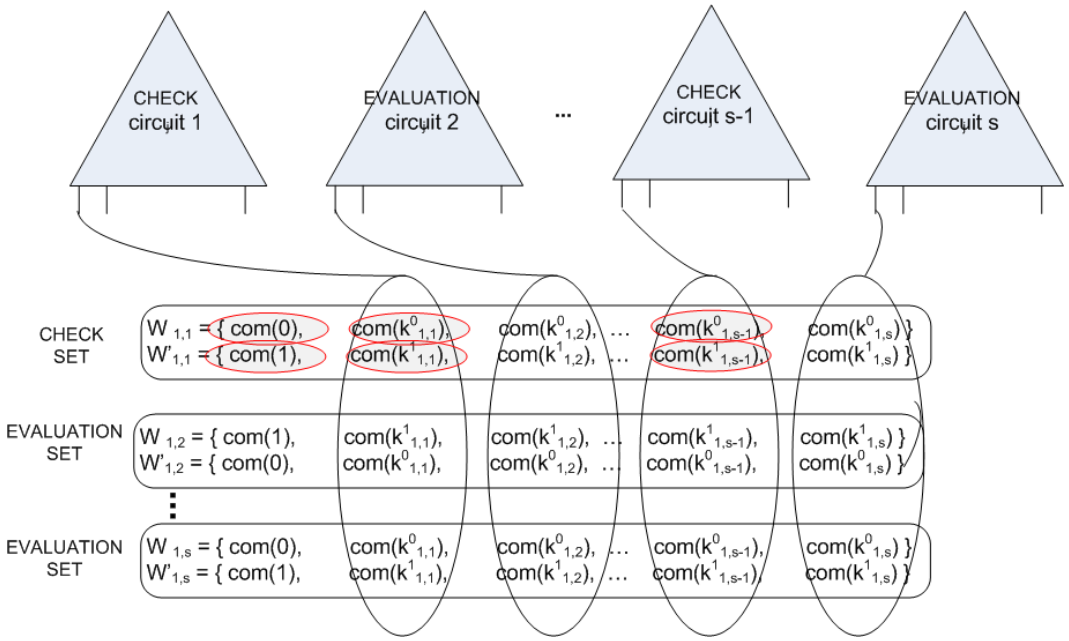


Figure 4.3: In every check-set, the commitment to the indicator bit, and the commitments corresponding to check-circuits are all opened.

Intuition. Having described the mechanism for checking consistency, we now provide some intuition as to why it is correct. A simple cut-and-choose check verifies that most of the evaluated circuits are correctly constructed. The main remaining issue is ensuring that P_1's inputs to most circuits are consistent. If P_1 wants to provide different inputs to a certain wire in two circuits, then *all* the $W_{i,j}$ (or $W'_{i,j}$) sets it opens in *evaluation-sets* must contain a commitment to 0 in the first circuit and a commitment to 1 in the other circuit. However, if any of these sets is chosen to be checked, and the circuits are among the checked circuits, then P_2 aborts. This means that if P_1 attempts to provide different inputs to two circuits and they are checked, it is almost surely caught. Now, since P_2 outputs the majority output of the evaluated circuits, the result is affected by P_1 providing different inputs only if these inputs affect a constant fraction of the circuits. But since all of these circuits must not be checked, P_1's probability of success is exponentially small in s.

Efficiency

The computation overhead is dominated by the oblivious transfers, as all other primitives are implemented using symmetric key operations. Each input bit of P_2 is replaced in the protocol by s new input bits and therefore $O(ns)$ OTs are required. In [81] it is shown how to use only $O(\max(n,s))$ new input bits, and consequently the number of OTs is reduced to $O(\max(n,s))$ (namely $O(1)$ OTs per input bit, assuming $n = \Omega(s)$).
The communication overhead of the protocol is dominated by sending s copies of the garbled circuit, and $2s(s+1)$ commitments for each of the n inputs of P_1. In the protocol, the original circuit C^0 is modified by replacing each of the n original input bits of P_2 with the exclusive-or of s of the new input bits, and therefore the size of the evaluated circuit C is $|C| = |C^0| + O(ns)$ gates. The communication overhead is therefore $O(s|C| + s^2 n) = O(s(|C^0| + ns) + s^2 n) = O(s|C^0| + s^2 n)$ times the length of the secret-keys (and ciphertexts) used to construct the garbled circuit.

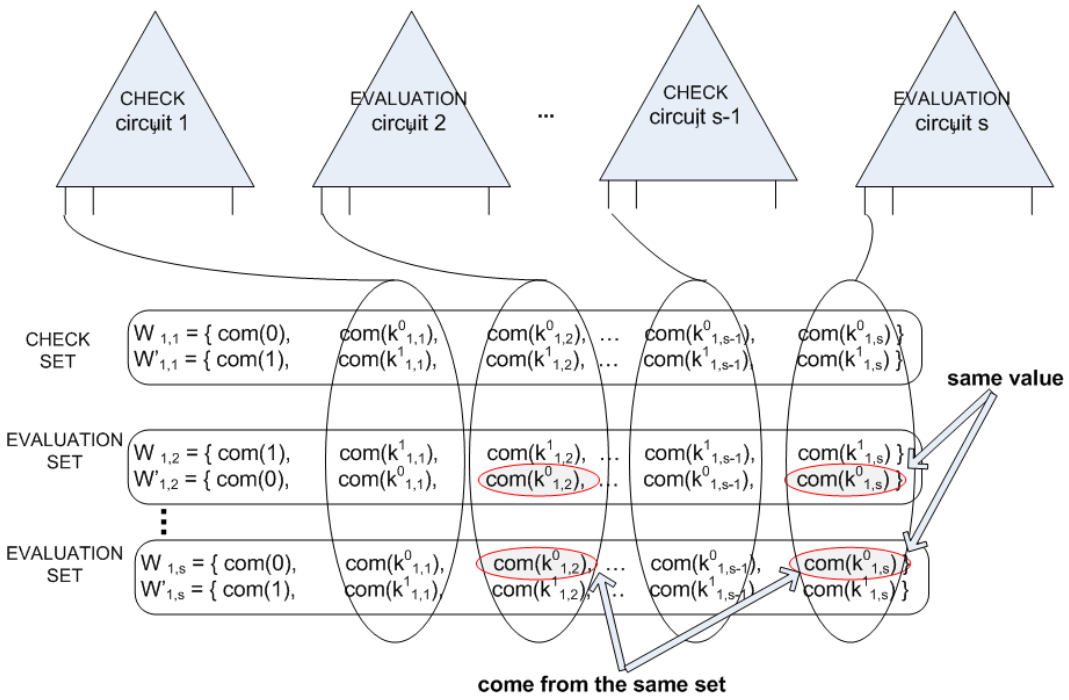


Figure 4.4: P_1 opens in the evaluation-sets, the commitments that correspond to its input. In every evaluation-set these commitments come from the same item in the pair.

4.2 Improved Efficiency Using Cut-and-Choose Oblivious Transfer

Although intuitively appealing, the cut-and-choose approach introduces a number of difficulties which significantly affect the efficiency of the protocol of [81]. First, since the parties need to evaluate $s/2$ circuits rather than one, there needs to be a mechanism to ensure that they use the same input in all evaluations (the solution for this for P_2's inputs is easy, but for P_1's inputs turned out to be hard). The mechanism used in [81] required constructing and sending $s^2\ell$ commitments. In the implementation by [107], they set $s = 160$ (to have a cheating probability of $2^{-s/4} = 2^{-40}$), and $\ell = 128$. Thus, the overhead due to these consistency proofs is the computation and transmission of $3,276,800$ commitments. Another problem that the protocol of [81] had to solve was that a malicious P_1 could input an incorrect key into one of the oblivious transfers used for P_2 to obtain the keys associated with its input wires in the garbled circuit. For example, it can set all the keys associated with 0 for P_2's first input bit to be garbage, thereby making it impossible for P_2 to decrypt any circuit if its first input bit indeed equals 0. In contrast, P_1 can make all of the other keys be correct. In this case, P_1 is able to learn P_2's first input bit, merely by whether P_2 obtains an output or not. The important observation is that the checks on the garbled circuit carried out by P_2 do not detect this attack because there is a separation between the cut-and-choose checks and the oblivious transfer. The solution to this problem in [81] requires making the circuit larger and significantly increasing the size of the inputs by replacing each input bit with the exclusive-or of multiple random input bits. Finally, the analysis of [81] yields an error of $2^{-s/17}$. Thus, in order to obtain an error level of 2^{-40} the parties need to exchange 680 circuits. We remark that it has been conjectured in [107] that the true error level of the protocol is $2^{-s/4}$; however, this has not been proved.

4.2.1 The new protocol

The new protocol, described in [82], solves the aforementioned problems, and reduces the error probability to $2^{-0.311s}$ (thus for an error of 2^{-40} it suffices to send only 128 circuits). The new protocol moderately increases the number of exponentiations, while reducing the number of circuits, completely removing the commitments, and also removing the need to increase the size of the inputs. The price for these improvements is that the new protocol relies heavily on the decisional Diffie-Hellman (DDH) assumption, while the protocol of [81] used general assumptions only. We now proceed to describe the main techniques used by the new protocol:

- The solution for ensuring consistency of P_1's inputs is to have P_1 determine the keys associated with its own input bits via a Diffie-Hellman pseudorandom synthesizer [97]. That is, P_1 chooses values $g^{a_1^0}, g^{a_1^1}, \ldots, g^{a_\ell^0}, g^{a_\ell^1}$ and $g^{r_1}, \ldots, g^{r_s}$ and then sets the keys associated with its ith input bit in the jth circuit to be $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$. Given all of the $\{g^{a_i^0}, g^{a_i^1}, g^{r_j}\}$ values and any subset of keys of P_1's input generated in this way, the remaining keys associated with its input are pseudo-random by the DDH assumption. Furthermore, it is possible for P_1 to efficiently prove that it is using the same input in all circuits when the keys have this nice structure.

- The reason that the inputs and circuits were needed to be made larger in [81] is due to the fact that the cut-and-choose circuit checks were separated from the oblivious transfer. In order to solve this problem, the protocol in [82] introduced a new primitive called *cut-and-choose oblivious transfer*. This is an ordinary oblivious transfer with the sender inputting many pairs $(x_1^0, x_1^1), \ldots, (x_s^0, x_s^1)$, and the receiver inputting bits $\sigma_1, \ldots, \sigma_s$. However, the receiver also inputs a set $J \subset [s]$ of size exactly $s/2$. Then, the receiver obtains $x_i^{\sigma_i}$ for every i (as in a regular oblivious transfer) along with both values (x_j^0, x_j^1) for every $j \in J$. The use of this primitive in the new protocol intertwines the oblivious transfer and the circuit checks and solves the aforementioned problem. It was also shown in [82] how to implement this primitive in a highly efficient way, under the DDH assumption.

4.3 Improved Amortized Overhead

Protocols that use the cut-and-choose technique in a straightforward manner require approximately $3s$ garbled circuits to obtain a bound of 2^{-s} on the cheating probability by the adversary. Recently, Lindell [79] showed that by executing another light secure two-party protocol, the number of garbled circuits can be reduced to s, which seems optimal given that 2^{-s} is the probability that a cut is bad and goes unnoticed by the circuit evaluator (meaning that all the check circuits are good and all the unchecked circuits are bad). The number of garbled circuits affects both computation time and communication. In most applications, when $|C|$ is large, sending s garbled circuits becomes the dominant overhead. (For example, [63] showed a prototype for garbling a circuit on GPUs, which generates more than 30 million gates per second. The communication size of this number of gates is about 15GB, and transferring 15GB of data most likely takes much more than a second.) Thus, further reducing the number of circuits is an important goal.
New results by [84] and concurrently by [61] reduce the number of circuits in cut-and-choose in the multiple-execution setting, where a pair of parties runs many executions of the protocol. This enables the parties to amortize the cost of the check-circuits over many executions.

Amortizing checks over multiple executions. In a single execution, P_1 constructs s circuits and asks P_2 to open a random subset of these circuits. If P_1 makes some of these circuits incorrect and some correct, then it can always succeed in cheating if P_2 opens all of the good circuits and the remaining are all bad. Since this bad event can happen with probability 2^{-s}, this approach to cut-and-choose seems to have a limitation of s circuits for 2^{-s} error. However, consider now the case that the parties wish to run N executions. One possibility is to simply prepare Ns circuits and work as in the single execution case. Alternatively, P_1 can prepare cN circuits (for some constant c); then P_2 can ask to open a subset of the circuits; finally, P_2 randomly assigns the remaining circuits to N small buckets of size B (where one bucket is used for every execution). The protocol suggested in [84], which is based on [79], has the property that P_1 can cheat only if there is a bucket in which *all* of the circuits are bad. The probability of this happening when not too many bad circuits are constructed by P_1 is very small, but if P_1 does construct many bad circuits then it will be caught even if a relatively small subset of circuits is checked. This idea is very powerful. Asymptotically only $O(s/\log N)$ circuits are needed on average per execution. The work in [84] contains a detailed analysis of the precise overhead of this scheme.

Chapter 5

Protocols with Preprocessing

The goal of this chapter is to give a coherent introduction and overview of some recent protocols in the so called *preprocessing model* (also known as the *trusted dealer* or *commodity based* model). We will focus on protocols that are secure in the *dishonest majority setting*, against *active adversaries*, and we are only going to consider *static corruptions*. We stress that due to space constraints, many important details (including proofs of security) are missing, and we refer the reader to the original papers for a more detailed exposition.
In the preprocessing model we assume that, before the protocol starts, the parties have access to some correlated randomness. That is, before the protocol starts a joint sample from some distribution $(r_1, \ldots, r_n) \leftarrow \mathcal{D}^\pi$ is taken and the sample r_i is given to party P_i. While each party only receives its own random string from this sample, these random strings are correlated as specified by the joint distribution. Note that the distribution is parametrized by the protocol, in the sense that the different protocols described in this chapter require the trusted dealer to sample from different distributions.
From a theoretical point of view, the correlated randomness model is interesting because it can be used to circumvent impossibility results for the plain model such as the impossibility of secure computation with unconditional security for dishonest majority. While constructing protocols with unconditional security might not be a very important goal in itself since a complete protocol will likely need some cryptographic assumptions anyway, even if just to realize private and authenticated point-to-point channels or to generate pseudo-randomness, there is another reason to be interested in protocols with unconditional security, namely *efficiency*: protocols relying on cryptographic assumptions are usually less efficient because they require to perform cryptographic operations which are orders of magnitude slower (especially when using public key cryptography) than the kind of operations that the protocols presented in this chapter will perform (field operations, XORs).

Instantiating the Trusted Dealer: In most practical applications we do not want to rely on a single, trusted dealer. In practice, the trusted dealer model can be instantiated in several different ways, including:

- MPC WITH PREPROCESSING. It is often the case that parties can use idle times before they have any input to run a secure "offline protocol" for generating and storing correlated randomness. The correlated randomness is later consumed by an "online protocol" which is executed once the inputs become available. This paradigm for MPC (i.e., for secure multi-party protocols) is particularly useful when it is important that the outputs are known shortly after the inputs are (i.e., for low-latency computation). Think of an electronic election or an auction: all the involved parties know well in advance that a

certain computation will take place at some point in the future. Therefore the parties can run a (more expensive) cryptographic protocol and then, when the inputs are known, run a more efficient protocol. We sometimes call the preprocessing phase the *offline phase*, while we call the part of the protocol described in this chapter the *online phase*. In the Chapter 6 several protocols for the offline phase will be presented.

- COMMODITY-BASED MPC. In the setting of commodity-based cryptography [14], the parties can "purchase" correlated randomness from one or more external servers. Security in this model is guaranteed as long as at most t of the servers are corrupted, for some specified threshold t, where corrupted servers may potentially collude with the parties. In contrast to the obvious solutions of employing a server as a trusted party or running an MPC protocol among the servers, the servers are only used during an offline phase before the inputs are known, and do not need to be aware of the existence of each other.

- HONEST-MAJORITY MPC. Recent large-scale practical applications of MPC [24, 22] employed three servers and assumed that at most one of these servers is corrupted by a semi-honest adversary. Protocols in the correlated randomness model can be translated into protocols in this 3-server model by simply letting one server generate the correlated randomness for the other two.

5.1 Warm-up: One Time Truth Table

To start, we will present a very simple protocol for MPC in the preprocessing model which will allow us to describe some of the issues that we will encounter later as well. The protocol is known as *one-time truth table* (OTTT) and was presented in [65]. The OTTT protocol is optimal in terms of communication complexity between the parties, but the size of the preprocessing is exponential in the input size, and can therefore only be used for functions with small domains. Still, it shows the power of the trusted dealer model and due to its simplicity can be explained very easily to an inexperienced audience (for instance an undergrad class).
Let f be a function that takes n inputs from some domain Z_m (for some integer m) and outputs a single element in Z_m. We will represent f with a truth table $T \in (Z_m)^{m^n}$.

The offline phase: The trusted dealer chooses some *random shifts* $s_1, \ldots, s_n \in Z_m$ and computes a permuted truth table $PT \in (Z_m)^{m^n}$ s.t.[1]

$$PT[i_1 + s_1, \ldots, i_n + s_n] = T[i_1, \ldots, i_n]$$

Then the trusted dealer chooses random hypermatrices $M_2, \ldots, M_n \in (Z_m)^{m^n}$ and defines

$$M_1 = PT - \sum_{i=2}^{n} M_i$$

Finally the trusted dealer outputs (s_i, M_i) to each party P_i.

The online phase: Each party P_i has an input $x_i \in Z_m$. The protocol proceeds as follows:

1. Each party P_i sends $u_i = x_i + s_i$ to all other parties.
2. Each party sends $z_i = M_i[u_1, \ldots, u_n]$ to all other parties.

[1] All additions + in the description of the protocol are modulo m.

3. All parties output $z = \sum_{i=1}^{n} z_i$.

The protocol is correct (by inspection) and can be easily shown to be secure against *passive corruptions*: the values u_i can be seen as one-time pad encryptions of the inputs using the random keys provided by the trusted dealer, and the shares of the output are uniformly random values under the constraint that they sum up to the right output (and can therefore be easily simulated).
It is also easy to see that the protocol is not secure against *active adversaries*: a corrupted party P_i^* can change its share of the output therefore leading to an incorrect output for all honest parties.[2]
To fix this issue, we need a way of making sure that a corrupted party cannot lie about the share she received from the trusted dealer. To fix this we let the trusted dealer compute MACs (*message authentication codes*) on all the shares it give to the parties.

Message Authentication Code. A MAC scheme has three algorithms $(\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ where Gen produces a MAC key k, which can be used to compute tags on messages as $t = \mathsf{Tag}_k(x)$. Finally the verification input $\mathsf{Ver}_k(t, x)$ outputs *accept* if t is a valid tag on x with key k or *reject* otherwise. Security is defined as a game between a challenger C and an adversary A: C samples a key k, then A is allowed q queries where he can get tags $t_1, \ldots, t_q$ on messages $x_1, \ldots, x_q$ of his choice. Now the adversary outputs (t', x'). We say that a MAC scheme is $(q, \epsilon)-$secure if no (polynomial time) adversary which is allowed q queries can output a pair (t', x') with $x' \neq x_i$ for all i such that t' is a valid MAC on x' with probability greater than ϵ.
We can now enhance the OTTT protocol in the following way: for each pair of parties P_i and P_j, the trusted dealer samples random MAC keys, computes tags on all entries of M_i and finally outputs the tags to P_i and the keys to P_j. Now the online phase is modified and then P_i is asked to send an entry of M_i to P_j it will also send a tag along. In this way P_j can, using the MAC key, verify that the value it received from P_i is the same that P_i received from the trusted dealer (since it is unfeasible for P_i to forge a MAC).
The modified protocol is now secure against active adversaries, and offers the same security level as the MAC scheme. Remember that when proving security against active adversaries the simulator needs to "extract" the input of the corrupted parties. This can be simply done since the simulator knows the value r_i for all corrupted parties, and can therefore compute $x_i' = u_i - r_i$. Now the simulator inputs this value to the ideal functionality and receives the right output z. Since the simulator is emulating the trusted dealer, the simulator knows all the MAC keys and can easily adjust the shares (and MACs) of the honest parties to make sure they sum up to z. At the same time, the adversary cannot lie about her share due to the security of the MAC scheme.
While most MAC schemes require cryptographic assumptions, note that in this application we use each MAC key only once, and it is therefore possible to use *information theoretic one-time MACs*.

One-Time MAC. The simplest MAC scheme that we can use in the above construction is probably the following:

$k \leftarrow \mathsf{Gen}(1^\kappa)$**:** Sample $k = (\alpha, \beta) \leftarrow (Z_p)^2$ for a prime p such that $\log_2 p > \kappa$.

[2] A corrupted party could also send different shares to different parties, thus leading to a situation where different honest parties have different outputs. We will not consider this in the following, and we will assume the presence of a broadcast channel between the parties.

$t \leftarrow \mathsf{Tag}_k(x)$**:** Compute a tag t on a value x with key k as $t = \alpha \cdot x + \beta$.

$\{\top, \bot\} \leftarrow \mathsf{Ver}_k(t, x)$**:** Output accept if $t' = \alpha \cdot x + \beta$ or reject otherwise.

It is clear that the MAC is *one-time secure*: an adversary that sees a pair (t, x) and manages to produce a different pair (t', x') with $x' \neq x$ can be trivially used to compute the key (α, β) (since p is prime) and this happens only with probability $O(1/p)$.
In the same way, an adversary who sees the MACs on two values using the same key can trivially compute the key and therefore compute the MAC on any other value of her choice.

5.2 The Arithmetic Black-Box

The protocol from the previous section can only be used for "small" functions, since the the samples that each party receive from the trusted dealer are exponential in the input size. Moreover, in the previous protocol the preprocessing phase must know which functions the parties want to compute.
A different and often more convenient model of computation is what we will call the *arithmetic black-box* (ABB) [39]: in this chapter we will describe protocols that allow the parties to simulate the existence of a trusted box that allows the parties to perform various arithmetic operations over some field Z_p, with p prime. Traditional choices of p are $p = 2$ (and therefore we talk about Boolean computation), or large p such that $1/p$ is negligible. Sometimes it is useful to choose a p that is larger than any internal value in the computation, so that it is possible to emulate integer computation inside the modulo p (and therefore "forget" about modular reduction).
The arithmetic black-box allows parties to perform the following commands:

Input: When all parties input $(\text{INPUT}, P_i, id_x)$ and party P_i inputs $(\text{INPUT}, P_i, id_x, x)$ the box stores the pair (id_x, x) (or aborts if that id was already present in memory).

Output: When all parties input $(\text{OUTPUT}, id_x, P_i)$, the box finds the pair (id_x, x) in memory and outputs the value x to P_i.

Add: When all parties input $(\text{ADD}, id_x, id_y, id_z)$, if there are pairs (id_x, x) and (id_y, y) in memory and id_z is an unused id, the box stores $(id_z, x + y)$.

Multiply: When all parties input $(\text{MUL}, id_x, id_y, id_z)$, if there are pairs (id_x, x) and (id_y, y) in memory and id_z is an unused id, the box stores $(id_z, x \cdot y)$.

This is the most basic arithmetic black-box. One could enhance it with other commands, such as generation of random values, multiplication by scalar or more, but since all these commands can be realized efficiently using the basic-box we do not include them here.

5.2.1 Implementing the Arithmetic Black-Box With Passive Security

We present here a simple protocol that implements the arithmetic black-box with passive security. This is very similar to the GMW protocol presented in Section 3.3. The protocol is in the trusted dealer model, and the only information that the trusted dealer needs to know about the way the arithmetic black-box is used is an upper bound on the number of inputs I and multiplications M which will be performed.
The protocol works as follows:

Notation: To emulate the storage of values in the arithmetic black box, we will use additive secret sharing between parties. That is, a value $x \in Z_p$ is stored by having all parties P_i store random shares $x_i \in Z_p$ under the condition that $\sum_i x_i = x$. We write $[x]$ (*x in a box*) to denote this situation.[3]

Note that it is possible to interpret a publicly known value a as a stored value by simply letting P_1 set her share $a_1 = a$ and all other parties P_j set their shares $a_j = 0$.

The Offline Phase: The trusted dealer, with parameters I and M (I being the upper bound on the number of input commands, M an upper bound on the number of multiplication operations) provides the parties with two kind of correlated randomness.

Random Value: The trusted dealer, for all $i \in \{1, \ldots, I\}$

1. Samples random $r^i \in Z_p$, and random $(r_1^i, \ldots, r_n^i) \in (Z_p)^n$ under the condition that $\sum_j r_j^i = r^i$;
2. Output $[r^i]$ (that is, output to each party P_j her share r_j^i of r^i);

Random Multiplication: The trusted dealer, for all $i \in \{1, \ldots, M\}$ generates three random values $[a^i], [b^i], [c^i]$ as described above, with the extra requirement that $c^i = a^i \cdot b_i$;

The Online Phase: The parties $P_1, \ldots, P_n$, after receiving their shares of random values and random multiplications from the trusted dealer, emulate the commands of the arithmetic black-box in the following way (note that the commands are presented in a different order here, since some will depend on previously defined ones):

Output: If party P_i is supposed to learn a secret value $[x]$, all other parties P_j with $j \neq i$ send their shares x_j to Alice which outputs $x = \sum_i x_i$ We write $x \leftarrow \mathsf{OutputTo}(P_i, [x])$ for short. We write also $x \leftarrow \mathsf{OutputAll}([x])$ as a shortcut for the case where everyone is supposed to learn the output;

Addition: If the parties hold two values $[x]$ and $[y]$ and want to compute a new representation $[z]$ such that $z = x + y$, each party P_i sets her new share $z_i = x_i + y_i$. It is trivial to check that now

$$z = \sum z_i = \sum (x_i + y_i) = \sum x_i + \sum y = x + y$$

In the same way, it is possible to compute $[z]$ such that $z = a \cdot x + b \cdot y + c$ for any publicly known (a, b, c) by applying local transformation on the shares.

To stress that no communication is involved in performing this operations, we write

$$[z] = a[x] + b[y] + c$$

as a shortcut.

Input: [4] If a party P_i is supposed to give an input x to the arithmetic black-box, the parties:

[3] Note that we are abusing of notation, since when we write x we sometimes refer to the name of the variable (the l-value of x) while sometimes we refer to the value associated to x (the r-value of x). When notation is unclear, we will use $n(x)$ to refer to the name of x and $v(x)$ for the value associated to x, and therefore to be correct we would write $[n(x)]$.

[4] The input phase as described here is an overkill if one wants passive security only, but since this section is only a warm-up to the following sections describing actively secure solutions, we opted for this solution that is going to work also for active security.

1. Find the first random sample $[r^\ell]$ which was not used yet;
2. Run $r^\ell \leftarrow \mathsf{OutputTo}(P_i, [r^\ell])$;
3. P_i computes $d = x - r^\ell$ and sends d to all other parties P_j;
4. All parties (locally) compute their shares of

$$[x] = [r^\ell] + d$$

We write $[x] \leftarrow \mathsf{Input}(P_i, x)$ for short.

Multiplication: If parties hold two values $[x]$ and $[y]$ and want to compute a new new representation $[z]$ such that $z = x \cdot y$, the parties:

1. Find the first random multiplication $[a^\ell], [b^\ell], [c^\ell]$ which was not used yet;
2. Run $d \leftarrow \mathsf{OutputAll}([x] - [a^\ell])$;
3. Run $e \leftarrow \mathsf{OutputAll}([y] - [b^\ell])$;
4. All parties (locally) compute their shares of

$$[z] = e \cdot [a^\ell] + d \cdot [b^\ell] + [c^\ell] + d \cdot e$$

Analysis, Correctness: The protocol is correct by inspection. The most interesting part is how multiplications gates are evaluated using a random, preprocessed multiplication. This trick is due to Beaver [13] and correctness can be easily checked:

$$ea + db + c + de = (ay - ab) + (bx - ab) + (ab) + (xy - ay - bx + ab) = xy$$

Analysis, (Passive) Security: The protocol is secure against passive corruptions: the command $\mathsf{OutputTo}$ leaks exactly the information which is supposed to, since the shares received by party P_i are uniformly random conditioned to the fact that they add up to the correct value (and can therefore be easily simulated); the command Add clearly does not leak any information since it does not involve any communication; in the last two commands, Input and Mul, we exploit the fact that the values obtained by the trusted dealer are uniformly random in the view of up to $n-1$ corrupted parties, and therefore the distribution of the revealed values d (in Input) and d, e (in Mul) is independent of the actual inputs and internal values stores in the black box.

Analysis, (Active) Security: The protocol described in this section is not secure against active adversaries. The reason for this is that during the $\mathsf{OutputTo}$ command (similarly to what we have already seen in the one-time truth table protocol), a party can change its share x_i (even as a function of the shares of the other $n-1$ parties) and can therefore fully control the output of the computation.
To address this problem, we need to enhance the representation of shared value $[x]$ with some MACs such that:

1. Parties cannot lie about their shares during the output phase;
2. It is still possible to compute linear combinations of representations without any communication;

In the next sections we will see a number of different MAC schemes that achieve both properties and therefore lead to fully-secure implementations of the arithmetic black-box. Crucially, we will only need to describe: 1) the semantic of the representation $[x]$ 2) how to perform an OutputTo command and finally 3) the semantic of the addition between shared values $[x] + [y]$. In particular, we will not have to modify how the Input and Mul command are implemented. To do so, remember that in the case of active security we need to be able to extract one parties input (this is not only an artefact of our security model, but is also ensuring concrete security properties such as input independence): now, let's say that a corrupted party P_i is giving input to the ABB: the simulator (which in the proof emulates the trusted dealer as well, and therefore knows all the values r^ℓ) can extract the input of a corrupted party simply by computing $x' = d - r^\ell$. Now, the simulator can input this to the ABB and run the protocol with P_i replacing all inputs of the honest parties (which are unknown to the simulator) with uniformly random values. Finally when an OutputTo command needs to be simulated, the simulator receives the value from the ABB and will adjust the shares of the simulated honest parties to "hit" the right output. Here it is crucial that the corrupted party cannot lie about its share during the OutputTo phase (due to the security of the MAC schemes which will be presented later), while at the same time the simulator can do this (similarly to what was done in the OTTT protocol) since the simulator is also emulating the trusted dealer and therefore knows all the MAC keys.
This "informal proof" of security shows that in the next sections we only need to focus on describing shared representations which support linear operations (without communication) and where we have the guarantee that during the reconstruction of the secret a corrupted party can only make the honest parties output the right value (or abort the protocol).

5.3 BeDOZA and TinyOT Online Phase

In this section we describe a protocol that generalizes the way secret values are shared in the protocols known as BeDOZa [20] and TinyOT [99]. The first protocol, BeDOZa, implements an ABB over a field Z_p for a large prime p, while TinyOT implements a Boolean Arithmetic Black-Box over Z_2. Here we will abstract this difference away and describe both protocols at once. To do so, we will use also a parameter k which must be such that p^{-k} is negligible. Therefore in BeDOZa $k = 1$ while in TinyOT $k > 60$.

Shared Value: In BeDOZa and TinyOT, to emulate the values in the arithmetic black box we will enhance the representation of the passive secure protocol with some MACs. First, for each pair of parties P_i, P_j, with $i \neq j$, party P_i owns a "global MAC key" $\alpha_{i,j} \in (Z_p)^k$. Then, for each shared value $[x]$ party P_i owns

1. A random share $x_i \in Z_p$ (under the condition that $\sum x_i = x$);
2. A "local MAC key" $\beta^x_{i,j} \in (Z_p)^k$ for each other party P_j, $j \neq i$;
3. A tag $t^x_{i,j} = x_i \cdot \alpha_{j,i} + \beta_{j,i}$ (note that party P_i has a tag computed on its own share x_i using the key owned by the other party P_j);

Note that keys (both global and local) and tags are k dimensional vectors, and the product in step 3 between x_i and $\alpha_{j,i}$ is a scalar product (that is, all entries in $\alpha_{j,i}$ are multiplied by x_i).

Output: When a party P_i is supposed to learn a value $[x]$

1. All other parties P_j, $j \neq i$ send the pair $(x_j, t^x_{i,j})$.
2. For all $j \neq i$, party P_i checks if $t^x_{i,j} = x_i \cdot \alpha_{i,j} + \beta^x_{i,j}$ and aborts if any check fails;
3. If all checks pass, P_i outputs $x = \sum x_i$.

Addition: If the parties hold two values $[x]$ and $[y]$ and want to compute a new representation $[z]$ such that $z = x + y$ each party P_i

1. Computes $z_i = x_i + y_i$;
2. Computes $t^z_{i,j} = t^x_{i,j} + t^y_{i,j}$ for all $j \neq i$;
3. Computes $\beta^z_{i,j} = \beta^x_{i,j} + \beta^y_{i,j}$ for all $j \neq i$;

Analysis: It is easy to see that it is possible to compute linear combinations of shares using only local computation. In fact, for all i, j

$$t^z_{i,j} = t^x_{i,j} + t^y_{i,j} = (x_i \cdot \alpha_{j,i} + \beta^x_{j,i}) + (y_i \cdot \alpha_{j,i} + \beta^y_{j,i}) = z_i \cdot \alpha_{j,i} + \beta^z_{j,i}$$

as it is supposed to be. The security of the OutputTo command follows from the fact that the MAC scheme used is secure: note that the MAC scheme used here is *almost* the same as the one-time MAC scheme introduced before, except that here the first component of the key α is kept constant over different authentications. This is crucial to allow to perform linear operations on the MACs, but it does not have any impact on security, since when the adversary sees q tags there are still $q + 1$ unknown values in $(Z_p)^k$ (q local keys β and one global key α).

Lazy MAC Check: In the passive secure protocol each party sends, for each OutputTo command, one element in Z_p. In the protocol presented here each party sends in addition a tag in $(Z_p)^k$, thus increasing significantly the communication complexity.
A possible optimization is to perform a *lazy MAC check*: the rationale here is that an adversary who lies about her share only creates a real problem at the stage where some values is actually given as output to a party. In particular, no (temporary) damage is caused when a party lies about her share during the OutputTo commands which happen as a part of the Input and Mul operations.
Therefore a possible strategy is to modify the output command as follows:

(Lazy) Output: When an output command is run as a subroutine in Mul or Input

1. Party P_i receives x_j from all P_j, $j \neq i$;
2. Party P_j updates $h_{j,i} = h_{j,i} + H(t^x_{j,i})$;
3. Party P_i updates
$$h_{j,i} = h_{j,i} + H(x_j \cdot \alpha_{i,j} + \beta^x_{i,j})$$
Where all $h_{i,j}$ are initially set to 0 and H is some hash function[5]; Note that the input to the hash function is the MAC that an honest P_j would have sent.
4. Party P_i outputs $x = \sum x_i$;

(Real) Output: When an output command is run because P_i should learn some output of the computation $[x]$, perform step 1-3 from the lazy output procedure and then:

4. Every party P_j sends $h'_{j,i}$ to party P_i.
5. If for any j, $h'_{j,i} \neq h_{j,i}$ party P_i aborts, otherwise P_i outputs $x = \sum x_i$.

[5] See discussion below.

Discussion: The intuition here is that both parties P_i and P_j keep a history $h_{j,i}$ of all the MACs that P_j *should have sent to* P_i thus, instead of checking the MACs every time a gate is evaluated, MACs (or hashes of MACs) are only checked before the actual output phase. Doing so we reduce the total communication complexity from $O((I+M+O)\cdot k\cdot \log_p)$ to $O((I+M)\cdot \log_p + O\cdot k\cdot \log_p)$, where (I, M, O) are the number of input, multiplication and output commands respectively. On the negative side, the opening now requires the use of a hash function. The hash function can be implemented in at least two ways, namely: Using a *cryptographic hash function* (such as the SHA family) with the disadvantage that the online phase now requires cryptographic operations (and computational assumptions) or using *universal hashing*, that is simply performing a field multiplication with some random value. This does not require cryptographic operations during the online phase, and can be implemented by having the parties coin-flip some random seed for a PRG and use this to generate the stream of random (unbiased) elements used for the hashing.

Efficiency: The BeDOZa and TinyOT MAC style have the advantage that they do not require any cryptographic operations and achieve unconditional security. On the negative side, to store the representation of a single value in Z_p each party must store $O(nk)$ elements in Z_p. This will be addressed in the next two sections.

5.4 SDPZ Online Phase

Here we present an abstraction of the online phase of the SPDZ protocol [40, 37], including the recent multiparty version of the TinyOT protocol [77]. In the previous section, the storage requirement of each party grows linearly with the number of parties n. This is due to the fact that each party must have a key and a tag for each other party. In SPDZ instead of computing MACs on the share held by the parties, MACs will be computed directly on the secret shared values (which, at the end of the day, is the values we are interested in reconstructing without errors). More in details:

Shared Value: Also in SPDZ we use additive secret sharing representation enhanced with some MACs. First, each party P_i has a share $\alpha_i \in (Z_p)^k$. These shares define a unique global key $\alpha = \sum \alpha_i$.

For each shared value $[x]$ party P_i owns

1. A random share $x_i \in Z_p$ (under the condition that $\sum x_i = x$);
2. A share of a tag t_i^x such that the sum of all shares $t^x = \sum t_i^x$ satisfies

$$t^x = x \cdot \alpha$$

Once again, tags and keys are k-dimensional vectors (where $k = 1$ when p is big enough so that $1/p$ is negligible).

OutputAll: In the SPDZ protocol the "basic" output function is one which gives output to all parties.[6] To open a value $[x]$.

[6] If only one party P_i is supposed to learn an output $[x]$, one can use the standard trick of letting P_i input a random value r to the ABB and then let the ABB output $[z] = [x] + [r]$ instead. This now introduces a chicken and egg problem, since in our protocol description we used the command OutputTo as a crucial component for implementing Input. In SPDZ this is solved by asking the trusted dealer to compute random values $[r^{i,\ell}]$, for each input $\ell \in \{1, \ldots, I_i\}$ of party P_i, where the trusted dealer also reveals the plaintext value $r^{i,\ell}$ to party P_i. Now this can be used to implement a Input phase very similar to the one described above.

1. All parties P_i broadcast their shares x_i and define $x = \sum_i x_i$;
2. All parties P_i compute values $v_i = x \cdot \alpha_i - t_i$;
3. All parties P_i perform a *simultaneous exchange*[7] of the v_i's, and abort if $\sum_i v_i \neq 0$;

Addition: If the parties hold two values $[x]$ and $[y]$ and want to compute a new representation $[z]$ such that $z = x + y$ each party P_i

1. Computes $z_i = x_i + y_i$;
2. Computes $t_i^z = t_i^x + t_i^y$;

Discussion: The protocol is correct by inspection: the key observation here is that instead of having MACs on shares (like in the previous protocol) here there is only one key and only one tag, and each party holds a share of both. During the opening phase, also the MAC checking procedure is shared in the sense that once the value x becomes public, every party can compute a "share" of the decision of whether to accept or reject. Now, when the MAC check accepts, the output is 0 and this leaks no information on the MAC key α. If the MAC check does not accept, we abort the protocol and therefore even a leak of the value α does not compromise the privacy of the inputs of the parties (since MACs are only used to check the correctness of the output).
The second crucial aspect is that when the parties exchange the value v_i we need to avoid the obvious *rushing attack*, where a corrupted party P_j waits for all other parties to send their values v_i and then sets $v_j = -\sum_{i \neq j} v_i$, thus making the MAC check accept any (incorrect) value x. To avoid this we need to make sure that the parties exchange their shares of their results simultaneously, and this can be simply done letting all parties *commit and open* their shares.
Note that by performing *lazy MAC check* as described above, each party only needs to perform one single commitment per output value.

Efficiency: The SPDZ online phase outperforms the BeDOZa/TinyOT online phase in terms of storage complexity, since each party only needs to store one element in $(Z_p)^k$ per shared value in Z_p, thus independent of the number of parties.

5.5 MiniMACs Online Phase

As we already discussed, in the online phase of BeDoZa/TinyOT each party has a *storage overhead* of

$$O(n \cdot \epsilon^{-1} \cdot \log p)$$

where ϵ is the security parameter. This is problematic for large number of parties n and when computing over small modulo p. The SPDZ representation presented in the previous section takes care of the dependence on n, but there is still a significant overhead when computing with small p. For instance this means that when $p = 2$ then one needs to store a vector of MACs of size $k = \epsilon^{-1}$ for each bit in the computation.
In the MiniMACs protocol [41, 38], this problem is addressed by letting parties store vectors of values (instead of single ones) together with joint MACs on the whole vector. This has the advantage of taking the storage overehead down to constant, but introduces the drawback of

[7] See discussion below.

having to perform SIMD[8] computation on the vector of shared data. This is not a problem when the application at hand requires to compute many copies of the same function on different inputs at the same time.
The main idea is to take a vector

$$x \in (Z_p)^k$$

encode it using some *linear code with constant rate*

$$E(x) \in (Z_p)^k$$

and now compute a single MAC for each entry in the encoded vector. Now, if an adversary wants to cheat and have the honest parties output $x' \neq x$, he needs to forge as many MACs as the hamming distance between $E(x')$ and $E(x)$. By picking the right code we can make sure that this only happens with negligible probability.

[8] Single instruction multiple data.

Chapter 6

Implementing the Offline Phase of Protocols with Preprocessing

In this Chapter we describe the state-of-the art in implementing the preprocessing phase for the MPC protocols discussed in the previous section. These can be divided, depending on the underlying cryptographic technology, in those based on homomorphic encryption (BeDOZa [20] and SPDZ [40]) and those based on oblivious transfer (TinyOT [99]).
The offline phase in SPDZ uses a single instance of a *somewhat homomorphic* encryption scheme, with distributed key generation and decryption procedures to generate triples shared across all parties simultaneously; BeDOZa uses an *semi-homomorphic* encryption scheme between every pair of parties. Since these implementations are quite similar in spirit, and SPDZ outperforms BeDOZa, we survey the details of the former and highlight the differences with BeDOZa.

6.1 SPDZ

Overview

SPDZ relies on a *s*omewhat homomorphic encryption scheme given by algorithms (KeyGen, Enc, Dec): in such a scheme ciphertexts can be homomorphically added, and a *single* homomorphic multiplication can be performed. In addition, we require protocols for distributed key generation and distributed decryption for this encryption scheme. Distributed key generation generates a public key and additive shares of a secret key, so that each share reveals no information on the underlying key. All parties can use the common public key to encrypt data, but to decrypt they must jointly use the distributed decryption algorithm, DistDec. This allows players holding all shares of the secret key to decrypt a ciphertext known by all the players. One of the main sources of inefficiency in BeDOZa, the use of zero-knowledge proofs is eliminated at the expense of introducing the distributed decryption procedure.
Throughout this section we will sometimes use bold letters to emphasize that we are dealing with vectors, rather than with single elements in $\mathbb{Z}_p$. The SPDZ protocol uses the BGV scheme [25], which has security based on the *ring learning with errors* assumption. The key advantage of this scheme is that ciphertexts can contain a vector of many plaintext messages, such that homomorphic addition and multiplication are applied component-wise on plaintext vectors. Thus, the encryption (decryption) algorithm uses an extra encoding (decoding) function that takes elements from the vector space $\mathbb{Z}_p^s$ to the ring over which the message space is effectively defined. The necessary properties are that decodings of $\mathsf{encode}(\mathbf{m})$ returns $\mathbf{m}$, and the ring product $\mathsf{encode}(\mathbf{m}_1) \cdot \mathsf{encode}(\mathbf{m}_2)$ decodes to the component-wise product $\mathbf{m}_1 \cdot \mathbf{m}_2$ (which also applies

for the sum of two encodings). This allows the cost of creating many random multiplicative triples to be very efficient in an amortized sense.

Generating SPDZ Triples

Overview. Recall that we wish to generate triples $[a], [b], [c]$ with a, b, randomly distributed in $\mathbb{Z}_p$, and $c = a \cdot b$. That is, each player P_i holds shares a_i, b_i, c_i, of the triple, and shares t_i^a, t_i^b, t_i^b of their corresponding tags t^a, t^b, t^c.
The idea is that P_i first generates their random shares $\mathbf{a}_i, \mathbf{b}_i$ and computes encryptions of these. All players broadcast their ciphertexts and sum them up using homomorphic addition to obtain public ciphertexts $\mathsf{c}_\mathbf{a} = \mathsf{Enc}(\mathbf{a})$, and $\mathsf{c}_\mathbf{b} = \mathsf{Enc}(\mathbf{b})$, where $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, and $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$. Now they can use homomorphic multiplication to calculate an encryption of the component-wise product $\mathbf{a} \cdot \mathbf{b}$ and then use distributed decryption so that each party obtains a cleartext share of this.
The main challenge is to do all this with active (or covert) security, so that an adversary who does not follow the protocol cannot learn any additional information, with good probability.

EncCommit. A key subprotocol of SPDZ is EncCommit, which enables the use of ciphertexts as commitments, binding players to their shares of a message, whilst still allowing commitments to be manipulated with the homomorphic properties of the encryption scheme. The output of EncCommit is a random message $\mathbf{m}_i$ to each player P_i, seen as a share of the message $\mathbf{m} = \mathbf{m}_1 + \cdots + \mathbf{m}_n$, and in addition every player gets the ciphertexts $\mathsf{c}_i = \mathsf{Enc}(\mathbf{m}_i)$ for $i = 1$ to n.
A passively secure protocol for EncCommit is straightforward: after distributed key generation has been performed, each player simply encrypts a random message and sends the ciphertext to all other players. The difficulty in implementing this protocol with active security is to ensure that messages and ciphertexts are properly generated: if a player can generate their ciphertext dishonestly, it is possible that a selective failure attack could be mounted, leaking information on an honest party's message share depending on whether decryption succeeds or not later on.

Reshare. As well as EncCommit and somewhat homomorphic encryption, there is one more important building block required to be able to generate multiplication triples. We need a protocol that allows parties to decrypt a ciphertext so that each party learns only a share of the message (recall that distributed decryption gives the message to every party). This is fairly straightforward to do with distributed decryption and EncCommit: each party uses EncCommit to commit to another random share, $\mathbf{f}_i$, and they add the ciphertexts of these shares to the ciphertext being decrypted, giving an encryption of $\mathbf{m}+\mathbf{f}$, where $\mathbf{f} = \mathbf{f}_1 + \cdots + \mathbf{f}_n$. Now invoking distributed decryption on this, parties can subtract their share $\mathbf{f}_i$ to get a valid share of $\mathbf{m}$.

Triple generation. The protocol for generating triples is given in Figure 6.2. It uses sub-protocols EncCommit and Reshare in a fairly straightforward manner as described earlier. Note that in step 2d, Reshare is used with the flag NewCiphertext, which causes it to output a fresh encryption of the product $\mathbf{a} \cdot \mathbf{b}$. This is required since the somewhat homomorphic encryption scheme only supports one multiplication; we then need to multiply $\mathsf{c}_{\mathbf{a}\cdot\mathbf{b}}$ by c_α to obtain the tag on $\mathbf{a} \cdot \mathbf{b}$, which would not be possible without a fresh encryption.

b
b

Protocol Reshare

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

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

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

Figure 6.1: The protocol for sharing $\mathbf{m} \in R_p$ on input $\mathsf{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$.

Procedure TripleGen
This produces at least $2n$ $[\cdot]$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

1. The players run EncCommit to obtain c_α, an encryption of the global MAC key α, so that party P_i knows some share α_i of α.
2. For $k \in \{1, \ldots, 2n\}$:
   (a) The players run EncCommit twice so that P_i obtains plaintexts $\mathbf{a}_i, \mathbf{b}_i$ and all players obtain $\mathsf{c}_{\mathbf{a}_i}$ and $\mathsf{c}_{\mathbf{b}_i}$, encryptions of $\mathbf{a}_i$ and $\mathbf{b}_i$.
   (b) The players compute $\mathsf{c}_{\mathbf{a}} \leftarrow \mathsf{c}_{\mathbf{a}_1} + \cdots + \mathsf{c}_{\mathbf{a}_n}$ and $\mathsf{c}_{\mathbf{b}} \leftarrow \mathsf{c}_{\mathbf{b}_1} + \cdots + \mathsf{c}_{\mathbf{b}_n}$. Define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$ and $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$, although no party can compute $\mathbf{a}$ or $\mathbf{b}$.
   (c) The players compute $\mathsf{c}_{\mathbf{a}\cdot\mathbf{b}} \leftarrow \mathsf{c}_{\mathbf{a}} \cdot \mathsf{c}_{\mathbf{b}}$.
   (d) The players execute $\mathsf{Reshare}(\mathsf{c}_{\mathbf{a}\cdot\mathbf{b}}, \mathsf{NewCiphertext})$ so that P_i obtains the share $\mathbf{c}_i$ and all players obtain a fresh ciphertext $\mathsf{c}_{\mathbf{c}}$ encrypting the plaintext $\mathbf{c} = \mathbf{c}_1 + \cdots + \mathbf{c}_n$.
   (e) The players compute $\mathsf{c}_{\gamma(\mathbf{a})} \leftarrow \mathsf{c}_{\mathbf{a}} \cdot \mathsf{c}_\alpha$, $\mathsf{c}_{\gamma(\mathbf{b})} \leftarrow \mathsf{c}_{\mathbf{b}} \cdot \mathsf{c}_\alpha$ and $\mathsf{c}_{\gamma(\mathbf{c})} \leftarrow \mathsf{c}_{\mathbf{c}} \cdot \mathsf{c}_\alpha$.
   (f) The players execute:
   $\mathsf{Reshare}(\mathsf{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$,
   $\mathsf{Reshare}(\mathsf{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$, and
   $\mathsf{Reshare}(\mathsf{c}_{\gamma(\mathbf{c})}, \mathsf{NoNewCiphertext})$
   to obtain shares $\gamma(\mathbf{a})_i$, $\gamma(\mathbf{b})_i$ and $\gamma(\mathbf{c})_i$.

Figure 6.2: Production of tuples and shared bits.

Sacrificing. The triples generated by the previous protocol may contain errors. This is because the distributed decryption protocol, used in Reshare, allows an adversary to introduce b

errors into the ciphertext. One solution to prevent this attacks is to use general techniques to make the protocol immune to active attacks; this would however require the use of expensive zero knowledge proofs. Instead, we use a *sacrificing* technique, whereby half of the triples are wasted in order to check correctness of the other half. This method, outlined below, is also used for BeDOZa and TinyOT, which have the same problem that triples could contain errors. Take two triples $([a_0], [b_0], [c_0]), ([a_1], [b_1], [c_1])$, and agree on a public random value $u \in \mathbb{Z}_p$ using a commitment scheme.

- Partially open $v = u \cdot [b_0] - [b_1]$ and $w = [a_0] + [a_1]$.
- Locally compute

$$\begin{aligned} x &= u \cdot [c_0] + [c_1] - [a_0] \cdot v - [b_1] \cdot w \\ &= [u \cdot (c_0 - a_0 \cdot b_0) + c_1 - a_1 \cdot b_1] \end{aligned}$$

  and check it partially opens to 0.
- Check the tags on the partially opened values, as explained in the OutputAll command of Chapter 5.
- Output $([a_0], [b_0], [c_0])$ as a valid multiplication triple.

If b_0, b_1, u are random elements of $\mathbb{Z}_p$ then so are the opened values v and w, provided one of the triples is then discarded. If othe output triple is not multiplicative, then $c_0 - a_0 \cdot b_0$ is invertible in $\mathbb{Z}_p$, and the only challenge value for which the check succeeds is $u = (c_1 - a_1 \cdot b_1)/(c_0 - a_0 \cdot b_0)$, so this happens with probability $1/p$. In fields where p^{-1} is negligible in the security parameter this already gives statistical security. For small fields, security can be increased by simply checking the triple several times, sacrificing more triples: to get security 2^k we need to sacrifice roughly $k/\log p$ triples. Alternatively, a more efficient procedure for small fields is given for the multi-party version of TinyOT [77], which performs cut-and-choose and randomly assigns the triples into buckets before sacrificing (similarly to the actively secure EncCommit protocol that is described below).

EncCommit: covert security

To implement EncCommit with covert security, a simple cut-and-choose based protocol can be used. Each player generates c message and ciphertext pairs and broadcasts the ciphertexts, along with a commitment to the random seeds used to generate the ciphertexts. The players jointly choose one set of ciphertexts to use as the output at random, and then open the commitments to the remaining $c - 1$ sets of ciphertexts and check they are well-formed. Clearly a cheating player can only cheat in one of the c ciphertexts, so will only succeed with probability $1/c$. If c is chosen large enough (e.g. 10 or 20) and there is a suitable penalty for cheating to act as a deterrent then this can give an acceptable level of security for many scenarios.

EncCommit: active security

In some cases, we may require that a player can only cheat with *negligible* probability in a security parameter, for instance 2^{-40}. Clearly this is infeasible to achieve with the simple cut-and-choose approach above. For active security there are two possible approaches: one from [40] using a zero knowledge proof of plaintext knowledge and another cut-and-choose based method from [37]. Below we sketch the cut-and-choose method:

Let P_i be the player producing ciphertexts to be verified by the other players. The protocol is parametrized by two natural numbers T, b where b divides T. We set $t = T/b$. The protocol produces as output t ciphertexts $\mathsf{c}_0, \ldots, \mathsf{c}_{t-1}$.
Each such ciphertext is generated according to the algorithm described earlier, and is therefore created from the public key and four polynomials m, v, e_0 and e_1. To make the notation easier to deal with below, we rename these as f_1, f_2, f_3, f_4. We can then observe that there exist ρ_l, for $l = 1, \ldots, 4$ such that $|f_l|_\infty \leq \rho_l$ except with negligible probability. Concretely, we can use $\rho_1 = p/2$, $\rho_2 = 1$ and $\rho_3 = \rho_4 = \rho$ where ρ can be determined by a tail-bound on the Gaussian distribution used for generating f_3, f_4.
Each player P_i also creates a set of random *reference ciphertexts* $\delta_0, \ldots, \delta_{2T-1}$ that are used to verify that $\mathsf{c}_0, \ldots, \mathsf{c}_{t-1}$ are well-formed and that P_i knows what they contain. Each δ_j is created from 4 polynomials $g_1, \ldots, g_4$ in the same way as above, but the polynomials are created with a different distribution. Namely, they are random subject to $|g_i|_\infty \leq 4 \cdot \delta \cdot \rho_i \cdot T \cdot \phi(m)$, where $\delta > 1$ is some constant.
The protocol proceeds as follows:

1. P_i is given some number of attempts to prove that his ciphertexts are correctly formed. The protocol is parametrized by a number M which is the maximal number of allowed attempts. We start by setting a counter $v = 1$.

2. P_i broadcasts the ciphertexts $\mathsf{c}_0, \ldots, \mathsf{c}_{t-1}$ and the ciphertexts $\delta_0, \ldots, \delta_{2T-1}$, These ciphertexts should be generated from seeds $s_0, \ldots, s_{2T-1}$ that are first sent through the random oracle whose output is used to generate the plaintext and randomness for the encryptions.

3. A random index subset of size T is chosen, and P_i must broadcast s_i for $i \in T$. Players check that each opened s_i indeed induces the ciphertext δ_i, and abort if this is not the case.

4. A random permutation π on T items is generated and the unopened ciphertexts are permuted according to π. We renumber the permuted ciphertexts and call them $\delta_0, \ldots, \delta_{T-1}$.

5. Now, for each c_i, the subset of ciphertexts $\{\delta_{bi+j} |\ j = 0, \ldots, b-1\}$ is used to demonstrate that c_i is correctly formed. This is called the block of ciphertexts assigned to c_i. We do as follows:

   (a) For each i, j do the following: let $f_1, \ldots, f_4$ and $g_1, \ldots, g_4$ be the polynomials used to form c_i, respectively δ_{bi+j}. Define $z_l = f_l + g_l$, for $l = 1, \ldots, 4$.

   (b) Player P_i checks that $|z_l|_\infty \leq 4 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m) - \rho_l$. If this is the case, he broadcasts z_l, for $l = 1, \ldots, 4$. Otherwise he broadcasts $\perp$.

   (c) In the former case players check that $|z_l|_\infty$ is in range for $l = 1, \ldots, 4$ and that the z_l's induce the ciphertext $\mathsf{c}_i + \delta_{bi+j}$.

   (d) At the end, players verify that for each c_i, P_i has correctly opened $\mathsf{c}_i + \delta_{bi+j}$ for all ciphertexts in the block assigned to c_i.

   (e) If all checks go through, output $\mathsf{c}_0, \ldots, \mathsf{c}_{t-1}$ and exit. Else, if $v < M$, increment v and go to step 2. Finally, if $v = M$, the prover has failed to convince us M times, so abort the protocol.

It is possible to adapt the protocol for proving that the plaintexts in c_i satisfy certain special properties. For instance, assume we want to ensure that the plaintext polynomial f_1 is a

constant polynomial, i.e., only the degree-0 coefficient is non-zero. We do this by generating the reference ciphertexts such that for each δ_i, the polynomial g_1 is also a constant polynomial. When opening we check that the plaintext polynomial is always constant. The proof of security is trivially adapted for this case.

Some intuition for why this works: after half the reference ciphertexts are opened, we know that except with exponentially small probability, almost all the unopened ciphertexts are well formed. A simulator will be able to extract randomness and plaintext for all the well formed ones. When we split the unopened δ_j's randomly in blocks of b ciphertexts, it is therefore very unlikely that some block contains only bad ciphertexts. It can be shown that the probability that this happens is at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$ [100].

Assume P_i is corrupt: now, if he survives one iteration of the test, and no block was completely bad, it follows that for every c_i, he has opened opened at least one $\mathsf{c}_i + \delta_{bi+j}$ where δ_{bi+j} was well formed. The simulator can therefore extract a way to open c_i since $\mathsf{c}_i = (\mathsf{c}_i + \delta_{bi+j}) - \delta_{bi+j}$. It will be able to compute polynomials f_l for c_i with $|f_l|_\infty \leq 8 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m)$. Therefore, if some c_i is not of this form, the prover can survive one iteration of the test with probability at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$. To survive the entire protocol, the prover needs to win in at least one of the M iterations, and this happens with probability at most $M \cdot t^{1-b} \cdot (e \cdot \ln(2))^{-b}$, by the union bound.

Assume P_i is honest: then when he decides whether to open a given ciphertext, the probability that a single coefficient is in range is $\frac{1}{4 \cdot \delta \cdot \phi(m) \cdot T}$. There are $4 \cdot \phi(m)$ coefficients in a single ciphertext and up to T ciphertexts to open, so by a union bound, P_i will not need to send $\perp$ at all, except with probability $1/\delta$. The probability that an honest prover fails to complete the protocol is hence $(1/\delta)^M$. We therefore see that the completeness error vanishes exponentially with increasing M, and in the soundness probability, we only lose $\log M$ bits of security.

It is easy to see that for each opening done by an honest prover, the polynomials z_l will have coefficients that are uniformly distributed in the expected range, so the protocol can be simulated.

Finally, note that in a normal run of the protocol, only one iteration is required, except with probability $1/\delta$. So in practice, what counts for the efficiency is the time we spend on one iteration.

6.2 BeDOZa

Overview. The preprocessing stage of BeDOZa uses a *semi-homomorphic* encryption scheme. This is a relaxed notion of additively homomorphic encryption (AHE), where ciphertexts can be added together to compute an encryption of the sum of messages. In semi-homomorphic encryption, it suffices that homomorphic addition can be performed, and decryption will succeed provided the underlying message does not grow too large. While SPDZ uses homomorphic multiplication and distributed decryption to obtain a multiplication triple shared across all parties, in BeDOZa we only rely on additive homomorphism so we must perform the multiplication in a pairwise fashion.

To do this, first all pairs of parties generate a public/private key pair for the semi-homomorphic encryption scheme. Now two parties P_i and P_j can generate random shares x_i, x_j, and compute their product as follows: P_i encrypts x_i under P_i's public key and sends $\mathsf{Enc}_i(x_i)$ to P_j. Now P_j generates a random value r_j and uses additive homomorphism to calculate $\mathsf{c} = x_j \cdot \mathsf{Enc}_i(x_i) + \mathsf{Enc}_i(r_j)$ and sends c to P_i. P_i outputs the decryption of c and P_j outputs $-r_j$, which add up to the product $x_i \cdot x_j$.

b
b

To bootstrap pairwise multiplication to multiplication of a value shared across n parties, we simply run the pairwise protocol between every pair of parties, where each party uses the same random share for every instance.
Once shares of a triple have been generated, the parties then run a similar protocol to add MACs to these shares. Since a MAC is simply the product of one party's share and another party's global key, added to a local MAC key, the same pairwise protocol for multiplication can be applied to achieve authentication: one party inputs their share and the other party inputs their global MAC key.

Zero knowledge proofs. The protocol sketched above is only passively secure, since there is no way to ensure that the parties generate valid ciphertexts. To turn this into a protocol with active security, we need two zero knowledge proofs for the semi-homomorphic encryption scheme. These are a proof of plaintext knowledge, Π_{PoPK}, to prove that parties correctly encrypt ciphertexts and know the underlying plaintext, and a proof of correct multiplication, Π_{PoCM}, which ensures that the second party correctly multiplies in their share using additive homomorphism. In BeDOZa the zero knowledge proof Π_{PoPK} has amortized complexity $O(\kappa+u)$ bits per instance proved, where the soundness error is 2^{-u}, whilst Π_{PoCM} has complexity $O(\kappa u)$. For the particular case of Paillier encryption, there is a more efficient version of Π_{PoCM} with complexity just $O(\kappa + u)$.

Comparison with SPDZ. In BeDOZa, the main cost of the protocol is the zero knowledge proofs for active security. In particular, Π_{PoCM}, the proof of correct multiplication, is very expensive. This is avoided in SPDZ by using just one instance of a somewhat homomorphic encryption scheme: since all parties perform the multiplication on public ciphertexts there is no need to prove correctness. However, SPDZ has additional overheads due to distributed key generation and distributed decryption. Key generation only needs to be performed once and distributed decryption is quite efficient (since it doesn't need to guarantee correct outputs) but both of these procedures cause the parameters of the BGV scheme to grow. Unfortunately, this increases the cost of all basic operations on ciphertexts, as well as communication. Implementations of SPDZ suggest that producing triples is more efficient than BeDOZa, and also has the benefit of the more efficient online phase. However, in the BeDOZa paper, it is conjectured that a modified form of the semi-homomorphic encryption schemes satisfy an additional property called *multiplication security*. If this is the case, the Π_{PoCM} protocol is not needed and the resulting protocol could be much more efficient, but this has not been proven.

6.3 TinyOT

Overview. The preprocessing of TinyOT uses oblivious transfer (OT) instead of building on homomorphic encryption. We start explaining how OT is used to generate tags on bits. Next, since OT is a 2-party primitive, these tags can be obtained in a pairwise fashion, so we need a method to bootstrap to global tags. Finally, we explain how to produce the correlated randomness.

Generating Pair-wise Tags

In order to distinguish pair of parties we will write $[r]^i_{j,\alpha_j}$ to mean that P_i holds a bit r and element $\mu_i \in Z_p$, and P_j holds two field elements ν_j and α_j, such that $\mu_i = \nu_j + r \cdot \alpha_j$. This is captured in the ideal $\mathcal{F}_{\mathsf{aBit}}$ functionality of Figure 6.3, that produced large batches (of size ℓ)

of authenticated bits. To implement it we proceed in two steps. (1) First, P_i and P_j execute a protocol to generate a large number of authenticated bits under a larger global key $\tilde{\alpha}_j$. This protocol is insecure in the sense that a dishonest P_i might learn some bits of the larger key. We can think of it as a *leaky* implementation of $\mathcal{F}_{\mathsf{aBit}}$. (2) Privacy amplification is used to give P_j a smaller but fully secure (pairwise) key α_j.

The Functionality $\mathcal{F}_{\mathsf{aBit}}$

Honest Parties

On input (aBit, i, j) from honest P_i and P_j, the functionality samples a random $\alpha_j \in \mathbb{F}$ and does the following

- For each $s \leq \ell$ it samples random bit r^s and random field element ν_j^s. It then sets $\mu_i^s = \nu_j^s + r^s \cdot \alpha_j$.
- It outputs $\{r^s, \mu_i^s\}_{s\leq\ell}$ to P_i and $(\alpha_j, \{\nu_j^s\}_{s\leq\ell})$ to P_j. Thus, both parties hold representations $\{[r^s]^i_{\alpha_j,j}\}_{s\leq\ell}$

Corrupted Parties

- If P_i is corrupted, the functionality waits for the environment to input pairs $\{r^s, \mu_i^s\}_{s\leq\ell}$, and it sets $\nu_j^s = \mu_i^s + r^s \cdot \alpha_j$. Then $(\alpha_j, \{\nu_j^s\}_{s\leq\ell})$ is returned to party P_j (with α_j sampled as above).
- If P_j is corrupted, the functionality waits for the environment to input the pair $(\alpha_j, \{\nu_j^s\}_{s\leq\ell})$, and it sets $\mu_i^s = \nu_j^s + r^s \cdot \alpha_j$. The pairs $\{r^s, \mu_i^s\}_{s\leq\ell}$ are returned to party P_i (with $\{r^s\}_{s\leq\ell}$ sampled as above).

Figure 6.3: Ideal Two-party Bit Authentication [99]

The leaky protocol in detail. The main idea of the protocol, described in Figure 6.4, exploits techniques of OT extensions: The two parties run many OTs with P_i playing the sender and P_j playing the receiver. In the kth OT P_i inputs large vectors $(\mathbf{x}_0^k, \mathbf{x}_1^k)$, and P_j inputs the kth bit of α_j, i.e. inputs $\alpha_j[k] = a_k$. An honest P_i sets his input vectors so that they define an additive sharing of a random vector $\mathbf{r}$ formed with the bits that he wishes to authenticate. Namely, he sets $\mathbf{x}_0^k \oplus \mathbf{x}_1^k = \mathbf{r}$, where $\mathbf{r}[k]$ is the kth authenticated bit. To test that P_i used the same vector $\mathbf{r}$ in every OT the parties randomly partition the OTs into pairs. Say that one such pair consists of the kth and sth OT. P_j then sends $b = a_k \oplus a_s$ to P_i and computes $\mathbf{d} = \mathbf{x}_{a_k}^k \oplus \mathbf{x}_{a_s}^s$. If P_i is honest he can also compute $\mathbf{d}$ as $\mathbf{d} = \mathbf{x}_0^k \oplus \mathbf{x}_0^s \oplus e \cdot \mathbf{r}$. On the other hand, it is not difficult to see that if P_i used different values for $\mathbf{r}$ in the kth and sth OT he can guess $\mathbf{d}$ with at most probability $\frac{1}{2}$. Therefore, to test that P_i behaved honestly, they compare their own values of $\mathbf{d}$ by securely exchanging the hashes of $\mathbf{d}$ and check they are equal. This is modeled with an standard $\mathcal{F}_{\text{EQ}}$ functionality. In case of inequality the protocol is aborted since this will indicate that one party is corrupted. As P_j reveals $a_k \oplus a_s$, the parties waste the sth OT and only use the output of the kth OT as output from the protocol—since a_s is uniformly random $a_k \oplus a_s$ leaks no information on a_s. Note that we cannot simply let P_i reveal $\mathbf{d}$, as a malicious P_j could send $b = 1 \oplus a_k \oplus a_s$: this would allow P_j to learn both $\mathbf{d}$ and $\mathbf{d} \oplus \mathbf{r}$, thus

leaking $\mathbf{r}$. Using $\mathcal{F}_{\text{EQ}}$ forces a malicious P_j to make the protocol abort unless he can guess a random message, which he can do only with negligible probability $2^{-\ell}$.

Protocol π_{LaBit}

1. P_i samples $\mathbf{r} \in_{\text{R}} \{0,1\}^\ell$ and for $k = 1, \ldots, 2\tau$ samples $\mathbf{x}_0^k \in_{\text{R}} \{0,1\}^\ell$.
2. P_j samples $(a_1, \ldots, a_{2\tau}) \in_{\text{R}} \{0,1\}^{2\tau}$.
3. The parties run a $\mathcal{F}_{\text{OT}}(2\tau, \ell)$ functionality, where for $k = 1, \ldots, 2\tau$ P_i inputs messages $\mathbf{x}_0^k$ and $\mathbf{x}_0^k \oplus \mathbf{r}$. P_j inputs choice bit a_k and receives $\mathbf{x}_{a_k}^k = \mathbf{x}_0^k \oplus a_k \cdot \mathbf{r}$.
4. P_j picks a uniformly random pairing π (a permutation $\pi : [2\tau] \to [2\tau]$ where $\forall k, \pi(\pi(k)) = k$), and sends π to P_i. Given a pairing π, let $\mathcal{S}_\pi = \{k | k \leq \pi(k)\}$, i.e., for each pair, add the smallest index to $\mathcal{S}_\pi$.
5. For all τ indices $k \in \mathcal{S}_\pi$:
   (a) P_j announces $b_k = a_k \oplus a_{\pi(k)}$.
   (b) P_j computes $\mathbf{d}_k = \mathbf{x}_{a_i}^k \oplus \mathbf{x}_{a_s}^s$ and P_i computes $\hat{\mathbf{d}}_k = \mathbf{x}_0^k \oplus \mathbf{x}_0^s \oplus b_k \cdot \mathbf{r}$.
   The parties then compare the strings $(\mathbf{d}_k)_{k \in \mathcal{S}_\pi}$ and $(\hat{\mathbf{d}}_k)_{k \in \mathcal{S}_\pi}$ using $\mathcal{F}_{\text{EQ}}(\tau\ell)$ and abort if they are different. Otherwise the protocol continues.
6. P_i outputs $(\mathbf{r}, (\mathbf{x}_0^k)_{k \in \mathcal{S}_\pi})$ and P_j outputs $(\mathbf{x}_{a_k}^k, a_k)_{k \in \mathcal{S}_\pi}$.

Figure 6.4: Leaky Pairwise Authentication From Oblivious Transfer

Removing the leakage. In Figure 6.5 we describe a protocol where one quarter of the bits of the global key might leak, and amplify it to the $\mathcal{F}_{\mathsf{aBit}}$ functionality. It takes π_{LaBit} as a subprocedure.

1. The parties invoke π_{LaBit} with global keys of length $\tau = \frac{22}{3}\psi$. The output to P_i is $(\hat{\mu}_{i,k}, r_k)_{k \in [\ell]}$. The output to P_j is $(\hat{\alpha}_j, (\hat{\nu}_{j,k})_{k \in [\ell]})$.
2. P_j samples $\mathbf{A} \in_{\text{R}} \{0,1\}^{\psi \times \tau}$, a random binary matrix with ψ rows and τ columns, and sends $\mathbf{A}$ to P_i.
3. P_i computes $\mathbf{u}_{i,k} = \mathbf{A}\hat{\mathbf{u}}_{i,k} \in \{0,1\}^\psi$, (where $\hat{\mathbf{u}}_{i,k}$ are the bits of $\hat{\mu}_{i,k}$, and $\mu_{i,k}$ is formed with bits $\mathbf{u}_{i,k}$) and outputs $(\mu_{i,k}, r_k)_{k \in [\ell]}$.
4. P_j computes $\mathbf{a}_j = \mathbf{A}\hat{\mathbf{a}}_j$ and $\mathbf{v}_{j,k} = \mathbf{A}\hat{\mathbf{v}}_{j,k}$ (where $\hat{\mathbf{a}}_j$ are the bits of $\hat{\alpha}_j$, $\hat{\mathbf{v}}_{i,k}$ are the bits of $\hat{\nu}_{i,k}$) and outputs $(\alpha_j, (\nu_{j,k})_{i \in [\ell]})$.

Figure 6.5: Reducing $\mathcal{F}_{\mathsf{aBit}}$ to Amplified π_{LaBit}

Correctness of the protocol is straight forward: We have that

$$\hat{\mu}_{i,k} = \hat{\nu}_{j,k} \oplus r_i \cdot \hat{\alpha_j} \ ,$$

so

$$\mu_{i,k} = \mathbf{A}\hat{\mu}_{i,k} = \mathbf{A}\nu_{j,k} \oplus r_k \mathbf{A}\hat{\alpha_k} = \nu_{j,k} \oplus r_i \cdot \alpha_j .$$

In addition it is clear that the protocol leaks no information on the r_i's to P_j: there is only communication from P_j to P_i. It is therefore sufficient to look at the case where P_i is corrupted. To prove security against corrupted P_i it is enough to see that α_j is uniformly random in the

view of P_i except with probability $2^{2-\psi}$. In [99] it was shown that there exists a failure event F such that: (1) F occurs with probability at most $2^{2-\psi}$. (2) When F does not occur, then α_j is uniform to P_i.

Exploiting the Output of $\mathcal{F}_{\mathsf{aBit}}$

We now change perspective and do not see $\mathcal{F}_{\mathsf{aBit}}$ as a mean to obtain pairwise authentications. Instead we will use it to produce additive sharings of the scalar product $x \cdot \delta$, where one party chooses the scalar x, but the field element δ is unknown and additively shared among all the parties. This will serve for two purposes: to authenticate bits towards *all* the parties, and to generate the triples needed for the multiplication gate.
In other words, for a given bit x we want to obtain a representation of the form $[x]^i_\delta = \{\langle x\rangle^i, \langle \mu\rangle^i, \langle \nu\rangle^{\mathcal{P}}, \langle \delta\rangle^{\mathcal{P}}\}$. Here $\langle\cdot\rangle^{\mathcal{I}}$ stands for an additive secret sharing reconstructable by the parties indexed in subset $\mathcal{I} \subseteq \mathcal{P}$. For example when we write $\langle x\rangle^{\mathcal{I}}$ we mean that for each $i \in \mathcal{I}$ party P_i holds a bit x_i, or share, such that $\sum_{i\in\mathcal{I}} x_i = x$. If we drop the set of indices from $[\cdot]$ we mean that the values are shared among the entire system of parties.
The idea behind the protocol of Figure 6.6 is as follows: Each pair of parties is given access to the $\mathcal{F}_{\mathsf{aBit}}$ functionality and use its output to form the new representation $[\cdot]$. In more detail, if P_i wants to authenticate a bit x_i towards the entire set of parties he proceeds in three steps. First, he generates a representation $[r_j]^i_{\alpha_j} = \{\langle r_j\rangle^i, \langle \hat{\mu}_i\rangle^i, \langle \hat{\nu}_j\rangle^j, \langle \hat{\alpha}_j\rangle^j\}$ on a random bit r_j with each other party P_j using $\mathcal{F}_{\mathsf{aBit}}$. Second, P_j switches to a share α_j of his chosing — as P_j needs to use the same share with the remaining parties. P_j does this sending $\sigma_i = \hat{\alpha}_j + \alpha_j$ to P_i who sets $\mu_i = \hat{\mu}_i + x_i \cdot \sigma_i$. Third, P_i sends $d_j = r_i + x_i$ to P_j, who can use it to obtain a valid share of the MAC on x_i. Namely, P_j sets $\nu_j = \hat{\nu}_j + d_j \cdot \alpha_j$. After completing the same steps with all the parties, P_i adds up $x_i \cdot \alpha_i$ to all his shares μ_i. In this way the system has obtained $[x_i]^i_\delta$.

Generating tags towards all the parties. To generate global authentications $[x]$, the parties use the protocol $\Pi_{\mathsf{Bootstrap}}$ (Figure Figure 6.6) to obtain $[x]^i_\alpha$ which are tuples of the form$\{\langle x\rangle^i, \langle \mu\rangle^i, \langle \nu\rangle^{\mathcal{P}}, \langle \alpha\rangle^{\mathcal{P}}\}$, such that $\mu = \nu + x \cdot \alpha$. Then, for $j \neq i$, party P_j sets his share of x to be zero, and $t^x_j = \nu_j$. Party P_i sets $t^x_i = \mu + \nu_i$. Thus, the parties obtain $[x]$.

Generating multiplicative triples. First we see how to generate a variant of the triples, i.e. bit quadruples $([e], [z], [x_0], [x_1])$ such that $z = x_e$. Later the quadruple is converted to a multiplicative triple.
Observe that tags and OTs are closely related: OTs can be seen as evaluation of affine functions, which is essentially what tags are. Seeing both as the same object means that a way to authenticate bits also gives us a way to generate OTs, and the other way around.
To produce bit quadruples (e, z, x_0, x_1), such that $z = x_e$, the parties will use a (secret) affine line in $\mathbb{F}$ parametrized by (ϑ, η). Note that $\Pi_{\mathsf{Bootstrap}}$ gives $[e_i]^i_\eta$, where e_i is known to P_i, and an additive sharing $\langle\eta\rangle$ is held by the system. Since fresh copies of $\Pi_{\mathsf{Bootstrap}}$ are used to generate OT quadruples and for tags' generation, to emphasize which η is used in each concrete execution we write $\Pi_{\mathsf{Bootstrap}}(\eta)$. Note, that η is not an input to the functionality but a shared random value produced when calling Initialise. Now, performing n independent queries of Share command on this copy $\Pi_{\mathsf{Bootstrap}}(\eta)$, the parties can generate

$$[e]^{\mathcal{P}}_\eta = [e_1]^1_\eta + \cdots + [e_n]^n_\eta. \tag{6.1}$$

b
b

The Protocol $\Pi_{\mathsf{Bootstrap}}$

Initialize: Each party P_i samples a random α_i. Define $\alpha = \alpha_1 + \cdots + \alpha_n$.

Share:

The parties do the following:

1. For each $j \neq i$, run the steps below. After completion, party P_i obtains $\{r_{i,j}, \mu_{i,j}\}_{j \neq i}$ whilst party P_j obtains $\nu_{i,j}$, such that $\mu_{i,j} = \nu_{i,j} + r_{i,j} \cdot \alpha_j$.
   (a) P_i and P_j call $\mathcal{F}_{\mathsf{aBit}}$ on input (aBit, i, j): The box samples a random $\hat{\alpha}_j$ and then produces
   $$[r]^i_{\hat{\alpha}_j, j} = (r, \hat{\mu}_i, \nu_j),$$
   such that $\hat{\mu}_i = \nu_j + r \cdot \hat{\alpha}_j$, and outputs $\{r, \hat{\mu}_i\}$ to P_i and $\{\hat{\alpha}_j, \nu_j\}$ to P_j.
   (b) P_j computes $\sigma_i = \alpha_j + \hat{\alpha}_j$ and sends σ_i to party P_i.
   (c) P_i sets $\mu_i = \hat{\mu}_i + r \cdot \sigma_i = \nu_j + r \cdot \alpha_j$.
2. Party P_i samples ε at random and sets $\mu_i = \varepsilon + \sum_{j \neq i} \mu_{i,j}$ and $\nu_i = \varepsilon + x \cdot \alpha_i$.
3. Party P_i sends $d_j = x + r_{i,j}$ to party P_j for all $j \neq i$.
4. For $j \neq i$, P_j sets $\nu_j = \nu_{i,j} + d_j \cdot \alpha_j$.
5. Output (μ_i, ν_i, α_i) to P_i and (ν_j, α_j) to party P_j, for $j \neq i$. The system now has $[x]^i_\alpha$.

Figure 6.6: Transforming Two-party Representations into $[\cdot]^i_\alpha$-representations

Thus, the system obtains two (secret) elements $\langle e \rangle$, $\langle \zeta \rangle$, such that $\zeta = \vartheta + e \cdot \eta$, for line $(\langle \vartheta \rangle, \langle \eta \rangle)$. Define $\chi_0 = \vartheta$ and $\chi_1 = \vartheta + \eta$, so it holds $\zeta = \chi_e$. The quadruple (e, z, x_0, x_1) is then given by the least significant bits of the corresponding field elements $(e, \zeta, \chi_0, \chi_1)$.
To add tags to each bit of the quadruple that the parties just generated, the protocol uses the $\Pi_{\mathsf{Bootstrap}}(\alpha)$ instance to obtain a sharing $\langle \alpha \rangle$ of the global key. Each party can now authenticate his shares of (e, z, x_0, x_1) querying Share command and obtaining $[e]$, $[z]$, $[x_0]$, $[x_1]$. We emphasize that the same α is used to authenticate all OT quadruples, thus $\Pi_{\mathsf{Bootstrap}}(\alpha)$ is fixed once and for all.
The parties now have sharings $[e]$, $[z]$, $[x_0]$, $[x_1]$, which could suffer from two possible errors induced by the corrupted parties: Firstly the multiplicative relation $z = x_e$ may not hold, and second the tags values may be inconsistent. For the latter problem we will check all the partially opened values as explained in Chapter 5 at the end of the offline phase. For the former case we sacrifice some quadruples to check another, as explained in Section 6.1. Finally, the triple is derived simply setting $[a] = [x_0] + [x_1]$, $[b] = [e]$, $[c] = [x_0] + [z]$.

Chapter 7

Specific Protocols for Private Set Intersection

Private set intersection (PSI) allows two parties P_1 and P_2 holding sets X and Y, respectively, to identify the intersection $X \cap Y$ without revealing any information about elements that are not in the intersection. The basic PSI functionality can be used in applications where two parties want to perform JOIN operations over database tables that they must keep private, e.g., private lists of preferences, properties, or personal records of clients or patients.
PSI is used for privacy-preserving computation of functionalities such as relationship path discovery in social networks [88], botnet detection [93], testing of fully-sequenced human genomes [7], proximity testing [98], or cheater detection in online games [27]. Another use case is measurement of the performance of web ad campaigns, by comparing purchases by users who were shown a specific ad to purchases of users who were not shown the ad. This is essentially a variant of PSI where the input of the web advertising party is the identities of the users who were shown the ad, and the input of the merchant, or of an agency that operates on its behalf, is the identities of the buyers.
PSI has been a very active research field, and there have been many suggestions for PSI protocols. The large number of protocols makes it non-trivial to perform comprehensive cross-evaluations. This is further complicated by the fact that many protocol designs have not been implemented and evaluated, were analyzed under different assumptions and observations, and were often optimized w.r.t. overall runtime while neglecting other relevant factors such as communication.
In this chapter, we give an overview on existing efficient PSI protocols that are based on public-key cryptography (cf. Section 7.2), generic secure computation (cf. Section 7.3), and oblivious transfer (cf. Section 7.4). We compare both the empirical performance of all protocols on the same platform and conclude with remarks on the protocols and their suitability for different scenarios (cf. Section 7.5). This chapter is a summary of the work presented in [108].

7.1 Notation and Security Definitions

We denote the parties as P_1 and P_2, and their respective input sets as X and Y with $|X| = n_1$ and $|Y| = n_2$. When the two input sets are of equal size, we use $n = n_1 = n_2$. We refer to elements from X as x and elements from Y as y and each element has bit-length σ. We write $b[i]$ for the i-th element of a list b, denote the bitwise-AND between two bit strings a and b of equal length as $a \wedge b$ and the bitwise-XOR as $a \oplus b$. We write $\binom{N}{1}$-OT_ℓ^m for m parallel 1-out-of-N oblivious transfers on ℓ-bit strings, and write OT_ℓ^m for $\binom{2}{1}$-OT_ℓ^m. In this chapter, Enc
b
b

Security	SYM (κ)	FFC and IFC (ρ)	ECC (φ)	Hash
128-bit	128	3 072	K-283	SHA-256

Table 7.1: NIST recommended key sizes for symmetric cryptography (SYM), finite field cryptography (FFC), integer factorization cryptography (IFC), elliptic curve cryptography (ECC) and hash functions.

denotes the encryption operation and the operation in an encrypted domain and Dec denotes the decryption operation.

Security parameters We denote the symmetric security parameter as κ, the asymmetric security parameter as ρ, the statistical security parameter as λ, and use the recommended key sizes of the NIST guideline [101], summarized in Tab. 7.1. We denote the bit size of elliptic curve points with φ, i.e., $\varphi = 284$ for the NIST recommended Koblitz curve K-283 using point compression.

7.2 Public-Key-Based PSI

In the following section we describe efficient PSI protocols based on public-key cryptography. We describe a very simple Diffie-Hellman-based PSI protocol (Section 7.2.1) and a RSA-based PSI protocol (Section 7.2.2). An advantage of these protocols is that they are relatively easy to implement, but they achieve only moderate performance since they require at least a linear number of relatively expensive public-key operations.

7.2.1 Diffie-Hellman-Based PSI

Probably the first PSI protocol was given in [62] (a similar construction was described in [86]). The protocol is secure based on the Decisional Diffie-Hellmann (DDH) assumption (a security proof appeared in [2]). The protocol works in the following way: Both parties agree on a cyclic group of prime order q and on a hash function H that is modeled as a random oracle. P_1 chooses a secret $\alpha \in_R \mathbb{Z}_q$ and P_2 chooses $\beta \in_R \mathbb{Z}_q$. P_1 then computes $(H(x_1))^\alpha, ..., (H(x_{n_1}))^\alpha$, permutes the order of the results and sends them to P_2. In parallel to that operation, P_2 computes $(H(y_1))^\beta, ..., (H(y_{n_2}))^\beta$, permutes the order of the results and sends them to P_1.
P_1 then raises each of the values that it received to the power of α and sends the results to P_2, while P_2 raises each of the values that it received to the power of β. P_2 then compares the values that it computed to those received from P_1. Note that the associativity of the exponentiation operation guarantees that if $x = y$ then $((H(x))^\alpha)^\beta = ((H(x))^\beta)^\alpha$ and this enables P_2 to identify the intersection. (This last comparison does not require any crypto operations and can be implemented efficiently using a hash table.)
Overall, P_1 and P_2 have to send $n_1 + n_2$ and n_2 group elements, respectively, and compute $n_1 + n_2$ exponentiations each. A major advantage of this protocol, apart from its simplicity, is that the two parties execute similar computations and can therefore work in parallel and in full utilization of their computing power. In addition, the exponentiation can be implemented using elliptic-curve cryptography, improving computation and, even more notably, communication overhead.

7.2.2 RSA-Based PSI

A protocol for private set intersection which uses a blinded RSA approach was introduced (among other protocols) in [34]. A detailed description of an efficient implementation of this protocol appeared in [35]. In the protocol, P_1 chooses an RSA key pair $\langle \text{pk}, \text{sk} \rangle = \langle (N, e), d \rangle$ and computes $x'_i = H(\text{Dec}_{\text{sk}}(x_i))$ for each element x_i in its input set, where H is a cryptographic hash function modeled as a random oracle. P_1 sends pk to P_2 and P_2 chooses a random element r_i for each y_i, blinds y_i as $\mu_i = y_i \cdot \text{Enc}_{\text{pk}}(r_i) \mod N$, and sends the values μ_i to P_1.
P_1 then decrypts the blinded values μ_i of P_2 as $\mu'_i = \text{Dec}_{\text{sk}}(\mu_i)$ and sends μ'_i and the hash values x'_i back to P_2. Finally, P_2 de-blinds and hashes μ'_i by computing $y'_i = H(\mu'_i / r_i \mod N)$, which it then can compare with the received x'_i values to identify intersecting elements.
In terms of communication, P_2 sends to P_1 n_2 ciphertexts, i.e., its blinded values, and P_1 sends $n_1 + n_2$ ciphertexts, i.e., its own hashed elements and P_2's signed and blinded values. The computation complexity is n_2 RSA encryptions for P_2 and $n_1 + n_2$ RSA decryptions for P_1, of which the n_2 decryptions of its own values can be computed before receiving the first message from P_2.
Observe that in this protocol one party (P_2) has to perform n_2 relatively light-weight RSA encryptions, whereas the other party (P_1) has to perform $n_1 + n_2$ computationally intensive RSA decryptions. This asymmetry makes it hard to improve performance by running the tasks of the two parties in parallel, as is possible with the DH-based solution. In addition, this protocol cannot be based on elliptic-curve cryptography.

7.3 Circuit-Based PSI

Unlike special purpose private set intersection protocols, the protocols that we describe in this section are based on a *generic* secure computation protocol that can be used for computing arbitrary functionalities. State-of-the-art for computing the PSI functionality is the sort-compare-shuffle (SCS) circuit of [59], which has size $\mathcal{O}(n \log n)$. The SCS circuit can be evaluated using secure computation protocols such as GMW (cf. Section 3.3) or Yao's garbled circuits (cf. Section 3.2).
The usage of generic protocols holds the advantage that the functionality of the protocol can easily be extended, without having to change the protocol or the security of the resulting protocol. For example, it is straightforward to change the protocol to compute the size of the intersection, or a function that outputs 1 iff the intersection is greater than some threshold, or compute a summation of values (e.g., revenues) associated with the items that are in the intersection. Computing these variants using other PSI protocols is non-trivial.

7.3.1 Sort-Compare-Shuffle Circuit for PSI

The straightforward way of using a circuit for PSI is to compare each input item of P_1 to each input of P_2. However, this approach results in a circuit of size $O(n^2)$. A more efficient approach is the *sort-compare-shuffle (SCS)* circuit described in [59] that has a size of $O(n \log n)$. (We refer here to the SCS circuit that uses the Waksman permutation for shuffling). The SCS circuit computes the intersection between two sets by first *sorting* both sets into a single sorted list, then *comparing* all neighboring elements for equality, and finally *shuffling* the intersecting elements in order to hide any information that could be obtained from the resulting order.

Sort To sort both sets into a single sorted list, both parties locally pre-sort their sets and merge them using a *bitonic merging circuit* [8]. In contrast to a sorting network, a bitonic merging circuit takes advantage of the fact that the inputs are already sorted and allows the parties to obtain a globally sorted list of $2n$ input elements using $n\log_2(2n)$ sorter circuits. A sorter circuit takes as input two elements x and y, swapping them if $x > y$ and preserving the order if $x \leq y$. Each sort gate consists of a comparison and a conditional swap sub-circuit.

Compare All elements in the sorted list are then compared to their neighbors to determine if a duplicate exists. Since each party's input consists of different values, duplicates only occur for items in the intersection of the two inputs. A duplicate item is passed on, whereas if no duplicate is found then the item is replaced by a special bottom symbol.

Shuffle Finally, all elements are shuffled using a Waksman permutation network [122]. An n input Waksman circuit consists of $n\log_2(n) - n + 1$ conditional swap gates, which either forward their two input elements or swap their order depending on the required randomly chosen output permutation.

Overall The overall size of the SCS circuit for input words of length σ is $\sigma(3n\log_2 n + 4n) - n$ gates, which is the sum of $2\sigma n\log_2(2n)$ AND gates for the sort circuit, $\sigma(3n-1) - n$ AND gates for the compare circuit, and $\sigma(n\log_2(n) - n + 1)$ for the shuffle circuit, where $n = \frac{n_1+n_2}{2}$. It is important to note that approximately 2/3 of the AND gates in the circuit are due to multiplexers.

7.3.2 Optimized Circuit-Based PSI

We describe in this section an optimization which greatly reduces the overhead of circuit based PSI for GMW (as is detailed in Fig. 7.1 in Section 7.5, the reduction in the runtime for inputs of size 2^{18} is about 40%). The optimization is based on a protocol proposed in [91].
As outlined in Section 7.3.1, the size of the SCS circuit is dominated by the multiplexer gates. In each multiplexer operation with σ-bit inputs x and y and a choice bit s, we compute $z[j] = s \wedge (x[j] \oplus y[j]) \oplus x[j]$ for each $1 \leq j \leq \sigma$ using σ AND gates in total. The evaluation of this multiplexer circuit in the GMW protocol requires random $\text{OT}_1^{2\sigma}$, namely 2σ random OTs of single-bit inputs. We observe that the same wire s is input to multiple AND gates which allows for the following optimization.
Consider an input wire u that is the input to multiple AND gates of the form $w[1] = (u \wedge v[1]), \ldots, w[\sigma] = (u \wedge v[\sigma])$. Similar to the evaluation of a single AND gate described in Section 3.3, these gates can be evaluated using a multiplication triple (cf. Section 3.3) generalized to vectors, which we call a *vector multiplication triple*.
A vector multiplication triple has the following form: $a_1, a_2 \in \{0,1\}$; $b_1, b_2, c_1, c_2 \in \{0,1\}^\sigma$, where P_i holds the shares labeled with i that satisfy the condition $(a_1 \oplus a_2) \wedge (b_1[j] \oplus b_2[j]) = c_1[j] \oplus c_2[j]$. To evaluate the AND gates, both parties compute $d_i = a_i \oplus u_i$ and $e_i[j] = b_i[j] \oplus v_i[j]$, exchange $d_i, e_i[j]$, set $d = d_1 \oplus d_2$, $e[j] = e_1[j] \oplus e_2[j]$, and $w_i[j] = (d \wedge e[j]) \oplus (d \wedge b_i[j]) \oplus (e[j] \wedge a_i) \oplus c_i[j]$.
The vector multiplication triple can be pre-computed analogously to the regular multiplication triples described in Section 3.3, but using random OT_σ^2, namely only two random OTs applied to σ-bit strings: The parties each choose $a_1, a_2 \in_R \{0,1\}$ and perform a random OT_σ^1 with P_1 acting as sender and P_2 acting as receiver with choice bit a_2, and a second random OT_σ^1 with P_2 acting as sender and P_1 acting as receiver with choice bit a_1. From these random OTs, P_i

obtains $b_i \in \{0,1\}^\sigma = x_0^i \oplus x_1^i$ and, analogously to the regular multiplication triple generation, a valid $c_i \in \{0,1\}^\sigma$.

7.4 OT-Based PSI

The recent PSI protocol of [44] uses Bloom Filters (BF) and OT to compute set intersection. We summarize Bloom filters in Section 7.4.1 and the PSI protocol of [44] in Section 7.4.2. We then present a redesigned optimized version of the protocol in Section 7.4.3. This optimization reduces the runtime for inputs of size 2^{18} by $55\% - 60\%$ (cf. Section 7.5, Fig. 7.1).

7.4.1 The Bloom Filter

A BF that represents a set of n elements consists of an m-bit string F and k independent uniform hash functions $h_1, ..., h_k$ with $h_i : \{0,1\}^* \mapsto [1, m]$, for $1 \leq i \leq k$. Initially, all bits in F are set to zero. An element x is inserted into the BF by setting $F[h_i(x)] = 1$ for all i. To query if the BF contains an item y, one checks all bits $F[h_i(y)]$. If there is at least one j such that $BF[h_j(y)] = 0$, then y is not in the BF. If, on the other hand, all bits $BF[h_i(y)]$ are set to one, then y is in the BF except for a false positive probability ε. An upper bound on ε can be computed as $\varepsilon = p^k(1 + O(\frac{k}{p}\sqrt{\frac{\ln m - k \ln p}{m}}))$, where $p = 1 - (1 - \frac{1}{m})^{kn}$. The authors of [44] propose to choose the number of hash functions as $k = 1/\varepsilon$ and the size of the BF as $m = kn/\ln 2 \approx 1.44kn$. In their experiments, they set $\varepsilon = 2^{-\kappa}$, resulting in $k = \kappa$ and a filter of size $m \approx 1.44\kappa n$.

7.4.2 Garbled Bloom Filter-Based PSI

For BF-based PSI, one cannot simply compute the bitwise AND of the BFs that represent each set, as this leaks information (see [44] for details). Instead, the authors of [44] introduced a variant of the BF, called Garbled Bloom Filter (GBF). Like a BF, a GBF G uses κ hash functions $h_1, ..., h_\kappa$, but instead of single bits, it holds shares of length ℓ at each position $G[i]$, for $1 \leq i \leq m$. These shares are chosen uniformly at random, subject to the constraint that for every element x contained in the filter G it holds that $\bigoplus_{j=1}^{\kappa} G[h_j(x)] = x$.
To represent a set X using a GBF G, all positions of G are initially marked as unoccupied. Each element $x \in X$ is then inserted as follows. First, the insertion algorithm tries to find a hash function $t \in [1...\kappa]$ such that $G[h_t(x)]$ is unoccupied (the probability of not finding such a function is equal to the probability of a false positive in the BF, which is negligible due to the choice of parameters). All other unoccupied positions $G[h_j(x)]$ are set to random ℓ-bit shares. Finally, $G[h_t(x)]$ is set to $G[h_t(x)] = x \oplus \left(\bigoplus_{j=1, j\neq t}^{\kappa} G[h_j(x)]\right)$ to obtain a valid sharing of x. We emphasize that because existing shares need to be re-used, the generation of the GBF cannot be fully parallelized. (We describe below in Section 7.4.3 how the protocol can be modified to enable a parallel execution.)
In the semi-honest secure PSI protocol of [44], P_1 generates a m-bit GBF G_X from its set X and P_2 generates a m-bit BF F_Y from its set Y. P_1 and P_2 then perform OT_ℓ^m, where for the i-th OT P_1 acts as a sender with input $(0, G_X[i])$ and P_2 acts as a receiver with choice bit $F_Y[i]$. Thereby, P_2 obtains an intersection GBF $G_{(X \wedge Y)}$, for which $G_{(X \wedge Y)}[i] = 0$ if $F_Y[i] = 0$ and $G_{(X \wedge Y)}[i] = G_X[i]$ if $F_Y[i] = 1$. P_2 can check whether an element y is in the intersection by checking whether $\bigoplus_{i=1}^{k} G_{(X \wedge Y)}[h_i(y)] \stackrel{?}{=} y$. (Note that P_2 cannot perform this check for any value which is not in its input set, since the probability that it learns all GBF locations

associated with that value is equal to the probability of a false positive, which is negligible due to the choice of parameters.) The bit-length of the shares in the GBF can be set to $\ell = \lambda$.

7.4.3 Random GBF-Based PSI

We introduce an optimization of the GBF-based PSI protocol of [44], which we call the *random Garbled Bloom Filter* protocol. The core idea is to have parties collaboratively generate a *random* GBF. This is in contrast to the original protocol where the GBF had to be of a specific structure (i.e., have the XOR of the entries of $x \in X$ be x). The modified protocol can be based on random OT extension (in fact, on a version of the protocol which is even more efficient than the original random OT extension). For each position in the filter, each party learns a random value if the corresponding bit in its BF is 1. P_1 then sends to P_2 the XOR of the GBF values corresponding to each of its inputs, and P_2 compares these values to the XOR of the GBF values of its own inputs.

We denote the primitive that enables this solution an oblivious pseudo-random generator (OPRG), which takes as inputs bits b_1, b_2 from each party, respectively, generates a random string s, and outputs to P_t s if $b_t = 1$ and nothing otherwise, for $t \in \{0, 1\}$. Additionally, we require that the parties remain oblivious to whether the other party obtained s. A protocol for computing this functionality is obtained by modifying the existing random OT extension protocol of [3] as follows.

Random OT extension is a special flavor of OT extension where, in the i-th OT, S has no input and outputs two random values (x_0^i, x_1^i), while R inputs a choice bit vector b and outputs $x_{b[i]}^i$ (cf. Section 3.1). The new functionality is obtained by having S ignore the x_0^i output that it receives, and ignore also the x_1^i output if $b_1 = 0$. Similarly, R ignores its output if $b_2 = 0$. The random OT extension protocol thus becomes more efficient, since the parties can ignore parts of the computation.

Our resulting Bloom filter-based protocol works as follows. First, P_1 and P_2 each generate a BF, F_X and F_Y respectively. They evaluate the OPRG with P_1 being the sender and P_2 being the receiver, using the bits of F_X and F_Y as inputs, to obtain random GBFs G_X and G_Y with entries in $\{0, 1\}^\ell$. For each element x_j in its set X, P_1 then computes $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_X[h_i(x_j)]$, with $1 \leq j \leq n_1$. Finally, P_1 sends all m_{P_1} values in random order to P_2, which identifies whether an element y in its set is in the intersection by checking whether a j exists such that $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_Y[h_i(y)]$.

7.4.4 Private Set-Inclusion and Hashing

We outline a very recent private set intersection protocol ([108]) that is based on the most efficient OT extension techniques, in particular the random OT functionality [99, 3] and the efficient 1-out-of-N OT of [73]. This PSI protocol scales very efficiently with an increasing set size.

We first describe the protocol for a private equality test (PEQT) between two elements x and y and then describe how to efficiently extend it for comparing y to a set $X = \{x_1, ..., x_n\}$. The resulting protocol can then be simply extended to perform PSI between sets X and Y by applying the parallel comparison protocol for each element $y \in Y$. Finally, the overhead of the protocol can be greatly improved using hashing as described in [108].

The Basic PEQT Protocol

In the most basic private equality test (PEQT) protocol, P_1 and P_2 check whether their σ-bit elements x and y are equal by engaging in random $\binom{2}{1}$ $\text{OT}^{\sigma}_{\ell}$, where P_2 uses the bits of y as its choice vector. From each random OT, P_1 obtains two uniformly distributed and random ℓ-bit strings (s^i_0, s^i_1), and P_2 obtains $s^i_{y[i]}$. P_1 then computes $m_{P_1} = \bigoplus_{i=1}^{\sigma} s^i_{x[i]}$ (the XOR of the strings corresponding to the binary representation of x) and sends it to P_2. P_2 compares this value to $m_{P_2} = \bigoplus_{i=1}^{\sigma} s^i_{y[i]}$ and decides that $x = y$ if and only if $m_{P_1} = m_{P_2}$.
The basic private equality test can be improved by using a base-N representation of the inputs and a $\binom{N}{1}$ OT in the protocol. Specifically, let $N = 2^{\eta}$. P_1 and P_2 check whether their σ-bit elements x and y are equal by representing them using $t = \sigma/\eta$ letters from an alphabet of size N, and then engaging in random $\binom{N}{1}$-OT^{t}_{ℓ}.
For this, P_2 cuts its σ-bit element y into t blocks $y[i]$ of bitlength η each: $y = y[1]||\dots||y[t]$; similarly, P_1 interprets $x = x[1]||\dots||x[t]$. In the i-th random $\binom{N}{1}$-OT, P_2 inputs $y[i]$ as choice bits and P_1 obtains N random and uniformly distributed ℓ-bit strings $(s^i_0, ..., s^i_{N-1})$; P_2 obtains $s^i_{y[i]}$. P_1 sends $m_{P_1} = \bigoplus_{i=1}^{t} s^i_{x[i]}$ to P_2 who compares it to $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and decides that $x = y$ iff $m_{P_1} = m_{P_2}$.

Private Set Inclusion Protocol

In a private set inclusion protocol, P_1 and P_2 check whether y equals any of the values in $X = \{x_1, ..., x_{n_1}\}$. The set inclusion protocol is similar to the basic PEQT protocol, but in order to perform multiple comparisons in parallel, the OTs are computed over longer strings, essentially transferring (in parallel) a random string for each element in the set X.
In more detail, both parties run a random $\binom{N}{1}$-$\text{OT}^{t}_{n_1\ell}$, where P_2 uses the bits of y as choice bits. Each received string is of length $n_1\ell$ bits. That is, in the i-th random OT, P_1 obtains N random strings $(s^i_0, ..., s^i_{N-1}) \in \{0,1\}^{n_1\ell}$, and P_2 obtains one random string $s^i_{y[i]}$. The strings are parsed as a list of n_1 sub-strings of length ℓ bits each. We refer to the j-th sub-string in these lists as $s^i_w[j]$, for $1 \le j \le n_1$ and $0 \le w < N$. Using these sub-strings, P_1 and P_2 can then compute the XOR of the strings corresponding to their respective inputs, compare the results and decide on equality, as was described in the basic PEQT protocol above. In more detail, P_1 computes $m_{P_1}[j] = \bigoplus_{i=1}^{t} s^i_{x_j[i]}[j]$ and sends the $n_1\ell$-bit string m_{P_1} to P_2. P_2 decides whether y matches any of the elements in X by computing $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and checking whether there exists an index j with $m_{P_1}[j] = m_{P_2}$.
Note that we now require that the value m_{P_2} and *all* the n_1 values $m_{P_1}[j]$ are distinct, which happens with probability $n_1 2^{-\ell}$. Thus, to achieve correctness with probability 1-$2^{-\lambda}$, we must increase the bit-length of the OTs to $\ell = \lambda + \log_2 n_1$. Also, note that P_2 learns the position j at which the match is found, which can be avoided by randomly permuting the inputs.

The OT-Based PSI Protocol

To obtain the final PSI protocol that computes $X \cap Y$, P_2 simply invokes the private set inclusion protocol of Section 7.4.4 for each $y \in Y$. Overall, to compute the intersection between sets X and Y of σ-bit elements, the protocol requires $n_2\sigma/\eta$ random $\binom{N}{1}$-OTs of $n_1(\lambda + \log_2 n_1)$ bit-strings and additionally $n_1 n_2(\lambda + \log_2 n_1)$ bits to be sent. Using the random $\binom{N}{1}$-OT of [73], the total amount of communication is $2n_2\sigma\kappa/\eta + n_1 n_2(\lambda + \log_2 n_1)$ bits. For large n_1 and n_2, this amount of communication grows too large for an efficient solution. In order to cope with large sets, one can use a hashing scheme. In a hashing scheme, both parties assign their elements to a bucket based on the output of a hash function and compare the elements that are in the same

bucket. There exist various hashing schemes such as simple hashing [109], balanced allocation hashing [5], and Cuckoo hashing [102]. Details about these schemes and their application to our protocol above are given in [108].

7.5 Experimental Comparison

In the following we experimentally compare the PSI protocols described above. We describe our benchmarking environment in Section 7.5.1 and then detail the comparison between the protocols in Section 7.5.2. Fig. 7.1 compares the single-threaded runtimes and communication complexities of all protocols over Gigabit LAN. More extensive experiments can be found in [108].

7.5.1 Benchmarking Environment

We ran our experiments on two Intel Core2Quad desktop PCs (**without** AES-NI extension) with 4 GB RAM, connected via Gigabit LAN. In the experiment, P_1 and P_2 held the same number of input elements $n = 2^{18}$ and were not allowed to perform any pre-computation. We use $\sigma = 32$ as the bit length of the elements. We use a statistical security parameter $\lambda = 40$ and a symmetric security parameter $\kappa = 128$ (other security parameters are chosen according to Tab. 7.1). For our set-inclusion protocol we set $\eta = 8$, i.e., use 1-out-of-2^8 OT extensions.

Implementations The implementation of the blind-RSA-based [34] and garbled Bloom- Filter [44] protocols were taken from the authors, but we used a hash-table to compute the last step in the blind-RSA protocol that finds the intersection (the original implementation used pairwise comparisons with quadratic runtime overhead). We implemented a state-of-the-art Yao's garbled circuits protocol (using garbled-row-reduction, point-and-permute, free-XOR, and pipelining, cf. Section 3.2) by building on the C++ implementation of [29] and using the fixed-key garbling scheme of [15]. For Yao's garbled circuits protocol, we evaluated a size-optimized version of the sort-compare-shuffle circuit (comparison circuits of size and depth σ) while for GMW we evaluated a depth-optimized version (comparison circuits of size 3σ and depth $\log_2 \sigma$) for σ-bit input values, cf. [112]. We implemented FFC (finite field cryptography) and IFC (integer factorization cryptography) using the GMP library (v. 5.1.2), ECC using the Miracl library (v. 5.6.1), symmetric cryptographic primitives using OpenSSL (v. 1.0.1e), and used the OT extension implementation of [3] which requires about 3 symmetric cryptographic operations per OT for the asymptotic performance analysis.
We argue that we provide a fair comparison, since all protocols are implemented in the same programming language (C/C++), run on the same hardware, and use the same underlying libraries for cryptographic operations.
For each protocol we measured the time from starting the program until the client outputs the intersecting elements. All runtimes are averaged over 10 executions. Our results are summarized in Fig. 7.1.

7.5.2 Performance Comparison

We divide the performance comparison in Fig. 7.1 into three categories, depending on whether the protocol is based on public-key operations (red), circuits (blue), or OT (green).

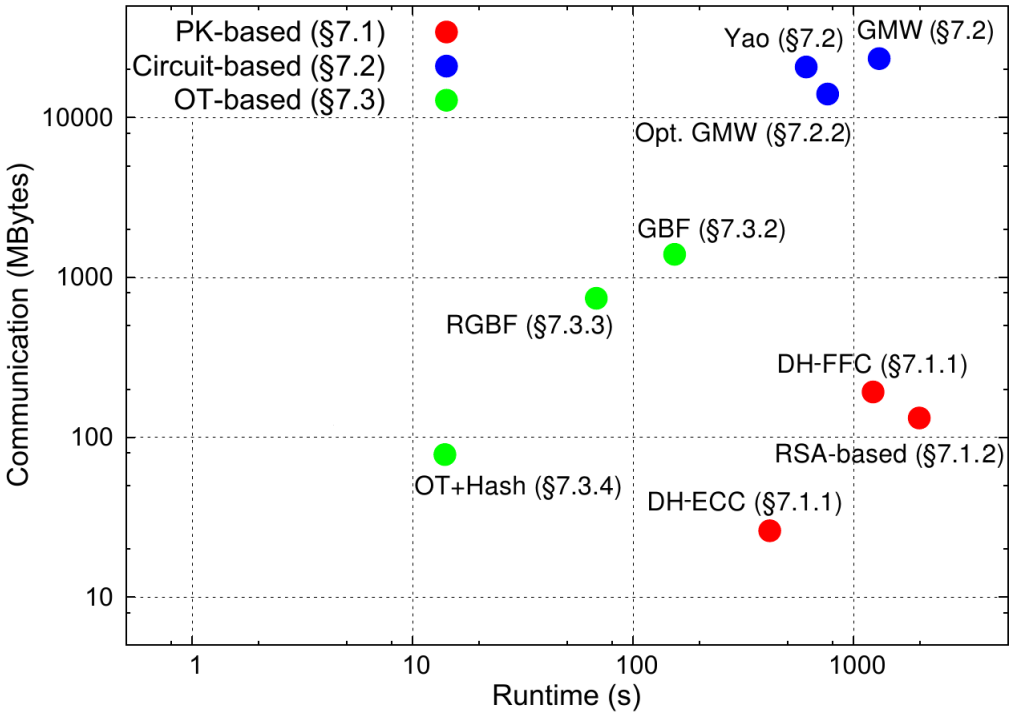


Figure 7.1: Runtime and communication of the outlined PSI protocols for $n = 2^{18}$ elements of $\sigma = 32$-bit length using $\kappa = 128$-bit security.

Public-Key-Based PSI (red) We observe that the DH-based protocol of [62] outperforms the RSA-based protocol of [34] when using FFC in runtime but has a larger communication overhead. The DH-based protocol using ECC improves the performance of both FFC-based protocols by a factor of 3 in runtime and has the lowest communication complexity among all protocols.
The major advantage of the public-key-based protocols is their simplicity as well as their small communication overhead.

Circuit-Based PSI (blue) Here we tested Yao- and GMW-based implementations, as well as an implementation of our optimized vector multiplication-triple-based GMW protocol (Section 7.3.2). The basic GMW protocol has the highest overall runtime and communication complexity. Our vector multiplication triple optimization reduces the runtime and communication of GMW by approximately 40%. In comparison, Yao's protocol is slightly faster but also requires more communication.
The circuit-based PSI protocols have the highest computation and communication complexity among all protocols that we tested. However, circuit-based protocols are of independent interest since their functionality can be easily adapted without requiring a new security proof.

OT-Based PSI (green) The random garbled Bloom filter protocol of Section 7.4.3 improves the original garbled Bloom filter protocol of [44] by more than a factor of two in runtime and by factor of 2-3 in communication. We also implemented the private set inclusion and hashing protocol in Section 7.4.4, where we used Cuckoo hashing. This protocol had the best runtime, and was about 5 times faster than the random garbled Bloom filter protocol. In terms of communication, the set inclusion protocol uses less than 10% of the communication of the random garbled Bloom filter protocol.

The OT-based PSI protocols have the lowest runtime among all tested protocols. Furthermore, the set-inclusion protocol achieves less communication than the public-key-based protocols that use FFC.

Chapter 8

Universal Verifiability

So far, we have focused on secure computation that assures correctness of the computation result only to those parties present at the time of the computation. However, there are various scenarios in which it should also be possible to verify correctness by parties not involved in the computation, i.e., when the computation should be *universally verifiable*. Such scenarios include the case when external parties need to be able to verify correctness for the "common good" (e.g., voters verifying an election result; or a competition watchdog verifying the computation of an optimal supply chain); such parties cannot be assumed to actively participate in every computation. Several such situations are described in PRACTICE Deliverable D12.1 [57]. However, even if parties interested in the computation would be willing to participate, if there are many of them that would cause an unacceptable overhead. Hence, these parties should not be actively involved in the computation, but they should be able to verify correctness of the outcome. Note that secure computation typically only guarantees correctness and privacy under some assumptions on the number and type of corrupted parties. Conversely, universal verifiability requires that verifiability holds *even if all parties are actively corrupted* (privacy however is still typically achieved only under assumptions on the adversary).
The problem of obtaining universally verifiable secure computation can be approached both from direction of secure computation and from the direction of verifiable computation. As described in earlier chapters, there is a considerable state-of-the-art in secure multi-party computation; it may be possible to adapt these techniques to make them universally verifiable. Dually, a considerable amount of work focusses on making computation efficiently verifiable (without privacy); it may be possible to adapt these techniques to make them privacy-friendly. In theory, combining privacy and verifiability is possible (any statement can be proven non-interactively, and this proof procedure can be performed in a multi-party way); but in practice it remains largely an open problem how to combine efficient (multi-party) computation with efficient verification.
In this chapter, we survey work relevant to universally verifiable secure computation. In Section 8.1, we discuss what adaptations to techniques for multi-party computation have been proposed to make them universally verifiable. In Section 8.2, we discuss verifiable computation techniques with a focus on potential applicability in the multi-party setting. Finally, in Section 8.3, we conclude with an overall review of the state-of-the-art of universally verifiable secure computation.

8.1 Making Secure Computation Universally Verifiable

Most of the literature on universal verifiability of secure computation has focused on particular application domains. The classical example of an application domain where universal verifiability is needed, is in e-voting. In this setting, several parties work together to determine the election result; although it may be unavoidable that these parties can learn individual votes by all colluding; they should at least not be able to manipulate the outcome of the election. In the traditional solution (e.g., [19, 1]), universal verifiability is achieved in the following steps: 1) each voter posts an encryption of his vote; 2) different parties verifiably shuffle the votes; and 3) the shuffled votes are verifiably decrypted.

A similar classical example is that of an auction, in which all bids but the winning one should remain private, but it should be possible to verify that the winning bid was highest. This can be achieved by encoding bids as encryptions of a fixed value F with a key corresponding to the bid value; and verifiably decrypting all bids with the keys corresponding to higher bid values to plaintexts other than F [111]. Note that in this case as well as in the e-voting case, verifiability is achieved by applying tricks specific to the application, rather than using general multi-party computation techniques (e.g., for e-voting, the computation of the votes is actually done in the clear; only the inputs are shuffled).

On the other hand, in the case of e-voting, other proposals for universal verifiability exist that can be generalised to achieve general universally verifiable multi-party computation. In particular, e-voting can be based on threshold homomorphic encryption as follows: 1) each voter posts an encryption of his votes; 2) all votes are homomorphically added; 3) the total is verifiably decrypted to obtain the election result (e.g., [33]). In effect, this is a verifiable multi-party computation of the homomorphic combination (e.g., sum) of the votes of the individual voters. (The observation that this is an instance of a more general MPC protocol was used by Groth [56] to analyse security properties of the voting schemes; unfortunately, he did not analyse universal verifiability.)

De Hoogh [42] was the first to consider universal verifiability of general multi-party computation. He proposes a construction based on threshold homomorphic encryption as in the above e-voting protocols; hence, this construction can be seen as a generalisation of the above approach to universally verifiable MPC. In particular, the construction is based on the protocols by Cramer, Damgård and Nielsen [31]. In these protocols, an arithmetic circuit is evaluated by homomorphic addition and private multiplication using threshold decrypted values. Active security is achieved by accompanying every decryption and multiplication by a Σ-protocol proving its correctness. De Hoogh [42] showed that applying the Fiat-Shamir heuristic [46] to these Σ-protocols preserves the security of the overall scheme; and then argued that the resulting transcript is a zero-knowledge proof that the computation was correct, hence the computation is universally verifiable. Indeed, at a high level, the transcript that the verifier sees consists of encryptions of the input, intermediary, and output values of the computation; and non-interactive zero knowledge proofs that they correspond to each other according to the computed circuit.

Recently, Baum, Damgård and Orlandi presented an extension to the SPDZ protocol to make it universally verifiable [9] (called "publicly auditable" in their work). Recall from Sections 5.4,6.1 that SPDZ [40] consists of an offline phase and an online phase. In the offline phase, random multiplication triples $c = a \cdot b$ and their message authentication codes (MACs) $\gamma_c = \alpha_c \cdot c$, $\gamma_a = \alpha_a \cdot a$, $\gamma_b = \alpha_b \cdot b$ are additively shared between the computation parties. In the online phase, these triples are used to quickly multiply shared values x, y by opening their differences $x - a$, $y - b$ and calculating a sharing of $x \cdot y$ from that using the well-known techniques of

Beaver [13]. The computation is made universally verifiable by sharing, along with each value of the computation (including the triples), also randomness for a Pedersen commitment to that value. The verifier sees all these commitments; openings of $x - a$, $y - b$ for multiplications; and the opening of the final result. With this, he can re-compute a commitment to the result himself, and compare it to the opening given by the computation parties. Assuming that the Pedersen commitments for the triples are correct, the binding property of the commitments ensures that the computation parties cannot convince the verifier of an incorrect result. Note that the verifier also needs to audit the full offline phase of the protocol to know that the triples are indeed correct. In contrast to the construction of [42], the auditor can only verify correctness of outputs he learns in the clear; he cannot verify if a certain encryption contains the correct output of the circuit evaluation.
The two generic approaches for universal verifiability by de Hoogh [42] and Baum et al. [9] share broadly the same characteristics. On the one hand, they do not greatly slow down the computation. In the case of [42], adding verifiability does not incur any cost for the computation parties compared to active security (but the base protocols are quite slow); in the case of [9], the online phase is roughly twice as slow because the commitment randomness for each computed value needs to be kept (but the protocols are quite efficient, so this factor two may not be a big problem). On the other hand, verification is quite heavy because the verifier needs to go through the full circuit: in [42], by checking zero-knowledge proofs of correct multiplication and decryption; and in [9], both by computing the commitments, and by verifying correctness of the triples used by them. De Hoogh [42] does point to the interesting possibility of speeding up verification by exploting the fact that for many practical problems (e.g. integer division, computation of matrix inverse/eigenvalues), it is easier to check a given solution than to compute it. Hence, the verifier does not need to go through the full computation circuit if instead the computation parties verifiably run a verification circuit.
Also, both techniques depend (for correctness as well as for privacy) on a set-up to be performed before the protocol. Either this set-up needs to be trusted by the verifier, or it needs to be performed in an auditable way; for both approaches, it is not clear how this auditable set-up can be practically achieved. Note, also, that both approaches have not been implemented, so it is not known how their performance plays out in practice.

8.2 Practical Verifiable Computation

In the previous section, the known techniques for proving multi-party computation correct have been described. The issue of proving computation correct has seen much more attention outside the field of multi-party computation, however, and in this section we will focus on those general techniques.
Even without employing multi-party computation or other means of providing input secrecy, verifiable computation can still be useful for performing outsourced computation to untrusted workers. In such a scenario, a computationally limited client can outsource a complex computation to a more powerful, but untrusted worker. The worker then provides, along with the outcome of the computation, a proof of correctness. In this scenario, it is essential that verifying the proof of correctness be more efficient than having the client carry out the computation by itself. There are non-cryptographic solutions to this problem such as replication, trusted hardware, attestation, and auditing; but each has assumptions on correlation of errors or trust relations that a cryptographic solution may avoid [123]. Achieving outsourcing to untrusted workers using cryptographic techniques has been an area of active research for the past three decades, but only in recent years have schemes offering nearly practical efficiency emerged.
b
b

In this section, we will briefly discuss the important results that serve as building blocks for most practical solutions. We will then detail those constructions that have proved sufficiently efficient and practical to see actual implementation.

8.2.1 Arguments and Probabilistically Checkable Proofs

In order to achieve efficient verification, a relaxed notion of what constitutes a proof is required, called an *argument* [26]. In contrast to "real" proofs, the existence of arguments for false statements is permitted. To be of any practical use, however, it must be infeasible for a computationally bounded prover to find such "incorrect" arguments. The soundness of arguments therefore only holds under computational assumptions on the prover.
One way to construct arguments is through the use of Probabilistically Checkable Proofs (PCPs) [6]. A PCP is a redundant encoding of a proof that allows a verifier to be convinced of the correctness of the proof by only inspecting a constant number of randomly selected locations (bits) in the PCP string. While this property implies very efficient verification, the highly redundant encoding makes direct application of PCPs impossible as a solution to the verifiable computation problem. Not only would the transmission of a PCP string be prohibitively expensive on the part of the verifier, which is required to be efficient, they are also typically too long for the prover to even construct explicitly.
In order to avoid transmitting entire PCP strings, cryptographic protocols were developed. These protocols are typically of an essentially interactive nature, in which the prover first commits to a PCP, after which the verifier queries, or challenges, the prover to reveal certain parts or properties of the PCP. It is important that the prover cannot predict the queries before they are presented, otherwise the prover would be able to break the soundness. An important result is due to Kilian [72], who constructed an efficient and succinct interactive argument system by using a novel bit commitment scheme and Merkle hash trees [87] to produce a short commitment to a PCP string the can be efficiently revealed in parts.

8.2.2 Practical Issues

Most efficient argument protocols require some form of set up, which depends on the function to be evaluated. Often, this set up involves large amounts of expensive cryptographic operations. To avoid this setup from being prohibitively expensive, it may be possible to reuse the same set of repetitions for the same computation on different input data. This way, the set up costs are amortized over many computations. It is also possible for the set up to be executed by a trusted party.
Employing a trusted party to perform the set up can also be used to achieve universal verifiability. In this case, the trusted party generates random challenge values and encrypts them using some form of (partially) homomorphic encryption on behalf of the verifier. The trusted party then publishes the encrypted challenge, eliminating the need for interactivity. When the prover has to compute its response to the verifier's challenge, it can do so under the encryption using the homomorphic properties, without being able to inspect the challenge values and thereby produce a fraudulent response. The evaluation of the verification conditions typically requires the use of pairings. Several of the concrete protocols discussed in the following sections use this approach to achieve universal verifiability under a trusted set up assumption.
Another well known method of eliminating interactivity, which is essential for universal verifiability, is by application of the Fiat-Shamir heuristic [46]. This is the approach taken by Micali in [89, 90], who transformed Kilian's protocol into an efficient non-interactive argument system b

and proved its soundness in the random oracle model. Although the Fiat-Shamir heuristic is not guaranteed to transform an interactive secure protocol into a non-interactive secure protocol, and there are theoretical objections to the random oracle model, this approach is useful in practice.
For a verifiable outsourced computation system to be considered practical, it is important that executing the verification procedure takes less time than executing the computation locally not only asymptotically, but also for real world applications. This also implies upper bounds on the amount of information the verifier can receive. Preferably, only a constant size proof is transmitted, but not all of the protocols described below will achieve this. Furthermore, it must also be possible for the prover to convince the verifier of the correctness of computation results with relatively little overhead.
The literature on state of the art implementation gives milliseconds as the typical order of magnitude for verification times and suggests the same order of magnitude for local execution. The typical proof times associated with this is on the order of minutes to hours, or even months for the most expensive systems. This discrepancy indicates that the these techniques can not yet be considered truly practical. The term *nearly practical* is used to indicate this. A quantitative comparison of state of the art implementations can be found in [123].

8.2.3 State of the Art Protocols and Implementations

In this section we will discuss the known protocols and implementations that offer near practical verifiability of general computation.

The IKO Protocol

Until the advent of the protocol by Ishai, Kushilevitz and Ostrovsky (IKO) in 2007 [66], all known constructions involving PCPs had the prover construct the entire PCP string and commit to it using hash trees. While most of the preceding research focussed on producing short PCPs, i.e., of size polynomial in the witness size, the IKO protocol avoided constructing the PCP string explicitly, thereby greatly improving the efficiency of the prover.
Instead of attempting to construct short PCPs, the IKO protocol uses a PCP which can be viewed as a linear function. While such a PCP may be exponentially long, the value at any position may be inspected efficiently by evaluating this linear function. By introducing a new commitment scheme for linear functions they were able to construct an argument system from any linear PCP. The linear PCP used in [66] is based on a proof for circuit satisfiability and its size is quadratic in the number of wires of the circuit.

Implementations The IKO protocol requires a large number of cryptographic operations to deliver the commitment to the linear PCP and a similarly large amount of communication for the challenge. The authors suggest batching computations, i.e., reusing a single commitment and a single challenge to prove correctness of many computations on different inputs to amortize the costs. In [114] batching and various other improvements to the IKO protocol were used to produce an implementation, called Pepper, achieving nearly practical efficiency for the production and verification of linear PCPs. Among the other improvements applied were switching from Boolean to arithmetic circuits, which can greatly reduce circuit complexity for computations which are arithmetic in nature; reducing the number of homomorphic encryptions required for the commitment protocol by orders of magnitude and optimization of the PCP itself.

Following Pepper, a system called Ginger was developed [115], offering theoretical improvements to greatly reduce the query cost of Pepper and extending the operations supported to suit more practical computations. Ginger further reduces the latency by benefiting from GPU-based parallelization.

Interactive Proofs for Muggles

The work of Goldwasser, Kalai and Rothblum [55] is not based on arguments, instead, their protocol is based on interactive proofs. In the proof setting, interactivity allows for far more efficient proofs than can be achieved non-interactively. Their protocol can also be used to create non-interactive arguments of correct computation. The authors only describe the possibility for designated verifier arguments, which depend on a public key of the verifier and are not transferable. Although the authors raise objections to the random oracle model, the protocol is also suitable for application of the Fiat-Shamir heuristic.
For interactive proofs, the prover is typically considered to be computationally unbounded. While this allows for the existence of very powerful proof techniques, this also limits the practical applicability of such protocols to verifiable outsourcing of computation, where the worker takes on the role of prover. Instead, [55] require that not only the verifier, but also the prover is computationally bounded. The computational boundedness of the prover implies even stricter bounds on the verifier, to the point that the verifier cannot even completely inspect the circuit whose output is being verified. This restricts the class of statements that can be proved to only computations represented by circuits of a highly "regular" nature, for which a short description exists.
The protocol works by proving correctness of the circuit evaluation layer by layer. Starting with the output layer, proving correctness of one layer is reduced to proving correctness of the preceding layer. This reduction is efficiently possible, because the verifier possesses a short description of the circuit and therefore of each layer. Finally, correctness of the output is reduced to correctness of the input, which the verifier can inspect.

Implementations Cormode, Mitzenmacher and Thaler [30] find that the GKR protocol is applicable in a streaming verifier setting, in which the verifier outsources some computation over a data stream to the prover, but the verifier's resources are limited so that it is not able to store the entire input stream. They present enhancements to the GKR protocol achieving prover time that is only quasilinear in the circuit size, as opposed to the original GKR protocol, which offers polynomial time in the circuit size. In addition to implementing the GKR protocol, they also present specialized protocols for proving correct computation for specific problems, such as matrix vector multiplication and pattern matching.
In a subsequent work [119], Thaler refines the protocol even further for circuits satisfying concretely specified regularity properties. The refined protocol achieves prover time linear in the circuit size. Although not all computations can be described using such regular circuits, this does cover important problems such as matrix multiplication. Additionally, Thaler applies the protocol to computations in which the same circuit, which does not necessarily satisfy the regularity property, is applied many times independently and in parallel to many input values and achieves prover time that scales linearly, rather than quasilinearly, in the number of input values. Finally, a specific protocol for proving square matrix multiplication is given with prover time quadratic in the height (or width) of the matrix, which means that asymptotically, the prover time is dominated by computing the matrix multiplication itself, not by the time needed to construct the proof.

Quadratic Arithmetic Programs

In 2013, Gennaro et al. introduced a new characterization of the complexity class NP, namely Quadratic Span Programs (QSP) [51]. While QSPs can describe any computation, this property is not used directly to specify and verify computations. Instead, the authors demonstrate how to efficiently construct a QSP for computing circuit satisfiability, given the circuit values.
In essence, circuit satisfiability can be verified using a QSP by checking that a special (circuit dependent) target polynomial divides a polynomial composed of a (circuit dependent) set of polynomials and the circuit's wire assignments. Polynomial divisibility can be efficiently interactively verified and the verification can be made non-interactive in the CRS model, using the general challenge hiding approach.
Arithmetic circuits are typically smaller than Boolean circuits representing the same computation and tend to be more practical for describing general computations. Quadratic Arithmetic Programs (QAP), the arithmetic counterpart to QSPs, can be used be used to prove satisfiability of arithmetic circuits and are therefore more suitable for real applications.

Implementations The most straightforward implementation of a QAP based verification system is Pinocchio [103], which, apart from introducing the necessary implementation refinements and reducing the QAP complexity, uses QAPs in the manner described by [51].
It was also observed in [21] that QAPs can be considered linear PCPs and that the QAP-based PCP for circuit satisfiability scales quasilinearly in the circuit size, in contrast to the PCP used in [66], which scales quadratically. Implementation of the QAP-based linear PCP in the Pepper line of systems based on the IKO protocol resulted in a system called Zaatar [113].
Although arithmetic circuits are sufficiently expressive for specifying general computations, it is more natural to do so using a high level programming language. [17] produced a C compiler for a random access machine architecture called TinyRAM, which generates arithmetic circuits. Unlike previous circuit compilers, that either don't support data dependent loops, control flow and memory access, or do support those at the cost of quadratic complexity, the [17] compiler manages to produce circuits of quasilinear complexity by supporting a non-deterministic witness representing a trace of the program's state or memory as it is being executed. They further use QAPs as an efficient means to prove circuit satisfiability and apply the numerous optimizations necessary to implement an argument system for general computation. The resulting system brings verifiable computation closer to practicality by avoiding the need for the computation to be specified as an arithmetic circuit.
The system of [17], as does Pinocchio, requires an expensive set-up stage which depends on the function to be computed. Although after the set-up the computation can be repeated many times and on different inputs, a new set-up is required when a different function is to be computed. The work of [17] was followed up by [18] by employing universal circuits to support a von Neumann architecture, called vnTinyRAM, supporting such programming techniques as self-modifying code and just-in-time compilation. This work further generalizes the set of programs supported for verifiable computation. The only restriction imposed by this system is an upper bound on the computation's running time, which is inherent in the use of universal circuits.

8.3 Universally Verifiable Secure Computation

The above discussion suggests that, although universally verifiable secure computation is possible today, it may profit from general verifiable computation results. Adding verifiability to

multi-party computation is possible in some cases without too much changes to existing protocols, but in this case the verifier needs to perform work for each gate in the circuit [42, 9].
On the other hand, recent results from (non-privacy-friendly) verifiable computation, e.g., [51, 17] allow very efficient verification: proofs in the order of hundreds of bytes, which can be verified in milliseconds. (All this is assuming a trusted set-up stage which depends on the computation to be performed, but this may be acceptable in many application domains.) However, in the verifiable computation literature, privacy has not been considered much (with the notable exception of [50], who define privacy in this setting, and provide a theoretical solution based on fully homomorphic encryption). In theory, obtaining efficiently verifiable proofs is possible (simply by making the construction of the proof a multi-party computation); but distributing the construction of these proofs in practice is still open. Also, the "best" proof for verification may not be the best proof for secure computation; alternative approaches to those discussed above, e.g., based on attribute-based encryption [104], may turn out to lend themselves better to translation to the secure computation setting.

Chapter 9

Summary and Conclusions

Secure multi-party computation (MPC) enables a set of untrusting parties to compute arbitrary functions of their confidential inputs while revealing nothing but the final output of the function, and ensuring the correctness of the computation. This technology is particularly relevant in cloud settings where clients store and use confidential data on external cloud servers.
Initial research in MPC showed very strong feasibility results demonstrating that any function can be securely and efficiently computed. However, these results were theoretical in nature and were not efficient in practice. In recent years there have been tremendous improvements in the practical performance of protocols for secure computation.
Partners in the PRACTICE project are among the pioneers and leaders in MPC research. In this deliverable we identified and summarized the best state of the art MPC technologies (some of them were developed by PRACTICE researchers). Furthermore, the deliverable will tremendously assist the project in identifying aspects that need to be improved for real world cloud deployments of MPC. These aspects will be optimized in PRACTICE, in order to be deployed in the next phase of the project, and allow for cloud computing on encrypted data. The techniques are already partially implemented by the project partners and could be shown in the project's second year demos.
The current state of the art can be summarized as follows:

- In the case of two parties, and security against semi-honest adversaries, the best protocols are variants of Yao's basic protocol for secure two-party computation. There have been many recent improvements of that protocol, improving both the evaluation of the gates, and the oblivious transfer operations applied to the inputs.

- In the case of two parties and security against malicious adversaries, there are recent efficient variants of Yao's protocol based on the cut-and-choose methodology. These protocols use multiple copies of the circuit, where some copies are checked for correctness and some are evaluated. Protocols of this type were greatly improved in recent years, especially in terms of their amortized complexity.

- In the case of more than two parties, and security against semi-honest adversaries, the best protocols are based on the methodology of the BGW and CCD protocols. These protocols have been used in actual commercial deployments and are ready for industrial use.

- In the case of more than two parties, and security against malicious adversaries, the best protocols belong to the SPDZ family. These protocols first run a preprocessing phase which is rather heavy but can be computed before the inputs to the computation are

known. The actual online computation which is run after the inputs are received is extremely efficient. This approach can also be applied to the two-party case.

- When it is needed to compute specific functions and the performance of the computation is of great importance, it makes sense to design specialized protocols rather than computing them using generic protocols. This task requires cryptographic expertise, whereas the usage of generic protocols only requires describing the function as a Boolean function.

Secure multi-party computation is still a dynamic research area where new protocols and new improvements are presented every year. The performance of secure computation will always be slower than that of naive insecure computation. However, the research community, and PRACTICE partners in particular, are working hard on closing the gap between the theory and practice of secure computation. We expect that, based on the results of the project, the technology will soon be ready for usage in computing on encrypted data in cloud environments.

Chapter 10

List of Abbreviations

EC	European Commission
ECC	Elliptic curve cryptography
FFC	Finite field cryptography
IFC	Integer factorization cryptography
FHE	Fully homomorphic encryption
MAC	Message authentication code
MPC	Multi-party computation
OPRG	Oblivious pseudo-random generator
OT	Oblivious transfer
PCP	Probabilistically checkable proofs
PEQT	Private equality test
PSI	Private set intersection
QAP	Quadratic arithmetic programs
QSP	Quadratic span programs
SCS	Sort-compare-shuffle

Bibliography

[1] Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

[2] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Management Of Data (SIGMOD'03)*, pages 86–97. ACM, 2003.

[3] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Computer and Communications Security (CCS'13)*, pages 535–548. ACM, 2013.

[4] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[5] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing*, 29(1):180–200, 1999.

[6] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *STOC*, pages 21–31. ACM, 1991.

[7] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Computer and Communications Security (CCS'11)*, pages 691–702. ACM, 2011.

[8] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[9] Carsten Baum, Ivan Damgrd, and Claudio Orlandi. Publicly auditable secure multi-party computation. Cryptology ePrint Archive, Report 2014/075, 2014. `http://eprint.iacr.org/`.

[10] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

[11] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology – CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[12] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Symposium on Theory of Computing (STOC'96)*, pages 479–488. ACM, 1996.

[13] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[14] Donald Beaver. Commodity-based cryptography (extended abstract). In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 446–455. ACM, 1997.

[15] M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Symposium on Security and Privacy (S&P'13)*, pages 478–492. IEEE, 2013.

[16] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[17] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and JuanA. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer Berlin Heidelberg, 2013.

[18] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive arguments for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013. `http://eprint.iacr.org/`.

[19] Josh Benaloh. Simple verifiable elections. In Dan S. Wallach and Ronald L. Rivest, editors, *EVT*. USENIX Association, 2006.

[20] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology – EUROCRYPT'11*, volume 6632 of *LNCS*, pages 169–188. Springer, 2011.

[21] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[22] D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In *Financial Cryptography and Data Security (FC'12)*, volume 7397 of *LNCS*, pages 57–64. Springer, 2012.

[23] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis (short paper). In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security. FC'12*, pages 57–64, 2012.

[24] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Financial cryptography and data security. chapter Secure Multiparty Computation Goes Live, pages 325–343. Springer-Verlag, Berlin, Heidelberg, 2009.

[25] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012.

[26] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988.

[27] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. OpenConflict: Preventing real time map hacks in online games. In *IEEE S&P'11*, pages 506–520. IEEE, 2011.

[28] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.

[29] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In *Cryptographers' Track at the RSA Conference (CT-RSA'12)*, volume 7178 of *LNCS*, pages 416–432. Springer, 2012.

[30] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 90–112, 2012.

[31] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–300. Springer Berlin Heidelberg, 2001.

[32] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994.

[33] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European Transactions on Telecommunications*, 8(5):481–490, 1997.

[34] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, pages 143–159. Springer, 2010.

[35] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing (TRUST'12)*, volume 7344, pages 55–73. LNCS, 2012.

[36] I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In *Security and Cryptography for Networks (SCN'12)*, pages 241–263, 2012.

[37] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[38] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the minimac protocol for secure computation. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International*

Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings, pages 398–415. Springer, 2014.

[39] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[40] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer Berlin Heidelberg, 2012.

[41] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.

[42] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation : secure linear programming*. PhD thesis, Eindhoven University of Technology, 2012.

[43] D. Demmler, T. Schneider, and M. Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security'14*, pages 893–908. USENIX, 2014.

[44] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *Computer and Communications Security (CCS'13)*, pages 789–800. ACM, 2013.

[45] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012. `http://eprint.iacr.org/`.

[46] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[47] Fairplay—secure function evaluation. `http://www.cs.huji.ac.il/project/Fairplay/`. Last accessed October 8^{th}, 2014.

[48] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2013.

[49] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*. Springer, 2014.

[50] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag.

b
b

[51] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer Berlin Heidelberg, 2013.

[52] O. Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, 2004.

[53] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Symposium on Theory of Computing (STOC'87)*, pages 218–229. ACM, 1987.

[54] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[55] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Cynthia Dwork, editor, *STOC*, pages 113–122. ACM, 2008.

[56] Jens Groth. Evaluating security of voting schemes in the universal composability framework. Cryptology ePrint Archive, Report 2002/002, 2002. `http://eprint.iacr.org/`.

[57] Georg Hafner, Mario Münzer, Ferdinand Brasser, Janus Dam Nielsen, Peter Sebastian Norholdt, Dan Bogdanov, Riivo Talviste, Liina Kamm, Marko J oemets, Meilof Veeningen, Niels de Vreede, Antonio Zilli, and Kurt Nielsen. PRACTICE Deliverable D12.1: application scenarios and their requirements, 2013. Available from `http://www.practice-project.eu`.

[58] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, 1st edition, 2010.

[59] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed System Security (NDSS'12)*. The Internet Society, 2012.

[60] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, pages 539–554. USENIX, 2011.

[61] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Garay and Gennaro [49], pages 458–475.

[62] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *ACM Conference on Electronic Commerce (EC'99)*, pages 78–86. ACM, 1999.

[63] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 169–178. ACM, 2013.

[64] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[65] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC*, pages 600–620, 2013.

[66] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short pcps. In *IEEE Conference on Computational Complexity*, pages 278–291. IEEE Computer Society, 2007.

[67] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 406–425. Springer, 2011.

[68] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In Wagner [121], pages 572–591.

[69] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In Naor [96], pages 97–114.

[70] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.

[71] Liina Kamm and Jan Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. Cryptology ePrint Archive, Report 2013/850, 2013. `http://eprint.iacr.org/`.

[72] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, New York, NY, USA, 1992. ACM.

[73] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology – CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[74] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology And Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.

[75] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[76] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In Garay and Gennaro [49], pages 440–457.

[77] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Garay and Gennaro [49], pages 495–512.

b
b

[78] B. Li, H. Li, G. Xu, and H. Xu. Efficient reduction of 1 out of n oblivious transfers in random oracle model. Cryptology ePrint Archive, Report 2005/279, 2005. `http://eprint.iacr.org/`.

[79] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology – CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 1–17. Springer, 2013.

[80] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2011.

[81] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Naor [96], pages 52–78.

[82] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[83] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *Security and Cryptography for Networks, 6th International Conference, SCN 2008, Amalfi, Italy, September 10-12, 2008. Proceedings*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

[84] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [49], pages 476–494.

[85] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[86] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE S&P'86*, pages 134–137. IEEE, 1986.

[87] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[88] G. Mezzour, A. Perrig, V. D. Gligor, and P. Papadimitratos. Privacy-preserving relationship path discovery in social networks. In *Cryptology and Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 189–208. Springer, 2009.

[89] Silvio Micali. Cs proofs (extended abstracts). In *FOCS*, pages 436–453. IEEE Computer Society, 1994.

[90] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[91] P. Mohassel and S. S. Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology – EUROCRYPT'13*, volume 7881 of *LNCS*, pages 557–574. Springer, 2013.

[92] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[93] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security Symposium*, pages 95–110. USENIX, 2010.

[94] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, 2005.

[95] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Electronic Commerce (EC'99)*, pages 129–139. ACM, 1999.

[96] Moni Naor, editor. *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*. Springer, 2007.

[97] Moni Naor and Omer Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. *J. Comput. Syst. Sci.*, 58(2):336–375, 1999.

[98] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Network and Distributed System Security (NDSS'11)*. The Internet Society, 2011.

[99] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[100] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[101] NIST. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Rev. 3). Technical report, National Institute of Standards and Technology (NIST), 2012.

[102] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA'01)*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001.

[103] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252, 2013.

[104] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *Proceedings of the*

9th International Conference on Theory of Cryptography, TCC'12, pages 422–439, Berlin, Heidelberg, 2012. Springer-Verlag.

[105] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In Wagner [121], pages 554–571.

[106] Jason Perry, Debayan Gupta, Joan Feigenbaum, and RebeccaN. Wright. Systematizing secure computation for research and decision support. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks*, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer International Publishing, 2014.

[107] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[108] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium (USENIX Security'14)*, pages 432–440. USENIX, 2014. Full version: `http://eprint.iacr.org/2014/447`.

[109] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science (RANDOM'98)*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998.

[110] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85. ACM, 1989.

[111] Kazue Sako. An auction protocol which hides bids of losers. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography*, volume 1751 of *Lecture Notes in Computer Science*, pages 422–432. Springer, 2000.

[112] T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, volume 7859 of *LNCS*, pages 275–292. Springer, 2013.

[113] Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *EuroSys*, pages 71–84. ACM, 2013.

[114] Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*. The Internet Society, 2012.

[115] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In Tadayoshi Kohno, editor, *USENIX Security Symposium*, pages 253–268. USENIX Association, 2012.

[116] Sharemind—Secure Database and Application Server. `http://sharemind.cyber.ee/`. Last accessed October 8^{th}, 2014.

[117] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[118] TASTY—Tool for Automating Secure Two-partY computations. `https://code.google.com/p/tastyproject/`. Last accessed October 8^{th}, 2014.

[119] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 71–89, 2013.

[120] VIFF—the Virtual Ideal Functionality Framework. `http://viff.dk`. Last accessed October 8^{th}, 2014.

[121] David Wagner, editor. *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*. Springer, 2008.

[122] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[123] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near-practicality. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:165, 2013.

[124] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

[125] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.