A Pragmatic Introduction to Secure Multi-Party Computation

Suggested Citation: David Evans, Vladimir Kolesnikov and Mike Rosulek, *A Pragmatic Introduction to Secure Multi-Party Computation*. NOW Publishers, 2018.

David Evans
University of Virginia
evans@virginia.edu

Vladimir Kolesnikov
Georgia Institute of Technology
kolesnikov@gatech.edu

Mike Rosulek
Oregon State University
rosulekm@eecs.oregonstate.edu

now
the essence of knowledge
Boston — Delft

A Pragmatic Introduction to Secure Multi-Party Computation

David Evans[1], Vladimir Kolesnikov[2] and Mike Rosulek[3]

[1] *University of Virginia; evans@virginia.edu*
[2] *Georgia Institute of Technology; kolesnikov@gatech.edu*
[3] *Oregon State University, rosulekm@eecs.oregonstate.edu*

ABSTRACT

Secure multi-party computation (MPC) has evolved from a theoretical curiosity in the 1980s to a tool for building real systems today. Over the past decade, MPC has been one of the most active research areas in both theoretical and applied cryptography. This book introduces several important MPC protocols, and surveys methods for improving the efficiency of privacy-preserving applications built using MPC. Besides giving a broad overview of the field and the insights of the main constructions, we overview the most currently active areas of MPC research and aim to give readers insights into what problems are practically solvable using MPC today and how different threat models and assumptions impact the practicality of different approaches.

David Evans, Vladimir Kolesnikov and Mike Rosulek, *A Pragmatic Introduction to Secure Multi-Party Computation*. NOW Publishers, 2018.

Contents

1

Introduction

Secure multi-party computation (MPC) enable a group to jointly perform a computation without disclosing any participant's private inputs. The participants agree on a function to compute, and then can use an MPC protocol to jointly compute the output of that function on their secret inputs without revealing them. Since its introduction by Andrew Yao in the 1980s, multi-party computation has developed from a theoretical curiosity to an important tool for building large-scale privacy-preserving applications.

This book provides an introduction to multi-party computation for practitioners interested in building privacy-preserving applications and researchers who want to work in the area. We provide an introduction to the foundations of MPC and describe the current state of the art. Our goal is to enable readers to understand what is possible today, and what may be possible in the future, and to provide a starting point for building applications using MPC and for developing MPC protocols, implementations, tools, and applications. As such, we focus on practical aspects, and do not provide formal proofs.

The term *secure computation* is used to broadly encompass all methods for performing computation on data while keeping that data secret. A computation method may also allow participants to confirm the result is indeed the output of the function on the provided inputs, which is known as *verifiable computation*.

There are two main types of secure and verifiable computation: *outsourced computation* and *multi-party computation*. Our focus is on multi-party computation, but first we briefly describe outsourced computation to distinguish it from multi-party computation.

1.1 Outsourced Computation

In an outsourced computation, one party owns the data and wants to be able to obtain the result of computation on that data. The second party receives and stores the data in an encrypted form, performs computation on the encrypted data, and provides the encrypted results to the data owner, without learning anything about the input data, intermediate values, or final result. The data owner can then decrypt the returned results to obtain the output.

Homomorphic encryption allows operations on encrypted data, and is a natural primitive to implement outsourced computation. With *partially-homomorphic encryption* schemes, only certain operations can be performed. Several efficient partially-homomorphic encryption schemes are known (Paillier, 1999; Naccache and Stern, 1998; Boneh *et al.*, 2005). Systems built on them are limited to specialized problems that can be framed in terms of the supported operations.

To provide *fully homomorphic encryption* (FHE), it is necessary to support a Turing-complete set of operations (e.g., both addition and multiplication) so that any function can be computed. Although the goal of FHE was envisioned by Rivest *et al.* (1978), it took more than 30 years before the first FHE scheme was proposed by Gentry (2009), building on lattice-based cryptography. Although there has been much recent interest in implementing FHE schemes Gentry and Halevi (2011), Halevi and Shoup (2014), and Chillotti *et al.* (2016), building secure, deployable, scalable systems using FHE remains an elusive goal.

In their basic forms, FHE and MPC address different aspects of MPC, and as such shouldn't be directly compared. They do, however, provide similar functionalities, and there are ways to adapt FHE to use multiple keys that enables multi-party computation using FHE (Asharov *et al.*, 2012; López-Alt *et al.*, 2012; Mukherjee and Wichs, 2016). FHE offers an asymptotic communication improvement in comparison with MPC, but at the expense of computational efficiency. State-of-the-art FHE implementations (Chillotti *et al.*, 2017) are thousands of times slower than two-party and multi-party

secure computation in typical applications and settings considered in literature. Ultimately, the relative performance of FHE and MPC depends on the relative costs of computation and bandwidth. For high-bandwidth settings, such as where devices connected within a data center, MPC vastly outperforms FHE. As FHE techniques improve, and the relative cost of bandwidth over computation increases, FHE-based techniques may eventually become competitive with MPC for many applications.

We do not specifically consider outsourcing computation or FHE further in this book, but note that some of the techniques developed to improve multi-party computation also apply to FHE and outsourcing. Shan *et al.* (2017) provide a survey of work in the area of outsourcing.

1.2 Multi-Party Computation

The goal of secure multi-party computation (MPC) is to enable a group of independent data owners who do not trust each other or any common third party to jointly compute a function that depends on all of their private inputs. MPC differs from outsourced computation in that all of the protocol participants are data owners who participate in executing a protocol. Chapter 2 provides a more formal definition of MPC, and introduces the most commonly considered threat models.

Brief history of MPC. The idea of secure computation was introduced by Andrew Yao in the early 1980s (Yao, 1982). That paper introduced a general notion of secure computation, in which m parties want to jointly compute a function $f(x_1, x_2, \ldots, x_m)$ where x_i is the i^{th} party's private input. In a series of talks over the next few years (but not included in any formal publication), Yao introduced the Garbled Circuits Protocol which we describe in detail in Section 3.1. This protocol remains the basis for many of the most efficient MPC implementations.

Secure computation was primarily of only theoretical interest for the next twenty years; it was not until the 2000s that algorithmic improvements and computing costs had reached a point where it became realistic to think about building practical systems using general-purpose multi-party computation. Fairplay (Malkhi *et al.*, 2004) was the first notable implementation of a general-purpose secure computation system. Fairplay demonstrated the possibility that

a privacy-preserving program could be expressed in a high level language and compiled to executables that could be run by the data-owning participants as a multi-party protocol. However, its scalability and performance limited its use to toy programs — the largest application reported in the Fairplay paper was computing the median two sorted arrays where each party's input is ten 16-bit numbers in sorted order, involving execution of 4383 gates and taking over 7 seconds to execute (with both parties connected over a LAN). Since then, the speed of MPC protocols has improved by more than five orders of magnitude due to a combination of cryptographic, protocol, network and hardware improvements. This enabled MPC applications to scale to a wide range of interesting and important applications.

Generic and specialized MPC. Yao's garbled circuits protocol is a *generic* protocol—it can be used to compute any discrete function that can be represented as a fixed-size circuit. One important sub-area of MPC focuses on specific functionalities, such as private set intersection (PSI). For specific functionalities, there may be custom protocols that are much more efficient than the best generic protocols. Specific functionalities can be interesting in their own right, but also can be natural building blocks for use in other applications. We focus mostly on generic MPC protocols, but include discussion of private set intersection (Section 3.8.1) as a particularly useful functionality.

1.3 MPC Applications

MPC enables privacy-preserving applications where multiple mutually distrusting data owners cooperate to compute a function. Here, we highlight a few illustrative examples of privacy-preserving applications that can be built using MPC. This list is far from exhaustive, and is meant merely to give an idea of the range and scale of MPC applications.

Yao's Millionaires Problem. The toy problem that was used to introduce secure computation is not meant as a useful application. Yao (1982) introduces it simply: "Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth." That is, the goal is to compute the Boolean result of $x_1 \leq x_2$ where x_1 is the first party's private input and x_2 is the second party's private input.

Although its a toy problem, Yao's Millionaires Problem can still be useful for illustrating issues in MPC applications.

Secure auctions. The need for privacy in auctions is well understood. Indeed, it is crucial for all participants, both bidders and sellers, to be able to rely on the privacy and non-malleability of bids. *Bid privacy* requires that no player may learn any other player's bid (other than perhaps revealing the winning bid upon the completion of the auction). *Bid non-malleability* means that a player's bid may not be manipulated to generate a related bid. For example, if a party generates a bid of $\$n$, then another party should not be able to use this bid to produce a bid of $\$n + 1$. Note that bid privacy does not necessarily imply bid non-malleability — indeed it is possible to design auction protocols that would hide a bid of $\$n$ while still allowing others to generate a related bid $\$n + 1$.

These properties are crucial in many standard bidding processes. For example, a sealed bid auction is an auction where bidders submit private (sealed) bids in attempts to purchase property, selling to the highest bidder. Clearly, the first bidder's bid value must be kept secret from other potential bidders to prevent those bidders from having an unfair advantage. Similarly, bid malleability may allow a dishonest bidder Bob to present a bid just slightly over Alice's bid, again, gaining an unfair advantage. Finally, the auction itself must be conducted correctly, awarding the item to the highest bidder for the amount of their bid.

A Vickrey auction is a type of sealed-bid auction where instead paying the value of their own bid, the highest bidder wins but the price paid is the value of the second-highest bid. This type of auction gives bidders an incentive to bid their true value, but requires privacy and non-malleability of each bid, and correctness in determining the winner and price.

MPC can be used to easily achieve all these features since it is only necessary to embed the desired properties into the function used to jointly execute the auction. All the participants can verify the function and then rely on the MPC protocol to provide high confidence that the auction will be conducted confidentially and fairly.

Voting. Secure electronic voting, in a simple form, is simply computation of the addition function which tallies the vote. Privacy and non-malleability of the vote (properties discussed above in the context of auctions) are essential for similar technical reasons. Additionally, because voting is a fundamental civil process, these properties are often asserted by legislation.

As a side note, we remark that voting is an example of an application which may require properties *not covered* by the standard MPC security definitions. In particular, the property of *coercion resistance* is not standard in MPC (but can be formally expressed and achieved (Küsters *et al.*, 2012)). The issue here is the ability of voters to *prove* to a third party how they voted. If such a proof is possible (e.g., a proof might exhibit the randomness used in generating the vote, which the adversary may have seen), then voter coercion is also possible. We don't delve into the specific aspects of secure voting beyond listing it here as a natural application of MPC.

Secure machine learning. MPC can be used to enable privacy in both the inference and training phases of machine learning systems.

Oblivious model inference allows a client to submit a request to a server holding a pre-trained model, keeping the request private from the server $\mathcal{S}$ and the model private from the client C. In this setting, the inputs to the MPC are the private model from $\mathcal{S}$, and the private test input from C, and the output (decoded only for C) is the model's prediction. An example of recent work in this setting include MiniONN (Liu *et al.*, 2017), which provided a mechanism for allowing any standard neural network to be converted to an oblivious model service using a combination of MPC and homomorphic encryption techniques.

In the training phase, MPC can be used to enable a group of parties to train a model based on their combined data without exposing that data. For the large scale data sets needed for most machine learning applications, it is not feasible to perform training across private data sets as a generic many-party computation. Instead, hybrid approaches have been designed that combine MPC with homomorphic encryption (Nikolaenko *et al.*, 2013b; Gascón *et al.*, 2017) or develop custom protocols to perform secure arithmetic operations efficiently (Mohassel and Zhang, 2017). These approaches can scale to data sets containing many millions of elements.

Other applications. Many other interesting applications have been proposed for using MPC to enable privacy. A few examples include privacy-preserving network security monitoring (Burkhart *et al.*, 2010), privacy-preserving genomics (Wang *et al.*, 2015a; Jagadeesh *et al.*, 2017), private stable matching (Doerner *et al.*, 2016), contact discovery (Li *et al.*, 2013; De Cristofaro *et al.*, 2013), ad conversion (Kreuter, 2017), and spam filtering on encrypted email (Gupta *et al.*, 2017).

1.3.1 Deployments

Although MPC has seen much success as a research area and in experimental use, we are still in the early stages of deploying MPC solutions to real problems. Successful deployment of an MPC protocol to solve a problem involving independent and mutually distrusting data owners requires addressing a number of challenging problems beyond the MPC execution itself. Examples of these problems include building confidence in the system that will execute the protocol, understanding what sensitive information might be inferred from the revealed output of the MPC, and enabling decision makers charged with protecting sensitive data but without technical cryptography background to understand the security implications of participating in the MPC.

Despite these challenges, there have been several successful deployments of MPC and a number of companies now focus on providing MPC-based solutions. We emphasize that in this early stage of MPC penetration and awareness, MPC is primarily deployed as an *enabler* of data sharing. In other words, organizations are typically not seeking to use MPC to add a layer of privacy in an otherwise viable application (we believe this is yet forthcoming). Rather, MPC is used to enable a feature or an entire application, which otherwise would not be possible (or would require trust in specialized hardware), due to the value of the shared data, protective privacy legislation, or mistrust of the participants.

Danish sugar beets auction. In what is widely considered to be the first commercial application of MPC, Danish researchers collaborated with the Danish government and stakeholders to create an auction and bidding platform for sugar beet production contracts. As reported in Bogetoft *et al.* (2009), bid privacy and auction security were seen as essential for auction participants.

The farmers felt that their bids reflected their capabilities and costs, which they did not want to reveal to Danisco, the only company in Denmark that processed sugar beets. At the same time, Danisco needed to be involved in the auction as the contracts were securities directly affecting the company.

The aucion was implemented as a three-party MPC among representatives for Danisco, the farmer's association (DKS) and the researchers (SIMAP project). As explained by Bogetoft *et al.* (2009), a three party solution was selected, partly because it was natural in the given scenario, but also because it allowed using efficient information theoretic tools such as secret sharing. The project led to the formation of a company, Partisia, that uses MPC to support auctions for industries such as spectrum and energy markets, as well as related applications such as data exchange (Gallagher *et al.*, 2017).

Estonian students study. In Estonia, a country with arguably the most advanced e-government and technology awareness, alarms were raised about graduation rates of IT students. Surprisingly, in 2012, nearly 43% of IT students enrolled in the previous five years had failed to graduate. One potential explanation considered was that the IT industry was hiring too aggressively, luring students away from completing their studies. The Estonian Association of Information and Communication Technology wanted to investigate by mining education and tax records to see if there was a correlation. However, privacy legislation prevented data sharing across the Ministry of Education and the Tax Board. In fact, k-anonymity-based sharing was allowed, but it would have resulted in low-quality analysis, since many students would not have had sufficiently large groups of peers with similar qualities.

MPC provided a solution, facilitated by the Estonian company Cybernetica using their Sharemind framework (Bogdanov *et al.*, 2008a). The data analysis was done as a three-party computation, with servers representing the Estonian Information System's Authority, the Ministry of Finance, and Cybernetica. The study, reported in Cybernetica (2015) and Bogdanov (2015), found that there was no correlation between working during studies and failure to graduate on time, but that more education was correlated with higher income.

Boston wage equity study. An initiative of the City of Boston and the Boston Women's Workforce Council (BWWC) aims to identify salary inequities

across various employee gender and ethnic demographics at different levels of employment, from executive to entry-level positions. This initiative is widely supported by the Boston area organizations, but privacy concerns prevented direct sharing of salary data. In response, Boston University researchers designed and implemented a web-based MPC aggregation tool, which allowed employers to submit the salary data privately and with full technical and legal protection, for the purposes of the study.

As reported by Bestavros *et al.* (2017), MPC enabled the BWWC to conduct their analysis and produce a report presenting their findings. The effort included a series of meetings with stakeholders to convey the risks and benefits of participating in the MPC, and considered the importance of addressing usability and trust concerns. One indirect result of this work is inclusion of secure multi-party computation as a requirement in a bill for student data analysis recently introduced in the United States Senate (Wyden, 2017).

Key management. One of the biggest problems faced by organizations today is safeguarding sensitive data as it is being used. This is best illustrated using the example of authentication keys. This use case lies at the core of the product offering of Unbound Tech (Unbound Tech, 2018). Unlike other uses of MPC where the goal is to protect data owned by multiple parties from exposure, here the goal is to protect from compromise the data owned by a single entity.

To enable a secure login facility, an organization must maintain private keys. Let's consider the example of shared-key authentication, where each user has shared a randomly chosen secret key with the organization. Each time the user U authenticates, the organization's server S looks up the database of keys and retrieves U's public key sk_U, which is then used to authenticate and admit U to the network by running key exchange.

The security community has long accepted that it is nearly impossible to operate a fully secure complex system, and an adversary will be able to penetrate and stealthily take control over some of the network nodes. Such an advanced adversary, sometimes called Advanced Persistent Threat (APT), aims to quietly undermine the organization. Naturally, the most prized target for APT and other types of attackers is the key server.

MPC can play a significant role in *hardening* the key server by splitting its functionality into two (or more) hosts, say, S_1 and S_2, and secret-sharing

key material among the two servers. Now, an attacker must compromise *both* S_1 and S_2 to gain access to the keys. We can run S_1 and S_2 on two different software stacks to minimize the chance that they will both be vulnerable to the exploit available to the malware, and operate them using two different sub-organizations to minimize insider threats. Of course, routine execution does need access to the keys to provide authentication service; at the same time, key should never be reconstructed as the reconstructing party will be the target of the APT attack. Instead, the three players, S_1, S_2, and the authenticating user U, will run the authentication inside MPC, *without ever reconstructing any secrets*, thus removing the singular vulnerability and hardening the defense.

1.4 Overview

Because MPC is a vibrant and active research area, it is possible to cover only a small fraction of the most important work in this book. We mainly discuss generic MPC techniques, focusing mostly on the two-party scenario, and emphasizing a setting where all but one of the parties may be corrupted. In the next chapter, we provide a formal definition of secure multi-party computation and introduce security models that are widely-used in MPC. Although we do not include formal security proofs in this book, it is essential to have clear definitions to understand the specific guarantees that MPC provides. Chapter 3 describes several fundamental MPC protocols, focusing on the most widely-used protocols that resist any number of corruptions. Chapter 4 surveys techniques that have been developed to enable efficient implementations of MPC protocols, and Chapter 5 describes methods that have been used to provide sub-linear memory abstractions for MPC.

Chapters 3–5 target the weak semi-honest adversary model for MPC (defined in Chapter 2), in which is it assumed that all parties follow the protocol as specified. In Chapter 6, we consider how MPC protocols can be hardened to provide security against active adversaries, and Chapter 7 explores some alternative threat models that enable trade-offs between security and efficiency. We conclude in Chapter 8, outlining the trajectory of MPC research and practice, and suggesting possible directions for the future.

2

Defining Multi-Party Computation

In this chapter, we introduce notations and conventions we will use throughout, define some basic cryptographic primitives, and provide a security definition for multi-party computation. Although we will not focus on formal security proofs or complete formal definitions, it is important to have clear security definitions to understand exactly what properties protocols are designed to provide. The protocols we discuss in later chapters have been proven secure based on these definitions.

2.1 Notations and Conventions

We will abbreviate *Secure Multi-Party Computation* as *MPC*, and will use it to denote secure computation among two or more participants. The term *secure function evaluation* (SFE) is often used to mean the same thing, although it can also apply to contexts where only one party provides inputs to a function that is evaluated by an outsourced server. Because two-party MPC is an important special case, which received a lot of targeted attention, and because two-party protocols are often significantly different from the general n-party case, we will use 2PC to emphasize this setting when needed.

We assume existence of direct secure channels between each pairs of

participating players. Such channels could be achieved inexpensively through a variety of means, and are out of scope in this book.

We denote encryption and decryption of a message m under key k as $\mathsf{Enc}_k(m)$ and $\mathsf{Dec}_k(m)$. We will refer to protocol participants interchangeably also as parties or players, and will usually denote them as P_1, P_2, etc. We will denote the adversary by $\mathcal{A}$.

A negligible function $\nu : \mathbb{N} \to \mathbb{R}$ is any function that approaches zero asymptotically faster than any inverse polynomial. In other words, for any polynomial p, $\nu(n) < 1/p(n)$ for all but finitely many n.

We will denote computational and statistical security parameters by κ and σ respectively. The computational security parameter κ governs the hardness of problems that can be broken by an adversary's offline computation — e.g., breaking an encryption scheme. In practice κ is typically set to a value like 128 or 256. Even when we consider security against computationally bounded adversaries, there may be some attacks against an interactive protocol that are not made easier by offline computation. For example, the interactive nature of a protocol may give the adversary only a single opportunity to violate security (e.g., by sending a message that has a special property, like predicting the random value that an honest party will chose in the next round). The statistical security parameter σ governs the hardness of these attacks. In practice, σ is typically set to a smaller value like 40 or 80. The correct way to interpret the presence of two security parameters is that security is violated only with probability $2^{-\sigma} + \nu(\kappa)$, where ν is a negligible function *that depends on the resources of the adversary.* When we consider computationally unbounded adversaries, we omit κ and require $\nu = 0$.

We will use symbol $\in_R$ to denote uniformly random sampling from a distribution. For example we write "choose $k \in_R \{0,1\}^\kappa$" to mean that k is a uniformly chosen κ-bit long string. More generally, we write "$v \in_R D$" to denote sampling according to a probability distribution D. Often the distribution in question is the output of a randomized algorithm. We write "$v \in_R A(x)$" to denote that v is the result of running randomized algorithm A on input x.

Let D_1 and D_2 be two probability distributions indexed by a security parameter, or equivalently two algorithms that each take a security parameter as input.[1] We say that D_1 and D_2 are *indistinguishable* if, for all algorithms A

[1] In the literature, D_1 and D_2 are often referred to as an *ensemble* of distributions.

there exists a negligible function ν such that:

$$\Pr[A(D_1(n)) = 1] - \Pr[A(D_2(n)) = 1] \leq \nu(n)$$

In other words, no algorithm behaves more than negligibly differently when given inputs sampled according to D_1 vs D_2. When we consider only *non-uniform, polynomial-time* algorithms A, the definition results in *computational indistinguishability*. When we consider *all* algorithms without regard to their computational complexity, we get a definition of *statistical indistinguishability*. In that case, the probability above is bounded by the *statistical distance* (also known as total variation distance) of the two distributions, which is defined as:

$$\Delta(D_1(n), D_2(n)) = \frac{1}{2} \sum_x \left| \Pr[x = D_1(n)] - \Pr[x = D_2(n)] \right|$$

Throughout this work, we use *computational security* to refer to security against adversaries implemented by non-uniform, polynomial-time algorithms. We use *information-theoretic security* (also known as *unconditional* or *statistical* security) to mean security against arbitrary adversaries (even those with unbounded computational resources).

2.2 Basic Primitives

Here, we provide definitions of a few basic primitives we use in our presentation. Several other useful primitives are actually special cases of MPC (i.e., they are defined as MPC of specific functions). These are defined in Section 2.4.

Secret Sharing. Secret sharing is another essential primitive, which is at the core of many MPC approaches. Informally, a (t, n)-secret sharing scheme splits the secret s into n shares, such that any $t - 1$ of the shares reveal no information about s, while any t shares allow complete reconstruction of the secret s. There are many variants of possible security properties of secret sharing schemes; we provide one definition, adapted from Beimel and Chor (1993), next.

Definition 2.1. Let D be the domain of secrets and D_1 be the domain of shares. Let $\mathsf{Shr} : D \to D_1^n$ be a (possibly randomized) sharing algorithm, and $\mathsf{Rec} : D_1^k \to D$ be a reconstruction algorithm. A (t, n)-secret sharing scheme is a pair of algorithms $(\mathsf{Shr}, \mathsf{Rec})$ that satisfies these two properties:

- *Correctness*. Let $(s_1, s_2, ..., s_n) = \mathsf{Shr}(s)$. Then,

$$\Pr[\forall k \geq t, Rec(s_{i_1}, ...s_{i_k}) = s] = 1.$$

- *Perfect Privacy*. Any set of shares of size less than t does not reveal anything about the secret in the information theoretic sense. More formally, for any two secrets $a, b \in D$ and any possible vector of shares $\boldsymbol{v} = v_1, v_2, ..., v_k$, such that $k < t$,

$$\Pr[\boldsymbol{v} = \mathsf{Shr}(a)|_k] = \Pr[\boldsymbol{v} = \mathsf{Shr}(b)|_k],$$

  where $|_k$ denotes appropriate projection on a subspace of k elements.

In many of our discussions we will use (n, n)-secret sharing schemes, where all n shares are necessary and sufficient to reconstruct the secret.

Random Oracle. Random Oracle (RO) is a heuristic model for the security of hash functions, introduced by Bellare and Rogaway (1993). The idea is to treat the hash function as a public, idealized random function. In the random oracle model, all parties have access to the public function $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$, implemented as a stateful oracle. On input string $x \in \{0,1\}^*$, H looks up its history of calls. If $H(x)$ had never been called, H chooses a random $r_x \in \{0,1\}^\kappa$, remembers the pair x, r_x and returns r_x. If $H(x)$ had been called before, H returns r_x. In this way, the oracle realizes a randomly-chosen function $\{0,1\}^* \rightarrow \{0,1\}^\kappa$.

The random oracle model is a heuristic model, because it captures only those attacks that treat the hash function H as a black-box. It deviates from reality in that it models a public function (e.g., a standardized hash function like SHA-256) as an inherently random object. In fact, it is possible to construct (extremely contrived) schemes that are secure in the random oracle model, but which are insecure whenever H is instantiated by *any* concrete function (Canetti *et al.*, 1998).

Despite these shortcomings, the random oracle model is often considered acceptable for practical applications. Assuming a random oracle often leads to significantly more efficient constructions. In this work we will be careful to state when a technique relies on the random oracle model.

2.3 Security of Multi-Party Computation

Informally, the goal of MPC is for a group of participants to learn the correct output of some agreed-upon function applied to their private inputs without revealing anything else. We now provide a more formal definition to clarify the security properties MPC aims to provide. First, we present the *real-ideal paradigm* which forms the conceptual core of defining security. Then we discuss two different adversary models commonly used for MPC. Finally, we discuss issues of composition—namely, whether security preserved in the natural way when a secure protocol invokes another subprotocol.

2.3.1 Real-Ideal Paradigm

A natural way to define security is to come up with a kind of a "laundry list" of things that constitute a violation of security. For example, the adversary should not be able to learn a certain predicate of another party's input, the adversary should not be able to induce impossible outputs for the honest parties, and the adversary should not be able to make its inputs depend on honest parties' inputs. Not only is this a tedious approach, but it is cumbersome and error-prone. It is not obvious when the laundry list could be considered complete.

The real-ideal paradigm avoids this pitfall completely by introducing an incredibly clear "ideal world" that implicitly captures all security guarantees, and defining security in relation to this ideal world. The definition of probabilistic encryption by Goldwasser and Micali (1984) is widely considered to be the first instance of using this approach to define and prove security, although they used different terminology.

Ideal World. In the ideal world, the parties securely compute the function $\mathcal{F}$ by privately sending their inputs to a completely trusted party $\mathcal{T}$, referred to as the *functionality*. Each party P_i has an associated input x_i, which is sends to $\mathcal{T}$, which simply computes $\mathcal{F}(x_1, \ldots, x_n)$ and returns the result to all parties. Often we will make a distinction between $\mathcal{F}$ as a trusted party (functionality) and the circuit C that such a party computes on the private inputs.

We can imagine an adversary attempting to attack the ideal-world interaction. An adversary can take control over any of the parties P_i, but not $\mathcal{T}$ (that is the sense in which $\mathcal{T}$ is described as a *trusted* party). The simplicity

of the ideal world makes it easy to understand the effect of such an attack. Considering our previous laundry list: the adversary clearly learns no more than $\mathcal{F}(x_1, \ldots, x_n)$ since that is the only message it receives; the outputs given to the honest parties are all consistent and legal; the adversary's choice of inputs is independent of the honest parties'.

Although the ideal world is easy to understand, the presence of a fully-trusted third party makes it imaginary. We use the ideal world as a benchmark against which to judge the security of an actual protocol.

Real World. In the real world, there is no trusted party. Instead, all parties communicate with each other using a protocol. The protocol π specifies for each party P_i a "next-message" function π_i. This function takes as input a security parameter, the party's private input x_i, a random tape, and the list of messages P_i has received so far. Then, π_i outputs either a next message to send along with its destination, or else instructs the party to terminate with some specific output.

In the real world, an adversary can corrupt parties—corruption at the beginning of the protocol is equivalent to the original party being an adversary. Depending on the threat model (discussed next), corrupt parties may either follow the protocol as specified, or deviate arbitrarily in their behavior.

Intuitively speaking, the real world protocol π is considered secure if any effect that an adversary can achieve in the real world can also be achieved by a corresponding adversary in the ideal world. Put differently, the goal of a protocol is to provide security in the real world (given a set of assumptions) that is equivalent to that in the ideal world.

2.3.2 Semi-Honest Security

A *semi-honest* adversary is one who corrupts parties but follows the protocol as specified. In other words, the corrupt parties run the protocol honestly but they may try to learn as much as possible from the messages they receive from other parties. Note that this may involve several colluding corrupt parties pooling their views together in order to learn information. Semi-honest adversaries are also considered *passive* in that they cannot take any actions other than attempting to learn private information by observing a view of a protocol execution. Semi-honest adversaries are also commonly called *honest-but-curious*.

The *view* of a party consists of its private input, its random tape, and the list of all messages received during the protocol. The view of an adversary consists of the combined views of all corrupt parties. Anything an adversary learns from running the protocol must be an efficiently computable function of its view. That is, without loss of generality we need only consider an "attack" in which the adversary simply outputs its entire view.

Following the real-ideal paradigm, security means that such an "attack" can also be carried out in the ideal world. That is, for a protocol to be secure, it must be possible in the ideal world to generate something indistinguishable from the real world adversary's view. Note that the adversary's view in the ideal world consists of nothing but inputs sent to $\mathcal{T}$ and outputs received from $\mathcal{T}$. So, an ideal-world adversary must be able to use this information to generate what looks like a real-world view. We refer to such an ideal-world adversary as a *simulator*, since it generates a "simulated" real-world view while in the ideal-world itself. Showing that such a simulator exists proves that there is nothing an adversary can accomplish in the real world that could not also be done in the ideal world.

More formally, let π be a protocol and $\mathcal{F}$ be a functionality. Let C be the set of parties that are corrupted, and let Sim denote a simulator algorithm. We define the following distributions of random variables:

- $\mathsf{Real}_\pi(\kappa, C; x_1, \ldots, x_n)$: run the protocol with security parameter κ, where each party P_i runs the protocol honestly using private input x_i. Let V_i denote the final view of party P_i, and let y_i denote the final output of party P_i.
  Output $\{V_i \mid i \in C\}, (y_1, \ldots, y_n)$.

- $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa, C; x_1, \ldots, x_n)$: Compute $(y_1, \ldots, y_n) \leftarrow \mathcal{F}(x_1, \ldots, x_n)$.
  Output $\mathsf{Sim}(C, \{(x_i, y_i) \mid i \in C\}), (y_1, \ldots, y_n)$.

A protocol is secure against semi-honest adversaries if the corrupted parties in the real world have views that are indistinguishable from their views in the ideal world:

Definition 2.2. A protocol π *securely realizes* $\mathcal{F}$ *in the presence of semi-honest adversaries* if there exists a simulator Sim such that, for every subset of corrupt parties C and all inputs $x_1, \ldots, x_n$, the distributions

$$\mathsf{Real}_\pi(\kappa, C; x_1, \ldots, x_n)$$

and

$$\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa, C; x_1, \ldots, x_n)$$

are indistinguishable (in κ).

In defining Real and Ideal we have included the outputs of all parties, even the honest ones. This is a way of incorporating a correctness condition into the definition. In the case that no parties are corrupt ($C = \emptyset$), the output of Real and Ideal simply consist of all parties' outputs in the two interactions. Hence, the security definition implies that protocol gives outputs which are distributed just as their outputs from the ideal functionality (and this is true even when $\mathcal{F}$ is randomized). Because the distribution of $y_1, \ldots, y_n$ in Real does not depend on the set C of corrupted parties (no matter who is corrupted, the parties all run honestly), it is not strictly necessary to include these values in the case of $C \neq \emptyset$, but we choose to include it to have a unified definition.

The semi-honest adversary model may at first glance seem exceedingly weak—simply reading and analyzing received messages barely even seems like an attack at all! It is reasonable to ask why such a restrictive adversary model is worth considering at all. In fact, achieving semi-honest security is far from trivial and, importantly, semi-honest protocols often serve as a basis for protocols in more robust settings with powerful attackers. Additionally, many realistic scenarios do correspond to semi-honest attack behavior. One such example is computing with players who are trusted to act honestly, but cannot fully guarantee that their storage might not be compromised in the future.

2.3.3 Malicious Security

A *malicious* (also known as *active*) may instead cause corrupted parties to deviate arbitrarily from the prescribed protocol in an attempt to violate security. A malicious adversary has all the powers of a semi-honest one in analyzing the protocol execution, but may also take any actions it wants during protocol execution. Note that this subsumes an adversary that can control, manipulate, and arbitrarily inject messages on the network (even through throughout this book we assume direct secure channels between each pair of parties). As before, security in this setting is defined in comparison to the ideal world, but there are two important additions to consider:

Effect on honest outputs. When the corrupt parties deviate from the protocol, there is now the possibility that honest parties' outputs will be affected. For example, imagine an adversary that causes two honest parties to output different things while in the ideal world all parties get identical outputs. This condition is somewhat trivialized in the previous definition — while the definition does compare real-world outputs to ideal-world outputs, these outputs have no dependence on the adversary (set of corrupted parties). Furthermore, we can/should make no guarantees on the final outputs of corrupt parties, only of the honest parties, since a malicious party can output whatever it likes.

Extraction. Honest parties follow the protocol according to a well-defined input, which can be given to $\mathcal{T}$ in the ideal world as well. In contrast, the input of a malicious party is not well-defined in the real world, which leads to the question of what input should be given to $\mathcal{T}$ in the ideal world. Intuitively, in a secure protocol, whatever an adversary can do in the real world should also be achievable in the ideal world by *some suitable choice of inputs* for the corrupt parties. Hence, we leave it to the simulator to choose inputs for the corrupt parties. This aspect of simulation is called *extraction*, since the simulator extracts an effective ideal-world input from the real-world adversary that "explains" the input's real-world effect. In most constructions, it is sufficient to consider *black-box* simulation, where the simulator is given access only to the oracle implementing the real-world adversary, and not its code.

When $\mathcal{A}$ denotes the adversary program, we write $\mathsf{corrupt}(\mathcal{A})$ to denote the set of parties that are corrupted, and use $\mathsf{corrupt}(\mathsf{Sim})$ for the set of parties that are corrupted by the ideal adversary, Sim. As we did for the semi-honest security definition, we define distributions for the real world and ideal world, and define a secure protocol as one that makes those distributions indistinguishable:

- $\mathsf{Real}_{\pi,\mathcal{A}}(\kappa; \{x_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\})$: run the protocol on security parameter κ, where each honest party P_i (for $i \notin \mathsf{corrupt}(\mathcal{A})$) runs the protocol honestly using given private input x_i, and the messages of corrupt parties are chosen according to $\mathcal{A}$ (thinking of $\mathcal{A}$ as a protocol next-message function for a collection of parties). Let y_i denote the output of each

honest party P_i and let V_i denote the final view of party P_i.
Output $(\{V_i \mid i \in \mathsf{corrupt}(\mathcal{A})\}, \{y_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\})$.

- $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa; \{x_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\})$: Run Sim until it outputs a set of inputs $\{x_i \mid i \in \mathsf{corrupt}(\mathcal{A})\}$. Compute $(y_1, \ldots, y_n) \leftarrow \mathcal{F}(x_1, \ldots, x_n)$. Then, give $\{y_i \mid i \in \mathsf{corrupt}(\mathcal{A})\}$ to Sim.[2] Let V^* denote the final output of Sim (a set of simulated views).
Output $(V^*, \{y_i \mid i \notin \mathsf{corrupt}(\mathsf{Sim})\})$.

Definition 2.3. A protocol π *securely realizes* $\mathcal{F}$ *in the presence of malicious adversaries* if for every real-world adversary $\mathcal{A}$ there exists a simulator Sim with $\mathsf{corrupt}(\mathcal{A}) = \mathsf{corrupt}(\mathsf{Sim})$ such that, for all inputs for honest parties $\{x_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\}$, the distributions

$$\mathsf{Real}_{\pi,\mathcal{A}}(\kappa; \{x_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\})$$

and

$$\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa; \{x_i \mid i \notin \mathsf{corrupt}(\mathsf{Sim})\})$$

are indistinguishable (in κ).

Note that the definition quantifies only over the inputs of honest parties $\{x_i \mid i \notin \mathsf{corrupt}(\mathcal{A})\}$. The interaction Real does not consider the corrupt parties to have any inputs, and the inputs of the corrupt parties in Sim is only determined indirectly (by the simulator's choice of what to send to $\mathcal{F}$ on the corrupt parties' behalf). While it would be possible to also define inputs for corrupt parties in the real world, such inputs would merely be "suggestions" since corrupt parties could choose to run the protocol on any other input (or behave in a way that is inconsistent with all inputs).

Reactive functionalities. In the ideal world, the interaction with the functionality consists of just a single round: inputs followed by outputs. It is possible to generalize the behavior of $\mathcal{F}$ so that it interacts with the parties over many rounds of interaction, keeping its own private internal state between rounds. Such functionalities are called *reactive*.

[2] To be more formal, we can write the simulator Sim as a pair of algorithms $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ which capture this two-phase process. Sim_1 (on input κ) outputs $\{x_i \mid i \in \mathsf{corrupt}(\mathcal{A})\}$ and arbitrary internal state Σ. Then Sim_2 takes input Σ and $\{y_i \mid i \in \mathsf{corrupt}(\mathcal{A})\}$, and gives output V^*.

One example of a reactive functionality is as the dealer in a poker game. The functionality must keep track of the state of all cards, taking input commands and giving outputs to all parties in many rounds.

Another example is an extremely common functionality called *commitment*. This functionality accepts a bit b (or more generally, a string) from P_1 and gives output "committed" to P_2, while internally remembering b. At some later time, if P_1 sends the command "reveal" (or "open") to the functionality, it gives b to P_2.

Security with abort. In any message-based two-party protocol, one party will learn the final output before the other. If that party is corrupt and malicious, they may simply refuse to send the last message to the honest party and thereby prevent the honest party from learning the output. However, this behavior is incompatible with our previous description of the ideal world. In the ideal world, if corrupt parties receive output from the functionality then all parties do. This property is called *output fairness* and not all functions can be computed with this property (Cleve, 1986; Gordon *et al.*, 2008; Asharov *et al.*, 2015a).

Typical results in the malicious setting provide a weaker property known as *security with abort*, which requires slightly modifying the ideal functionality as follows. First, the functionality is allowed to know the identities of the corrupt parties. The functionality's behavior is modified to be slightly reactive: after all parties have provided input, the functionality computes outputs and delivers the outputs to the *corrupt parties only*. Then the functionality awaits either a "deliver" or "abort" command from the corrupted parties. Upon receiving "deliver", the functionality delivers the outputs to all the honest parties. Upon receiving "abort", the functionality delivers an abort output ($\perp$) to all the honest parties.

In this modified ideal world, an adversary is allowed to learn the output before the honest parties and to prevent the honest parties from receiving any output. It is important to note, however, that whether an honest party aborts can depend only on the corrupt party's outputs. In particular, it would violate security if the honest party's abort probability to depended on its own input.

Usually the possibility of blocking outputs to honest parties is not written explicitly in the description of the functionality. Instead, it is generally understood that when discussing security against malicious adversaries, the

adversary has control over output delivery to honest parties and output fairness is not expected.

Adaptive corruption. We have defined both the real and ideal worlds so that the identities of the corrupted parties are fixed throughout the entire interaction. This provides what is known as security against *static corruption*. It is also possible to consider scenarios where an adversary may choose which parties to corrupt during the protocol execution, possibly based on what it learns during the interaction. This behavior is known as *adaptive corruption*.

Security against adaptive corruption can be modeled in the real-ideal paradigm, by allowing the adversary to issue a command of the form "corrupt P_i". In the real world, this results in the adversary learning the current view (including private randomness) of P_i and subsequently taking over control of its protocol messages. In the ideal world, the simulator learns only the input and outputs of the party upon corruption, and must use this information to generate simulated views. Of course, the views of parties are correlated (if P_i sends a message to P_j, then that message is included in both parties' views). The challenge of adaptive security is that the simulator must produce views piece-by-piece. For example, the simulator may be asked to produce a view of P_i when that party is corrupted. Any messages sent by P_j to P_i must be simulated without knowledge of P_j's private input. Later, the simulator might be asked to provide a view of P_j (including its private randomness) that "explains" its protocol messages as somehow consistent with whatever private input it had.

In this work we consider only static corruption, following the vast majority of work in this field.

2.3.4 Hybrid Worlds and Composition

In the interest of modularity, it is often helpful to design protocols that make use of other ideal functionalities. For example, we may design a protocol π that securely realizes some functionality $\mathcal{F}$, where the parties of π also interact with another functionality $\mathcal{G}$ in addition to sending messages to each other. Hence, the real world for this protocol includes $\mathcal{G}$, while the ideal world (as usual) includes only $\mathcal{F}$. We call this modified real world the *$\mathcal{G}$-hybrid world*.

A natural requirement for a security model is *composition*: if π is a $\mathcal{G}$-hybrid protocol that securely realizes $\mathcal{F}$ (i.e., parties in π send messages and also interact with an ideal $\mathcal{G}$), and ρ is a protocol that securely realizes $\mathcal{G}$, then composing π and ρ in the natural way (replacing every invocation of $\mathcal{G}$ with a suitable invocation of ρ) also results in a secure protocol for $\mathcal{F}$. While we have not defined all of the low-level details of a security model for MPC, it may be surprising that some very natural ways of specifying the details *do not guarantee composability* of secure protocols!

The standard way of achieving guaranteed composition is to use the *universal composability* (UC) framework from Canetti (2001). The UC framework augments the security model that we have sketched here with an additional entity called the *environment*, which is included in both the ideal and real worlds. The purpose of the environment is to capture the "context" in which the protocol executes (e.g., the protocol under consideration is invoked as a small step in some larger calling protocol). The environment chooses inputs for the honest party and receives their outputs. It also may interact arbitrarily with the adversary.

The same environment is included in the real and ideal worlds, and its "goal" is to determine whether it is instantiated in the real or ideal world. Previously we defined security by requiring certain real and ideal views to be indistinguishable. In this setting, we can also absorb any distinguisher of these views into the environment itself. Hence, without loss of generality, the environment's final output can be just a single bit which can be interpreted as the environment's "guess" of whether it is instantiated in the real or ideal world.

Next, we define the real and ideal executions, where Z is an environment:

- $\mathsf{Real}_{\pi,\mathcal{A},Z}(\kappa)$: run an interaction involving adversary $\mathcal{A}$ and environment Z. When Z generates an input for an honest party, the honest party runs protocol π, and gives its output to Z. Finally, Z outputs a single bit, which is taken as the output of $\mathsf{Real}_{\pi,\mathcal{A},Z}(\kappa)$.

- $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},Z}(\kappa)$: run an interaction involving adversary (simulator) Sim and environment Z. When Z generates an input for an honest party, the input is passed directly to functionality $\mathcal{F}$ and the corresponding output is given to Z (on behalf of that honest party). The output bit of Z is taken as the output of $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},Z}(\kappa)$.

Definition 2.4. A protocol π *UC-securely realizes* $\mathcal{F}$ if for all real-world adversaries $\mathcal{A}$ there exists a simulator Sim with $\mathsf{corrupt}(\mathcal{A}) = \mathsf{corrupt}(\mathsf{Sim})$ such that, for all environments Z:

$$\left| \Pr[\mathsf{Real}_{\pi,\mathcal{A},Z}(\kappa) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},Z}(\kappa) = 1] \right| \text{ is negligible (in } \kappa).$$

Since the definition quantifies over all environments, we can always consider absorbing the adversary $\mathcal{A}$ into the environment Z, so that what is left over is the so-called "dummy adversary" (which simply forwards protocol messages as instructed by Z).

In other (non UC-composable) security models, the ideal-world adversary (simulator) can depend arbitrarily on the real-world adversary. In particular, the simulator can do things like internally run the adversary and repeatedly rewind that adversary to a previous internal state. Many protocols are proven in these weaker model where the composability may be restricted. Sequential composition security (i.e., security for protocols which call functionalities in a sequential manner) holds for all protocols discussed in this book.

In the UC model such rewinding is not possible since the adversary can be assumed to be absorbed into the environment, and the simulator is not allowed to depend on the environment. Rather, the simulator must be a *straight-line simulator*: whenever the environment wishes to send a protocol message, the simulator must reply immediately with a simulated response. A straight-line simulator must generate the simulated transcript in one pass, whereas the previous definitions allowed for the simulated transcript or view to be generated without any restrictions. Assuming the other primitives (e.g., OT and commitments) used in these protocols provide UC-security, the malicious secure protocols described in this book are all UC-secure.

2.4 Specific Functionalities of Interest

Here, we define several functionalities that have been identified as particularly useful building blocks for building MPC protocols.

Oblivious Transfer. Oblivious Transfer (OT) is an essential building block for secure computation protocols. It is theoretically equivalent to MPC as shown by Kilian (1988): given OT, one can build MPC without any additional assumptions, and, similarly, one can directly obtain OT from MPC.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. $\mathcal{S}$ has input secrets $x_0, x_1 \in \{0,1\}^n$, and $\mathcal{R}$ has a selection bit $b \in \{0,1\}$.

FUNCTIONALITY:

- $\mathcal{R}$ receives x_b, $\mathcal{S}$ receives $\perp$.

Figure 2.1: 1-out-of-2 OT functionality $\mathcal{F}^{\text{OT}}$.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. Length of committed string n.

FUNCTIONALITY:

- $\mathcal{S}$ sends a string $s \in \{0,1\}^n$ to $\mathcal{F}^{\text{Comm}}$. $\mathcal{F}^{\text{Comm}}$ sends **committed** to $\mathcal{R}$.
- At some later time, $\mathcal{S}$ sends **open** to $\mathcal{F}^{\text{Comm}}$. $\mathcal{F}^{\text{Comm}}$ sends s to $\mathcal{R}$.

Figure 2.2: Commitment functionality $\mathcal{F}^{\text{Comm}}$.

The standard definition of 1-out-of-2 OT involves two parties, a Sender $\mathcal{S}$ holding two secrets x_0, x_1 and a receiver $\mathcal{R}$ holding a choice bit $b \in \{0,1\}$. OT is a protocol allowing $\mathcal{R}$ to obtain x_b while learning nothing about the "other" secret x_{1-b}. At the same time, $\mathcal{S}$ does not learn anything at all. More formally:

Definition 2.5. A 1-out-of-2 OT is a cryptographic protocol securely implementing the functionality $\mathcal{F}^{\text{OT}}$ of Figure 2.1.

Later in this chapter we define more formally what it means for a cryptographic protocol to be *secure* in this setting.

Many variants of OT may be considered. A natural variant is 1-out-of-k OT, in which $\mathcal{S}$ holds k secrets, and $\mathcal{R}$ has a choice selector from $[0, ..., k-1]$. We discuss protocols for implementing OT efficiently in Section 3.7.

Commitment. Commitment is a fundamental primitive in many cryptographic protocols. A commitment scheme allows a sender to commit to a

PARAMETERS:

1. Two parties: Prover $\mathcal{P}$ and Verifier $\mathcal{V}$.

FUNCTIONALITY:

- $\mathcal{P}$ sends (C, x) to $\mathcal{F}^{\text{zk}}$, where $C : \{0,1\}^n \to \{0,1\}$ is a Boolean circuit with 1 output bit, and $x \in \{0,1\}^n$. If $C(x) = 1$ then $\mathcal{F}^{\text{zk}}$ sends (**proven**, C) to $\mathcal{V}$. Otherwise, it sends $\perp$ to $\mathcal{V}$.

Figure 2.3: Zero-knowledge proof functionality $\mathcal{F}^{\text{zk}}$.

secret value, and reveal it at some later time to a receiver. The receiver should learn nothing about the committed value before it is revealed by the sender (a property refered to as *hiding*), while the sender should not be able to change its choice of value after committing (the *binding* property).

Commitment is rather simple and inexpensive in the random oracle model. To commit to x, simply choose a random value $r \in_R \{0,1\}^\kappa$ and publish the value $y = H(x\|r)$. To later reveal, simply announce x and r.

Definition 2.6. *Commitment* is a cryptographic protocol implementing the functionality $\mathcal{F}^{\text{Comm}}$ of Figure 2.2.

Zero-Knowledge Proof. A zero-knowledge (ZK) proof allows a prover to convince a verifier that it knows x such that $C(x) = 1$, without revealing any further information about x. Here C is a public predicate.

As a simple example, suppose G is a graph and that Alice knows a 3-coloring χ for G. Then Alice can use a ZK proof to convince Bob that G is 3-colorable. She constructs a circuit C_G that interprets its input as an encoding of a 3-coloring and checks whether it is a legal 3-coloring of G. She uses (C_G, χ) as input to the ZK proof. From Bob's point of view, he receives output (**proven**, C_G) if and only if Alice was able to provide a valid 3-coloring of G. At the same time, Alice knows that Bob learned nothing about her 3-coloring χ other than the fact that *some* legal χ exists.

Definition 2.7. A *zero-knowledge proof* is a cryptographic protocol implementing the functionality $\mathcal{F}^{\text{zk}}$ of Figure 2.3.

There are several variants of ZK proofs identified in the literature. Our specific variant is more precisely a zero-knowledge *argument of knowledge*. The distinctions between these variants are not crucial for the level of detail we explore in this book.

2.5 Further Reading

The real-ideal paradigm was first applied in the setting of MPC by Goldwasser *et al.* (1985), for the special case of zero-knowledge. Shortly thereafter the definition was generalized to arbitrary MPC by Goldreich *et al.* (1987). These definitions contain the important features of the real-ideal paradigm, but resulted in a notion of security (against malicious adversaries) that was not preserved under composition. In other words, a protocol could be secure according to these models *when executed in isolation*, but may be totally insecure when two protocol instances are run concurrently.

The definition of security that we have sketched in this book is the Universal Composition (UC) framework of Canetti (2001). Protocols proven secure in the UC framework have the important composition property described in Section 2.3.4, which in particular guarantees security of a protocol instance no matter what other protocols are executing concurrently. While the UC framework is the most popular model with this property, there are other models with similar guarantees (Pfitzmann and Waidner, 2000; Hofheinz and Shoup, 2011). The details of all such security models are extensive and subtle. However, a significantly simpler model is presented by Canetti *et al.* (2015), which is equivalent to the full UC model for the vast majority of cases. Some of the protocols we describe are secure in the random oracle model. Canetti *et al.* (2014) describe how to incorporate random oracles into the UC framework.

Our focus in this book is on the most popular security notions — namely, semi-honest security and malicious security. The literature contains many variations on these security models, and some are a natural fit for real-world applications. We discuss some alternative security models in Chapter 7.

3

Fundamental MPC Protocols

In this chapter we survey several important MPC approaches, covering the main protocols and presenting the intuition behind each approach.

All of the approaches discussed can be viewed as a form of computing under encryption, or, more specifically, as secret-sharing the input data and computing on the shares. For example, an encryption $\mathsf{Enc}_k(m)$ of a message m with a key k can be seen as secret-sharing m, where one share is k and the other is $\mathsf{Enc}_k(m)$. We present several fundamental protocols illustrating a variety of generic approaches to secure computation, as summarized in Figure 3.1. All of the protocols of this section target the semi-honest adversary model (Section 2.3.2). We discuss malicious-secure variants in Chapter 6. All of these

protocol	# parties	# rounds	circuit
Yao's GC (Section 3.1)	2	constant	Boolean
GMW (Section 3.2)	many	circuit depth	Boolean or arithmetic
BGW (Section 3.3)	many	circuit depth	Boolean or arithmetic
BMR (Section 3.5)	many	constant	Boolean
GESS (Section 3.6)	2	constant	Boolean *formula*

Figure 3.1: Summary of semi-honest MPC protocols discussed in this chapter.

protocols build on oblivious transfer, which we discuss how to implement efficiently in Section 3.7.

3.1 Yao's Garbled Circuits Protocol

Yao's Garbled Circuits protocol (GC) is the most widely known and celebrated MPC technique. It is usually seen as best-performing, and many of the protocols we cover build on Yao's GC. While not having the best known communication complexity, it runs in constant rounds and avoids the costly latency associated with approaches, such as GMW (described in Section 3.2), where the number of communication rounds scales with the circuit depth.

3.1.1 GC Intuition

The main idea behind Yao's GC approach is quite natural. Recall, we wish to evaluate a given function $\mathcal{F}(x, y)$ where party P_1 holds $x \in X$ and P_2 holds $y \in Y$. Here X and Y are the respective domains for the inputs of P_1 and P_2.

Function as a look-up table. First, let's consider a function $\mathcal{F}$ for which the input domain is small and we can efficiently enumerate all possible input pairs, (x, y). The function $\mathcal{F}$ can be represented as a look-up table T, consisting of $|X| \cdot |Y|$ rows, $T_{x,y} = \langle \mathcal{F}(x, y) \rangle$. The output of $\mathcal{F}(x, y)$ is obtained simply by retrieving $T_{x,y}$ from the corresponding row.

This gives us an alternative (and much simplified!) view of the task at hand. Evaluating a look-up table can be done as follows. P_1 will encrypt T by assigning a randomly-chosen strong key to *each* possible input x and y. That is, for each $x \in X$ and each $y \in Y$, P_1 will choose $k_x \in_R \{0,1\}^\kappa$ and $k_y \in_R \{0,1\}^\kappa$. It will then encrypt T by encrypting each element $T_{x,y}$ of T with *both* keys k_x and k_y, and send the encrypted (and randomly permuted!) table $\langle \mathsf{Enc}_{k_x,k_y}(T_{x,y}) \rangle$ to P_2.

Now our task is to enable P_2 to decrypt (only) the entry $T_{x,y}$ corresponding to players' inputs. This is done by having P_1 send to P_2 the keys k_x and k_y. P_1 knows its input x, and hence simply sends key k_x to P_2. The key k_y is sent to P_2 using a 1-out-of-$|Y|$ Oblivious Transfer (Section 2.4). Once P_2 receives k_x and k_y, it can obtain the output $\mathcal{F}(x, y)$ by decrypting $T_{x,y}$ using those keys. Importantly, no other information is obtained by P_2. This is because P_2 only

has a single pair of keys, which can only be used to open (decrypt) a single table entry. We stress that, in particular, it is important that neither partial key, k_x or k_y, by itself can be used to obtain partial decryptions or even determine whether the partial key was used in the obtaining a specific encryption.[1]

Point-and-Permute. A careful reader may wonder how P_2 knows which row of the table T to decrypt, as this information is dependent on the inputs of both parties, and, as such, is sensitive.

The simplest way to address this is to encode some additional information in the encrypted elements of T. For example, P_1 may append a string of σ zeros to each row of T. Decrypting the wrong row with high probability ($p = \frac{1}{2^\sigma}$) will produce an entry which will not end with σ zeros, and hence will be rejected by P_2.

While the above approach works, it is inefficient for P_2, who expects to need to decrypt half of the rows of the table T. A much better approach, often called *point-and-permute*,[2] was introduced by Beaver *et al.* (1990). The idea is to interpret part of the key (namely, the last $\lceil \log |X| \rceil$ bits of the first key and the last $\lceil \log |Y| \rceil$ bits of the second key) as a pointer to the permuted table T, where the encryption will be placed. To avoid collisions in table row allocation, P_1 must ensure that the pointer bits don't collide within the space of keys k_x or within the space of k_y; this can be done in a number of ways. Finally, strictly speaking, key size must be maintained to achieve the corresponding level of security. As a consequence, rather than viewing key bits as a pointer, players will append the pointer bits to the key and maintain the desired key length.

In the subsequent discussions, we assume that the evaluator knows which row to decrypt. In protocol presentations, we may or may not explicitly include the point-and-permute component, depending on context.

[1] Consider a counter-example. Suppose P_2 was able to determine that a key k_x that it received from P_1 was used in encrypting r_x rows. Because some input combinations may be invalid, the encrypted look-up table T may have a unique number of rows relying on k_x, which will reveal x to P_1, violating the required security guarantees.

[2] This technique was not given a name by Beaver *et al.* (1990). Rather, this name came to be widely used by the community around 2010, as GC research progressed and need for a name arose. This technique is different from the *permute and point* technique introduced and coined in the information-theoretic garbled circuit construction of Kolesnikov (2005), which we discuss in Section 3.6.

Managing look-up table size. Clearly, the above solution is inefficient as it scales linearly with the domain size of $\mathcal{F}$. At the same time, for small functions, such as those defined by a single Boolean circuit gate, the domain has size 4, so using a look-up table is practical.

The next idea is to represent $\mathcal{F}$ as a Boolean circuit C and evaluate each gate using look-up tables of size 4. As before, P_1 generates keys and encrypts look-up tables, and P_2 applies decryption keys without knowing what each key corresponds to. However, in this setting, we cannot reveal the plaintext output of intermediate gates. This can be hidden by making the gate output also a key whose corresponding value is unknown to the evaluator, P_2.

For each wire w_i of C, P_1 assigns two keys k_i^0 and k_i^1, corresponding to the two possible values on the wire. We will refer to these keys as *wire labels*, and to the plaintext wire values simply as *wire values*. During the execution, depending on the inputs to the computation, each wire will be associated with a specific plaintext value and a corresponding wire label, which we will call *active value* and *active label*. We stress that the evaluator can know only the active *label*, but not its corresponding *value*, and not the *inactive* label.

Then, going through C, for each gate G with input wires w_i and w_j, and output wire w_t, P_1 builds the following encrypted look-up table:

$$T_G = \begin{pmatrix} \mathsf{Enc}_{k_i^0,k_j^0}(k_t^{G(0,0)}) \\ \mathsf{Enc}_{k_i^0,k_j^1}(k_t^{G(0,1)}) \\ \mathsf{Enc}_{k_i^1,k_j^0}(k_t^{G(1,0)}) \\ \mathsf{Enc}_{k_i^1,k_j^1}(k_t^{G(1,1)}) \end{pmatrix}$$

For example, if G is an AND gate, the look-up table will be:

$$T_G = \begin{pmatrix} \mathsf{Enc}_{k_i^0,k_j^0}(k_t^0) \\ \mathsf{Enc}_{k_i^0,k_j^1}(k_t^0) \\ \mathsf{Enc}_{k_i^1,k_j^0}(k_t^0) \\ \mathsf{Enc}_{k_i^1,k_j^1}(k_t^1) \end{pmatrix}$$

Each cell of the look-up table encrypts the *label corresponding to the output computed by the gate*. Crucially, this allows the evaluator P_2 to obtain the intermediate active labels on internal circuit wires and use them in the evaluation of $\mathcal{F}$ under encryption without ever learning their semantic value.

P_1 permutes the entries in each of the look-up tables (usually called *garbled tables* or *garbled gates*), and sends all the tables to P_2. Additionally, P_1 sends (only) the active labels of all wires corresponding to the input values to P_2. For input wires belonging to P_1's inputs to $\mathcal{F}$, this is done simply by sending the wire label keys. For wires belonging to P_2's inputs, this is done via 1-out-of-2 Oblivious Transfer.

Upon receiving the input keys and garbled tables, P_2 proceeds with the evaluation. As discussed above, P_2 must be able to decrypt the correct row of each garbled gate. This is achieved by the point-and-permute technique described above. In our case of a 4-row garbled table, the point-and-permute technique is particularly simple and efficient — one pointer bit is needed for each input, so there are two total pointer bits added to each entry in the garbled table. Ultimately, P_2 completes evaluation of the garbled circuit and obtains the keys corresponding to the output wires of the circuit. These could be sent to P_1 for decryption, thus completing the private evaluation of $\mathcal{F}$.

We note that a round of communication may be saved and sending the output labels by P_2 for decryption by P_1 can be avoided. This can be done simply by P_1 including the decoding tables for the output wires with the garbled circuit it sends. The decoding table is simply a table mapping each label on each *output* wire to its semantics (i.e. the corresponding plaintext value. Now, P_2 obtaining the output labels will look them up in the decoding table and obtain the output in plaintext.

At an intuitive level, at least, it is easy to see that this circuit-based construction is secure in the semi-honest model. Security against a corrupt P_1 is easy, since (other than the OT, which we assume has been separately shown to satisfy the OT security definition) that party receives no messages in the protocol! For a corrupt P_2, security boils down to the observation that the evaluator P_2 never sees both labels for the same wire. This is obviously true for the input wires, and it holds inductively for all intermediate wires (knowing only one label on each incoming wire of the gate, the evaluator can only decrypt one ciphertext of the garbled gate). Since P_2 does not know the correspondence between plaintext values and the wire labels, it has no information about the plaintext values on the wires, except for the output wires where the association between labels and values is explicitly provided by P_1. To simulate P_2's view, the simulator $\mathsf{Sim}_{\mathsf{P}_2}$ chooses random active labels for

each wire, simulates the three "inactive" ciphertexts of each garbled gate as dummy ciphertexts, and produces decoding information that decodes the active output wires to the function's output.

3.1.2 Yao's GC Protocol

Figure 3.2 formalizes Yao's gate generation, and Figure 3.3 summarizes Yao's GC protocol. For simplicity of presentation, we describe the protocol variant based on Random Oracle (defined in Section 2.2), even though a weaker assumption (the existence of pseudo-random functions) is sufficient for Yao's GC construction. The Random Oracle, denoted by H, is used in implementing garbled row encryption. We discuss different methods of instantiating H in practice in Section 4.1.4). The protocol also uses Oblivious Transfer, which requires public-key cryptography.

For each wire label, a pointer bit, p_i, is added to the wire label key following the point-and-permute technique described in Section 3.1.1. The pointer bits leak no information since they are selected randomly, but they allow the evaluator to determine which row in the garbled table to decrypt, based on the pointer bits for the two active wires it has for the inputs. In Section 4.1 we discuss several ways for making Yao's GC protocol more efficient, including reducing the size of the garbled table to just two ciphertexts per gate (Section 4.1.3) and enabling XOR gates to be computed without encryption (Section 4.1.2).

3.2 Goldreich-Micali-Wigderson (GMW) Protocol

As noted before, computation under encryption can be naturally viewed as operating on secret-shared data. In Yao's GC, the secret sharing of the active wire value is done by having one player (generator) hold two possible wire labels w_i^0, w_i^1, and the other player (evaluator) hold the active label w_i^b. In the GMW protocol (Goldreich *et al.*, 1987; Goldreich, 2004), the secret-sharing of the wire value is more direct: the players hold additive shares of the active wire value.

The GMW protocol (or just "GMW") naturally generalizes to more than two parties, unlike Yao's GC, which requires novel techniques to generalize to more than two parties (see Section 3.5).

PARAMETERS:
Boolean circuit C implementing function $\mathcal{F}$, security parameter κ.

GC GENERATION:

1. *Wire Label Generation*. For each wire w_i of C, randomly choose wire labels,
$$w_i^b = (k_i^b \in_R \{0,1\}^\kappa, p_i^b \in_R \{0,1\}),$$
such that $p_i^b = 1 - p_i^{1-b}$.

2. *Garbled Circuit Construction*. For each gate G_i of C in topological order:

   (a) Assume G_i is a 2-input Boolean gate implementing function g: $w_c = g(w_a, w_b)$, where input labels are $w_a^0 = (k_a^0, p_a^0), w_a^1 = (k_a^1, p_a^1), w_b^0 = (k_b^0, p_b^0), w_b^1 = (k_b^1, p_b^1)$, and the output labels are $w_c^0 = (k_c^0, p_c^0), w_c^1 = (k_c^1, p_c^1)$.

   (b) Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set
   $$e_{v_a,v_b} = H(k_a^{v_a} \,||\, k_b^{v_b} \,||\, i) \oplus w_c^{g_i(v_a,v_b)}$$
   Sort entries e in the table by the input pointers, placing entry e_{v_a,v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$

3. *Output Decoding Table*. For each circuit-output wire w_i (the output of gate G_j) with labels $w_i^0 = (k_i^0, p_i^0), w_i^1 = (k_i^1, p_i^1)$, create garbled output table for both possible wire values $v \in \{0,1\}$. Set
$$e_v = H(k_i^v \,||\, \text{“out”} \,||\, j) \oplus v$$
(Because we are xor-ing with a single bit, we just use the lowest bit of the output of H for generating the above e_v.) Sort entries e in the table by the input pointers, placing entry e_v in position p_i^v. (There is no conflict, since $p_i^1 = p_i^0 \oplus 1$.)

Figure 3.2: Yao's Garbled Circuit protocol: GC generation

PARAMETERS: Parties P_1 and P_2 with inputs $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$ respectively. Boolean circuit C implementing function $\mathcal{F}$.

PROTOCOL:

1. P_1 plays the role of GC generator and runs the algorithm of Figure 3.2. P_1 then sends the obtained GC $\widehat{C}$ (including the output decoding table) to P_2.

2. P_1 sends to P_2 active wire labels for the wires on which P_1 provides input.

3. For each wire w_i on which P_2 provides input, P_1 and P_2 execute an Oblivious Transfer (OT) where P_1 plays the role of the Sender, and P_2 plays the role of the Receiver:

   (a) P_1's two input secrets are the two labels for the wire, and P_2's choice-bit input is its input on that wire.

   (b) Upon completion of the OT, P_2 receives active wire label on the wire.

4. P_2 evaluates received $\widehat{C}$ gate-by-gate, starting with the active labels on the input wires.

   (a) For gate G_i with garbled table $T = (e_{0,0}, \ldots e_{1,1})$ and whose active input labels are $w_a = (k_a, p_a), w_b = (k_b, p_b)$, P_2 computes active output label $w_c = (k_c, p_c)$:

$$w_c = H(k_a \mid\mid k_b \mid\mid i) \oplus e_{p_a,p_b}$$

5. Obtaining output using output decoding tables. Once all gates of $\widehat{C}$ are evaluated, using "out" for the second key to decode the final output gates, P_2 obtains the final output labels which are equal to the plaintext output of the computation. P_2 sends the obtained output to P_1, and they both output it.

Figure 3.3: Yao's Garbled Circuit Protocol

3.2.1 GMW Intuition

The GMW protocol can work both on Boolean and arithmetic circuits. We present the two-party Boolean version first, and then briefly explain how the protocol can be generalized to the arithmetic setting, as well as to more than two parties. As with Yao's protocol, we assume players P_1 with input x and P_2 with input y have agreed on the Boolean circuit C representing the computed function $\mathcal{F}(x, y)$.

The GMW protocol proceeds as follows. For each input bit $x_i \in \{0, 1\}$ of $x \in \{0, 1\}^n$, P_1 generates a random bit $r_i \in_R \{0, 1\}$ and sends all r_i to P_2. Next, P_1 obtains a secret sharing of each x_i among P_1 and P_2 by setting its share to be $x_i \oplus r_i$. Symmetrically, P_2 generates random bit masks for its inputs y_i and sends the masks to P_1, secret sharing its input similarly.

P_1 and P_2 proceed in evaluating C gate by gate. Consider gate G with input wires w_i and w_j and output wire w_k. Let P_1 holds shares s_i^1 and s_j^1 on w_i and w_j, and P_2 hold shares s_i^2 and s_j^2 on the two wires. Without loss of generality, assume C consists of NOT, XOR and AND gates.

Both NOT and XOR gates can be evaluated without any interaction. A NOT gate is evaluated by P_1 flipping its share of the wire value, which flips the shared wire value. An XOR gate on wires w_i and w_j is evaluated by players xor-ing the shares they already hold. That is, P_1 computes its output share as $s_k^1 = s_i^1 \oplus s_j^1$, and P_2 correspondingly computes its output share as $s_k^2 = s_i^2 \oplus s_j^2$. The computed shares s_k^1, s_k^2 indeed are shares of the active output value: $s_k^1 \oplus s_k^2 = (s_i^1 \oplus s_j^1) \oplus (s_i^2 \oplus s_j^2) = (s_i^1 \oplus s_i^2) \oplus (s_j^1 \oplus s_j^2) = v_1 \oplus v_2$.

Evaluating an AND gate requires interaction and uses 1-out-of-4 OT a basic primitive. From the point of view of P_1, its shares s_i^1, s_j^1 are fixed, and P_2 has two Boolean input shares, which means there are four possible options for P_2. If P_1 knew P_2's shares, then evaluating the gate under encryption would be trivial: P_2 can just reconstruct the active input values, compute the active output value and secret-share it with P_2. While P_2 cannot do that, it can do the next best thing: prepare such a secret share for *each* of P_2's possible inputs, and run 1-out-of-4 OT to transfer the corresponding share. Specifically, let

$$S = S_{s_i^1, s_j^1}(s_i^2, s_j^2) = (s_i^1 \oplus s_i^2) \wedge (s_j^1 \oplus s_j^2)$$

be the function computing the gate output value from the shared secrets on the two input wires. P_1 chooses a random mask bit $r \in_R \{0, 1\}$ and prepares a

table of OT secrets:

$$T_G = \begin{pmatrix} r \oplus S(0,0) \\ r \oplus S(0,1) \\ r \oplus S(1,0) \\ r \oplus S(1,1) \end{pmatrix}$$

Then P_1 and P_2 run an 1-out-of-4 OT protocol, where P_1 plays the role of the sender, and P_2 plays the role of the receiver. P_1 uses table rows as each of the four input secrets, and P_2 uses its two bit shares as the selection to choose the corresponding row. P_1 keeps r as its share of the gate output wire value, and P_2 uses the value it receives from the OT execution.

Because of the way the OT inputs are constructed, the players obtain a secret sharing of the gate output wire. At the same time, it is intuitively clear that the players haven't learned anything about the other player's inputs or the intermediate values of the computation. This is because effectively only P_2 receives messages, and by the OT guarantee, it learns nothing about the three OT secrets it did not select. The only thing it learns is its OT output, which is its share of a random sharing of the output value and therefore leaks no information about the plaintext value on that wire. Likewise, P_1 learns nothing about the selection of P_2.

After evaluating all gates, players reveal to each other the shares of the output wires to obtain the output of the computation.

Generalization to more than two parties. We now sketch how to generalize this to the setting where n players $\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_n$ evaluate a boolean circuit $\mathcal{F}$. As before, player P_j secret-shares its input by choosing $\forall i \neq j, r_i \in_R \{0,1\}$, and sending r_i to each P_i. The parties $\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_n$ proceed by evaluating C gate-by-gate. They evaluate each gate G as follows:

- For an XOR gate, the players locally add their shares. Like the two-party case, no interaction is required and correctness and security are assured.
- For an AND gate $c = a \wedge b$, let $a_1, ...a_n, b_1, ..., b_n$ denote the shares of

a, b respectively held by the players. Consider the identity

$$c = a \wedge b = (a_1 \oplus \cdots \oplus a_n) \wedge (b_1 \oplus \cdots \oplus b_n)$$
$$= \left(\bigoplus_{i=1}^{n} a_i \wedge b_i \right) \oplus \left(\bigoplus_{i \neq j} a_i \wedge b_j \right)$$

Each player P_j computes $a_j \wedge b_j$ locally to obtain a sharing of $\bigoplus_{i=1}^{n} a_i \wedge b_i$. Further, each pair of players $\mathsf{P}_i, \mathsf{P}_j$ jointly computes the shares of $a_i \wedge b_j$ as described above in the two-party GMW. Finally, each player outputs the XOR of all obtained shares as the sharing of the result $a \wedge b$.

3.3 BGW protocol

One of the first multi-party protocols for secure computation is due to Ben-Or, Goldwasser, and Wigderson (Ben-Or *et al.*, 1988), and is known as the "BGW" protocol. Another somewhat similar protocol of Chaum, Crépau, and Damgård was published concurrently (Chaum *et al.*, 1988) with BGW, and the two protocols are often considered together. For concreteness, we present here the BGW protocol for n parties, which is somewhat simpler.

The BGW protocol can be used to evaluate an arithmetic circuit over a field $\mathbb{F}$, consisting of addition, multiplication, and multiplication-by-constant gates. The protocol is heavily based on Shamir secret sharing (Shamir, 1979), and it uses the fact that Shamir secret shares are homomorphic in a special way—the underlying shared value can be manipulated obliviously, by suitable manipulations to the individual shares.

For $v \in \mathbb{F}$ we write $[v]$ to denote that the parties hold Shamir secret shares of a value v. More specifically, a dealer chooses a random polynomial p of degree at most t, such that $p(0) = v$. Each party P_i then holds value $p(i)$ as their share. We refer to t as the *threshold* of the sharing, so that any collection of t shares reveals no information about v.

The invariant of the BGW protocol is that for every wire w in the arithmetic circuit, the parties hold a secret-sharing $[v_w]$ of the value v_w on that wire. Next, we sketch the protocol with a focus on maintaining this invariant.

Input wires. For an input wire belonging to party P_i, that party knows the value v on that wire in the clear, and distributes shares of $[v]$ to all the parties.

Addition gate. Consider an addition gate, with input wires α, β and output wire γ. The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of $[v_\alpha + v_\beta]$. Suppose the incoming sharings correspond to polynomials p_α and p_β, respectively. If each party P_i locally adds their shares $p_\alpha(i) + p_\beta(i)$, then the result is that each party holds a point on the polynomial $p_\gamma(x) \stackrel{\text{def}}{=} p_\alpha(x) + p_\beta(x)$. Since p_γ also has degree at most t, these new values comprise a valid sharing $p_\gamma(0) = p_\alpha(0) + p_\beta(0) = v_\alpha + v_\beta$.

Note that addition gates require no communication among the parties. All steps are local computation. The same idea works to multiply a secret-shared value by a public constant — each party simply locally multiplies their share by the constant.

Multiplication gate. Consider a multiplication gate, with input wires α, β and output wire γ. The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of the product $[v_\alpha \cdot v_\beta]$. As above, the parties can locally multiply their individual shares, resulting in each party holding a point on the polynomial $q(x) = p_\alpha(x) \cdot p_\beta(x)$. However, in this case the resulting polynomial may have degree as high as $2t$ which is too high.

In order to fix the excessive degree of this secret sharing, the parties engage in a degree-reduction step. Each party P_i holds a value $q(i)$, where q is a polynomial of degree at most $2t$. The goal is to obtain a valid secret-sharing of $q(0)$, but with correct threshold.

The main observation is that $q(0)$ can be written as a linear function of the party's shares. In particular,

$$q(0) = \sum_{i=1}^{2t+1} \lambda_i q(i)$$

where the λ_i terms are the appropriate Lagrange coefficients. Hence the degree-reduction step works as follows:

1. Each party[3] P_i generates and distributes a threshold-t sharing of $[q(i)]$. To simplify the notation, we do not give names the polynomials that underly these shares. However, it is important to keep in mind that each party P_i chooses a polynomial of degree at most t whose constant coefficient is $q(i)$.

[3] Technically, only $2t + 1$ parties need to do this

2. The parties compute $[q(0)] = \sum_{i=1}^{2t+1} \lambda_i [q(i)]$, using local computations. Note that the expression is in terms of addition and multiplication-by-constant applied to secret-shared values.

Since the values $[q(i)]$ were shared with threshold t, the final sharing of $[q(0)]$ also has threshold t, as desired.

Note that multiplication gates in the BGW protocol require communication/interaction, in the form of parties sending shares of $[q(i)]$. Note also that we require $2t + 1 \leq n$, since otherwise the n parties do not collectively have enough information to determine the value $q(0)$, as q may have degree $2t$. For that reason, the BGW protocol is secure against t corrupt parties, for $2t < n$ (i.e., an honest majority).

Output wires. For an output wire α, the parties will eventually hold shares of the value $[v_\alpha]$ on that wire. Each party can simply broadcast its share of this value, so that all parties can learn v_α.

3.4 MPC From Preprocessed Multiplication Triples

A convenient paradigm for constructing MPC protocols is to split the problem into a *pre-processing* phase (before the parties' inputs are known) and an *online* phase (after the inputs are chosen). The pre-processing phase can produce correlated values for the parties, which they can later "consume" in the online phase. This paradigm is also used in some of the leading malicious-secure MPC protocols discussed in Section 6.

Intuition. To get an idea of how to defer some of the protocol effort to the pre-processing phase, recall the BGW protocol. The only real cost in the protocol is the communication that is required for every multiplication gate. However, it is not obvious how to move any of the related costs to a pre-processing phase, since the costs are due to manipulations of secret values that can only be determined in the online phase (i.e., they are based on the circuit inputs). Nonetheless, Beaver (1992) showed a clever way to move the majority of the communication to the pre-processing phase.

A *Beaver triple* (or *multiplication triple*) refers to a triple of secret-shared values $[a], [b], [c]$ where a and b are randomly chosen from the appropriate

field, and $c = ab$. In an offline phase, such Beaver triples can be generated in a variety of ways, such as by simply running the BGW multiplication subprotocol on random inputs. One Beaver triple is then "consumed" for each multiplication gate in the eventual protocol.

Consider a multiplication gate with input wires α, β. The parties hold secret sharings of $[v_\alpha]$ and $[v_\beta]$. To carry out the multiplication of v_α and v_β using a Beaver triple $[a], [b], [c]$, the parties do the following:

1. Using local computation, compute $[v_\alpha - a]$ and publicly open $d = v_\alpha - a$ (i.e., all parties announce their shares). While this value depends on the secret value v_α, it is masked by the random value a and therefore reveals no information about v_α.[4]

2. Using local computation, compute $[v_\beta - b]$ and publicly open $e = v_\beta - b$.

3. Observe the following identity:

$$\begin{aligned} v_\alpha v_\beta &= (v_\alpha - a + a)(v_\beta - b + b) \\ &= (d + a)(e + b) \\ &= de + db + ae + ab \\ &= de + db + ae + c \end{aligned}$$

   Since d and e are public, and the parties hold sharings of $[a], [b], [c]$, they can compute a sharing of $[v_\alpha v_\beta]$ by local computation only:

$$[v_\alpha v_\beta] = de + d[b] + e[a] + [c]$$

Using this technique, a multiplication can be performed using only two openings plus local computation. Overall, each party must broadcast two field elements per multiplication, compared to n field elements (across private channels) in the plain BGW protocol. While this comparison ignores the cost of generating the Beaver triples in the first place, there are methods for generating triples in a batch where the amortized cost of each triple is a constant number of field elements per party (Beerliová-Trubíniová and Hirt, 2008).

[4] Since a is used as essentially a one-time pad (and b similarly below), this triple $[a], [b], [c]$ cannot be reused again in a different multiplication gate.

Abstraction. While the BGW protocol (specifically, its degree-reduction step) deals with the details of Shamir secret shares, the Beaver-triples approach conveniently abstracts these away. In fact, it works as long as the parties have an abstract "sharing mechanism" $[v]$ with the following properties:

- *Additive homomorphism*: Given $[x]$ and $[y]$ and a public value z, parties can obtain any of $[x + y]$, $[x + z]$, $[xz]$, without interaction.
- *Opening*: Given $[x]$, parties can choose to reveal x to all parties.
- *Privacy*: An adversary (from whatever class of adversaries is being considered) can get no information about x from $[x]$.
- *Beaver triples*: For each multiplication gate, the parties have a random triple $[a], [b], [c]$ where $c = ab$.
- *Random input gadgets*: For each input wire belonging to party P_i, the parties have a random $[r]$, where r is known only to P_i. During the protocol, when P_i chooses its input value x for this wire, it can announce $\delta = x - r$ to all parties (leaking nothing about x), and they can locally compute $[x] = [r] + \delta$ from the homomorphic properties.

As long as these properties are true of an abstract sharing mechanism, the Beaver-triples approach is secure. In fact, the paradigm is also secure in the presence of *malicious* adversaries, as long as the opening and privacy properties of the sharing mechanism hold against such adversaries. Specifically, a malicious adversary cannot falsify the opening of a shared value. We use this fact later in Section 6.6.

Instantiations. Clearly Shamir secret sharing gives rise to an abstract sharing scheme $[\cdot]$ that satisfies the above properties with respect to adversaries who corrupt at most $t < n/2$ parties.

Another suitable method of sharing is simple additive sharing over a field $\mathbb{F}$. In additive sharing, $[v]$ signifies that each party P_i holds v_i where $\sum_{i=1}^{n} v_i = v$. This mechanism satisfies the appropriate homomorphic properties, and is secure against $n - 1$ corrupt parties. When using $\mathbb{F} = \{0, 1\}$, we obtain an offline-online variant of the GMW protocol (since the field operations in this case correspond to AND and XOR). Of course, an arbitrary $\mathbb{F}$ is possible as well, leading to a version of GMW for arithmetic circuits.

3.5 Constant-Round Multi-Party Computation: BMR

After Yao's (two-party) GC protocol was proposed, several *multi-party* protocols appeared, including Goldreich-Micali-Wigderson (GMW) (Goldreich, 2004; Goldreich *et al.*, 1987), presented in detail above in Section 3.2, Ben Or-Goldwasser-Wigderson (BGW) (Ben-Or *et al.*, 1988), Chaum-Crepeau-Damgård (CCD) (Chaum *et al.*, 1988). All of these protocols have a number of rounds linear in the depth of the circuit C computing $\mathcal{F}$. The Beaver-Micali-Rogaway (BMR) protocol (Beaver *et al.*, 1990) runs in a constant (in the depth of the circuit C) number of rounds, while achieving security in against any $t < n$ number of corruptions among the n participating parties.

3.5.1 BMR Intuition

The BMR protocols adapt the main idea of Yao's GC to a multi-party setting. GC is chosen as a starting point due to its round-efficiency. However, a naïve attempt to port the GC protocol from the 2PC into the MPC setting gets stuck at the stage of sending the generated GC to the evaluators. Indeed, the circuit generator knows all the secrets (wire label correspondences), and if it colludes with any of the evaluators, the two colluding parties can learn the intermediate wire values and violate the security guarantees of the protocol.

The basic BMR idea is to perform a *distributed* GC generation, so that no single party (or even a proper subset of all parties) knows the GC generation secrets – the label assignment and correspondence. This GC generation can be done *in parallel* for all gates using MPC. This is possible by first generating (in parallel) all wire labels independently, and then independently and in parallel generating garbled gate tables. Because of parallel processing for all gates/wires, the GC generation is *independent* of the depth of the computed circuit C. As a result, the GC generation circuit C_{GEN} is constant-depth for all computed circuits C (once the security parameter κ is fixed). Even if the parties perform MPC evaluation of C_{GEN} that depends on the depth of C_{GEN}, the overall BMR protocol will still have constant rounds overall.

The MPC output, the GC produced by securely evaluating C_{GEN}, may be delivered to a designated player, say P_1, who will then evaluate it similarly to Yao's GC. The final technicality here is how to deliver the active input labels to P_1. There are several ways how this may be achieved, depending on how

exactly the MPC GC generation proceeded. Perhaps, it is conceptually simplest to view this as part of the GC generation computation.

In concrete terms, the above approach is not appealing due to potentially high costs of distributed generation of encryption tables, requiring the garbled row encryption function (instantiated as a PRF or hash function) evaluation inside MPC. Several protocols were proposed, which allow the PRF/hash evaluation to be extracted from inside the MPC and instead be done locally by the parties while providing the output of PRF/hash into the MPC. The underlying idea of such an approach is to assign different portions of each label to be generated by different players. That is, a wire w_a's labels w_a^v are a concatenation of sublabels $w_{a,j}^v$ each generated by P_j. Then, for a gate G_i with input labels $w_a^{v_a}, w_b^{v_b}$ and the output label $w_c^{v_c}$, the garbled row corresponding to input values v_a, v_b and output value v_c can simply be:

$$e_{v_a,v_b} = w_c^{v_c} \bigoplus_{j=1..n} (F(i, w_{a,j}^{v_a}) \oplus F(i, w_{b,j}^{v_b})), \tag{3.1}$$

where $F : \{0,1\}^\kappa \mapsto \{0,1\}^{n\cdot\kappa}$ is a PRG extending κ bits into $n \cdot \kappa$ bits.

The generation of the garbled table row is almost entirely done locally by each party. Each P_j computes $F(i, w_{a,j}^{v_a}) \oplus F(i, w_{b,j}^{v_b})$ and submits it to the MPC, which simply xors all the values to produce the garbled row.

However, we are not quite done. Recall that the GC evaluator P_1 will reconstruct active labels. A careful reader would notice that knowledge of its own contributed sublabel will allow it to identify which plaintext value the active label corresponds to, violating the security guarantee.

The solution is to for each player P_j to add a "flip" bit $f_{a,j}$ to each wire w_a. The xor of the n flip bits, $f_a = \bigoplus_{j=1..n} f_{a,j}$, determines which plaintext bit corresponds to the wire label w_a^v. The flip bits will be an additional input into the garbling MPC. Now, with the addition of the flip bits, no subset of players will know the wire flip bit, and hence even the recognition and matching of the sublabel will not allow the evaluator to match the label to plaintext value, or to compute the inactive label in full.

We sketch a concrete example of an efficient BMR garbling in Figure 3.4; BMR evaluation is straightforward based on the garbling technique.

PARAMETERS: Boolean circuit C implementing function $\mathcal{F}$.
Let $F : \{0,1\}^\kappa \mapsto \{0,1\}^{n\cdot\kappa+1}$ be a PRG.

PLAYERS: P_1, P_2, ..., P_n with inputs $x_1, ..., x_n \in \{0,1\}^k$.

GC GENERATION:

1. For each wire w_i of C, each P_j randomly chooses wire sublabels, $w^b_{i,j} = (k^b_{i,j}, p^b_{i,j}) \in_R \{0,1\}^{\kappa+1}$, such that $p^b_{i,j} = 1 - p^{1-b}_{i,j}$, and flip-bit shares $f_{i,j} \in_R \{0,1\}$. For each wire w_i, P_j locally computes its underlying-MPC input,

$$I_{i,j} = (F(w^0_{i,j}), F(w^1_{i,j}), p^0_{i,j}, f_{i,j}).$$

2. For each gate G_i of C in parallel, all players participate in n-party MPC to compute the garbled table, taking as input all players' inputs $x_1, ..., x_n$ as well as pre-computed values $I_{i,j}$, by evaluating the following function:

   1. Assume G_i is a 2-input Boolean gate implementing function g, with input wires w_a, w_b and output wire w_c.
   2. Compute pointer bits $p^0_a = \bigoplus_{j=1..n} p^0_{a,j}, p^0_b = \bigoplus_{j=1..n} p^0_{b,j}, p^0_c = \bigoplus_{j=1..n} p^0_{c,j}$, and set $p^1_a = 1 - p^0_a, p^1_b = 1 - p^0_b, p^1_c = 1 - p^0_c$. Similarly compute flip bits f_a, f_b, f_c by xor-ing the corresponding flip bit shares submitted by the parties. Amend the semantics of the wires according to the flip bits by xor-ing f_a, f_b, f_c in the label index as appropriate (included in the next steps).
   3. Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set

$$e_{v_a,v_b} = w^{v_c \oplus f_c}_c \bigoplus_{j=1..n} (F(i, w^{v_a \oplus f_a}_{a,j}) \oplus F(i, w^{v_b \oplus f_b}_{b,j})),$$

   where $w^0_c = w^0_{c,1} \,||\, \ldots \,||\, w^0_{c,n} \,||\, p^0_c, w^1_c = w^0_{c,1} \,||\, \ldots \,||\, w^1_{c,n} \,||\, p^1_c$. Sort entries e in the table, placing entry e_{v_a,v_b} in position $(p^{v_a}_a, p^{v_b}_b)$.
   4. Output to P_1 the computed garbled tables, as well as active wire labels inputs of C, as selected by players' inputs, $x_1, ..., x_n$.

Figure 3.4: BMR Multi-Party GC Generation

3.6 Information-Theoretic Garbled Circuits

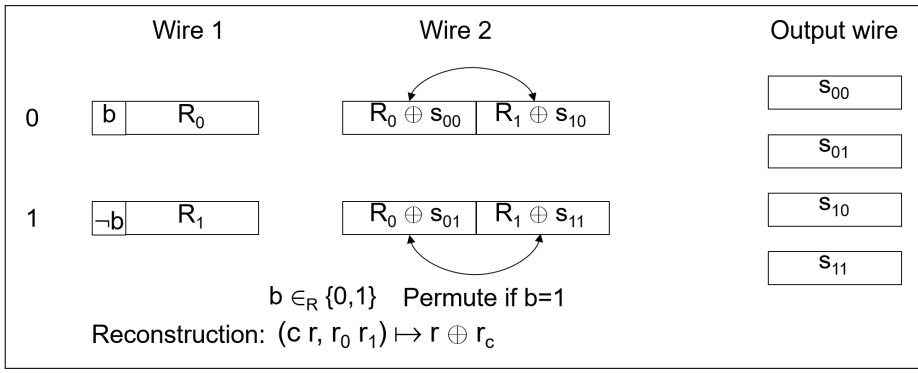
Yao's GC and the GMW protocol present two different flavors of the use of secret sharing in MPC. In this section, we discuss a third flavor, where the secrets are shared not among players, but *among wires*. This construction is also interesting because it provides information-theoretic security in the OT-hybrid setting, meaning that no computational hardness assumptions are used in the protocol beyond what is used in the underlying OT. An important practical reason to consider IT GC is that it presents a trade-off between communication bandwidth and latency: it needs to send less data than Yao GC at the cost of additional communication rounds. While most research on practical MPC focuses on low round complexity, we believe some problems which require very wide circuits, such as those that arise in machine learning, may benefit from IT GC constructions.

Information-theoretic constructions typically provide stronger security at a higher cost. Surprisingly, this is not the case here. Intuitively, higher performance is obtained because information-theoretic encryption allows the encryption of a bit be a single bit rather than the security parameter in length. Further, information-theoretic encryption here is done with bitwise XOR and bit shufflings, rather than with standard primitives such as AES.

We present the Gate Evaluation Secret Sharing (GESS) scheme of Kolesnikov (2005) (Kolesnikov (2006) provides details), which is the most efficient information-theoretic analog of GC. The main result of Kolesnikov (2005) is a two-party protocol for a Boolean formula F with communication complexity $\approx \sum d_i^2$, where d_i is the depth of the i-th leaf of F.

At a high level, GESS is a secret-sharing scheme, designed to allow evaluation under encryption of a Boolean gate G. The output wire labels of G are the two secrets from which P_1 produces four secret shares, one corresponding to each of the wire labels of the two input wires. GESS guarantees that a valid combination of shares (one share per wire) can be used to reconstruct the corresponding label of the output wire. This is similar to Yao's GC, but GESS does not require the use of garbled tables, and hence can be viewed as a generalization of Yao's GC. Similarly to Yao's GC approach, the secret sharing can be applied gate-by-gate without the need to decode or reconstruct the plaintext values.

Consider a two-input Boolean gate G. Given the possible output values



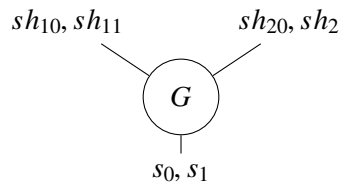


Figure 3.5: GESS for Boolean gate

s_0, s_1 and the semantics of the gate G, P_1 generates input labels (sh_{10}, sh_{11}), (sh_{20}, sh_{21}), such that each possible pair of encodings $(sh_{1,i}, sh_{2,j})$ where $i, j \in \{0, 1\}$, allows reconstructing $G(i, j)$ but carries no other information. Now, if P_2 obtains shares corresponding to gate inputs, it would be able to reconstruct the label on the output wire, and nothing else.

This mostly corresponds to our intuition of secret sharing schemes. Indeed, the possible gate outputs play the role of secrets, which are shared and then reconstructed from the input wires encodings (shares).

3.6.1 GESS for Two-Input Binary Gates

We present GESS for the 1-to-1 gate function $G : \{0, 1\}^2 \mapsto \{00, 01, 10, 11\}$, where $G(0, 0) = 00, G(0, 1) = 01, G(1, 0) = 10, G(1, 1) = 11$. Clearly, this is a generalization of the Boolean gate functionality $G : \{0, 1\}^2 \mapsto \{0, 1\}$.

Let the secrets domain be $\mathcal{D}_S = \{0, 1\}^n$, and four (not necessarily distinct) secrets $s_{00}, ...s_{11} \in \mathcal{D}_S$ are given. The secret s_{ij} corresponds to the value $G(i, j)$ of the output wire.

The intuition for the design of the GESS scheme is as follows (see illustration in Figure 3.5). We first randomly choose two strings $R_0, R_1 \in_R \mathcal{D}_S$ to be the shares sh_{10} and sh_{11} (corresponding to 0 and 1 of the first input wire). Now consider sh_{20}, the share corresponding to 0 of the second input wire. We want this share to produce either s_{00} (when combined with sh_{10}) or s_{10} (when combined with sh_{11}). Thus, the share sh_{20} will consist of two blocks. One, block $s_{00} \oplus R_0$, is designed to be combined with R_0 and reconstruct s_{00}. The other, $s_{10} \oplus R_1$, is designed to be combined with R_1 and reconstruct s_{10}. Share sh_{21} is constructed similarly, setting blocks to be $s_{01} \oplus R_0$ and $s_{11} \oplus R_1$.

Both leftmost blocks are designed to be combined with the same share R_0, and both rightmost blocksare designed to be combined with the same share R_1. Therefore, we append a 0 to R_0 to tell *Rec* to use the left block of the second share for reconstruction, and append a 1 to R_1 to tell *Rec* to use the right block of the second share for reconstruction. Finally, to hide information leaked by the order of blocks in shares, we randomly choose a bit b and if $b = 1$ we reverse the order of blocks in *both* shares of wire 2 and invert the appended pointer bits of the shares of wire 1. Secret reconstruction proceeds by xor-ing the wire-1 share (excluding the pointer bit) with the first or second half of the wire-2 share as indexed by the pointer bit.

3.6.2 Reducing Share Growth

Note the inefficiency of the above construction, causing the shares corresponding to the second input wire be double the size of the gate's secrets. While, in some circuits we can avoid the exponential (in depth) secret growth by balancing the direction of greater growth toward more shallow parts of the circuit, a more efficient solution is desirable. We discuss only AND and OR gates, since NOT gates are implemented simply by flipping the wire label semantics by the Generator. GESS also enables XOR gates without any increase the share sizes. We defer discussion of this to Section 4.1.2, because the XOR sharing in GESS led to an important related improvement for Yao's GC.

For OR and AND gates in the above construction, either the left or the right blocks of the two shares are equal (this is because $s_{00} = s_{01}$ for the AND gate, and $s_{10} = s_{11}$ for the OR gate). We use this property to reduce the size of the shares when the secrets are of the above form. The key idea is to view the shares of the second wire as being the same, except for one block.

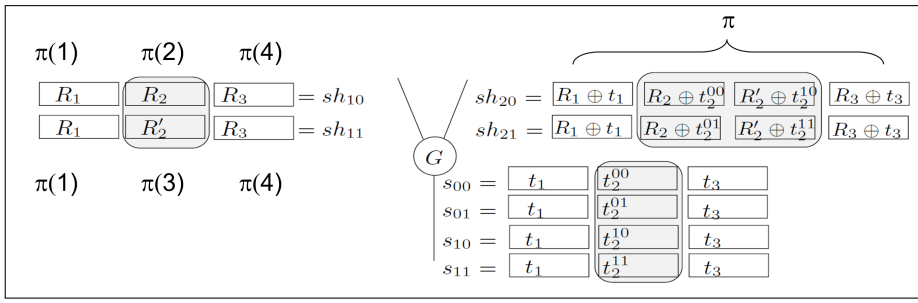


Figure 3.6: Improved GESS for Boolean gate

Suppose each of the four secrets consists of n blocks and the secrets differ only in the j^{th} block, as follows:

$$s_{00} = (\ t_1\ \ \ldots\ \ t_{j-1}\ \ t_j^{00}\ \ t_{j+1}\ \ \ldots\ \ t_n\ ),$$
$$\ldots$$
$$s_{11} = (\ t_1\ \ \ldots\ \ t_{j-1}\ \ t_j^{11}\ \ t_{j+1}\ \ \ldots\ \ t_n\ ),$$

where $\forall i = 1..n$: $t_i, t_j^{00}, t_j^{01}, t_j^{10}, t_j^{11} \in \{0,1\}^k$ for some k. It is convenient to consider the *columns* of blocks, spanning across the shares. Every column (with the exception of the j-th) consists of four equal blocks, where the value j is private.

For simplicity, we show the main ideas by considering a special case where the four secrets consist of $n = 3$ blocks each, and $j = 2$ is the index of the column of distinct blocks. This intuition is illustrated on Figure 3.6. The scheme naturally generalizes from this intuition; Kolesnikov (2005) provides a formal presentation.

The idea is to share the secrets "column-wise", treating each of the three columns of blocks of secrets as a tuple of subsecrets and sharing this tuple separately, producing the corresponding subshares. Consider sharing column 1. All four subsecrets are equal (to t_1), and we share them trivially by setting both subshares of the first wire to a random string $R_1 \in_R D$, and both subshares of the second wire to be $R_1 \oplus t_1$. Column 3 is shared similarly. We share column 2 as in previous construction (highlighted on the diagram), omitting the last step of appending the pointers and applying the permutation. This preliminary assignment of shares (still leaking information due to order of blocks) is shown on Figure 3.6. Note that the reconstruction of secrets is done by xor-ing the

corresponding blocks of the shares, and, importantly, the procedure is the same for both types of sharing we use. For example, given shares sh_{10} and sh_{21}, we reconstruct the secret $s_{01} = (R_1 \oplus (R_1 \oplus t_1), \quad R_2 \oplus (R_2 \oplus t_2^{01}), \quad R_3 \oplus (R_3 \oplus t_3))$.

The remaining permute-and-point step is to apply (the same) random permutation π to reorder the four columns of both shares of wire 2 and to append $(\log 4)$-bit pointers to each block of the shares of wire 1, telling the reconstructor which block of the second share to use. Note that the pointers appended to both blocks of column 1 of wire 1 are the same. The same holds for column 3. Pointers appended to blocks of column 2 are different. For example, if the identity permutation was applied, then we will append "1" to both blocks R_1, "2" to R_2, "3" to R_2', and "4" to both blocks R_3. This leads to the punchline: because G is either an OR or an AND gate, both tuples of shares maintain the property that all but one of the pairs of corresponding blocks are equal between the shares of the tuple. This allows *repeated* application (i.e., continuing sharing) of GESS for OR and AND gates.

Finally, to put it all together, we sketch the GESS-based MPC protocol. P_1 represents the function $\mathcal{F}$ as a *formula* F. Then, starting with the output wires of F and taking the plaintext output wire labels as secrets, P_1 applies GESS scheme repeatedly to all gates of the circuit, assigning the GESS shares to gates' input wires until he assigns the labels to formula inputs. Then P_1 transfers to P_2 active wires on the input labels, and P_2 repeatedly uses GESS reconstruction procedure to obtain output labels of F.

3.7 Oblivious Transfer

Oblivious Transfer, defined in Section 2.4, is an essential building block for secure computation protocols, and an inherently asymmetric primitive. Impagliazzo and Rudich (1989) showed that a reduction from OT to a symmetric-key primitive (one-way functions, PRF) implies that P≠NP. However, as first observed by Beaver (1996), a *batched* execution of OT only needs a small number of public key operations. Beaver's construction was non-black-box in the sense that a PRF needed to be represented as a circuit and evaluated as MPC. As a consequence, Beaver's result was mainly of theoretical interest. Ishai *et al.* (2003) changed the state of affairs dramatically by proposing an extremely efficient batched OT which only required κ of public key operations for the entire batch and two or three hashes per OT.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. $\mathcal{S}$ has input secrets $x_1, x_2 \in \{0,1\}^n$, and $\mathcal{R}$ has a selection bit $b \in \{0,1\}$.

PROTOCOL:

1. $\mathcal{R}$ generates a public-private key pair sk, pk, and samples a random key, pk', from the public key space. If $b = 0$, $\mathcal{R}$ sends a pair (pk, pk') to $\mathcal{S}$. Otherwise (if $b = 1$), $\mathcal{R}$ sends a pair (pk', pk) to $\mathcal{S}$.
2. $\mathcal{S}$ receives (pk_0, pk_1) and sends back to $\mathcal{R}$ two encryptions $e_0 = \mathsf{Enc}_{pk_0}(x_0), e_1 = \mathsf{Enc}_{pk_1}(x_1)$.
3. $\mathcal{R}$ receives e_0, e_1 and decrypts the ciphertext e_b using sk. $\mathcal{R}$ is unable to decrypt the second ciphertext as it does not have the corresponding secret key.

Figure 3.7: Public key-based semi-honest OT.

3.7.1 Public Key-Based OT

We start with the basic public key-based OT in the semi-honest model. The construction, presented in Figure 3.7, is very simple indeed.

The security of the construction assumes the existence of public-key encryption with the ability to sample a random public key without obtaining the corresponding secret key. The scheme is secure in the semi-honest model. The Sender $\mathcal{S}$ only sees the two public keys sent by $\mathcal{R}$, so cannot predict with probability better than $\frac{1}{2}$ which key was generated without the knowledge of the secret key. Hence, the view of $\mathcal{S}$ can be simulated simply by sending two randomly-chosen public keys.

The Receiver $\mathcal{R}$ sees two encryptions and has a secret key to decrypt only one of them. The view of $\mathcal{R}$ is also easily simulated, given $\mathcal{R}$'s input and output. $\mathsf{Sim}_{\mathcal{S}}$ will generate the public key pair and a random public key, and set the simulated received ciphertexts to be 1) the encryption of the received secret under the generated keypair and 2) the encryption of zero under the randomly chosen key. The simulation goes through since the difference with the real

execution is only in the second encryption, and distinguisher will not be a tell apart the encryption of zero from another value due to the encryption security guarantees. Note that this semi-honest protocol provides no security against a malicious sender—the Sender $\mathcal{S}$ can simply generate two public-private key pairs, (sk_0, pk_0) and (sk_1, pk_1) and send (pk_0, pk_1) to $\mathcal{R}$, and decrypt both received ciphertexts to learn both x_1 and x_2.

3.7.2 Public Key Operations in OT

The simple protocol in Figure 3.7 requires one public key operation for both the sender and receiver for each selection bit. As used in a Boolean circuit-based MPC protocol such as Yao's GC, it is necessary to perform an OT for each input bit of the party executing the circuit. For protocols like GMW, evaluating each gate requires an OT. Hence, several works have focused on reducing the number of public key operations to perform a large number of OTs.

Beaver's non-black-box construction. Beaver (1996) proposed bootstrapping Yao's GC protocol to generate a polynomial number of OTs from a small number of public key operations. As discussed in Section 3.1, the GC protocol for computing a circuit C requires m OTs, where m is the number of input bits provided by P_2. Following the OT notation, we call P_1 (the generator in GC) the sender $\mathcal{S}$, and P_2 (the evaluator in GC) the receiver $\mathcal{R}$. Let m be a desired number of OTs that will now be performed as a batch. $\mathcal{S}$'s input will be m pairs of secrets $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$, and $\mathcal{R}$'s input will be m-bit selection string $b = (b_1, \ldots, b_m)$.

We now construct a circuit C that implements a function F which takes only a small number of input bits from $\mathcal{R}$, but outputs the result of polynomial number of OTs to $\mathcal{R}$. The input of $\mathcal{R}$ to F will be a randomly chosen κ-bit string r. Let G be a pseudo-random generator expanding κ bits into m bits. $\mathcal{R}$ will send to $\mathcal{S}$ its input string masked with the pseudo-random string, $b \oplus G(r)$. Then, $\mathcal{S}$'s input to F will be m pairs of secrets $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$ as well as the m-bit string $b \oplus G(r)$. Given r, the function F computes the m-bit expansion $G(r)$ and unmasks the input $b \oplus G(r)$, obtaining the selection string b. Then F simply outputs to $\mathcal{R}$ the corresponding secrets x_{b_i}. Only κ input bits are provided by $\mathcal{R}$, the circuit evaluator, so only a constant number of κ OTs are needed to perform m OTs.

Reducing the number of public key operations. The construction of Beaver (1996) shows a simple way to reduce the number of asymmetric operations required to perform m OTs to a fixed security parameter, but is not efficient in practice because of the need to execute a large GC. Recall, our goal is to use a small number k of base-OTs, plus only symmetric-key operations, to achieve $m \gg k$ effective OTs. Here, k is chosen depending on the computational security parameter κ; in the following we show how to choose k. Below we describe the OT extension by Ishai *et al.* (2003) that achieves m 1-out-of-2 OT of random strings, in the presence of semi-honest adversaries.

We follow the notation of Kolesnikov and Kumaresan (2013), as it explicates the coding-theoretic framework for OT extension. Suppose the receiver $\mathcal{R}$ has choice bits $r \in \{0,1\}^m$. $\mathcal{R}$ chooses two $m \times k$ matrices (m rows, k columns), T and U. Let $\boldsymbol{t}_j, \boldsymbol{u}_j \in \{0,1\}^k$ denote the j-th row of T and U, respectively. The matrices are chosen at random, so that:

$$\boldsymbol{t}_j \oplus \boldsymbol{u}_j = r_j \cdot 1^k \stackrel{\text{def}}{=} \begin{cases} 1^k & \text{if } r_j = 1 \\ 0^k & \text{if } r_j = 0 \end{cases}$$

The sender $\mathcal{S}$ chooses a random string $s \in \{0,1\}^k$. The parties engage in k instances of 1-out-of-2 string-OT, *with their roles reversed*, to transfer to sender $\mathcal{S}$ the columns of either T or U, depending on the sender's bit s_i in the string s it chose. In the i-th OT, $\mathcal{R}$ provides inputs $\boldsymbol{t}^i$ and $\boldsymbol{u}^i$, where these refer to the i-th *column* of T and U, respectively. $\mathcal{S}$ uses s_i as its choice bit and receives output $\boldsymbol{q}^i \in \{\boldsymbol{t}^i, \boldsymbol{u}^i\}$. Note that these are OTs of strings of length $m \gg k$ — the *length* of OT messages is easily extended, e.g., by encrypting and sending the two m-bit long strings, and using OT on short strings to send the right decryption key.

Now let Q denote the matrix obtained by the sender, whose columns are $\boldsymbol{q}^i$. Let $\boldsymbol{q}_j$ denote the jth row. The key observation is that

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [r_j \cdot s] = \begin{cases} \boldsymbol{t}_j & \text{if } r_j = 0 \\ \boldsymbol{t}_j \oplus s & \text{if } r_j = 1 \end{cases} \tag{3.2}$$

Let H be a Random Oracle (RO)[5]. Then $\mathcal{S}$ can compute two random strings $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$, of which $\mathcal{R}$ can compute only one, via $H(\boldsymbol{t}_j)$,

[5] As pointed out by Ishai *et al.* (2003), it is sufficient to assume that H is a correlation-robust hash function, a weaker assumption than RO. A special assumption is required because the same s is used for every resulting OT instance.

of $\mathcal{R}$'s choice. Indeed, following Equation 3.2, $\boldsymbol{q}_j$ equals either $\boldsymbol{t}_j$ or $\boldsymbol{t}_j \oplus s$, depending on $\mathcal{R}$'s choice bit r_j. It is immediate then that $\boldsymbol{t}_j$ equals either $\boldsymbol{q}_j$ or $\boldsymbol{q}_j \oplus s$, depending on $\mathcal{R}$'s choice bit r_j. Note that $\mathcal{R}$ has no information about s, so intuitively it can learn only one of the two random strings $H(\boldsymbol{q}_j), H(\boldsymbol{q}_j \oplus s)$. Hence, each of the m rows of the matrix can be used to produce a single 1-out-of-2 OT of random strings.

To extend this to the more usual 1-out-of-2 OT of two given secrets s_0, s_1, we add the following step to the above. $\mathcal{S}$ now additionally encrypts the two OT secrets with the two keys $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$ and sending the two encryptions (e.g. $H(\boldsymbol{q}_j) \oplus s_0$ and $H(\boldsymbol{q}_j \oplus s) \oplus s_1$) to $\mathcal{R}$. As $\mathcal{R}$ can obtain exactly one of $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$, he can obtain only the corresponding secret s_i.

Coding interpretation and cheaper 1-out-of-2^ℓ OT. In IKNP, the receiver prepares secret shares of T and U such that each row of $T \oplus U$ is either all zeros or all ones. Kolesnikov and Kumaresan (2013) interpret this aspect of IKNP as a *repetition code* and suggest using other codes instead.

Consider how we might use the IKNP OT extension protocol to realize 1-out-of-2^ℓ OT. Instead of a choice bit r_i for the receiver, r_i will now be an ℓ-bit string. Let C be a linear error correcting code of dimension ℓ and codeword length k. The receiver will prepare matrices T and U so that $\boldsymbol{t}_j \oplus \boldsymbol{u}_j = C(r_j)$.

Now, generalizing Equation 3.2 the sender $\mathcal{S}$ receives

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \tag{3.3}$$

where "$\cdot$" now denotes bitwise-AND of two strings of length k. (Note that when C is a repetition code, this is exactly Equation 3.2.)

For each value $r' \in \{0,1\}^\ell$, the sender associates the secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$, which it can compute for all $r' \in \{0,1\}^\ell$. At the same time, the receiver can compute one of these values, $H(\boldsymbol{t}_j)$. Rearranging Equation 3.3, we have:

$$H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r_j) \cdot s])$$

Hence, the value that the receiver can learn is the secret value that the sender associates with the receiver's choice string $r' = r_j$.

At this point, OT of random strings is completed. For OT of chosen strings, the sender will use each $H(\boldsymbol{q}_i \oplus [C(r) \cdot s])$ as a key to encrypt the r-th OT

message. The receiver will be able to decrypt only one of these encryptions, namely one corresponding to its choice string r_j.

To argue that the receiver learns *only one* string, suppose the receiver has choice bits r_j but tries to learn also the secret $H(\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s])$ corresponding to a different choice $\tilde{r}$. We observe:

$$\begin{aligned}\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s] &= \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \oplus [C(\tilde{r}) \cdot s] \\ &= \boldsymbol{t}_j \oplus [(C(r_j) \oplus C(\tilde{r})) \cdot s]\end{aligned}$$

Importantly, everything in this expression is known to the receiver except for s. Now suppose the minimum distance of C is κ (the security parameter). Then $C(r_j) \oplus C(\tilde{r})$ has Hamming weight at least κ. Intuitively, the adversary would have to guess at least κ bits of the secret s in order to violate security. The protocol is secure in the RO model, and can also be proven under the weaker assumption of correlation robustness, following Ishai *et al.* (2003) and Kolesnikov and Kumaresan (2013).

Finally, we remark that the width k of the OT extension matrix is equal to the length of codewords in C. The parameter k determines the number of base OTs and the overall cost of the protocol.

The IKNP protocol sets the number of OT matrix columns to be $k = \kappa$. To achieve the same concrete security as IKNP OT, the KK13 protocol (Kolesnikov and Kumaresan, 2013) requires setting $k = 2\kappa$, to account for the larger space required by the more efficient underlying code C.

3.8 Custom Protocols

All of the secure computation protocols discussed so far in this chapter are generic circuit-based protocols. Circuit-based protocols suffer from linear bandwidth cost in the size of the circuit, which can be prohibitive for large computations. There are significant overheads with circuit-based computation on large data structures, compared to, say, a RAM (Random Access Machine) representation. In Section 5 we discuss approaches for incorporating sublinear data structures into generic circuit-based protocols.

Another approach is to design a customized protocol for a particular problem. This has some significant disadvantages over using a generic protocol. For one, it requires designing and proving the security of a custom protocol. It also may not integrate with generic protocols, so even if there is an efficient custom

protocol for computing a particular function, privacy-preserving applications often require additional pre-processing or post-processing around that function to be useful, so it may not be possible to use a custom protocol without also developing methods for connecting it with a generic protocol. Finally, although hardening techniques are known for generic protocols (Section 6), it may not be possible to (efficiently) harden a customized protocol to work in a malicious security setting.

Nevertheless, several specialized problems do benefit from tailored solutions and the performance gains possible with custom protocols may be substantial. In this work we briefly review one such practically important problem: *private set intersection*.

3.8.1 Private Set Intersection (PSI)

The goal of private set intersection (PSI) is to enable a group of parties to jointly compute the intersection of their input sets, without revealing any other information about those sets (other than upper bounds on their sizes). Although protocols for PSI have been built upon generic MPC (Huang *et al.*, 2012a), more efficient custom protocols can be achieved by taking advantage of the structure of the problem.

We will present current state-of-the art two-party PSI (Kolesnikov *et al.*, 2016). It is built on the protocol of Pinkas *et al.* (2015), which heavily uses Oblivious PRF (OPRF) as a subroutine. OPRF is an MPC protocol which allows two players to evaluate a PRF F, where one of the players holds the PRF key k, and the other player holds the PRF input x, and the second player gets $F_k(x)$. We first describe how to obtain PSI from OPRF, and then we briefly discuss the OPRF construction. The improvement of Kolesnikov *et al.* (2016) is due to developing a faster OPRF.

PSI from OPRF. We now describe the Pinkas-Schneider-Segev-Zohner (PSSZ) construction (Pinkas *et al.*, 2015) building PSI from an OPRF. For concreteness, we describe the parameters used in PSSZ when the parties have roughly the same number n of items.

The protocol relies on Cuckoo hashing (Pagh and Rodler, 2004) with 3 hash functions, which we briefly review now. To assign n items into b bins using Cuckoo hashing, first choose random functions $h_1, h_2, h_3 : \{0,1\}^* \to [b]$

and initialize empty bins $\mathcal{B}[1, \ldots, b]$. To hash an item x, first check to see whether any of the bins $\mathcal{B}[h_1(x)]$, $\mathcal{B}[h_2(x)]$, $\mathcal{B}[h_3(x)]$ are empty. If so, then place x in one of the empty bins and terminate. Otherwise, choose a random $i \in \{1, 2, 3\}$, evict the item currently in $\mathcal{B}[h_i(x)]$ and replace it with x, and then recursively try to insert the evicted item. If this process does not terminate after a certain number of iterations, then the final evicted element is placed in a special bin called the *stash*.

PSSZ uses Cuckoo hashing to implement PSI. First, the parties choose 3 random hash functions h_1, h_2, h_3 suitable for 3-way Cuckoo hashing. Suppose Alice has input set X and Bob has input set Y, where $|X| = |Y| = n$. Bob maps his items into $1.2n$ bins using Cuckoo hashing and a stash of size s. At this point Bob has at most one item per bin and at most s items in his stash — he pads his input with dummy items so that each bin contains exactly one item and the stash contains exactly s items.

The parties then run $1.2n + s$ instances of an OPRF, where Bob plays the role of receiver and uses each of his $1.2n + s$ items as input to the OPRF. Let $F(k_i, \cdot)$ denote the PRF evaluated in the i-th OPRF instance. If Bob has mapped item y to bin i via Cuckoo hashing, then Bob learns $F(k_i, y)$; if Bob has mapped y to position j in the stash, then Bob learns $F(k_{1.2n+j}, y)$.

On the other hand, Alice can compute $F(k_i, \cdot)$ for any i. So she computes sets of candidate PRF outputs:

$$H = \{F(k_{h_i(x)}, x) \mid x \in X \text{ and } i \in \{1, 2, 3\}\}$$
$$S = \{F(k_{1.2n+j}, x) \mid x \in X \text{ and } j \in \{1, \ldots, s\}\}$$

She randomly permutes elements of H and elements of S and sends them to Bob. Bob can identify the intersection of X and Y as follows. If Bob has an item y mapped to the stash, he checks whether the associated OPRF output is present in S. If Bob has an item y mapped to a hashing bin, he checks whether its associated OPRF output is in H.

Intuitively, the protocol is secure against a semi-honest Bob by the PRF property. For an item $x \in X \setminus Y$, the corresponding PRF outputs $F(k_i, y)$ are pseudorandom. Similarly, if the PRF outputs are pseudorandom even under related keys, then it is safe for the OPRF protocol to instantiate the PRF instances with related keys.

The protocol is correct as long as the PRF does not introduce any further

collisions (i.e., $F(k_i, x) = F(k_{i'}, x')$ for $x \neq x'$). We must carefully set the parameters required to prevent such collisions.

More efficient OPRF from 1-out-of-∞ OT. Kolesnikov *et al.* (2016) developed an efficient OPRF construction for the PSI protocol, by pushing on the coding idea from Section 3.7.2. The main technical observation is pointing out that the code C need not have many of the properties of error-correcting codes. The resulting pseudorandom codes enable an 1-out-of-∞ OT, which can be used to produce an efficient PSI.

In particular,

1. it makes no use of decoding, thus the code does not need to be efficiently decodable, and

2. it requires only that for all possibilities r, r', the value $C(r) \oplus C(r')$ has Hamming weight at least equal to the computational security parameter κ. In fact, it is sufficient even if the Hamming distance guarantee is only probabilistic — i.e., it holds with overwhelming probability over choice of C (we discuss subtleties below).

For ease of exposition, imagine letting C be a random oracle with suitably long output. Intuitively, when C is sufficiently long, it should be hard to find a near-collision. That is, it should be hard to find values r and r' such that $C(r) \oplus C(r')$ has low (less than a computational security parameter κ) Hamming weight. A random function with output length $k = 4\kappa$ suffices to make near-collisions negligible (Kolesnikov *et al.*, 2016).

We refer to such a function C (or family of functions, in our standard-model instantiation) as a *pseudorandom code* (PRC), since its coding-theoretic properties — namely, minimum distance — hold in a cryptographic sense.

By relaxing the requirement on C from an error-correcting code to a pseudorandom code, we *remove the a-priori bound* on the size of the receiver's choice string! In essence, the receiver can use *any string* as its choice string; the sender can associate a secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$ for any string r'. As discussed above, the receiver is only able to compute $H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r) \cdot s])$ — the secret corresponding to its choice string r. The property of the PRC is that, with overwhelming probability, all other values of $\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s]$ (that a

polytime player may ever ask) differ from $\boldsymbol{t}_j$ in a way that would require the receiver to guess at least κ bits of s.

Indeed, we can view the functionality achieved by the above 1-out-of-∞ OT as a kind of OPRF. Intuitively, $r \mapsto H(\boldsymbol{q} \oplus [C(r) \cdot s])$ is a function that the sender can evaluate on any input, whose outputs are pseudorandom, and which the receiver can evaluate only on its chosen input r.

The main subtleties in viewing 1-out-of-∞ OT as OPRF are:

1. the fact that the receiver learns slightly more than the output of this "PRF" — in particular, the receiver learns $\boldsymbol{t} = \boldsymbol{q} \oplus [C(r) \cdot s]$ rather than $H(\boldsymbol{t})$; and,

2. the fact that the protocol realizes many instances of this "PRF" but with related keys — s and C are shared among all instances.

Kolesnikov *et al.* (2016) show that this construction can be securely used in place of the OPRF in the PSSZ protocol, and can scale to support private intersections of sets (of any size element) with $n = 2^{20}$ over a wide area network in under 7 seconds.

Set intersection of multiple sets can be computed iteratively by computing pairwise intersections. However, extending the above 2PC PSI protocol to the multi-party setting is not immediate. Several obstacles need to be overcome, such as the fact that in 2PC computation one player learns the set intersection of the two input sets. In the multi-party setting this information must be protected. Efficient extension of the above PSI protocol to the multi-party setting was proposed by Kolesnikov *et al.* (2017a).

3.9 Further Reading

In this book, we aim to provide an easy to understand and exciting introduction to MPC, so omit a lot of formalization and proofs. Yao's GC, despite its simplicity, has several technical proof subtleties, which are first noticed and written out in the first formal account of Yao's GC (Lindell and Pinkas, 2009). The GMW protocol was introduced by Goldreich *et al.* (1987), but Goldreich (2004) provides a cleaner and more detailed presentation. The BGW and CCD protocols were developed concurrently by Ben-Or *et al.* (1988) and Chaum *et al.* (1988). Beaver *et al.* (1990) considered constant-round multiparty

protocols. A more detailed protocol presentation and discussion can be found in Phillip Rogaway's Ph.D. thesis (Rogaway, 1991).

Recently, a visual cryptography scheme for secure computation without computers was designed based on the GESS scheme (D'Arco and De Prisco, 2014; D'Arco and De Prisco, 2016). The OT extension of (Ishai *et al.*, 2003) is indeed one of the most important advances in MPC, and there are several extensions. Kolesnikov and Kumaresan (2013) and Kolesnikov *et al.* (2016) propose random 1-out-of-n OT and 1-out-of-∞ OT at a cost similar to that of 1-out-of-2 OT. The above schemes are in the semi-honest model; maliciously-secure OT extensions were proposed Asharov *et al.* (2015b) and Keller *et al.* (2015) (the latter is usually seen as simpler and more efficient of the two).

Custom PSI protocols have been explored in many different settings with different computation vs. communication costs and a variety of trust assumptions. Hazay and Lindell (2008) presented a simple and efficient private set intersection protocol that assumes one party would perform computations using a trusted smartcard. Kamara *et al.* (2014) present a server-aided private set intersection protocol, which, in the case of the semi-honest server, computes the private set intersection of billion-element sets in about 580 seconds while sending about 12.4 GB of data. This is an example of asymmetric trust, which we discuss further in Section 7.2.

There has been much research on custom protocols beyond PSI, but it is surprisingly rare to find custom protocols that substantially outperform fast generic MPC implementations of the same problem.

4

Implementation Techniques

Although secure computation protocols (as described in Section 3) were known since the 1980s, the first full implementation of a generic secure computation system was Fairplay (Malkhi *et al.*, 2004). Fairplay compiles a high-level description of a function into a circuit, described using a custom-designed Secure Hardware Description Language (SHDL). This circuit could then be executed as a protocol by a generator program and an evaluator program running over a network.

As a rough indication of the costs of MPC with Fairplay, the largest benchmark reported for Fairplay was finding the median of two sorted input arrays containing ten 16-bit numbers from each party. This required executing 4383 gates, and took over 7 seconds on a local area network (the time was then dominated by the oblivious transfer, not the garbled circuit execution). Modern MPC frameworks can execute millions of gates per second, and scale to circuits computing complex functions on large inputs, with hundreds of billions of gates.

Perhaps a factor of ten of the improvement can be attributed to general improvements in computing and network bandwidth (the Fairplay results are on a LAN with 618 Mbps, compared to 4 Gbps routinely available today), but the rest of the 3–4 orders of magnitude improvements are due primarily to the

advances in implementation techniques described in this section. These include optimizations that reduce the bandwidth and computational costs of executing a GC protocol (Section 4.1), improved circuit generation (Section 4.2), and protocol-level optimizations (Section 4.3). We focus on improvements to Yao's GC protocol, as the most popular generic MPC protocol, although some of the improvements discussed apply to other protocols as well. Section 4.4 briefly surveys tools and languages that have been developed for implementing privacy-preserving applications using MPC.

4.1 Less Expensive Garbling

The main costs of executing a garbled circuits protocol are the bandwidth required to transmit the garbled gates and the computation required to generate and evaluate the garbled tables. In a typical setting (LAN or WAN and moderate computing resources such as smartphone or a laptop), bandwidth is the main cost of executing GC protocols. There have been many improvements to the traditional garbling method introduced in Section 3.1.2; we survey the most significant ones next. Table 4.1 summarizes the impact of garbing improvements on the bandwidth and computation required to generate and evaluate a garbled gate. We described point-and-permute in Section 3.1.1; the other techniques are described in the next subsections.

technique	size		calls to H	
	XOR	AND	XOR	AND
classical	4	4	4	4
point-and-permute (1990) (Section 3.1.1)	4	4	4, 1	4, 1
row reduction (GRR3) (1999) (Section 4.1.1)	3	3	4, 1	4, 1
FreeXOR + GRR3 (2008) (Section 4.1.2)	0	3	0	4, 1
half gates (2015) (Section 4.1.3)	0	2	0	4, 2

Table 4.1: Garbling techniques (based on Zahur *et al.* (2015)). Size is number of "ciphertexts" (multiples of κ bits) transmitted per gate. Calls to H is the number of evaluations of H needed to evaluate each gate. When the number is different for the generator and evaluator, the numbers shown are the *generator calls*, *evaluator calls*.

4.1.1 Garbled Row Reduction

Naor *et al.* (1999) introduced *garbled row reduction* (GRR) as a way to reduce the number of ciphertexts transmitted per gate. The key insight is that it is not necessary for each ciphertext to be (unpredictable) encryption of a wire label. Indeed, one of the entries in each garbled table can be fixed to a predetermined value (say 0^κ), and hence need not be transmitted at all. For example, consider the garbled table below, where a and b are the input wires, and c is the output:

$H(a^1 \,\|\, b^0) \oplus\ c^0$
$H(a^0 \,\|\, b^0) \oplus\ c^0$
$H(a^1 \,\|\, b^1) \oplus\ c^1$
$H(a^0 \,\|\, b^1) \oplus\ c^0$

Since c^0 and c^1 are just arbitrary wire labels, we can select $c^0 = H(a^1 \,\|\, b^0)$. Thus, one of the four ciphertexts in each gate (say, the first one when it is sorted according point-and-permute order) will always be the all-zeroes string and does not need to be sent. We call this method *GRR3* since only three ciphertexts need to be transmitted for each gate.

Pinkas *et al.* (2009) describe a way to further reduce each gate to two ciphertexts, applying a polynomial interpolation at each gate. Because this is not compatible with the FreeXOR technique described next, however, it was rarely used in practice. The later half-gates technique (Section 4.1.3) achieves two-ciphertext AND gates and is compatible with FreeXOR, so supersedes the interpolation technique of Pinkas *et al.* (2009).

4.1.2 FreeXOR

One of the results of Kolesnikov (2005) was the observation that the GESS sharing for XOR gates can be done without any growth of the share sizes (Section 3.6). Kolesnikov (2005) found a lower bound for the minimum share sizes, explaining the necessity of the exponential growth for independent secrets. This bound, however, did not apply to XOR gates (or, more generally, to "even" gates whose truth table had two zeros and two ones).

As introduced in Section 3.6, XOR sharing for GESS can simply be done as follows. Let $s_0, s_1 \in \mathcal{D}_\mathcal{S}$ be the output wire secrets. Choose $\Delta \in_R \mathcal{D}_\mathcal{S}$ and set the shares $sh_{10} = \Delta, sh_{11} = s_0 \oplus s_1 \oplus \Delta, sh_{20} = s_0 \oplus \Delta, sh_{21} = s_1 \oplus \Delta$. The share reconstruction procedure on input sh_1, sh_2, outputs $sh_1 \oplus sh_2 = s_i$.

The above GESS XOR gate construction cannot be directly plugged into Yao's GC, though, since standard Yao's GC requires wires to be independently random, while in GESS XOR the shares on all of the gate's wires have the same offset. Indeed, it is easy to see that in the above XOR GESS, $sh_{10} \oplus sh_{11} = sh_{20} \oplus sh_{21} = s_0 \oplus s_1 = \Delta$. This breaks both correctness and security of GC. Indeed, correctness is broken because GESS XOR will not produce the pre-generated output wire secrets. Standard GC security proofs fail since GESS XOR creates correlations across wire labels, which are encryption keys. Even more problematic with respect to security is the fact that keys and encrypted messages are related to each other, creating *circular* dependences.

FreeXOR: Integrating GESS XOR into GC. FreeXOR is a GC technique introduced by Kolesnikov and Schneider (2008b). Their work is motivated by the fact that in GESS an XOR gate costs nothing (no garbled table needed, and share secrets don't grow), while traditional Yao's GC pays full price of generating and evaluating a garbled table for XOR gates. The GC FreeXOR construction enables the use of GESS XOR construction by adjusting GC secrets generation to repair the correctness broken by the naïve introduction of GESS XOR into GC, and observing that a strengthening of the assumptions on the encryption primitives used in the construction of GC garbled tables is sufficient for security of the new scheme.

FreeXOR integrates GESS XOR into GC by requiring that *all* the circuit's wires labels are generated with the same offset Δ. That is, we require that for each wire w_i of GC $\widehat{C}$ and its labels w_i^0, w_i^1, it holds that $w_i^0 \oplus w_i^1 = \Delta$, for a randomly chosen $\Delta \in_R \{0,1\}^\kappa$. Introducing this label correlation enables GESS XOR to correctly reconstruct output labels.

To address the security guarantee, FreeXOR uses Random Oracle (RO) for encryption of gates' output labels, instead of the weaker (PRG-based) encryption schemes allowed by Yao GC. This is necessary since inputs to different instances of H are correlated by Δ, and furthermore different values masked by H's output are also correlated by Δ. The standard security definition of a PRG does not guarantee that the outputs of H are pseudorandom in this case, but a random oracle does. Kolesnikov and Schneider mention that a variant of *correlation robustness*, a notion weaker than RO, is sufficient (Kolesnikov and Schneider, 2008b). In an important theoretical clarification of the FreeXOR

required assumptions, Choi *et al.* (2012b) show that the standard notion of correlation robustness is indeed not sufficient, and pin down the specific variants of correlation robustness needed to prove the security of FreeXOR.

The full garbling protocol for FreeXOR is given in Figure 4.1. The FreeXOR GC protocol proceeds identically to the standard Yao GC protocols of Figure 3.3, except that in Step 4, P_2 processes XOR gates without needing any ciphertexts or encryption: for an XOR-gate G_i with garbled input labels $w_a = (k_a, p_a), w_b = (k_b, p_b)$, the output label is directly computed as $(k_a \oplus k_b, p_a \oplus p_b)$.

Kolesnikov *et al.* (2014) proposed a generalization of FreeXOR called fleXOR. In fleXOR, an XOR gate can be garbled using 0, 1, or 2 ciphertexts, depending on structural and combinatorial properties of the circuit. FleXOR can be made compatible with GRR2 applied to AND gates, and thus supports two-ciphertext AND gates. The half gates technique described in the next section, however, avoids the complexity of fleXOR, and reduces the cost of AND gates to two ciphertexts with full compatibility with FreeXOR.

4.1.3 Half Gates

Zahur *et al.* (2015) introduced an efficient garbling technique that requires only two ciphertexts per AND gate and fully supports FreeXOR. The key idea is to represent an AND gate as XOR of two *half gates*, which are AND gates where one of the inputs is known to one of the parties. Since a half gate requires a garbled table with two entries, it can be transmitted using the garbled row reduction (GRR3) technique with a single ciphertext. Implementing an AND gate using half gates requires constructing a *generator half gate* (where the generator knows one of the inputs) and an *evaluator half gate* (where the evaluator knows one of the inputs). We describe each half gate construction next, and then show how they can be combined to implement an AND gate.

Generator Half Gate. First, consider the case of an AND gate where the input wires are a and b and the output wire is c. The generator half-AND gate computes $v_c = v_a \wedge v_b$, where v_a is somehow known to the circuit generator. Then, when v_a is false, the generator knows v_c is false regardless of v_b; when v_a is false, $v_c = v_b$. We use a^0, b^0, and c^0 to denote the wire labels encoding false for wires a, b, and c respectively. Using the FreeXOR design, the wire label

PARAMETERS:
Boolean Circuit C implementing function $\mathcal{F}$, security parameter κ.
Let $H : \{0,1\}^* \mapsto \{0,1\}^{\kappa+1}$ be a hash function modeled by a RO.

PROTOCOL:

1. Randomly choose global key offset $\Delta \in_R \{0,1\}^\kappa$.
2. For each input wire w_i of C, randomly choose its 0 label,
$$w_i^0 = (k_i^0, p_i^0) \in_R \{0,1\}^{\kappa+1}.$$
Set the other label $w_i^1 = (k_i^1, p_i^1) = (k_i^0 \oplus \Delta, p_i^0 \oplus 1)$.
3. For each gate G_i of C in topological order
   (a) If G_i is an XOR-gate $w_c = \mathsf{XOR}(w_a, w_b)$ with input labels $w_a^0 = (k_a^0, p_a^0), w_b^0 = (k_b^0, p_b^0), w_a^1 = (k_a^1, p_a^1), w_b^1 = (k_b^1, p_b^1)$:
      i. Set garbled output value $w_c^0 = (k_a^0 \oplus k_b^0, p_a \oplus p_b)$
      ii. Set garbled output value $w_c^1 = (k_a^0 \oplus k_b^0 \oplus \Delta, p_a \oplus p_b \oplus 1)$.
   (b) If G_i is a 2-input gate $w_c = g_i(w_a, w_b)$ with garbled labels $w_a^0 = (k_a^0, p_a^0), w_b^0 = (k_b^0, p_b^0), w_a^1 = (k_a^1, p_a^1), w_b^1 = (k_b^1, p_b^1)$:
      i. Randomly choose output label $w_c^0 = (k_c^0, p_c^0) \in_R \{0,1\}^{\kappa+1}$
      ii. Set output label $w_c^1 = (k_c^1, p_c^1) = (k_c^0 \oplus R, p_c^0 \oplus 1)$.
      iii. Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set
$$e_{v_a,v_b} = H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a,v_b)}.$$
      Sort entries e in the table by the input pointers, placing entry e_{v_a,v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$.
4. Compute the output tables, as in Figure 3.3.

Figure 4.1: FreeXOR Garbling

for b is either b^0 or $b^1 = b^0 \oplus \Delta$. The generator produces the two ciphertexts:

$$H(b^0) \oplus c^0$$
$$H(b^1) \oplus c^0 \oplus v_a \cdot \Delta$$

These are permuted according to the pointer bits of b^0, according to the point-and-permute optimization.

To evaluate the half gate and obtain $v_a \wedge v_b$, the evaluator takes a hash of its wire label for b (either b^0 or b^1) and decrypts the appropriate ciphertext. If the evaluator has b^0, it can compute $H(b^0)$ and obtain c^0 (the correct semantic false output) by xor-ing it with the first ciphertext. If the evaluator has $b^1 = b^0 \oplus \Delta$, it computes $H(b^1)$ to obtain $c^0 \oplus v_a \cdot \Delta$. If $v_a = 0$, this is c^0; if $v_a = 1$, this is $c^1 = c^0 \oplus \Delta$. Intuitively, the evaluator will never know both b^0 and b^1, hence the inactive ciphertext appears completely random. This idea was also used implicitly by Kolesnikov and Schneider, 2008b, Fig. 2, in the context of programming components of a universal circuit.

Crucially for performance, the two ciphertexts can be reduced to a single ciphertext by selecting c^0 according to the garbled row-reduction (Section 4.1.1).

Evaluator Half Gate. For the evaluator half gate, $v_c = v_a \wedge v_b$, the evaluator knows the value of v_a when the gate is evaluated, and the generator knows neither input. Thus, the evaluator can behave differently depending on the known plaintext value of wire a. The generator provides the two ciphertexts:

$$H(a^0) \oplus c^0$$
$$H(a^1) \oplus c^0 \oplus b^0$$

The ciphertexts are not permuted here—since the evaluator already knows v_a, it is fine (and necessary) to arrange them deterministically in this order. When v_a is false, the evaluator knows it has a^0 and can compute $H(a^0)$ to obtain output wire c^0. When v_a is true, the evaluator knows it has a^1 so can compute $H(a^1)$ to obtain $c_0 \oplus b_0$. It can then xor this with the wire label it has for b, to obtain either c_0 (false, when $b = b_0$) or $c^1 = c^0 \oplus \Delta$ (true, when $b^1 = b^0 \oplus \Delta$), without learning the semantic value of b or c. As with the generator half gate, using garbled row-reduction (Section 4.1.1) reduces the two ciphertexts to a single ciphertext. In this case, the generator simply sets $c^0 = H(a^0)$ (making the first ciphertext all zeroes) and sends the second ciphertext.

Combining Half Gates. It remains to show how the two half gates can be used to evaluate a gate $v_c = v_a \wedge v_b$, in a garbled circuit, where neither party can know the semantic value of either input. The trick is for the generator to generate a uniformly random bit r, and to transform the original AND gate into two half gates involving r:

$$v_c = (v_a \wedge r) \oplus (v_a \wedge (r \oplus v_b))$$

This has the same value as $v_a \wedge v_b$ since it distributes to $v_a \wedge (r \oplus r \oplus v_b)$. The first AND gate $(v_a \wedge r)$ can be garbled with a generator half-gate. The second AND gate $(v_a \wedge (r \oplus v_b))$ can be garbled with an evaluator half-gate, but only if $r \oplus v_b$ is leaked to the evaluator. Since r is uniformly random and not known to the evaluator, this leaks no sensitive information to the evaluator. The generator does not know v_b, but can convey $r \oplus v_b$ to the evaluator without any overhead, as follows. The generator will choose r to be the point-and-permute pointer bit of the false wire label on wire b, which is already chosen uniformly randomly. Thus, the evaluator learns $r \oplus v_b$ directly from the pointer bit on the wire it holds for b without learning anything about v_b.

Since the XOR gates do not require generating and sending garbled tables by using FreeXOR, we can compute an AND gate with only two ciphertexts, two invocations of H, and two "free" XOR operations. Zahur *et al.* (2015) proved the security of the half-gates scheme for any H that satisfies correlation robustness for naturally derived keys. In all settings, including low-latency local area networks, both the time and energy cost of bandwidth far exceed the cost of computing the encryptions (see the next section for how H is computed in this and other garbling schemes), and hence the half-gates method is preferred over any other known garbling scheme. Zahur *et al.* (2015) proved that no garbling scheme in a certain natural class of "linear" schemes[1] could use fewer than two ciphertexts per gate. Hence, under these assumptions the half-gates scheme is bandwidth-optimal for circuits composed of two-input binary gates (see Section 4.5 for progress on alternatives).

[1] See Zahur *et al.* (2015) for a precise formulation of this class. Roughly speaking, the half-gates scheme is optimal among schemes that are allowed to call a random oracle and perform fixed linear operations on wire labels / garbled gate information / oracle outputs, where the choice of these operations depends only on standard point-and-permute (Section 3.1.1) bits.

4.1.4 Garbling Operation

Network bandwidth is the main cost for garbled circuits protocols in most practical scenarios. However, computation cost of GC is also substantial, and is dominated by calls to the encryption function implementing the random oracle H in garbling gates, introduced in Section 3.1.2. Several techniques have been developed to reduce that cost, in particular by taking advantage of built-in cryptographic operations in modern processors.

Since 2010, Intel cores have included special-purpose AES-NI instructions for implementing AES encryption, and most processors from other vendors include similar instructions. Further, once AES a key is set up (which involves AES round keys generation), the AES encryption is particularly fast. This combination of incentives motivated Bellare *et al.* (2013) to develop fixed-key AES garbling schemes, where H is implemented using fixed-key AES as a cryptographic permutation.

Their design is based on a *dual-key cipher* (Bellare *et al.*, 2012), where two keys are both needed to decrypt a ciphertext. Bellare *et al.* (2012) show how a secure dual-key cipher can be built using a single fixed-key AES operation under the assumption that fixed-key AES is effectively a random permutation. Since the permutation is invertible, it is necessary to combine the permutation with the key using the Davies-Meyer construction (Winternitz, 1984): $\rho(K) = \pi(K) \oplus K$. Bellare *et al.* (2013) explored the space of secure garbling functions constructed from a fixed-key permutation, and found the fastest garbling method using $\pi(K||T)[1:k] \oplus K \oplus X$ where $K \leftarrow 2A \oplus 4B$, A and B are the wire keys, T is a tweak, and X is the output wire.

Gueron *et al.* (2015) pointed out that the assumption that fixed-key AES behaves like a random permutation is non-standard and may be questionable in practice (Biryukov *et al.*, 2009; Knudsen and Rijmen, 2007). They developed a fast garbling scheme based only on the more standard assumption that AES is a pseudorandom function. In particular, they showed that most of the performance benefits of fixed-key AES can be obtained just by carefully pipelining the AES key schedule in the processor.

Note also that the FreeXOR optimization also requires stronger than standard assumptions (Choi *et al.*, 2012b), and the half-gates method depends on FreeXOR. Gueron *et al.* (2015) showed a garbling construction alternative to FreeXOR that requires only standard assumptions, but requires a single

ciphertext for each XOR gate. Moreover, their construction is compatible with a scheme for reducing the number of ciphertexts needed for AND gates to two (without relying on FreeXOR, as is necessary for half gates). The resulting scheme has higher cost than the half-gates scheme because of the need to transmit one ciphertext for each XOR, but shows that it is possible to develop efficient (within about a factor of two of the cost of half gates) garbling schemes based only on standard assumptions.

4.2 Optimizing Circuits

Since the main cost of executing a circuit-based MPC protocol scales linearly with the size of the circuit, any reduction in circuit size will have a direct impact on the cost of the protocol. Many projects have sought ways to reduce the sizes of circuits for MPC. Here, we discuss a few examples.

4.2.1 Manual Design

Several projects have manually designed circuits to minimize the costs of secure computation (Kolesnikov and Schneider, 2008b; Kolesnikov *et al.*, 2009; Pinkas *et al.*, 2009; Sadeghi *et al.*, 2010; Huang *et al.*, 2011b; Huang *et al.*, 2012a), often focusing on reducing the number of non-free gates when FreeXOR is used. Manual circuit design can take advantage of opportunities that are not found by automated tools, but because of the effort required to manually design circuits, is only suitable for widely-used circuits. We discuss one illustrative example next; similar approaches have been used to design common building blocks optimized for secure computation such as multiplexers and adders, as well as more complex functions like AES (Pinkas *et al.*, 2009; Huang *et al.*, 2011b; Damgård *et al.*, 2012a).

Oblivious permutation. Shuffling an array of data in an oblivious permutation is an important building block for many privacy-preserving algorithms, including private set intersection (Huang *et al.*, 2012a) and square-root ORAM (Section 5.4). A basic component of an oblivious permutation, as well as many other algorithms, is a *conditional swapper* (also called an *X switching block* by Kolesnikov and Schneider (2008b) and Kolesnikov and Schneider (2008a)), which takes in two inputs, a_1 and a_2, and produces two outputs, b_1 and b_2.

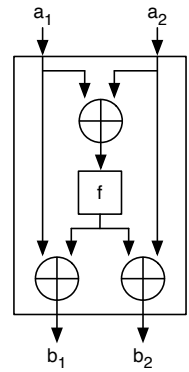


Figure 4.2: X switching block

Depending on the value of the swap bit p, either the outputs match the inputs ($b_1 = a_1$ and $b_2 = a_2$) or the outputs are the inputs in swapped order ($b_1 = a_2$ and $b_2 = b_1$). The swap bit p is known to the circuit generator, but must not be revealed to the evaluator. Kolesnikov and Schneider (2008b) provided a design for a swapper that takes advantage of FreeXOR and requires only a two-row garbled table (which can be reduced to a single ciphertext using garbled row reduction from Section 4.1.1). The swapper is implemented as:

$$\begin{aligned} b_1 &= a_1 \oplus (p \wedge (a_1 \oplus a_2)) \\ b_2 &= a_2 \oplus (p \wedge (a_1 \oplus a_2)) \end{aligned} \tag{4.1}$$

The swapper is illustrated in Figure 4.2. There the block f is set to 0 if no swapping is desired, and to 1 to implement swapping. The f block is implemented as a conjunction of the input with the programming bit p, so Figure 4.2 corresponds to Equation 4.1.

Since wire outputs can be reused, $p \wedge (a_1 \oplus a_2)$ only needs to be evaluated once. Referring back to the half-gates garbling and the notation of Section 4.1.3, when p is known to the generator, this conjunction is a generator half gate. As noted above, applying GRR3 allows this to be implemented with a single ciphertext.[2] With the above conditional swapper, a random oblivious permutation can be produced by the circuit generator by selecting a random permutation

[2]Indeed, the idea for half-gates garbling (Section 4.1.3) came from this X switching block design from Kolesnikov and Schneider (2008b).

and configuring the swappers in a Waksman network (Waksman, 1968) as necessary to produce it (Huang *et al.*, 2012a). Several permutation block designs, including truncated permutation blocks, are presented by Kolesnikov and Schneider (2008a).

Low-depth circuits: optimizing for GMW. For most of this chapter, we focus on Yao's GC, where a single round of communication is sufficient and the cost of execution scales with the size of the circuit. While most of the circuit-based optimizations apply to other protocols also, it is important to consider variations in cost factors when designing circuits for other protocols. In particular, in the GMW protocol (Section 3.2) each AND gate evaluation requires an OT, and hence a round of communication. AND gates on the same level can be batched and executed in the same round. Thus, unlike Yao's GC where the cost of execution is independent of its depth, for GMW protocol executions the cost depends heavily on the depth of the circuit.

Choi *et al.* (2012a) built an efficient implementation of GMW by using OT precomputation (Beaver, 1995) to reduce the on-line computation to a simple XOR, and OT extension protocols (Section 3.7.2) to reduce the overall cost of OT. A communication round (two messages) is still required during evaluation for each level of AND gates, however, so circuit execution time depends on the depth of the circuit. Schneider and Zohner (2013) provided further optimizations to the OT protocols for use in GMW, and designed low-depth circuits for several specific problems. Since GMW supports FreeXOR, the effective depth of a circuit is the maximum number of AND gates on any path. Schneider and Zohner (2013) were able to produce results on low-latency networks with GMW that were competitive with Yao's GC implementations by designing low-depth circuits for addition, squaring, comparison, and computing Hamming weight, and by using single-instruction multiple data (SIMD) instructions to pack operations on multiple bits, following on the approach used by Sharemind (Bogdanov *et al.*, 2008a).

4.2.2 Automated Tools

Boolean circuits go back to the earliest days of computing (Shannon, 1937). Because they have core applications in computing (e.g., in hardware components), there are a number of tools that have been developed to produce

efficient Boolean circuits. The output of some of these tools can be adapted to circuit-based secure computation.

CMBC-GC. Holzer *et al.* (2012) used a model checking tool as the basis for a tool that compiles C programs into Boolean circuits for use in a garbled circuits protocol. CBMC (Clarke *et al.*, 2004) is a bounded model checker designed to verify properties of programs written in ANSI C. It works by first translating an input program (with assertions that define the properties to check) into a Boolean formula, and then using a SAT solver to test the satisfiability of that formula. CBMC operates at the level of bits in the machine, so the Boolean formula it generates is consistent with the program semantics at the bit level. When used as a model checker, CBMC attempts to find a satisfying assignment of the Boolean formula corresponding to the input program. If a satisfying assignment is found, it corresponds to a program trace that violates an assertion in the program. CBMC unrolls loops and inlines recursive function calls up to the given model-checking bound, removing cycles from the program. For many programs, CBMC can statically determine the maximum number of loop iterations; when it cannot, programmers can use annotations to state this explicitly. When used in bounded model checking, an assertion is inserted that will be violated if the unrolling was insufficient. Variables are replaced by bit vectors of the appropriate size, and the program is converted to single-static assignment form so that fresh variables are introduced instead of assigning to a given variable more than once.

Normally, CBMC would convert the program to a Boolean formula, but internally it is represented as a circuit. Hence, CBMC can be used as a component in a garbled circuits compiler that translates an input program in C into a Boolean circuit. To build CBMC-GC, Holzer *et al.* (2012) modified CBMC to output a Boolean circuit which can be then executed in circuit-based secure computation framework (such as the one from Huang *et al.* (2011b), which was used by CMBC-GC). Since CBMC was designed to optimize circuits for producing Boolean formulas for SAT solvers, modifications were done to produce better circuits for garbled circuit execution. In particular, XOR gates are preferred in GC execution due to the FreeXOR technique (whereas the corresponding costs in model checking favor AND gates). To minimize the number of non-free gates, Holzer *et al.* (2012) replaced the built-in circuits

CMBC would use for operations like addition and comparison, with designs that minimize costs with free XOR gates.

TinyGarble. Another approach to generating circuits for MPC is to leverage the decades of effort that have been invested in hardware circuit synthesis tools. Hardware description language (HDL) synthesis tools transform a high-level description of an algorithm, which could be written in a programming language or common HDL language such as Verilog, into a Boolean circuit. A synthesis tool optimizes a circuit to minimize its size and cost, and then outputs a *netlist*, which is a straightforward description of a circuit as a list of logic gates with connected inputs and outputs.

Conventional hardware synthesis tools, however, do not generate circuits suitable for MPC protocols because they generate circuits that may have cycles and they are designed to optimize for different costs that are encountered the MPC execution (in particular, they assume the cost of gates based on hardware implementations). With TinyGarble, Songhori *et al.* (2015) overcame these problems in adapting circuit synthesis tools for generating circuits for MPC, and Yao's GC in particular.

The approach of TinyGarble is to use sequential logic in a garbled circuit. In a sequential circuit, instead of just connecting gates in a combinational way where each gate's outputs depend only on its inputs and no cycles are permitted, a circuit also can maintain state. In hardware, circuit state would be stored in a memory element (such as a flip flop), and updated with each clock cycle. To execute (generate and send) a garbled circuit, TinyGarble instead unrolls a sequential circuit, so the stored state is an additional input to the circuit, and new garbled gates are generated for each iteration. This means the representation is compact, even for large circuit executions, which allows performance improvement due to the ability to store the circuit in processor cache and avoid expensive RAM accesses. This method trades off a slight increase in the number of garbled gates to execute for a reduction in the size of the circuit representation.

In addition, TinyGarble uses a custom circuit synthesis library to enable the circuit synthesis tool to produce cost-efficient circuits for MPC. This includes a library of custom-designed circuits for common operations like a multiplexer, based on previous designs (Section 4.2.1). Another input to a circuit synthesis

tool is a *technology library*, that describes the logic units available on the target platform and their costs and constraints, and use this to map a structural circuit to a gate-level netlist. To generate circuits that take advantage of FreeXOR, the custom library developed for TinyGarble sets the area of an XOR gate to 0, and the area of other gates to a cost that reflects the number of ciphertexts required. When the circuit synthesis tool is configured to minimize circuit area, this produces circuits with an optimized number of non-XOR gates.

Songhori *et al.* (2015) report 67% reduction in the number of non-XOR gate in 1024-bit multiplication as compared to prior automatically generated circuits for same function. When looking at a more diverse function set (implementation of the MIPS processor), the circuit generation has modest performance improvement as compared to prior work (for example, the synthesized MIPS CPU circuit size is reduced by less than 15% in the number of non-XOR gates compared to a straightforward assembly of MIPS CPU from constituent blocks).

4.3 Protocol Execution

The main limit on early garbled circuit execution frameworks, starting with Fairplay (Malkhi *et al.*, 2004), is that they needed to generate and store the entire garbled circuit. Early on, researchers focused on the performance for smaller circuits and developed tools that naïvely generate and store the entire garbled circuit. This requires a huge amount of memory for all but trivial circuits, and limited the size of inputs and complexity of functions that could be computed securely. In this section, we discuss various improvements to the way MPC protocols are executed that have overcome these scaling issues and eliminated much of the overhead of circuit execution.

Pipelined Execution. Huang *et al.* (2011b) introduced *garbled circuit pipelining*, which eliminated the need for either party to ever store the entire garbled circuit. Instead of generating the full garbled circuit and then sending it, the circuit generation and evaluation phases are interleaved. Before the circuit execution begins, both parties instantiate the circuit structure, which is small relative to the size of the full garbled circuit since it can reuse components and is made of normal gate representations instead of non-reusable garbled gates.

To execute the protocol, the generator produces garbled gates in an order that is determined by the topology of the circuit, and transmits the garbled tables to the evaluator as they are produced. As the client receives them, it associates each received garbled table with the corresponding gate of the circuit. Since the order of generating and evaluating the circuit is fixed according to the circuit (and must not depend on the parties' private inputs), keeping the two parties synchronized requires essentially no overhead. As it evaluates the circuit, the evaluator maintains a set of live wire labels and evaluates the received gates as soon as all their inputs are ready. This approach allows the storage for each gate to be reused after it is evaluated, resulting in much smaller memory footprint and greatly increased performance.

Compressing Circuits. Pipelining eliminates the need to store the entire garbled circuit, but still requires the full structural circuit to be instantiated. Sequential circuits, mentioned earlier in Section 4.2.2, are one way to overcome this by enabling the same circuit structure to be reused but require a particular approach to circuit synthesis and some additional overhead to maintain the sequential circuit state. Another approach, initiated by Kreuter *et al.* (2013), uses lazy generation from a circuit representation that supports bounded loops. Structural circuits are compactly encoded using a *Portable Circuit Format* (PCF), which is generated by a circuit compiler and interpreted as the protocol executes. The input to the the circuit compiler is intermediate-level stack machine bytecode output by the LCC front-end compiler (Fraser and Hanson, 1995), enabling the system to generate MPC protocols from different high-level programs. The circuit compiler takes in an intermediate-level description of a program to execute as an MPC, and outputs a compressed circuit representation. This representation is then used as the input to interpreters that execute the generator and evaluator for a Yao's GC protocol, although the same representation could be used for other interpreters to executed different circuit-based protocols.

The key insight enabling PCF's scalability is to evaluate loops without unrolling them by reusing the same circuit structure while having each party locally maintain the loop index. Thus, new garbled tables can be computed as necessary for each loop execution, but the size of the circuit, and local memory needed, does not grow with the number of iterations. PCF represents

Boolean circuits in a bytecode language where each input is a single bit, and the operations are simple Boolean gates. Additional operations are provided for duplicating wire values, and for making function calls (with a return stack) and indirect (only forward) jumps. Instructions that do not involve executing Boolean operators do not require any protocol operations, so can be implemented locally by each party. To support secure computation, garbled wire values are represented by unknown values, which cannot be used as the conditions for conditional branches. The PCF compiler implemented several optimizations to reduce the cost of the circuits, and was able to scale to circuits with billions of gates (e.g., over 42 billion gates, of which 15 billion were non-free, to compute 1024-bit RSA).

Mixed Protocols. Although generic MPC protocols such as Yao's GC and GMW can execute any function, there are often much more efficient ways to implement specific functions. For example, additively homomorphic encryption schemes (including Paillier (1999) and Damgård and Jurik (2001)) can perform large additions much more efficiently than can be done with Boolean circuits.

With homomorphic encryption, instead of jointly computing a function using a general-purpose protocol, P_1 encrypts its input and sends it to P_2. P_2 then uses the encryption homomorphism to compute (under encryption) a function on the encrypted input, and sends the encrypted result back to P_1. Unless the output of the homomorphic computation is the final MPC result, its plaintext value cannot be revealed. Kolesnikov *et al.* (2010) and Kolesnikov *et al.* (2013) describe a general mechanism for converting between homomorphically-encrypted and garbled GC values. The party that evaluates the homomorphic encryption, P_2, generates a random mask r, which is added to the output of the homomorphic encryption, $\mathsf{Enc}(x)$, before being sent to P_1 for decryption. Thus the value received by P_1 is $\mathsf{Enc}(x + r)$, which P_1 can decrypt to obtain $x + r$. To enter x into the GC evaluation, P_1 provides $x + r$ and P_2 provides r as their inputs into the GC. The garbled circuit performs the subtraction to cancel out the mask, producing a garbled representation of x. Several works have developed customized protocols for particular tasks that combine homomorphic encryption with generic MPC (Brickell *et al.*, 2007; Huang *et al.*, 2011c; Nikolaenko *et al.*, 2013a; Nikolaenko *et al.*, 2013b).

The TASTY compiler (Henecka *et al.*, 2010) provides a language for

describing protocols involving both homomorphic encryption and garbled circuits. It compiles a high-level description into a protocol combining garbled circuit and homomorphic encryption evaluation. The ABY (Arithmetic, Boolean, Yao) framework of Demmler *et al.* (2015) support Yao's garbled circuits and two forms of secret sharing: arithmetic sharings based on Beaver multiplication triples (Section 3.4) and Boolean sharings based on GMW (Section 3.2). It provides efficient methods for converting between the three secure encodings, and for describing a function that can be executed using a combination of the three protocols. Kerschbaum *et al.* (2014) developed automated methods for selecting which protocol performs best for different operations in a secure computation.

Outsourcing MPC. Although it is possible to run MPC protocols directly on low-power devices, such as smartphones (Huang *et al.*, 2011a), the high cost of bandwidth and the limited energy available for mobile devices makes it desirable to outsource the execution of an MPC protocol in a way that minimizes the resource needed for the end user device without compromising security. Several schemes have been proposed for off-loading most of the work of GC execution to an untrusted server including Salus (Kamara *et al.*, 2012) and (Jakobsen *et al.*, 2016).

We focus here on the scheme from Carter *et al.* (2016) (originally published earlier (Carter *et al.*, 2013)). This scheme targets the scenario where a mobile phone user wants to outsource the execution of an MPC protocol to a cloud service. The other party in the MPC is a server that has high bandwidth and computing resources, so the primary goal of the design is to make the bulk of the MPC execution be between the server and cloud service, rather than between the server and mobile phone client. The cloud service may be malicious, but it is assumed not to collude with any other party. It is a requirement that no information about either the inputs or outputs of the secure function evaluation are leaked to the cloud service. This security notion of a non-colluding cloud is formalized by Kamara *et al.* (2012). The Carter *et al.* (2016) protocol supports malicious security, building on several techniques, some of which we discuss in Section 6.1. To obtain the inputs with lower resources from the client, the protocol uses an outsourced oblivious transfer protocol. To provide privacy of the outputs, a blinding circuit is added to the

original circuit that masks the output with a random pad known only to the client and server. By moving the bulk of the garbled circuit execution cost to the cloud service, the costs for the mobile device can be dramatically reduced.

4.4 Programming Tools

Many programming tools have been developed for building privacy-preserving applications using MPC. These tools vary by the input languages they support, how they combine the input program into a circuit and how the output is represented, as well as the protocols they support. Table 4.2 provides a high-level summary of selected tools for building MPC applications. We don't attempt to provide a full survey of MPC programming tools here, but describe one example of a secure computation programming framework next.

Obliv-C. The Obliv-C language is a strict extension of C that supports all C features, along with new data types and control structures to support data-oblivious programs that will be implemented using MPC protocols. Obliv-C is designed to provide high-level programming abstractions while exposing the essential data-oblivious nature of such computations. This allows programmers to implement libraries for data-oblivious computation that include low-level optimizations without needing to specify circuits.

In Obliv-C, a datatype declared with the **obliv** type modifier is oblivious to the program execution. It is represented in encrypted form during the protocol execution, so nothing in the program execution can depend on its semantic value. The only way any values derived from secret data can be converted back to a semantic value is by calling an explicit reveal function. When this function is invoked by both parties on the same variable, the value is decrypted by the executing protocol, and its actual value is now available to the program.

Control flow of a program cannot depend on oblivious data since its semantic value is not available to the execution. Instead, Obliv-C provides oblivious control structures. For example, consider the following statement where x and y are **obliv** variables:

```
obliv if (x > y) x = y;
```

Since the truth value of the x > y condition will not be known even at runtime, there is no way for the execution to know if the assignment occurs. Instead,

```
typedef struct {
    obliv int *arr;
    obliv int sz;
    int maxsz;
} Resizeable;

void writeArray(Resizeable *r, obliv int index, obliv int val) obliv;

// obliv function, may be called from inside oblivious conditional context
void append(Resizable *r, obliv int val) obliv {
    ~obliv(_c) {
        r→arr = reallocateMem(r→arr, r→maxsz + 1);
        r→maxsz++;
    }
    writeArray(r, r→sz, val);
    r→sz++;
}
```

Figure 4.3: Example use of an unconditional block (extracted from Zahur and Evans (2015)).

every assignment statement inside an oblivious conditional context must use "multiplexer" circuits that select based on the semantic value of the comparison condition within the MPC whether to perform the update or have no effect. Within the encrypted protocol, the correct semantics are implemented to ensure semantic values are updated only on the branch that would actually be executed based on the oblivious condition. The program executing the protocol (or an analyst reviewing its execution) cannot determine which path was actually executed since all of the values are encrypted within the MPC.

Updating a cleartext value z within an oblivious conditional branch would not leak any information, but would provide unexpected results since the update would occur regardless of whether or not the oblivious conditional is true. Obliv-C's type system protects programmers from mistakes where non-**obliv** values are updated in conditional contexts. Note that the type checking is not necessary for security since the security of the **obliv** values is enforced at runtime by the MPC protocol. It only exists to help the programmers avoid mistakes by providing compile time errors for non-sensical code.

To implement low-level libraries and optimizations, however, it is useful

for programmers to escape that type system. Obliv-C provides an unconditional block construct that can be used within an oblivious context but contains code that executes unconditionally. Figure 4.3 shows an example of how an unconditional block (denoted with ~**obliv**(var)) can be used to implement oblivious data structures in Obliv-C. This is an excerpt of an implementation of a simple resizable array implemented using a **struct** that contains oblivious variables representing the content and actual size of the array, and an opaque variable representing its maximum possible size. While the current length of the array is unknown (since we might append() while inside an **obliv if**), we can still use an unconditional block to track a conservative upper bound of the length. We use this variable to allocate memory space for an extra element when it might be needed.

This simple example illustrates how Obliv-C can be used to implement low-level optimizations for complex oblivious data structures, without needing to implement them at the level of circuits. Obliv-C has been used to implement libraries for data-oblivious data structures supporting random access memory including Square-Root ORAM (Section 5.4) and Floram (Section 5.5), and to implement some of the largest generic MPC applications to date including stable matching at the scale needed for the national medical residency match (Doerner *et al.*, 2016), an encrypted email spam detector (Gupta *et al.*, 2017), and a commercial MPC spreadsheet (Calctopia, Inc., 2017).

4.5 Further Reading

Many methods for improving garbling have been proposed beyond the ones covered in Section 4.1. As mentioned in Section 4.1.3, the half-gates scheme is bandwidth optimal under certain assumptions. Researchers have explored several ways to reduce bandwidth by relaxing those assumptions including garbling schemes that are not strictly "linear" in the sense considered in the optimality proof (Kempka *et al.*, 2016), using high fan-in gates (Ball *et al.*, 2016) and larger lookup tables (Dessouky *et al.*, 2017; Kennedy *et al.*, 2017). MPC protocols are inherently parallelizable, but with further effort circuits can be designed to maximize the benefits of parallel execution (Buescher and Katzenbeisser, 2015). GPUs also provide an opportunity for further speeding up MPC execution (Husted *et al.*, 2013).

Many other MPC programming tools have been developed. Wysteria (Ras-

Tool / Input Language	Output/Execution	Protocols
ABY / Custom low-level Demmler *et al.*, 2015	Virtual machine executes protocol	Arithmetic and Boolean sharings and GC (§4.3)
EMP / C++ Library Wang *et al.*, 2017a	Compiled to executable	Authenticated Garbling (§6.7), others
Frigate / Custom (C-like) Mood *et al.*, 2016	Interprets compact Boolean circuit	Yao's GC, malicious-secure with DUPLO
Obliv-C / C + extensions Zahur and Evans, 2015	Source-to-source (C)	Yao's GC, Dual Execution (§7.6)
PICCO / C + extensions Zhang *et al.*, 2013	Source-to-source (C)	3+-party secret-sharing

Table 4.2: Selected MPC Programming tools. In this table, we focus on tools that are recently or actively developed, and that provide state-of-the-art performance. The DUPLO extension is from (Kolesnikov *et al.*, 2017b). All of the listed tools are available as open source code: ABY at https://github.com/encryptogroup/ABY; EMP at https://github.com/emp-toolkit; Frigate at https://bitbucket.org/bmood/frigaterelease; Obliv-C at https://oblivc.org; PICCO at https://github.com/PICCO-Team/picco.

togi *et al.*, 2014) provides a type system that supports programs that combine local and secure computation. SCAPI (Bar-Ilan Center for Research in Applied Cryptography and Cyber Security, 2014; Ejgenberg *et al.*, 2012) provides Java implementations of many secure computation protocols. We focused on tools mostly building on garbled circuit protocols, but many tools implement other protocols. For example, the SCALE-MAMBA system (Aly *et al.*, 2018) compiles programs written in a custom Python-like language (MAMBA) to execute both the offline and online phases of secure computation protocols built on BDOZ and SPDZ (Section 6.6.2). We focused on programming tools in the dishonest majority setting, but numerous tools have been built supporting other threat models. In particular, very efficient implementations are possible with assuming three-party, honest-majority model, most notably Sharemind (Bogdanov *et al.*, 2008b) and Araki *et al.* (2017) (Section 7.1.2).

5

Oblivious Data Structures

Standard circuit-based execution is not well-suited to programs relying on random access to memory. For example, executing a simple array access where the index is a private variable (we use the $< z >$ notation to indicate that the variable z is private, with its semantic value protected by MPC),

```
a[<i>] = x
```

requires a circuit that scales linearly in the size of the array a. A natural circuit consists of N multiplexers, as shown in Figure 5.1. This method, where every element of a data structure is touched to perform an oblivious read or an update, is known as *linear scan*. For practical computations on large data structures, it is necessary to provide sublinear access operations. However, any access that only touches a subset of the data potentially leaks information about protected data in the computation.

In this chapter, we discuss several extensions to circuit-based MPC designed to enable efficient applications using large data structures. One strategy for providing sublinear-performance data structures in oblivious computation is to design data structures that take advantage of predictable access patterns. Indeed, it is not necessary to touch the entire data structure if the parts that are accessed do not depend on any private data (Section 5.1). A more general strategy, however, requires providing support for arbitrary memory access with sublinear

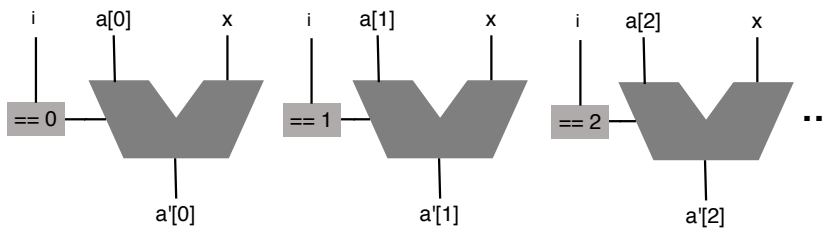


Figure 5.1: A single array access requiring n multiplexers.

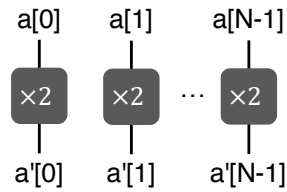


Figure 5.2: Oblivious array update with predictable access pattern.

cost. This cannot be achieved within a general-purpose MPC protocol, but can be achieved by combining MPC with oblivious RAM (Sections 5.2–5.5).

5.1 Tailored Oblivious Data Structures

In some programs the access patterns are predictable and known in advance, even though they may involve private data. As a simple example, consider this loop that doubles all elements of an array of private data:

```
for (i = 0; i < N; i++) {
    a[i] = 2 * a[i]
}
```

Instead of requiring N linear scan array accesses for each iteration (with $\Theta(N^2)$ total cost), the loop could be unrolled to update each element directly, as shown in Figure 5.2. Since the access pattern required by the algorithm is completely predictable, there is no information leakage in just accessing each element once to perform its update.

Most algorithms access data in a way that is not fully and as obviously predictable as in the above example. Conversely, usually it is done in a way that

is not fully data-dependent. That is, it might be *a priori* known (i.e., known independently of the private inputs) that some access patterns are guaranteed to never occur in the execution. If so, an MPC protocol that does not include the accesses that are known to be impossible regardless of the private data may still be secure. Next, we describe oblivious data structures designed to take advantage of predictable array access patterns common in many algorithms.

Oblivious Stack and Queue. Instead of implementing a stack abstraction using an array, an efficient oblivious stack takes advantage of the inherent locality in stack operations—they always involve the top of the stack. Since stack operations may occur in conditional contexts, though, the access pattern is not fully predictable. Hence, an oblivious stack data structure needs to provide conditional operations which take as an additional input a protected Boolean variable that indicates whether the operation actually occurs. For example, the <stack>.condPush(<b>, <v>) operation pushes v on the stack when the semantic value of b is true, but has no impact on the semantic state of the stack when b is false.

A naïve implementation of condPush would be to use a series of multiplexers to select for each element of the resulting stack either the current element, stack[i] when b is false, or the previous element, stack[i − 1] (or pushed value v, for the top element) when b is true. As with a linear scan array, however, this implementation would still require a circuit whose size scales with the maximum current size of the stack for each stack operation.

A more efficient data structure uses a hierarchical design, dividing the stack representation into a group of buffers where each has some slack space so it is not necessary to touch the entire data structure for each operation. The design in Zahur and Evans (2013), inspired by Pippenger and Fischer (1979), divides the stack into buffers where the level-i buffer has 5×2^i data slots. The top of the stack is represented by level 0. For each level, the data slots are managed in blocks/groups of 2^i slots; thus, for level 1, each data is always added in a block of two data items. For each block, a single bit is maintained that tracks whether the block is currently empty. For each level, a 3-bit counter, t, keeps track of the location of the next empty block available (0–5).

Figure 5.3 depicts an example of a conditional stack which starts off with some data already inserted and two condPush operations are illustrated. The

starting state in the figure depicts a state where none of the t values exceed 3, and hence there is guaranteed sufficient space for two conditional push operations. A multiplexer is used to push the new value into the correct slot based on the t_0 value, similar to the naïve stack circuit design described above. However, in this case, the cost is low since this is applied to a fixed 5-element array. After two conditional push operations, however, with the starting $t_0 = 3$, the level 0 buffer could be full. Hence, it is necessary to perform a shift, which either has no impact (if $t_0 \leq 3$), or pushes one block from level 0 into level 1 (as shown in Figure 5.3). After the shift, two more conditional push operations can be performed. This hierarchical design can extend to support any size stack, with shifts for level i generated for every 2^i `condPush` operations. A similar design can support conditional pop operations, where left shifts are required for level i after every 2^i `condPop` operations. The library implementing the conditional stack keeps track of the number of stack operations to know the minimum and maximum number of possible elements at each level, and inserts the necessary shifts to prevent overflow and underflows.

For all circuit-based MPC protocols, the primary cost is the bandwidth required to execute the circuit, which scales linearly in the number of gates. The cost depends on the maximum possible number of elements at each point in the execution. For a stack known to have at most N elements, k operations access level i at most $\lfloor k/2^i \rfloor$ times since we need a right shift for level i after every 2^i conditional push operations (and similarly, need a left shift after 2^i conditional pop operations).

However, the operations at the deeper levels are more expensive since the size of each block of elements at level i is 2^i, requiring $\Theta(2^i)$ logic gates to move. So, we need $\Theta(2^i \times k/2^i) = \Theta(k)$-sized circuits at level i. Thus, with $\Theta(\log N)$ levels, the total circuit size for k operations is $\Theta(k \log n)$ and the amortized cost for each conditional stack operation is $\Theta(\log N)$.

Other Oblivious Data Structures. Zahur and Evans (2013) also present a similar oblivious hierarchical queue data structure, essentially combining two stacks, one of which only supports push operations and the other that only supports pop operations. Since these stacks only need to support one of the conditional operations, instead of using a 5-block buffer at each level they use a 3-block buffer. Moving data between the two stacks requires an additional

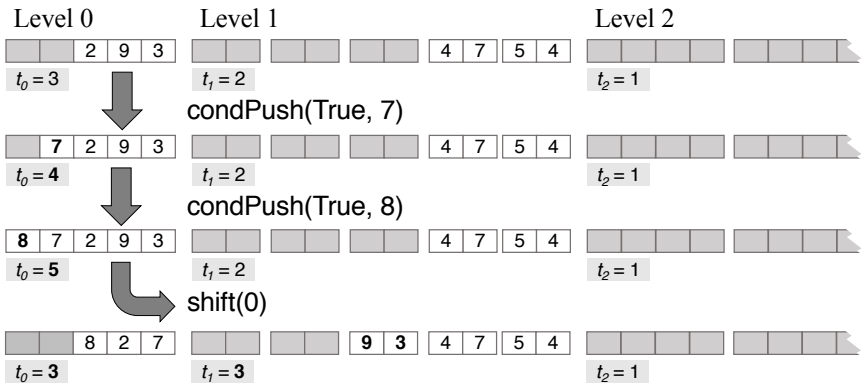


Figure 5.3: Illustration of two conditional push operations for oblivious stack. The shift(0) operation occurs after every two condPush operations.

multiplexer. Similarly to the oblivious stack, the amortized cost for operations on the oblivious queue is $\Theta(\log N)$.

Data structures designed to provide sublinear-cost oblivious operations can be used for a wide range of memory access patterns, whenever there is sufficient locality and predictability in the code to avoid the need to provide full random access. Oblivious data structures may also take advantage of operations that can be batched to combine multiple updates into single data structure scan. For example, Zahur and Evans (2013) present an associative map structure where a sequence of reads and writes with no internal dependencies can be batched into one update. This involves constructing the new value of the data structure by sorting the original values and the updates into a array, and only keeping the most recent value of each key-value pair. This allows up to N updates to be performed with a circuit of size $\Theta(N \log^2 N)$, with an amortized cost per update of $\Theta(\log^2 N)$.

The main challenge is writing programs to take advantage of predictable memory access patterns. A sophisticated compiler may be able to identify predictable access patterns in typical code and perform the necessary transformations automatically, but this would require a deep analysis of the code and no suitable compiler currently exists. Alternatively, programmers can manually rewrite code to use libraries that implement the oblivious data structures and manage all of the bookkeeping required to carry out the necessary shift

operations. Another strategy for building efficient oblivious data structures is to build upon a general-purpose Oblivious RAM, which is the focus of the rest of this chapter.

5.2 RAM-Based MPC

Oblivious RAM (ORAM) was introduced by Goldreich and Ostrovsky (1996) as a memory abstraction that allows arbitrary read and write operations without leaking any information about which locations are accessed. The original ORAM targeted the client-server setting, where the goal is to enable a client to outsource data to an untrusted server and perform memory operations on that outsourced data without revealing the data or access patterns to the server.

Ostrovsky and Shoup (1997) first proposed the general idea of using ORAM to support secure multi-party computation by splitting the role of the ORAM server between two parties. Gordon *et al.* (2012) were the first to propose a specific method for adapting ORAM to secure computation (RAM-MPC, also often called RAM-SC). In (Gordon *et al.*, 2012), the ORAM state is jointly maintained by both parties (client and server) within a secure computation protocol. The key idea is to have each party store a share of the ORAM's state, and then use a general-purpose circuit-based secure computation protocol to execute the ORAM access algorithms.

An ORAM system specifies an initialization protocol that sets up the (possibly already populated) storage structure, and an access protocol that implements oblivious read and write operations on the structure. To satisfy the oblivious memory goals, an ORAM system must ensure that the observable behaviors reveal nothing about the elements that are accessed. This means the physical access patterns produced by the access protocol for any same-length access sequences must be indistinguishable.

The initialization protocol takes as input an array of elements and initializes an oblivious structure with those elements without revealing anything about the initial values other than the number of elements. Assuming the access protocol is secure, it is always possible to implement the initialization protocol by performing the access protocol once for each input element. The costs of initializing an ORAM this way, however, may be prohibitive, especially as used in secure computation.

The RAM-MPC construction proposed by Gordon *et al.* (2012) implemented an ORAM-based secure multi-party computation. To implement an ORAM access, a circuit is executed within a 2PC that translates the oblivious logical memory location into a set of physical locations that must be accessed to perform the access. The physical locations are then revealed to the two parties, but the ORAM design guarantees that these leak no information about the logical location accessed. Each party then retrieves the data shares stored in those locations, and passes them into the MPC. To complete the access, a logical write occurs, as follows. The circuit executing within the MPC produces new data elements to be written back to each of the physical locations. These locations are output in plaintext, together with the data shares to be written into parties' local physical storage.

Gordon *et al.* (2012) proved that combining semi-honest 2PC with the semi-honest ORAM protocol where ORAM state is split between the two parties and operations are implemented within the 2PC results in a secure RAM-based MPC in the semi-honest model.[1]

5.3 Tree-Based RAM-MPC

The construction of Gordon *et al.* (2012) builds on the tree-based ORAM design of Shi *et al.* (2011). The underlying data structure in the tree-based ORAM storing N elements is a binary tree of height $\log N$, where each node in the tree holds $\log N$ elements. Each logical memory location is mapped to a random leaf node, and the logical index and value of the data element is stored in encrypted form in one of the nodes along the path between the tree root and that leaf. To access a data item, the client needs to map the item's location to its associated leaf node and retrieve from the server all nodes along the path from the root to that node. Each of the nodes along the path is decrypted and scanned to check if it is the requested data element.

A careful reader will notice the following technical difficulty. Performing data look up requires mapping the logical location to the leaf node. However, the size of this map is linear in N, and hence the map cannot be maintained by the client with sublinear storage. The solution is to store the location map

[1] RAM-MPC can also be made to work in the malicious security model (Afshar *et al.*, 2015), but care must be taken to ensure that data stored outside the MPC is not corrupted.

by the server using another instance of tree-based ORAM. Importantly, since the size of the index map is smaller than the size of the data, the size of the second ORAM tree can be smaller than N, as each item in the second tree can store several mappings. To support larger ORAMs, a sequence of look-up trees might be used, where only the smallest tree is stored by the client.

In RAM-MPC, the trees are secret-shared between the two parties. To access an element, the parties execute a 2PC protocol that takes the shares of the logical index and outputs (in shares to each party) the physical index for the next level. A linear scan is used on the reconstructed elements along the search path to identify the one corresponding to the requested logical index. In the final tree, the shares of the requested data element are output to the higher-level MPC protocol. Note that the data in each node could also be stored in an ORAM to avoid the need for a linear scan, but since the bucket sizes are small here and (at least with this ORAM design) there is substantial overhead required to support an ORAM, a simple linear scan is preferred.

To complete the data access procedure, we need to ensure that the subsequent access results in an oblivious access pattern, even if the same element is accessed again. To achieve this in tree-based ORAMs, the accessed logical location is re-mapped to a random leaf node, and the updated data value is inserted into the root node in the tree, ensuring its availability in the subsequent access. To prevent the root node from overflowing, the protocol of Shi *et al.* (2011) uses a balancing mechanism that pushes items down the tree after each ORAM access. Randomly-selected elements are *evicted* and are moved down the tree by updating both of the child nodes of the selected nodes (this is done to mask which leaf-path contains the evicted element).

Intuitively, the access pattern is indistinguishable from a canonical one, and hence an adversary cannot distinguish between two accesses. Indeed, every time an element is accessed, it is moved to a random-looking location. Further, every access retrieves a complete path from the root to a leaf, so as long as the mapping between logical locations and leaves is random and not revealed to the server, the server learns no information about which element is accessed. The scheme does have the risk that a node may overflow as evicted elements are moved down the tree, and not be able to store all of the elements required. The probability of an overflow after k ORAM accesses with each node holding $O(\log(\frac{kN}{\delta}))$ elements is shown by Shi *et al.* (2011) to be less than δ, which

is why the number of elements in each node is set to $O(\log N)$ to make the overflow probability negligible. The constant factors matter, however. Gordon *et al.* (2012) simulated various configurations to find that a binary search on a 2^{16} element ORAM (that is, only 16 operations) could be implemented with less than 0.0001 probability of overflow for $\delta = 32$.

Variations on this design improved the performance of tree-based ORAM for MPC have focused on using additional storage (called a *stash*) to store overflow elements and reduce the sizes of the buckets needed to provide negligible failure probability, as well as on improving the eviction algorithm.

Path ORAM. Path ORAM (Stefanov *et al.*, 2013) added a stash to the design as a fixed-size auxiliary storage for overflow elements, which would be scanned on each request. The addition of a small stash enabled a more efficient eviction strategy than the original binary-tree ORAM. Instead of selecting two random nodes at each level for eviction and needing to update both child nodes of the selected nodes to mask the selected element, Path ORAM performed evictions on the access path from the root to the accessed node, moving elements along this path from the root towards the leaves as much as possible. Since this path is already accessed by the request, no additional masking is necessary to hide which element is evicted. The Path ORAM design was adapted by Wang *et al.* (2014a) to provide a more efficient RAM-MPC design, and they presented a circuit design for a more efficient eviction circuit.

Circuit ORAM. Further advances in RAM-MPC designs were made both by adapting improvements in traditional client-server ORAM designs to RAM-MPC, as well as by observing differences between the costs and design space options between the MPC and traditional ORAM setting and designing ORAM schemes focused on the needs of MPC.

Wang *et al.* (2014a) argued that the main cost metric for ORAM designs used in circuit-based secure computation should be *circuit complexity*, whereas client-server designs were primarily evaluated based on (client-server) bandwidth metrics. When used within an MPC protocol, the execution costs of an ORAM are dominated by the bandwidth costs of executing the circuits needed to carry out ORAM accesses and updates within the MPC protocol.

Wang *et al.* (2015b) proposed Circuit ORAM, an ORAM scheme designed

specifically for optimal circuit complexity for use in RAM-MPC. Circuit ORAM replaced the complex eviction method of Path ORAM with a more efficient design where the eviction can be completed with a single scan of the blocks on the eviction path, incorporating both the selection and movement of data blocks within a single pass. Their key insight is to perform two metadata scans first, so as to determine which blocks are to be moved, together with their new locations. These scans can be run on the metadata labels, which are much smaller than the full data blocks. After these scans have determined which blocks to move, the actual data blocks can be moved using a single pass along the path from the stash-root to the leaf, storing at most one block of data to relocate as it proceeds.

Because the metadata scans are on much smaller data than the actual blocks, the concrete total cost of the scan is minimized. Wang *et al.* (2015b) proved that with block size of $D = \Omega(\log^2 N)$, Circuit ORAM can achieve statistical failure probability of δ for block by setting the stash size to $O(\log \frac{1}{\delta})$ with circuit size of $O(D(\log N + \log \frac{1}{\delta}))$. The optimizations in Circuit ORAM reduce the effective cost of ORAM (measured by the number of non-free gates required in a circuit) by a factor of over 30 compared to the initial binary-tree ORAM design for a representative 4GB data size with 32-bit blocks.

5.4 Square-Root RAM-MPC

Although the first proposed ORAM designs were hierarchical, early RAM-MPC designs did not adopt these constructions because their implementation seemed to require implementing a pseudo-random function (PRF) within the MPC, and using the outputs of that function to perform an oblivious sort. Both of these steps would be very expensive to do in a circuit-based secure computation circuit, so RAM-MPC designs favored ORAMs based on the binary-tree design which did not require sorting or a private PRF evaluation.

Zahur *et al.* (2016) observed that the classic square-root ORAM design of Goldreich and Ostrovsky (1996) could in fact be adapted to work efficiently in RAM-MPC by implementing an oblivious permutation where the PRF required for randomizing the permutation would be jointly computed by the two parties *outside* of the generic MPC. This led to a simple and efficient ORAM design, which, unlike tree-based ORAMs, has zero statistical failure probability, since there is no risk that a block can overflow. The design maintains a public

set, Used, of used physical locations (revealing no information since logical locations are assigned randomly to physical ones, and only accessed at most once), and an oblivious stash at each level that stores accessed blocks. Since the stash contains private data, it is stored in encrypted form as wire labels within the MPC. Unlike in the tree-based ORAM designs where the stash is used as a probabilistic mechanisms to deal with node overflows, in Square-Root ORAM the stash is used deterministically on each access. Each access adds one element to each of the level stashes, and all of the stashes must be linearly scanned on every access. If an accessed element is found in the stash, to preserve the obliviousness, the look-up continues with a randomly selected element. The size of each stash determines the number of accesses that can be done between reshufflings. Optimal results are obtained by setting the size to $\Theta(\sqrt{(N)})$, hence the name "square-root ORAM". The oblivious shuffling is performed using a Waksman network (Waksman, 1968), which requires $n \log_2 n - n + 1$ oblivious swaps to permute n elements. Using the design from Huang *et al.* (2012a), this can be done with one ciphertext per oblivious swap.

One major advantage of the Square-Root ORAM design is its concrete performance, including initialization. All that is required to initialize is produce a random permutation and obliviously permute all the input data, generating the oblivious initial position map using the same method as the update protocol. Compared to earlier ORAM designs, where initialization was done with repeated writes, this approach dramatically reduces the cost of using the ORAM in practice. Without considering initialization, Square-Root ORAM has a per-access cost that is better than linear scan, once there are more than 32 blocks (for typical block sizes of 16 or 32 bytes), whereas Circuit ORAM is still more expensive than linear scanning up to 2^{11} blocks. Although Square-Root ORAM has asymptotically worse behavior than Circuit ORAM, its concrete costs per access are better for ORAM sizes up to 2^{16} blocks. For such large ORAMs, initialization costs become an important factor — initializing a Square-Root ORAM requires $\Theta(\log N)$ network round trips, compared to $\Theta(N \log N)$ for Circuit ORAM. Initializing a Circuit ORAM with $N = 2^{16}$ blocks would take several days, and its asymptotic benefits would only be apparent for very expensive computations.

5.5 Floram

Doerner and Shelat (2017) observed that even the sublinear-cost requirement, which was an essential design aspect of traditional ORAM systems, was not necessary to be useful in RAM-MPC. Since the cost of secure computation far exceeds the cost of standard computation, ORAM designs that have linear cost "outside of MPC", but reduce the computation performed "inside MPC", may be preferred to sublinear ORAM designs. With this insight, Doerner and Shelat (2017) revisited the Distributed Oblivious RAM approach of Lu and Ostrovsky (2013) and based a highly scalable and efficient RAM-MPC scheme on two-server private information retrieval (PIR). The scheme, known as Floram (*Function-secret-sharing Linear ORAM*), can provide over 100× improvements over Square-Root ORAM and Circuit ORAM across a range of realistic parameters.

Distributed Oblivious RAM relaxes the usual security requirement of ORAM (the indistinguishability of server traces). Instead, the ORAM server is split into two non-colluding servers, and security requirement is that the memory access patterns are indistinguishable based on any single server trace (but allowed to be distinguishable if the traces of both servers are combined). We note that it is not immediately obvious how to use this this primitive in constructing two-party MPC, since it requires two non-colluding servers *in addition* to the third player—the ORAM client.

Private information retrieval (PIR) enables a client to retrieve a selected item from a server, without revealing to the server which item was retrieved (Chor *et al.*, 1995). Traditionally, PIR schemes are different from ORAM in that they only provide read operations, and that they allow a linear server access cost whereas ORAM aims for amortized sublinear retrieval cost.

A *point function* is a function that outputs 0 for all inputs, except one:

$$P^{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0, & \text{otherwise.} \end{cases}$$

Gilboa and Ishai (2014) introduced the notion of *distributed point functions* (DPF), where a point function is secret-shared among two players with shares that have sizes sublinear in the domain of the function, hiding the values of both α and β. The output of each party's evaluation of the secret-shared function is a share of the output and a bit indicating if the output is valid: $y_p^x = P_p^{\alpha,\beta}(x)$

(party p's share output of the function, and $t_p^x = (x = \alpha)$ (a share of 1 if $x = \alpha$, otherwise a share of 0). Gilboa and Ishai (2014) showed how this could be used to efficiently implement two-server private information retrieval, and Boyle *et al.* (2016b) improved the construction.

The Floram design uses secret-shared distributed point functions to implement a two-party oblivious write-only memory (OWOM) and both a two-party oblivious read-only memory (OROM). The ORAM is constructed by composing the OWOM and OROM, but since it is not possible to read from the write-only memory, Floram uses a linear-scan stash to store written elements until it is full, at which point the state of the ORAM is refreshed by converting the write-only memory into oblivious read-only memory, replacing the previous OROM and resetting the OWOM. In the OWOM, values are stored using XOR secret sharing. To write to an element, all elements are updated by xor-ing the current value with the output of a generated distributed point function—so, the semantic value of the update is 0 for all elements other than the one to be updated, and the difference between the current value and updated value for the selected element.

Reading. In the OROM, each stored value is masked by a PRF evaluated at its index and the masked value is secret-shared between the two OROMs. To read element i from the OROM, each party obtains k_p^i from the MPC corresponding to its key for the secret-shared DPF. Then, it evaluates $P_{k_p^i}(x)$ on each element of its OROM and combines all the results with xor. For each element other than $x = i$ the output is its share of 0, so the resulting sum is its share of the requested value, v_p^i. This value is input into the MPC, and xor-ed with the value provided by the other party to obtain $R^i = v_1^i \oplus v_2^i$. To obtain the actual value of element i, R^i is xor-ed with the output of $\mathsf{PRF}_k(i)$, computed within the MPC. The PRF masking is necessary to avoid leaking any information when the OROM is refreshed. Each read requires generating a DPF ($O(\log N)$ secure computation and communication), $O(N)$ local work for the DPF evaluation at each element, and constant-size (independent of N) secure computation to compute the PRF for unmasking. In addition, each read requires scanning the stash within the MPC using a linear scan in case the requested element has been updated since the last refresh. Hence, the cost of the scheme depends on how large a stash is needed to amortize the refresh cost.

Refreshing. Once the stash becomes full, the ORAM needs to be refreshed by converting the OWOM into a new OROM and clearing the stash. This is done by having each party generate a new PRF key (k_1 generated by P_1, k_2 generated by P_2) and masking all of the values currently stored in its OWOM with keyed PRF, $W'_p[i] = \mathsf{PRF}_{k_p}(i) \oplus W_p[i]$, for each party, $p \in \{1, 2\}$. The masked values are then exchanged between the two parties. The OROM values are computed by xor-ing the value received from the other party for each cell with their own masked value to produce $R[i] = \mathsf{PRF}_{k_1}(i) \oplus \mathsf{PRF}_{k_2}(i) \oplus W_1[i] \oplus W_2[i]$, where $v[i] = W_1[i] \oplus W_2[i]$. Each party passes in its PRF key to the MPC, so that values can be unmasked within MPC reads by computing $\mathsf{PRF}_{k_1}(i) \oplus \mathsf{PRF}_{k_2}(i)$ within the MPC. This enables the read value to be unmasked for used within the MPC, without disclosing the private index i. Thus, refreshing the stash requires $O(N)$ local computation and communication, and no secure computation. Because the refresh cost is relatively low, the optimal access period is $O(\sqrt{N})$ (with low constants, so their concrete implementation used $\sqrt{N}/8$).

Floram offers substantial performance improvements over Square-Root ORAM and all other prior ORAM constructions used in RAM-MPC, even though its asymptotic costs are linear. The linear-cost operations of the OROM and OWOM are implemented outside the MPC, so even though each access requires $O(N)$ computation, the concrete cost of this linear work is much less that the client computation done within the MPC. In Doerner and Shelat (2017)'s experiments, the cost of the secure computation is the dominant cost up to ORAMs with 2^{25} elements, after which the linear local computation cost becomes dominant. Floram was over able to scale to support ORAMs with 2^{32} four-byte elements with an average access time of 6.3 seconds over a LAN.

Floram also enables a simple and efficient initialization method using the same mechanism as used for refreshing. The Floram design can also support reads and writes where the index is not private very efficiently—the location i can be read directly from the OWOM just by passing in the secret-shared values in location i of each party's share into the MPC. Another important advantage of Floram is that instead of storing wire labels as is necessary for other RAM-MPC designs, which expands the memory each party must store by factor κ (the computational security parameter), each party only needs to store a secret share of the data which is the same as the original size of the data for each the OROM and OWOM.

5.6 Further Reading

Many other data structures have been proposed for efficient MPC, often incorporating ORAM aspects. (Keller and Scholl, 2014) proposed efficient MPC data structures for arrays, built on top of ORAM designs. (Wang *et al.*, 2014b) devised oblivious data structures including priority queues that take advantage of sparse and predictable access patterns in many applications, and presented a general pointer-based technique to support efficient tree-based access patterns.

We only touched on the extensive literature on oblivious RAM, focusing on designs for MPC. ORAM continues to be an active research area, with many different design options and tradeoffs to explore. Buescher *et al.* (2018) study various MPC-ORAM designs in application settings and developed a compiler that selects a suitable ORAM for the array-accesses in a high-level program. Faber *et al.* (2015) proposed a three-party ORAM based on Circuit ORAM that offers substantial cost reduction in the three-party, honest majority model. Another new direction that may be useful for MPC ORAM is to allow some amount of limited leakage of the data access pattern to gain efficiency (Chan *et al.*, 2017; Wagh *et al.*, 2018).

6

Malicious Security

So far, we have focused on semi-honest protocols which provide privacy and security only against passive adversaries who follow the protocol as specified. The semi-honest threat model is very weak. It makes assumptions that underestimate the power of realistic adversaries, for most scenarios. This chapter discusses several protocols, summarized in Figure 6.1, that are designed to resist malicious adversaries.

6.1 Cut-and-Choose

Yao's GC protocol is not secure against malicious adversaries. In particular, P_1 is supposed to generate and send a garbled version of $\mathcal{F}$ to P_2. A malicious

Protocol	Parties	Rounds	Based on
Cut-and-Choose (§6.1–6.4)	2	constant	Yao's GC (§3.1)
GMW compiler (§6.5.1)	many	(inherited)	any semi-honest
BDOZ & SPDZ (§6.6)	many	circuit depth	preprocessing
Authenticated garbling (§6.7)	many	constant	BMR (§3.5)

Figure 6.1: Summary of malicious MPC protocols discussed in this chapter.

P_1 may send the garbling of a different circuit that P_2 had not agreed to evaluate, but P_2 has no way to confirm the circuit is correct. The output of the maliciously-generated garbled circuit may leak more than P_2 has agreed to reveal (for instance, P_2's entire input).

Main idea: check some circuits, evaluate others. The standard way to address this problem is a technique called *cut-and-choose*, a general idea that goes back at least to Chaum (1983), who used it to support blind signatures.

To use cut-and-choose to harden Yao's GC protocol, P_1 generates many independent garbled versions of $\mathcal{F}$ and sends them to P_2. P_2 then chooses some random subset of these circuits and asks P_1 to "open" them by revealing all of the randomness used to generate the chosen circuits. P_2 then verifies that each opened garbled circuit is a correctly garbled version of the agreed-upon circuit $\mathcal{F}$. If any of the opened circuits are found to be generated incorrectly, P_2 knows P_1 has cheated and can abort. If all of the opened circuits are verified as correct, P_2 continues the protocol. Since the opened circuits have had all of their secrets revealed, they cannot be used for the secure computation. However, if all of the opened circuits were correct, P_2 has some confidence that most of the unopened circuits are also correct. These remaining circuits can then be evaluated as in the standard Yao protocol.

Generating an output. Cut-and-choose cannot ensure with non-negligible probability that *all* unopened circuits are correct. Let's say that P_1 generates s garbled circuits, and each one is checked with independent probability $\frac{1}{2}$. Then if P_1 generates only one of the circuits incorrectly, then with (non-negligible) probability $\frac{1}{2}$ that circuit will not be chosen for checking and will become one of the evaluation circuits. In this event P_2 may get inconsistent outputs from the evaluated circuits.

If P_2 sees inconsistent outputs, then it is obvious to P_2 that P_1 is misbehaving. It is tempting to suggest that P_2 should abort in this case. However, to do so would be insecure! Suppose P_1's incorrect circuits are designed to be selectively incorrect in a way that *depends on* P_2*'s input.* For example, suppose an incorrect circuit gives the wrong answer if the first bit of P_2's input is 1. Then, P_2 will only see disagreeing outputs if its first bit of input is 1. If P_2 aborts in this case, the abort will then leak the first bit of its input. So we are in

a situation where P_2 knows for certain that P_1 is cheating, but must continue as if there was no problem to avoid leaking information about its input.

Traditionally, cut-and-choose protocols address this situation by making P_2 consider only the *majority* output from among the evaluated circuits. The cut-and-choose parameters (number of circuits, probability of checking each circuit) are chosen so that

$$\Pr[\textit{majority of evaluated circuits incorrect} \ \wedge \textit{all check circuits correct}]$$

is negligible. In other words, if all of the check circuits are correct, P_2 can safely assume that the majority of the evaluation circuits are correct too. This justifies the choice to use the majority output.

Input consistency. In cut-and-choose-based protocols, P_2 evaluates several garbled circuits. The fact that there are several circuits presents an additional problem: a malicious party may try to use *different inputs* to different garbled circuits. *Input consistency* refers to the problem of ensuring that both parties use the same input to all circuits.

Ensuring consistency of P_2's inputs (i.e. preventing a malicious P_2 from submitting different inputs) is generally easier. Recall that in Yao's protocol, P_2 picks up garbled input using oblivious transfer (OT) as receiver. The parties can perform a single OT for each bit of P_2's input, whose payload is the garbled input corresponding to that bit, for *all* circuits. Because P_1 prepares the OT payloads, they can ensure that P_2 only receives garbled inputs corresponding to the same input for all circuits.

Ensuring input consistency for P_1 is more challenging. One approach (see Section 6.8 for others), proposed by shelat and Shen (2011), has the parties evaluate the function $((x, r), y) \mapsto (\mathcal{F}(x, y), H(x, r))$ where H is a 2-universal hash function. The 2-universal property is that for all $z \neq z'$, $\Pr[H(z) = H(z')] = \frac{1}{2^\ell}$, where the probability is over the random choice of H and ℓ is the output length of H in bits. The idea is to make P_1 first commit to its (garbled) inputs; then P_2 chooses a random H which determines the circuit to garble. Since P_1's inputs are fixed before H is chosen, any inconsistent inputs will lead to different outputs from H, which P_2 can detect. In order to ensure that the H-output does not leak information about P_1's input x, P_1 also includes additional randomness r as argument to H, which hides x when r

is sufficiently long. As shown by shelat and Shen (2011), multiplication by a random Boolean matrix is a 2-universal hash function. When such a function H is made public, the resulting circuit computing H consists of exclusively XOR operations, and therefore adds little cost to the garbled circuits when using the FreeXOR optimization.

Selective abort. Another subtle issue is even if all garbled circuits are correct, P_1 may still cheat by providing incorrect garbled inputs in the oblivious transfers. Hence it does not suffice to just check the garbled circuits for correctness. For instance, P_1 may select its inputs to the OT so that whenever P_2's first input bit is 1, P_2 will pick up garbage wire labels (and presumably abort, leaking the first input bit in the process). This kind of attack is known as a *selective abort* attack (sometimes called *selective failure*). More generally, we care about when P_1 provides *selectively* incorrect garbled inputs in some OTs (e.g., P_1 provides a correct garbled input for 0 and incorrect garbled input for 1), so that whether or not P_2 receives incorrect garbled inputs depends on P_2's input.

An approach proposed by Lindell and Pinkas (2007) (and improved in shelat and Shen (2013)) uses what are called *k-probe-resistant* matrices. The idea behind this technique is to agree on a public matrix M and for P_2 to randomly encode its true input y into $\tilde{y}$ so that $y = M\tilde{y}$. Then the garbled circuit will compute $(x, \tilde{y}) \mapsto \mathcal{F}(x, M\tilde{y})$ and P_2 will use the bits of $\tilde{y}$ (rather than y) as its inputs to OT. The k-probe-resistant property of M is that for any nonempty subset of rows of M, their XOR has Hamming weight at least k. Lindell and Pinkas (2007) show that if M is k-probe-resistant, then the joint distribution of any k bits of $\tilde{y}$ is uniform—in particular, it is independent of P_2's true input y. Furthermore, when M is public, the computation of $\tilde{y} \mapsto M\tilde{y}$ consists of only XOR operations and therefore adds no cost to the garbled circuit using FreeXOR (though the increased size of $\tilde{y}$ contributes more oblivious transfers).

The k-probe-resistant encoding technique thwarts selective abort attacks in the following way. If P_1 provides selectively incorrect garbled inputs in at most k OTs, then these will be selectively picked up by P_2 according to at most k specific bits of $\tilde{y}$, which are completely uniform in this case. Hence, P_2's abort condition is input-independent. If on the other hand P_1 provides incorrect garbled inputs in more than k OTs, then P_2 will almost surely abort—at least with probability $1 - 1/2^k$. If k is chosen so that $1/2^k$ is negligible (e.g., if

$k = \sigma$, the statistical security parameter), then any input-dependent differences in abort probability are negligible.

Concrete parameters. Cut-and-choose mechanisms involve two main parameters: the *replication factor* is the number of garbled circuits that P_1 must generate, and the *checking probability* refers to the probability with which a garbled circuit is chosen to be checked in the cut-and-choose phase. An obvious goal for any cut-and-choose protocol is to minimize the replication factor needed to provide adequate security.

In the cut-and-choose protocol described above, the only way an adversary can break security of the mechanism is to cause a majority of evaluation circuits to be incorrect, while still making all checked circuits correct. The adversary's task can be captured in the following abstract game:

- The player (arbitrarily) prepares ρ balls, each one is either red or green. A red ball represents an incorrectly-garbled circuit while a green ball represents a correct one.
- A random subset of exactly c balls is designated to be *checked.* If any checked ball is red, the player loses the game.
- The player wins the game if the majority of the unchecked balls are red.

We want to find the smallest ρ and best c so that no player can win the game with probability better than $2^{-\lambda}$. The analysis of shelat and Shen (2011) found that the minimal replication factor is $\rho \approx 3.12\lambda$, and the best number of items to check is $c = 0.6\rho$ (surprisingly, not 0.5ρ).

Cost-aware cut-and-choose. The results from shelat and Shen (2011) provide an optimal number of check and evaluation circuits, assuming the cost of each circuit is the same. However, some circuits are evaluated and others are checked, and these operations do not have equal cost. In particular, the computational cost of evaluating a garbled circuit is about 25–50% the cost of checking a garbled circuit for correctness since evaluating just involves executing one path through the circuit, while checking must verify all entries in the garble tables. Also, some variants of cut-and-choose (e.g., Goyal *et al.* (2008)) allow P_1 to send only a hash of a garbled circuit up-front, before P_2

chooses which circuits to open. To open a circuit, P_1 can simply send a short seed that was used to derive all the randomness for the circuit. P_2 can then recompute the circuit and compare its hash to the one originally sent by P_1. In protocols like this, the communication cost of a checked circuit is almost nothing—only evaluation circuits require significant communication.

Zhu *et al.* (2016) study various cut-and-choose games and derive parameters with optimal cost, accounting for the different costs of checking and evaluating a garbled circuit.

6.2 Input Recovery Technique

In the traditional cut-and-choose mechanisms we have described so far, the evaluator (P_2) evaluates many garbled circuits and reports the majority output. As previously mentioned, the overhead of this method is well understood—the replication factor for security $2^{-\lambda}$ is roughly 3.12λ. Reducing this replication factor requires a different approach to the entire cut-and-choose mechanism.

Lindell (2013) and Brandão (2013) independently proposed cut-and-choose protocols breaking this replication factor barrier. These protocols give P_2 a way of identifying the correct output in the event that some of the evaluated circuits disagree. Hence, the only way for a malicious P_1 to break security is to force *all* of the evaluated circuits to be incorrect in the same way (rather than simply forcing a *majority* of them to be incorrect as in the previous protocols). Suppose there are ρ circuits, and each one is checked with independent probability $\frac{1}{2}$. The only way to cheat is for all of the correct circuits to be opened and all of the incorrect circuits to be evaluated, which happens with probability $2^{-\rho}$. In short, one can achieve $2^{-\lambda}$ security with replication factor only $\rho = \lambda$ rather than $\rho \approx 3.12\lambda$. The protocol of Lindell (2013) proceeds in two phases:

1. The parties do a fairly typical cut-and-choose with P_1 generating many garbled circuits, and P_2 choosing some to check and evaluating the rest. Suppose P_2 observes different outputs from different garbled circuits. Then, P_2 obtains output wire labels for the same wire corresponding to opposite values (e.g., a wire label encoding 0 on the first output wire of one circuit, a label encoding 1 on the first output wire of another circuit). When P_1 is honest, it is infeasible to obtain such contradictory wire labels. Hence, the wire labels serve as "proof of cheating". But,

for the same reasons as mentioned above, P_2 must not reveal whether it obtained such a proof since that event may be input-dependent and leak information about P_2's private input.

2. In the second phase, the parties do a malicious-secure computation of an *input recovery* function: if P_2 can provide proof of cheating in phase 1, then the function "punishes" P_1 by revealing its input to P_2. In that case, P_2 has both parties' inputs and can simply compute the correct output locally. Otherwise, when P_2 does not provide proof of cheating, P_2 learns nothing from this second phase. Either way, P_1 learns nothing from this second phase.

There are many subtle details that enable this protocol to work. Some of the most notable are:

- The secure computation in the second phase is done by using a traditional (majority-output) cut-and-choose protocol. However, the size of this computation can be made to depend only on the size of P_1's input. In particular, it does not depend on the size of the circuit the parties are evaluating in phase 1.

- In order to make the circuits for the second phase small, it is helpful if all garbled circuits share the same output wire labels. When this is the case, opening any circuit would reveal all output wire labels for all evaluation circuits and allows P_2 to "forge" a proof of cheating. Hence the check circuits of phase 1 cannot be opened until the parties' inputs to phase 2 have been fixed.

- The overall protocol must enforce that P_1 uses the same input to all circuits *in both phases*. It is important that if P_1 uses input x in phase 1 and cheats, it cannot prevent P_2 from learning that same x in phase 2. Typical mechanisms for input consistency (such as the 2-universal hash technique described above) can easily be adapted to ensure consistency across both phases in this protocol.

- For the reasons described previously, P_2 cannot reveal whether it observed P_1 cheating in the first phase. Analogously, the protocol gives P_2 two avenues to obtain the final output (either when all phase-one

circuits agree, or by recovering P_2's input in phase two and computing the output directly), but P_1 cannot learn which one was actually used. It follows that the parties must *always* perform the second phase, even when P_1 is honest.

6.3 Batched Cut-and-Choose

As a motivating scenario, consider the case where two parties know in advance that they would like to perform N secure evaluations of the same function f (on unrelated inputs). In each secure computation, P_1 would be required to generate many garbled circuits of f for each cut-and-choose. The amortized costs for each evaluation can be reduced by performing a single cut-and-choose for all N evaluation instances.

Consider the following variant of the cut-and-choose abstract game:

1. The player (arbitrarily) prepares $N\rho + c$ balls, each one is either red or green.

2. A random subset of exactly c balls is designated to be *checked*. If any checked ball is red, the player loses the game.

3. [new step] The unchecked balls are randomly assigned into N buckets, with each bucket containing exactly ρ balls.

4. [modified step] The player wins if any bucket contains *only* red balls (in a different variant, one might specify that the player wins if any bucket contains a majority of red balls).

This game naturally captures the following high-level idea for a cut-and-choose protocol suitable for a batch of N evaluations of the same function. First, P_1 generates $N\rho + c$ garbled circuits. P_2 randomly chooses c of them to be checked and randomly assigns the rest into N buckets. Each bucket contains the circuits to be evaluated in a particular instance. Here we are assuming that each instance will be secure as long as it includes at least one correct circuit (for example, using the mechanisms from Section 6.2).

Intuitively, it is now harder for the player (adversary) to beat the cut-and-choose game, since the evaluation circuits are further randomly assigned to

buckets. The player must get lucky not only in avoiding detection during checking, but also in having many incorrect circuits placed in the same bucket.

Zhu and Huang (2017) give an asymptotic analysis showing that replication $\rho = 2 + \Theta(\lambda/\log N)$ suffices to limit the adversary to success probability $2^{-\lambda}$. Compare this to single-instance cut-and-choose which requires replication factor $O(\lambda)$.[1] The improvement over single-instance cut-and-choose is not just asymptotic, but is significant for reasonable values of N. For instance, for $N = 1024$ executions, one achieves a security level of 2^{-40} if P_1 generates 5593 circuits, of which only 473 are checked. Then only $\rho = 5$ circuits are evaluated in each execution.

Lindell and Riva (2014) and concurrently Huang *et al.* (2014) described batch cut-and-choose protocols following the high-level approach described above. The former protocol was later optimized and implemented in Lindell and Riva (2015). The protocols use the input-recovery technique so that each instance is secure as long as at least one correct circuit is evaluated.

6.4 Gate-level Cut-and-Choose: LEGO

In batch cut-and-choose, the amortized cost per bucket/instance decreases as the number of instances increases. This observation was the foundation of the *LEGO paradigm* for malicious-secure two-party computation, introduced by Nielsen and Orlandi (2009). The main idea is to do a batch cut-and-choose on *individual garbled gates* rather than on entire garbled circuits:

1. P_1 generates a large number of independently garbled NAND gates, and the parties perform a batch cut-and-choose on them. P_2 chooses some gates to check and randomly assigns the remaining gates into buckets.

2. The buckets of gates are assembled into a garbled circuit in a process called *soldering* (described in more detail below).

    - The gates within a single bucket are connected so that they collectively act like a fault-tolerant garbled NAND gate, which correctly

[1] The replication factor in this modified game measures only the number of evaluation circuits, whereas for a single instance we considered the total number (check and evaluation) of circuits. In practice, the number of checked circuits in batch cut-and-choose is quite small, and there is little difference between amortized number of *total* circuits vs. amortized number of *evaluation* circuits.

computes the NAND function as long as a majority of gates in the bucket are correct.

- The fault-tolerant garbled gates are connected to form the desired circuit. The connections between garbled gates transfer the garbled value on the output wire of one gate to the input wire of another.

3. P_2 evaluates the single conceptual garbled circuit, which is guaranteed to behave like a correct garbled circuit with overwhelming probability.

We now describe the soldering process in more detail, using the terminology of Frederiksen *et al.* (2013). The paradigm requires a *homomorphic commitment*, meaning that if P_1 commits to values A and B independently, it can later either decommit as usual, or can generate a decommitment that reveals *only* $A \oplus B$ to P_2.

P_1 prepares many individual garbled gates, using the FreeXOR technique. For each wire i, P_1 chooses a random "zero-label" k_i^0; the other label for that wire is $k_i^1 = k_i^0 \oplus \Delta$, where Δ is the FreeXOR offset value common to all gates. P_1 sends each garbled gate, and commits to the zero-label of each wire, as well as to Δ (once and for all for all gates). In this way, P_1 can decommit to k_i^0 or to $k_i^1 = k_i^0 \oplus \Delta$ using the homomorphic properties of the commitment scheme.

If a gate is chosen to be checked, then P_1 cannot open all wire labels corresponding to the gate. This would reveal the global Δ value and break the security of all gates. Instead, P_2 chooses a one of the four possible input combinations for the gate at random, and P_1 opens the corresponding input and output labels (one label per wire). Then, P_2 can check that the gate evaluates correctly on this combination. An incorrectly-garbled gate can be therefore caught only with probability $\frac{1}{4}$ (Zhu and Huang (2017) provides a way to increase this probability to $\frac{1}{2}$). This difference affects the cut-and-choose parameters (e.g., bucket size) by a constant factor.

Soldering corresponds to connecting various wires (attached to individual gates) together, so that the logical value on a wire can be moved to another wire. Say that wire u (with zero-label k_u^0) and wire v (with zero-label k_v^0) are to be connected. Then P_1 can decommit to the solder value $\sigma_{u,v} = k_u^0 \oplus k_v^0$. This value allows P_2 to transfer a garbled value from wire u to wire v during circuit evaluation. For example, if P_2 holds wire label $k_u^b = k_u^0 \oplus b \cdot \Delta$, representing unknown value b, then xor-ing this wire label with the solder value $\sigma_{u,v}$ results

in the appropriate wire label on wire v:

$$k_u^b \oplus \sigma_{u,v} = (k_u^0 \oplus b \cdot \Delta) \oplus (k_u^0 \oplus k_v^0) = k_v^0 \oplus b \cdot \Delta = k_v^b$$

Gates within a bucket are assembled into a fault-tolerant NAND gate by choosing the first gate as an "anchor" and soldering wires of other gates to the matching wire of the anchor (i.e., solder the left input of each gate to the left input of the anchor). With ρ gates in a bucket, this gives ρ ways to evaluate the bucket starting with the garbled inputs of the anchor gate—transfer the garbled values to another gate, evaluate the gate, and transfer the garbled value back to the anchor. If all gates are correct, all ρ of the evaluation paths will result in an identical output wire label. If some gates are incorrect, the evaluator takes the majority output wire label.

The LEGO paradigm can take advantage of the better parameters for batch cut-and-choose, even in the single-execution setting. If the parties wish to securely evaluate a circuit of N gates, the LEGO approach involves a replication factor of $O(1) + O(\lambda/\log N)$, where λ is the security parameter. Of course, the soldering adds many extra costs that are not present in circuit-level cut-and-choose. However, for large circuits the LEGO approach gives a significant improvement over circuit-level cut-and-choose that has replication factor λ.

Variations on LEGO. The LEGO protocol paradigm has been improved in a sequence of works (Frederiksen *et al.*, 2013; Frederiksen *et al.*, 2015; Zhu and Huang, 2017; Kolesnikov *et al.*, 2017b; Zhu *et al.*, 2017). Some notable variations include:

- Avoiding majority-buckets in favor of buckets that are secure if even one garbled gate is correct (Frederiksen *et al.*, 2015).
- Performing cut-and-choose at the level of component subcircuits consisting of multiple gates (Kolesnikov *et al.*, 2017b).
- Performing cut-and-choose with a fixed-size pool of gates that is constantly replenished, rather than generating all of the necessary garbled gates up-front (Zhu *et al.*, 2017).

6.5 Zero-Knowledge Proofs

An alternative to the cut-and-choose approach is to convert a semi-honest protocol into a malicious-secure protocol by incorporating a proof that the protocol was executed correctly. Of course, the proof cannot reveal the secrets used in the protocol. Goldreich *et al.* (1987) shows how to use zero-knowledge (ZK) proofs to turn any semi-honest MPC protocol into one that is secure against malicious adversaries (Section 6.5.1).

Zero-knowledge proofs are a special case of malicious secure computation, and were introduced in Section 2.4. ZK proofs allow a prover to convince a verifier that it knows x such that $C(x) = 1$, without revealing any additional information about x, where C is a public circuit.

6.5.1 GMW Compiler

Goldreich, Micali, and Wigderson (GMW) showed a compiler for secure multi-party computation protocols that uses ZK proofs (Goldreich *et al.*, 1987). The compiler takes as input any protocol secure against semi-honest adversaries, and generates a new protocol for the same functionality that is secure against malicious adversaries.

Let π denote the semi-honest-secure protocol. The main idea of the GMW compiler is to run π and prove in zero-knowledge that every message is the result of running π *honestly*. The honest parties abort if any party fails to provide a valid ZK proof. Intuitively, the ZK proof ensures that a malicious party can either run π honestly, or cheat in π but cause the ZK proof to fail. If π is indeed executed honestly, then the semi-honest security of π ensures security. Whether or not a particular message is consistent with honest execution of π depends on the parties' private inputs. Hence, the ZK property of the proofs ensures that this property can be checked without leaking any information about these private inputs.

Construction. The main challenge in transforming a semi-honest protocol into an analogous malicious-secure protocol is to precisely define the circuit that characterizes the ZK proofs. Two important considerations are:

1. Each party must prove that each message of π is consistent with honest execution of π, *on a consistent input.* In other words, the ZK proof

should prevent parties from running π honestly, but with different inputs in different rounds.

2. The "correct" next message of π is a function of not only the party's private input, but also their private random tape. π guarantees security only when each party's random tape is chosen uniformly. In particular, the protocol may be insecure if the party runs honestly but on some adversarially-chosen random tape.

The first consideration is addressed by having each party commit to its input upfront. Then all ZK proofs refer to this commitment: e.g., the following message is consistent with an honest execution of π, on the input that is contained inside the public commitment.

The second consideration is addressed by a technique called *coin-tossing into the well*. For concreteness, we focus on the ZK proofs generated by P_1. Initially P_1 produces a commitment to a random string r. Then P_2 sends a value r' in the clear. Now P_1 must run π with $r \oplus r'$ as the random tape. In this way, P_1 does not have unilateral control over its effective random tape $r \oplus r'$ — it is distributed uniformly even if P_1 is corrupt. P_1's ZK proofs can refer to the commitment to r (and the public value r') to guarantee that π is executed with $r \oplus r'$ as its random tape.

The full protocol description is given in Figure 6.2.

6.5.2 ZK from Garbled Circuits

Jawurek, Kerschbaum, and Orlandi (JKO) presented an elegant zero-knowledge protocol based on garbled circuits (Jawurek *et al.*, 2013). Since zero-knowledge is a special case of malicious secure computation, one can obviously base zero-knowledge on any cut-and-choose-based 2PC protocol. However, these protocols require many garbled circuits. The JKO protocol on the other hand achieves zero-knowledge using only one garbled circuit.

The main idea is to use a single garbled circuit for both evaluation and checking. In standard cut-and-choose, opening a circuit that is used for evaluation would reveal the private input of the garbled circuit generator. However, the verifier in a zero-knowledge protocol has no private input. Thus, the verifier can play the role of circuit garbler.

PARAMETERS: Semi-honest-secure two-party protocol $\pi = (\pi_1, \pi_2)$, where $\pi_b(x, r, T)$ denotes the next message for party P_b on input x, random tape r, and transcript so far T. A commitment scheme Com.

PROTOCOL π^*: (P_1 has input x_1 and P_2 has input x_2)

1. For $b \in \{1, 2\}$, P_b chooses random r_b and generates a commitment c_b to (x_b, r_b) with decommitment δ_b.
2. For $b \in \{1, 2\}$, P_b chooses and sends random r'_{3-b} (a share of the counterpart's random tape).
3. The parties alternate between $b = 1$ and $b = 2$ as follows, until the protocol terminates:
   (a) Let T be the transcript of π-messages so far (initially empty). P_b computes and sends the next π-message, $t = \pi_b(x_b, r_b \oplus r'_b, T)$. If instead π_b terminates, P_b terminates as well (with whatever output π_b specifies).
   (b) P_b acts as prover in a ZK proof with private inputs x_b, r_b, δ_b, and public circuit $C[\pi_b, c_b, r'_b, T, t]$ that is defined as:

   $C[\pi, c, r', T, t](x, r, \delta)$:
   return 1 iff δ is a valid opening of commitment c to (x, r) and $t = \pi(x, r \oplus r', T)$.

   The other party P_{3-b} aborts if verification of the ZK proof fails.

Figure 6.2: GMW compiler applied to a semi-honest-secure protocol π.

Suppose the prover P_1 wishes to prove $\exists w : \mathcal{F}(w) = 1$ where $\mathcal{F}$ is a public function. The JKO protocol proceeds in the following steps:

1. The verifier P_2 generates and sends a garbled circuit computing $\mathcal{F}$.
2. The prover picks up garbled inputs for w, using oblivious transfer.
3. The prover evaluates the circuit and obtains the output wire label (corresponding to output 1) and generates a commitment to this wire label.

4. The verifier opens the garbled circuit and the prover checks that it was generated correctly. If so, then the prover opens the commitment to the output wire label.

5. The verifier accepts if the prover successfully decommits to the 1 output wire label of the garbled circuit.

The protocol is secure against a cheating prover because at the time P_1 generates a commitment in step 3, the garbled circuit has not yet been opened. Hence, if P_1 does not know an input that makes $\mathcal{F}$ output 1, it is hard to predict the 1 output wire label at this step of the protocol. The protocol is secure against a cheating verifier because the prover only reveals the result of the garbled circuit after the circuit has been confirmed to be generated correctly.

Because the garbled circuits used for this protocol only need to provide authenticity and not privacy, their garbled tables can be implemented less expensively than for standard Yao's. Zahur *et al.* (2015) show that the half-gates method can be used to reduce the number of ciphertexts needed for a privacy-free garbled circuit to a single ciphertext for each AND gate, and no ciphertexts for XOR gates.

6.6 Authenticated Secret Sharing: BDOZ and SPDZ

Recall the approach for secret-sharing based MPC using Beaver triples (Section 3.4). This protocol paradigm is malicious-secure given suitable Beaver triples and any sharing mechanism such that:

1. Sharings are additively homomorphic,

2. Sharings hide the underlying value against a (malicious) adversary, and

3. Sharings can be opened reliably, even in the presence of a malicious adversary.

In this section we describe two sharing mechanisms with these properties: BDOZ (Section 6.6.1) and SPDZ (Section 6.6.2).

6.6.1 BDOZ Authentication

The Bendlin-Damgård-Orlandi-Zakarias (BDOZ or BeDOZa) technique (Bendlin *et al.*, 2011) incorporates information-theoretic MACs into the secret shares. Let $\mathbb{F}$ be the underlying field, with $|\mathbb{F}| \geq 2^\kappa$ where κ is the security parameter. Interpreting $K, \Delta \in \mathbb{F}$ as a key, define $\mathsf{MAC}_{K,\Delta}(x) = K + \Delta \cdot x$.

This construction is an information-theoretic one-time MAC. An adversary who sees $\mathsf{MAC}_{K,\Delta}(x)$ for a chosen x cannot produce another valid MAC, $\mathsf{MAC}_{K,\Delta}(x')$, for $x \neq x'$. Indeed, if an adversary could compute such a MAC, then it could compute Δ:

$$\begin{aligned}
&(x - x')^{-1}\Big(\mathsf{MAC}_{K,\Delta}(x) - \mathsf{MAC}_{K,\Delta}(x')\Big) \\
&= (x - x')^{-1}\Big(K + \Delta x - K - \Delta x'\Big) \\
&= (x - x')^{-1}(\Delta(x - x')) = \Delta
\end{aligned}$$

But seeing only $\mathsf{MAC}_{K,\Delta}(x)$ perfectly hides Δ from the adversary. Hence, the probability of computing a MAC forgery is bounded by $1/|\mathbb{F}| \leq 1/2^\kappa$, the probability of guessing a randomly chosen field element Δ.

In fact, the security of this MAC holds even when an honest party has many MAC keys that all share the same Δ value (but with independently random K values). We refer to Δ as the global MAC key and K as the local MAC key.

The idea of BDOZ is to authenticate each party's shares with these information-theoretic MACs. We start with the two-party case. Each party P_i generates a global MAC key Δ_i. Then $[x]$ denotes the secret-sharing mechanism where P_1 holds x_1, m_1 and K_1 and P_2 holds x_2, m_2 and K_2 such that:

1. $x_1 + x_2 = x$ (additive sharing of x),
2. $m_1 = K_2 + \Delta_2 x_1 = \mathsf{MAC}_{K_2,\Delta_2}(x_1)$ (P_1 holds a MAC of its *share* x_1 under P_2's MAC key), and
3. $m_2 = K_1 + \Delta_1 x_2 = \mathsf{MAC}_{K_1,\Delta_1}(x_2)$ (P_2 holds a MAC of its *share* x_2 under P_1's MAC key).

Next, we argue that this sharing mechanism satisfies the properties required by the Beaver-triple paradigm (Section 3.4):

sharing	P_1 has	P_2 has
$[x]$	x_1 K_1 $\mathsf{MAC}_{K_2,\Delta_2}(x_1)$	x_2 K_2 $\mathsf{MAC}_{K_1,\Delta_1}(x_2)$
$[x']$	x'_1 K'_1 $\mathsf{MAC}_{K'_2,\Delta_2}(x'_1)$	x'_2 K'_2 $\mathsf{MAC}_{K'_1,\Delta_1}(x'_2)$
$[x + x']$	$x_1 + x'_1$ $K_1 + K'_1$ $\mathsf{MAC}_{K_2+K'_2,\Delta_2}(x_1 + x'_1)$	$x_2 + x'_2$ $K_2 + K'_2$ $\mathsf{MAC}_{K_1+K'_1,\Delta_1}(x_2 + x'_2)$

Figure 6.3: BDOZ authenticated sharing

- Privacy: the individual parties learn nothing about x since they only hold one additive share, x_p, and m_p reveals nothing about x without knowing the other party's keys (which are never revealed).

- Secure opening: To open, each party announces its (x_p, m_p), allowing both parties to learn $x = x_1 + x_2$. Then, P_1 can use its MAC key to check whether $m_2 = \mathsf{MAC}_{K_1,\Delta_1}(x_2)$ and abort if this is not the case. P_2 performs an analogous check on m_1. Note that opening this sharing to any different value corresponds exactly to the problem of breaking the underlying one-time MAC.

- Homomorphism: The main idea is that when all MACs in the system use the same Δ value, the MACs become homomorphic in the necessary way. That is,

$$\mathsf{MAC}_{K,\Delta}(x) + \mathsf{MAC}_{K',\Delta}(x') = \mathsf{MAC}_{K+K',\Delta}(x + x')$$

  Here we focus on adding shared values $[x] + [x']$; the other required forms of homomorphism work in a similar way. The sharings of $[x]$ and $[x']$ and the resulting BDOZ sharing of $[x + x']$ is shown in Figure 6.3.

The BDOZ approach generalizes to n parties in a straightforward (but expensive) way. All parties have global MAC keys. In a single sharing $[x]$, the

parties have additive shares of x and each party's share is authenticated under *every other party's* MAC key.

Generating triples. The BDOZ sharing method satisfies the security and homomorphism properties required for use in the abstract Beaver-triples approach. It remains to be seen how to generate Beaver triples in this format.

Note that BDOZ shares work naturally even when the payloads (i.e., x in $[x]$) are restricted to a subfield of $\mathbb{F}$. The sharings $[x]$ are then homomorphic with respect to that subfield. A particularly useful case is to use BDOZ for sharings of single bits, interpreting $\{0, 1\}$ as a subfield of $\mathbb{F} = \mathrm{GF}(2^\kappa)$. Note that $\mathbb{F}$ must be exponentially large for security (authenticity) to hold.

The state of the art method for generating BDOZ shares of bits is the scheme used by Tiny-OT (Nielsen *et al.*, 2012). It uses a variant of traditional OT extension (Section 3.7.2) to generate BDOZ-authenticated bits $[x]$. It then uses a sequence of protocols to securely multiply these authenticated bits needed to generate the required sharings for Beaver triples.

6.6.2 SPDZ Authentication

In BDOZ sharing, each party's local part of $[x]$ contains a MAC for every other party. In other words, the storage requirement of the protocol scales linearly with the number of parties. A different approach introduced by Damgård, Pastro, Smart, and Zakarias (SPDZ, often pronounced "speeds") (Damgård *et al.*, 2012b) results in constant-sized shares for each party.

As before, we start with the two-party setting. The main idea is to have a global MAC key Δ that is not known to either party. Instead, the parties hold Δ_1 and Δ_2 which can be thought of as shares of a global $\Delta = \Delta_1 + \Delta_2$. In a SPDZ sharing $[x]$, P_1 holds (x_1, t_1) and P_2 holds (x_2, t_2), where $x_1 + x_2 = x$ and $t_1 + t_2 = \Delta \cdot x$. Thus, the parties hold additive shares of x and of $\Delta \cdot x$. One can think of $\Delta \cdot x$ as a kind of "0-time information-theoretic MAC" of x.

This scheme clearly provides privacy for x. Next, we show that it also provides the other two properties required for Beaver triples:

- Secure opening: We cannot have the parties simply announce their shares, since that would reveal Δ. It is important that Δ remain secret throughout the entire protocol. To open $[x]$ without revealing Δ, the protocol proceeds in 3 phases:

1. The players announce only x_1 and x_2. This determines the (unauthenticated) candidate value for x.
2. Note that if this candidate value for x is indeed correct, then

$$\begin{aligned}(\Delta_1 x - t_1) + (\Delta_2 x - t_2) &= (\Delta_1 + \Delta_2)x - (t_1 + t_2) \\ &= \Delta x - (\Delta x) \\ &= 0\end{aligned}$$

   Furthermore, P_1 can locally compute the first term $(\Delta_1 x - t_1)$ and P_2 can compute the other term. In this step of the opening, P_1 *commits* to the value $\Delta_1 x - t_1$ and P_2 commits to $\Delta_2 x - t_2$.
3. The parties open these commitments and abort if their sum is not 0. Note that if the parties had simply announced these values one at a time, then the last party could cheat by choosing the value that causes the sum to be zero. By using a commitment, the protocol forces the parties to know these values in advance.

To understand the security of this opening procedure, note that when P_1 commits to some value c, it expects P_2 to also commit to $-c$. In other words, the openings of these commitments are easily simulated, which implies that they leak nothing about Δ.

It is possible to show that if a malicious party is able to successfully open $[x]$ to a different x', then that party is able to guess Δ. Since the adversary has no information about Δ, this event is negligibly likely.

- Homomorphism: In a SPDZ sharing $[x]$, the parties' shares consist of additive shares of x and additive shares of $\Delta \cdot x$. Since each of these are individually homomorphic, the SPDZ sharing naturally supports addition and multiplication-by-constant.

  To support addition-by-constant, the parties must use their additive shares of Δ as well. Conceptually, they locally update $x \mapsto x + c$ and locally update $\Delta x \mapsto \Delta x + \Delta c$. This is illustrated in Figure 6.4.

Generating SPDZ shares. Since SPDZ shares satisfy the properties needed for abstract Beaver-triple-based secure computation, the only question remains

sharing	P_1 has	P_2 has	sum of P_1 and P_2 shares
$[x]$	x_1	x_2	x
	t_1	t_2	Δx
$[x+c]$	$x_1 + c$	x_2	$x + c$
	$t_1 + \Delta_1 c$	$t_2 + \Delta_2 c$	$\Delta(x + c)$

Figure 6.4: SPDZ authenticated secret sharing.

how to generate Beaver triples in the SPDZ format. The paper that initially introduced SPDZ (Damgård *et al.*, 2012b) proposed a method involving somewhat homomorphic encryption. Followup work suggests alternative techniques based on efficient OT extension (Keller *et al.*, 2016).

6.7 Authenticated Garbling

Wang *et al.* (2017b) introduced an *authenticated garbling* technique for multiparty secure computation that combines aspects of information-theoretic protocols (e.g., authenticated shares and Beaver triples) and computational protocols (e.g., garbled circuits and BMR circuit generation). For simplicity, we describe their protocol in the two-party setting but many (not all) of the techniques generalize readily to the multi-party setting (Wang *et al.*, 2017c).

A different perspective on authenticated shares of bits. One starting point is the BDOZ method for authenticated secret-sharing of *bits*. Recall that a 2-party BDOZ sharing $[x]$ corresponds to the following information:

sharing	P_1 has	P_2 has
$[x]$	x_1	x_2
	K_1	K_2
	$T_1 = K_2 \oplus x_1\Delta_2$	$T_2 = K_1 \oplus x_2\Delta_1$

Since we consider x, x_1, x_2 to be bits, the underlying field is $\mathbb{F} = \mathrm{GF}(2^\kappa)$ and we write the field addition operation as $\oplus$. An interesting observation is that:

$$\underbrace{(K_1 \oplus x_1\Delta_1)}_{\text{known to } \mathsf{P}_1} \oplus \underbrace{(K_1 \oplus x_2\Delta_1)}_{\text{known to } \mathsf{P}_2} = (x_1 \oplus x_2)\Delta_1 = x\Delta_1$$

Hence, a side-effect of a BDOZ sharing $[x]$ is that parties hold additive shares of $x\Delta_1$, where Δ_1 is P_1's global MAC key.

Distributed garbling. Consider a garbled circuit in which the garbler P_1 chooses wire labels k_i^0, k_i^1 for each wire i. Departing from the notation from Section 3.1.2, we will let the superscript b in k_i^b denote the public "point-and-permute" pointer bit of a wire label (that the evaluator learns), rather than its semantic value true/false. We let p_i denote the corresponding pointer bit, so that $k_i^{p_i}$ is the label representing false.

We focus on a single AND gate with input wires a, b and output wire c. Translating the standard garbled circuit construction into this perspective (i.e., organized according to the pointer bits), we obtain the following garbled table:

$$\begin{aligned}
e_{0,0} &= H(k_a^0 \| k_b^0) \oplus k_c^{p_c \oplus p_a \cdot p_b} \\
e_{0,1} &= H(k_a^0 \| k_b^1) \oplus k_c^{p_c \oplus p_a \cdot \overline{p_b}} \\
e_{1,0} &= H(k_a^1 \| k_b^0) \oplus k_c^{p_c \oplus \overline{p_a} \cdot p_b} \\
e_{1,1} &= H(k_a^1 \| k_b^1) \oplus k_c^{p_c \oplus \overline{p_a} \cdot \overline{p_b}}
\end{aligned}$$

Using FreeXOR, $k_i^1 = k_i^0 \oplus \Delta$ for some global value Δ. In that case, we can rewrite the garbled table as:

$$\begin{aligned}
e_{0,0} &= H(k_a^0 \| k_b^0) \oplus k_c^0 \oplus (p_c \oplus p_a \cdot p_b)\Delta \\
e_{0,1} &= H(k_a^0 \| k_b^1) \oplus k_c^0 \oplus (p_c \oplus p_a \cdot \overline{p_b})\Delta \\
e_{1,0} &= H(k_a^1 \| k_b^0) \oplus k_c^0 \oplus (p_c \oplus \overline{p_a} \cdot p_b)\Delta \\
e_{1,1} &= H(k_a^1 \| k_b^1) \oplus k_c^0 \oplus (p_c \oplus \overline{p_a} \cdot \overline{p_b})\Delta
\end{aligned}$$

One of the main ideas in authenticated garbling is for the parties to construct such garbled gates in a somewhat distributed fashion, in such a way that neither party knows the p_i permute bits.

Instead, suppose the parties only have *BDOZ sharings* of the form $[p_a], [p_b], [p_a \cdot p_b], [p_c]$, where neither party knows these p_i values in the clear. Suppose further that P_1 chooses the garbled circuit's wire labels so that $\Delta = \Delta_1$ (i.e., its global MAC key from BDOZ). The parties therefore have additive shares of $p_a\Delta$, $p_b\Delta$, and so on. They can use the homomorphic properties of additive secret sharing to locally obtain shares of $(p_c \oplus p_a \cdot p_b)\Delta$, $(p_c \oplus p_a \cdot \overline{p_b})\Delta$, and so on.

Focusing on the first ciphertext in the garbled table, we can see:

$$e_{0,0} = \underbrace{H(k_a^0 \| k_b^0) \oplus k_c^0}_{\text{known to } \mathsf{P}_1} \oplus \underbrace{(p_c \oplus p_a \cdot p_b)\Delta}_{\text{parties have additive shares}}$$

Hence, using only local computation (P_1 simply adds the appropriate value to its share), parties can obtain additive shares of $e_{0,0}$ and all other rows in the garbled table.

In summary, the distributed garbling procedure works by generating BDOZ-authenticated shares of random permute bits $[p_i]$ for every wire in the circuit, along with Beaver triples $[p_a], [p_b], [p_a \cdot p_b]$ for every AND gate in the circuit. Then, using only local computation, the parties can obtain additive shares of a garbled circuit that uses the p_i values as its permute bits. P_1 sends its shares of the garbled circuit to P_2, who can open it and evaluate.

Authenticating the garbling. As in Yao's protocol, the garbler P_1 can cheat and generate an incorrect garbled circuit — in this case, by sending incorrect additive shares. For example, P_1 can replace the "correct" $e_{0,0}$ value in some gate by an incorrect value, while leaving the other three values intact. In this situation, P_2 obtains an incorrect wire label whenever the logical input to this gate is (p_a, p_b). Even assuming P_2 can detect this condition and abort, this leads to a *selective abort* attack for a standard garbled circuit. By observing whether P_2 aborts, P_1 learns whether the input to this gate was (p_a, p_b).

However, this is not actually a problem for distributed garbling. While it is still true that P_2 aborts if and only if the input to this gate was (p_a, p_b), P_1 has no information about p_a, p_b — hiding these permute bits from P_1 causes P_2's abort probability to be input-independent!

Constructing a garbled circuit with secret permute bits addresses the problem of privacy against a corrupt P_1. However, P_1 may still break the correctness of the computation. For instance, P_1 may act in a way that flips one of the $p_c \oplus p_a \cdot p_b$ bits. To address this, Wang *et al.* (2017b) relies on the fact that the parties have BDOZ sharings of the p_i indicator bits. These sharings determine the "correct" pointer bits that P_2 should see. For example, if the input wires to an AND gate have pointer bits $(0, 0)$ then the correct pointer bit for the output wire is $p_c \oplus p_a \cdot p_b$. To ensure correctness of the computation, it suffices to ensure that P_2 always receives the correct pointer bits. As discussed

previously, the parties can obtain authenticated BDOZ sharings of $p_c \oplus p_a \cdot p_b$. We therefore augment the garbled circuit so that each ciphertext contains not only the output wire label, but also P_1's authenticated BDOZ share of the "correct" pointer bit. The BDOZ authentication ensures that P_1 cannot cause P_2 to view pointer bits that are inconsistent with the secret p_i values, without aborting the protocol. Now as P_2 evaluates the garbled circuit, it checks for each gate that the visible pointer bit is authenticated by the associated MAC.

This protocol provided dramatic cost improvements. Wang *et al.* (2017b) reports on experiments using authenticated garbling to execute malicious secure protocols over both local and wide area networks. In a LAN setting, it can execute over 800,000 AND gates per second and perform a single private AES encryption in 16.6 ms (of which 0.93 ms is online cost) on a 10Gbps LAN and 1.4 s on a WAN (77 ms is online cost). In a batched setting where 1024 AES encryptions are done, the amortized total cost per private encryption drops to 6.66 ms (113 ms in a WAN). As a measure of the remarkable improvement in MPC execution, the fastest AES execution as a semi-honest LAN protocol in 2010 was 3300 ms total time (Henecka *et al.*, 2010), so in a matter of eight years the time required to execute a malicious secure protocol dropped to approximately $\frac{1}{200}$ that of the best semi-honest protocol!

6.8 Further Reading

Cut-and-choose existed as a folklore technique in the cryptographic literature. Different cut-and-choose mechanisms were proposed by Mohassel and Franklin (2006) and Kiraz and Schoenmakers (2006), but without security proofs. The first cut-and-choose protocol for 2PC with a complete security proof was due to Lindell and Pinkas (2007).

We presented one technique from Lindell and Pinkas (2007) for dealing with the selective abort attacks. The subtle nature of selective abort attacks was first observed by Kiraz and Schoenmakers (2006), and fixed using a different technique — in that work, by modifying the oblivious transfers into a variant called committed OT.

We presented one technique for the problem of input consistency in cut-and-choose (from shelat and Shen (2011)). Many other input-consistency mechanisms have been proposed including Lindell and Pinkas (2007), Lindell and Pinkas (2011), Mohassel and Riva (2013), and shelat and Shen (2013).

We described the BDOZ and SPDZ approaches to authenticated secret-sharing. Other efficient approaches include Damgård and Zakarias (2013) and Damgård *et al.* (2017). Various approaches for efficiently generating the authenticated shares needed for the SPDZ approach are discussed by Keller *et al.* (2016) and Keller *et al.* (2018).

The GMW paradigm transforms a semi-honest-secure protocol into a malicious-secure one. However, it generally does not result in a protocol with practical efficiency. This is due to the fact that it treats the semi-honest-secure protocol in a *non-black-box* way — the parties must prove (in zero knowledge) statements about the next-message function of the semi-honest-secure protocol, which in the general case requires expressing that function as a circuit. Jarecki and Shmatikov (2007) propose a malicious variant of Yao's protocol that is similar in spirit to the GMW paradigm, in the sense that the garbling party proves correctness of each garbled gate (although at a cost of public-key operations for each gate).

A black-box approach for transforming a semi-honest-secure protocol into a malicious-secure one (Ishai *et al.*, 2007; Ishai *et al.*, 2008) is known as "MPC in the head." The idea is for the actual parties to imagine a protocol interaction among "virtual" parties. Instead of running an MPC protocol for the desired functionality, the actual parties run an MPC that realizes the behavior of the virtual parties. However, the MPC that is used to simulate the virtual parties can satisfy a weaker security notion (than what the overall MPC-in-the-head approach achieves), and the protocol being run among the virtual parties needs to be only semi-honest secure. For the special case of zero-knowledge proofs, the MPC-in-the-head approach results in protocols that are among the most efficient known (Giacomelli *et al.*, 2016; Chase *et al.*, 2017; Ames *et al.*, 2017; Katz *et al.*, 2018).

7

Alternative Threat Models

In this chapter, we consider some different assumptions about threats that lead to MPC protocols offering appealing security-performance trade-offs. First, we relax the assumption that any number of participants may be dishonest and discuss protocols designed to provide security only when a majority of the participants behave honestly. Assuming an honest majority allows for dramatic performance improvements. Then, we consider alternatives to the semi-honest and malicious models that have become standard in MPC literature, while still assuming that any number of participants may be corrupted. As discussed in the previous chapter, semi-honest protocols can be elevated into the malicious model, but this transformation incurs a significant cost overhead which may not be acceptable in practice. At the same time, real applications present a far more nuanced set of performance and security constraints. This prompted research into security models that offer richer trade-offs between security and performance. Section 7.1 discusses protocols designed to take advantage of the assumption that the majority of participants are honest. Section 7.2 discusses scenarios there trust between the participants is asymmetric, and the remaining sections present protocols designed to provide attractive security-performance trade-offs in settings motivated by practical scenarios.

7.1 Honest Majority

So far we have considered security against adversaries who may corrupt any number of the participants. Since the purpose of security is to protect the honest parties, the worst-case scenario for a protocol is that $n - 1$ out of n parties are corrupted.[1] In the two-party case, it is indeed the only sensible choice to consider one out of the two parties to be corrupt.

However, in the multi-party setting it often is reasonable to consider restricted adversaries that cannot corrupt as many parties as they want. A natural threshold is *honest majority*, where the adversary may corrupt strictly less than $\frac{n}{2}$ of the n parties. One reason this threshold is natural is that, assuming an honest majority, *every* function has an information-theoretically secure protocol (Ben-Or *et al.*, 1988; Chaum *et al.*, 1988), while there exist functions with no such protocol in the presence of $\lceil n/2 \rceil$ corrupt parties.

7.1.1 Building on Garbled Circuits

Yao's protocol (Section 3.1) provides semi-honest security, but to make it secure against malicious adversaries requires significant modifications with high computation and communication overhead (Section 6.1).

Mohassel *et al.* (2015) propose a simple *3-party variant* of Yao's protocol that is secure against a malicious adversary that corrupts at most one party (that is, the protocol is secure assuming an honest majority). Recall that the main challenge in making Yao's protocol secure against malicious attacks is to ensure that the garbled circuit is generated correctly. In the protocol of Mohassel *et al.* (2015), the main idea is to let two parties $\mathsf{P}_1, \mathsf{P}_2$ play the role of circuit garbler and one party P_3 play the role of circuit evaluator. First, P_1 and P_2 agree on randomness to be used in garbling, then they both garble the designated circuit *with the same randomness*, and send it to P_3. Since at most one of the garblers is corrupt (by the honest majority assumption), at least one of the garblers is guaranteed to be honest. Therefore, P_3 only needs to check that both garblers send identical garbled circuits. This ensures that the (unique)

[1] We consider only *static* security, where the adversary's choice of the corrupted parties is made once-and-for-all, at the beginning of the interaction. It is also possible to consider *adaptive* security, where parties can become corrupted throughout the protocol's execution. In the adpative setting, it does indeed make sense to consider scenarios where all parties are (eventually) corrupted.

garbled circuit is generated correctly. Other protocol issues relevant for the malicious case (like obtaining garbled inputs) are handled in a similar way, by checking responses from the two garbling parties for consistency.

One additional advantage of the 3-party setting is that there is no need for oblivious transfer (OT) as in the 2-party setting. Instead of using OT to deliver garbled inputs to the evaluator P_3, we can let P_3 secret-share its input and send one share (in the clear!) to each of P_1 and P_2. These garblers can send garbled inputs for each of these shares, and the circuit can be modified to reconstruct these shares before running the desired computation. Some care is required to ensure that the garblers send the correct garbled inputs in this way (Mohassel *et al.* (2015) provide details). Overall, the result is a protocol that avoids all OT and thus uses only inexpensive symmetric-key cryptography.

The basic protocol of Mohassel *et al.* (2015) has been generalized to provide additional properties like fairness (if the adversary learns the output, then the honest parties do) and guaranteed output delivery (all honest parties will receive output) (Patra and Ravi, 2018). Chandran *et al.* (2017) extend it to provide security against roughly $\sqrt{n}$ out of n corrupt parties.

7.1.2 Three-Party Secret Sharing

The honest-majority 3-party setting also enables some of the fastest general-purpose MPC implementations to date. These protocols achieve their high performance due to their extremely low communication costs — in some cases, as little as one bit per gate of the circuit!

It is possible to securely realize any functionality information-theoretically in the presence of an honest majority, using the classical protocols of Ben-Or *et al.* (1988) and Chaum *et al.* (1988). In these protocols, every wire in the circuit holds a value v, and the invariant of these protocols is that the parties collectively hold some additive secret sharing of v. As in Section 3.4, let $[v]$ denote such a sharing of v. For an addition gate $z = x + y$, the parties can compute a sharing $[x + y]$ from sharings $[x]$ and $[y]$ by local computation only, due to the additive homomorphism property of the sharing scheme. However, interaction and communication are required for multiplication gates to compute a sharing $[xy]$ from sharings $[x]$ and $[y]$.

In Section 3.4 we discussed how to perform such multiplications when pre-processed triples of the form $[a], [b], [ab]$ are available. It is also possible

to perform multiplications with all effort taking place during the protocol (i.e., not in a pre-processing phase). For example, the protocol of Ben-Or *et al.* (1988) uses Shamir secret sharing for its sharings $[v]$, and uses an interactive multiplication subprotocol in which all parties generate shares-of-shares and combine them linearly.

The protocols in this section are instances of this general paradigm, highly specialized for the case of 3 parties and 1 corruption ("1-out-of-3" setting). Both the method of sharing and the corresponding secure multiplication subprotocol are the target of considerable optimizations.

The Sharemind protocol of Bogdanov *et al.* (2008a) was the first to demonstrate high performance in this setting. Generally speaking, for the 1-out-of-3 case, it is possible to use a secret sharing scheme with threshold 2 (so that any 2 shares determine the secret). The Sharemind protocol instead uses 3-out-of-3 additive sharing, so that in $[v]$ party P_i holds value v_i such that $v = v_1 + v_2 + v_3$ (in an appropriate ring, such as $\mathbb{Z}_2$ for Boolean circuits). This choice leads to a simpler multiplication subprotocol in which each party sends 7 ring elements.

Launchbury *et al.* (2012) describe an alternative approach in which each party sends only 3 ring elements per multiplication. Furthermore, the communication is in a round-robin pattern, where the only communication is in the directions $\mathsf{P}_1 \to \mathsf{P}_2 \to \mathsf{P}_3 \to \mathsf{P}_1$. The idea behind multiplication is as follows. Suppose two values x and y are additively shared as $x = x_1 + x_2 + x_3$ and $y = y_1 + y_2 + y_3$, where party P_i holds x_i, y_i. To multiply x and y, it suffices to compute all terms of the form $x_a \cdot y_b$ for $a, b \in \{1, 2, 3\}$. Already P_i has enough information to compute $x_i y_i$, but the other terms are problematic. However, if each P_i sends its shares around the circle (i.e., P_1 sends to P_2, P_2 sends to P_3, and P_3 to P_1), then every term of the form $x_a y_b$ will computable by some party. Each party will now hold two of the x_i shares and two of the y_i shares. This still perfectly hides the values of x and y from a single corrupt party since we are using 3-out-of-3 sharing. The only problem is that shares of xy are correlated to shares of x and y, while it is necessary to have an independent sharing of xy. So, the parties generate a random additive sharing of zero and locally add it to their (non-random) sharing of xy.

Araki *et al.* (2016) propose a secret sharing scheme that is a variant of

replicated secret sharing. The sharing of a value v is defined as:

$$\mathsf{P}_1 \text{ holds } (x_1, x_3 - v) \qquad \mathsf{P}_2 \text{ holds } (x_2, x_1 - v) \qquad \mathsf{P}_3 \text{ holds } (x_3, x_2 - v)$$

where the x_i values are a random sharing of zero: $0 = x_1 + x_2 + x_3$. They describe a multiplication subprotocol in which each party sends only one ring element (in a round-robin fashion as above).[2] One of the ways in which communication is minimized in this approach is that the multiplication subprotocol generates its needed randomness *without any interaction.* Suppose there are three keys, k_1, k_2, k_3, for a pseudorandom function where P_1 holds (k_1, k_2), P_2 holds (k_2, k_3), and P_3 holds (k_3, k_1). Then, the parties can non-interactively generate an unbounded number of random sharings of zero. The i-th sharing of zero is generated via:

- P_1 computes $s_1 = F_{k_1}(i) - F_{k_2}(i)$
- P_2 computes $s_3 = F_{k_2}(i) - F_{k_3}(i)$
- P_3 computes $s_3 = F_{k_3}(i) - F_{k_1}(i)$

Here F is a pseudorandom function whose output is in the field. As desired, $s_1 + s_2 + s_3 = 0$ and the sharing is indeed random from the perspective of a single corrupt party. Such random sharings of zero are the only randomness needed by the multiplication subprotocol of Araki *et al.* (2016).

As a result of such minimal communication, Araki *et al.* (2016) report MPC performance of over 7 billion gates per second. This performance suffices to replace a single Kerberos authentication server with 3 servers running MPC, so that no single server ever sees user passwords, and is able to support loads of 35,000 logins per second.

These results are secure against one *semi-honest* corrupt party. Furukawa *et al.* (2017) extend the approach of Araki *et al.* (2016), and show how to achieve security against one *malicious* party (for the case of Boolean circuits). Their protocol follows the high-level approach described in Section 3.4, by generating multiplication triples $[a], [b], [ab]$. These triples use the secret-sharing technique of Araki *et al.* (2016). The starting point to generate such triples is also the multiplication protocol of Araki *et al.* (2016). Here, an

[2] Their protocol works as long as the number 3 is invertible in the ring. In particular, it can be used for Boolean arithmetic over $\mathbb{Z}_2$ as well as $\mathbb{Z}_{2^k}$.

important property of this protocol is used: a single corrupt party cannot cause the output of multiplication to be an *invalid* sharing, only a (valid) sharing of a different value. Hence, the adversary can cause any triple generated in this way to have the form $[a], [b], [ab]$ or $[a], [b], [\overline{ab}]$ (since this idea only applies for sharings of single bits). Starting with this observation, collections of triples are used to "cross-check" each other and guarantee their correctness.

7.2 Asymmetric Trust

Although the standard models assume all parties are equally distrusting, many realistic scenarios have asymmetric trust. For example, consider the setting where one of the two participants of a computation is a well-known business, such as a bank (denoted by P_1), providing service to another, less trusted, participant, bank's customer, denoted by P_2. It may be safe to assume that P_1 is unlikely to actively engage in cheating by deviating from the prescribed protocol. Indeed, banks today enjoy full customer trust and operate on all customer data in plaintext. Customers are willing to rely on established regulatory and legal systems, as well as the long-term reputation of the bank, rather than on cryptographic mechanisms, to protect their funds and transactions. Today, we not only trust the bank to correctly perform requested transactions, but we also trust that the bank will not misuse our data and will keep it in confidence.

However, there may be several reasons why a cautious customer who trusts the bank's intent may want to withhold certain private information and execute transactions via MPC. One is the *unintentional* data release. As any organization, bank may be a target of cyber attacks, and data stored by the bank, including customer data, may simply be stolen. Having employed MPC to safeguard the data eliminates this possibility, since the bank will not have the sensitive data in the first place. Another reason might be the legally mandated audits and summons of the data. As an organization, a bank may have a presence in several jurisdictions with different rules on data retention, release and reporting. Again, MPC will serve as a protection for unpredictable future data releases.

Hence, given the existing trust to the bank it seems reasonable to employ semi-honest model to protect the customer. However, having upgraded customer privacy by moving from plaintext operation to semi-honest MPC (and correspondingly placing the customer as a semi-honest player), we now actually

greatly compromised bank's security. Indeed, where previously the customer has been a passive participant simply providing the input to the transaction, now the customer has opportunities to cheat within the semi-honest MPC protocol. Given the relative ease of creating an account at a bank, and the opportunity for financial gain by an improperly conducted financial transaction, a bank's customer is naturally incentivized to cheat.

This scenario suggests a hybrid model where one player is assumed to be semi-honest and the other is assumed to be malicious. Luckily, Yao's GC and many of its natural variants are *already secure against malicious evaluator*! Indeed, assuming the OT protocol used by P_2 to obtain its input wires provides security against a malicious evaluator, P_2, the GC evaluator proceeds simply by decrypting garbled tables in a single, pre-determined way. Any deviation by the GC evaluator in the GC evaluation will simply result in a failure to obtain an output label. This is a very convenient feature of Yao's GC, which allows for an order of magnitude or more cost savings in settings where the security policy can assume a semi-honest generator (P_1).

Server-aided secure computation is another noteworthy example of taking advantage of asymmetric trust. The Salus system (Kamara *et al.*, 2012) considers a natural setting where one participant is a special external party without input or output, whose goal is to assist the other parties in securely evaluating the function on their inputs. The external party might be a cloud service provider with larger computational resources at its disposal and, importantly, sufficient status to be partially trusted by all the other participants. For example, this player may be trusted to behave semi-honestly, while other players may be assumed to be malicious. An important assumption made by Kamara *et al.* (2012) is that the server *does not collude* with any of the other players, even if the server is malicious. Substantial performance improvements can be obtained by the Salus systems and subsequent work by taking advantage of the asymmetric trust and the collusion restriction. For example, Kamara *et al.* (2014) present a server-aided private set intersection protocol, which, in the case of the semi-honest server, computes PSI of billion-element sets in about 580 seconds while sending about 12.4 GB of data.

7.3 Covert Security

While reasonable in many settings, at times the above semi-honest/malicious asymmetric trust model is insufficient for applications where P_1 may have incentives to cheat. Even for a mostly trusted bank, cheating is nevertheless a real possibility, and bank customers and even external auditors have no tools to ensure compliance. At the same time, moving fully into the malicious model will incur a large performance penalty that may not be acceptable.

An effective practical approach may be to enable probabilistic checks on the generator, as first proposed by Aumann and Lindell (2007). Their idea is to allow the evaluator (P_2) to challenge the generator (P_1) at random to prove that the garbled circuit was constructed correctly. If P_1 is unable to prove this, then P_2 will know that P_1 is cheating and will react accordingly. The guarantee achieved is that a cheating player will be caught with a certain fixed probability (e.g., $\epsilon = \frac{1}{2}$) known as the *deterrence factor*.

This simple and natural idea can be formalized in the definition in several ways. First, it is essential that P_1 cannot forge an invalid proof or elude or withdraw from a validity proof challenge on a garbled circuit it produced. On the other hand, we accept that if P_2 did not challenge P_1 on an improperly-generated circuit, then P_1 was not caught cheating, and can win (i.e., learn something about P_2's private input or corrupt the function output). However, various privacy guarantees with respect to P_2's input may be considered. Aumann and Lindell (2007) propose three formulations:

1. *Failed simulation*. The idea is to allow the simulator (of the cheating party) to fail sometimes. "Fail" means that its output distribution is not indistinguishable from the real one. This corresponds to an event of successful cheating. The model guarantees that the probability that the adversary is caught cheating is at least ϵ times the probability that the simulator fails.

   One serious issue with the above definition is that it *only* requires that if cheating occurred in the real execution, the cheater will be caught with probability ϵ. The definition does not prevent the cheating player from deciding *when* to cheat (implicitly) based on the *honest* player's input. In particular, P_1 could attempt cheat only on the more valuable inputs of P_2 (e.g., natural protocols exist which allow P_1 to attempt to cheat

only when P_2's first bit is 0).

2. *Explicit cheat*. This formulation introduces an explicit ability to cheat to an ideal-model adversary (i.e., the simulator). This only slightly complicates the definition, but allows us to naturally prescribe that cheating in the ideal model can only occur *independently* of the other players' inputs. In the ideal model, the cheating player, upon sending a cheat instruction to the trusted party, will obtain the honest players' inputs. At the same time, the honest players in the ideal model will output $\mathsf{corrupted}_i$ (i.e., detect cheating by P_i) with probability ϵ, thus requiring that the same happens in the real model.

   Although this model is much more convincing, it has the drawback that the malicious player is allowed to obtain inputs of the honest parties even when cheating is detected. As Aumann and Lindell noted, "there is less deterrence to not rob a bank if when you are caught you are allowed to keep the stolen money." (Aumann and Lindell, 2007)

3. *Strong explicit cheat*. This is the same as the explicit cheat formulation, with the exception that the cheating ideal-model adversary *is not allowed* to obtain the honest players' inputs in the case where cheating is detected.

The first two (strictly weaker) models of Aumann and Lindell (2007) did not gain significant popularity mainly because the much stronger third model admits protocols of the same or very similar efficiency as the weaker ones. The *strong explicit cheat* model became standard due to its simplicity, effectiveness in deterrence, and the discovery of simple and efficient protocols that achieve it. We present one such simple and efficient 2PC protocol next.

7.3.1 Covert Two-Party Protocol

Since the work of Aumann and Lindell (2007), significant progress in efficient OT has produced several extremely efficient *malicious* OT protocols (Asharov *et al.*, 2015b; Keller *et al.*, 2015), with the latter having overhead over the semi-honest OT of only 5%. As a result, we don't consider covert OT security, and assume a maliciously-secure OT building block. It is important to remember, however, that a maliciously-secure protocol does not guarantee the players submit prescribed inputs. In particular, while malicious OT ensures correct

and private OT execution, deviating players can submit arbitrary OT inputs, such as invalid wire labels.

Next, we overview the method of Aumann and Lindell (2007) for building a covert 2PC protocol from Yao's GC and malicious OT. For simplicity, we assume deterrence factor of $\epsilon = \frac{1}{2}$. It is straightforward to efficiently generalize this for any non-negligible ϵ. First, we present the basic idea and point out missing pieces, which we then address.

Core Protocol. Aumann and Lindell go along the lines of the cut-and-choose approach and propose that P_1 generates and sends to P_2 *two* GCs. The two garbled circuits $\widehat{C}_0$ and $\widehat{C}_1$ are generated from random seeds s_0 and s_1 respectively by expanding them using a pseudo-random generator (PRG). Upon receipt, P_2 flips a coin $b \in \{0, 1\}$ and asks P_1 to *open* the circuit $\widehat{C}_b$ by providing s_b. Because the GCs are constructed deterministically from a seed via a PRG expansion, opening is achieved simply by sending the seed to the verifier. This allows P_2 to check the correctness of the generated garbled circuit $\widehat{C}_b$ by constructing the same garbled circuit from the provided seed, and comparing it to the copy that was sent. This guarantees that a badly constructed $\widehat{C}$ will be detected with probability $\epsilon = \frac{1}{2}$, which is needed to satisfy the strong explicit cheat definition.

However, a malicious P_1 can also perform OT-related attacks. For example, P_1 can flip the semantics of the labels on P_2's input wires, effectively silently flipping P_2's input. Similarly, P_1 can set *both* P_2's input wire labels to the same value, effectively setting P_2's input to a fixed value. Another attack is the selective abort attack discussed in Section 6.1, where one of the two OT secrets is set to be a dummy random value, resulting in an selectively aborted evaluation that allows P_1 to learn a bit of P_2's input.

As a result, we must ensure that a OT input substitution by P_1 is caught at least with probability equal to the deterrence factor ϵ. Note that input substitution by P_2 is allowed as it simply corresponds to P_2 choosing a different MPC input, a behavior allowed by the security definition.

Next, we discuss defenses to these attacks.

Preventing OT Input Substitution and Selective Abort. The solution enhances the basic protocol described above to provide additional protections

against OT input substitution by P_1. First, recall that the *entire* GC is generated from a seed, which includes the input wire labels. Further, OT is run prior to P_2's challenge being announced, so P_1 will not be able to adjust its tactic in response to the challenge and provide honest OT inputs in the challenge execution and malicious OT inputs in the live execution. At the same time, OT delivers only one of the two inputs to P_2, and P_2 cannot verify that the other OT input was correctly submitted by P_1, leaving open the possibility of the selective abort attack, described above.

This is not satisfactory in the strong explicit cheat formulation, since this model requires that if cheating is detected, the dishonest party cannot learn anything about the honest player's input. There are several ways to address this. Aumann and Lindell suggest using the inputs XOR tree idea, which was earlier proposed by Lindell and Pinkas (2007). The idea is to modify the circuit C being computed and correspondingly change the semantics of P_2's input wires as follows. Instead of each of P_2's input bit x_i, the new circuit C' will have σ inputs $x_i^1, \ldots, x_i^\sigma$, which are random with the restriction that $x_i = \bigoplus_{j \in \{1..i\}} x_i^j$. For each x_i of C, the new circuit C' will start by xor-ing the σ inputs $x_i^1, \ldots, x_i^\sigma$ so as to recover x_i inside the circuit. C' then continues as the original C would with the reconstructed inputs. The new circuit computes the same function, but now P_1 can only learn information if it is correctly guesses *all* σ random x_i^j, an event occurring with statistically negligible probability.

Alternate Keys. We additionally mention the following mainly theoretical attack discussed by Aumann and Lindell (2007) to underline the subtleties of even seemingly simple MPC protocols. The issue is that an adversary can construct (at least in theory) a garbled circuit with two sets of keys, where one set of keys decrypts the circuit to the specified one and another set of keys decrypts the circuit to an incorrect one. This is not a real issue in most natural GC constructions, but one can construct a tailored GC protocol with this property. The existence of two sets of keys is problematic because the adversary can supply "correct keys" to the circuits that are opened and "incorrect keys" to the circuit that is evaluated. Aumann and Lindell prevent this by having P_1 commit to these keys and send the commitments together with the garbled circuits. Then, instead of P_2 just sending the keys associated with its input, it sends the appropriate decommitments.

Protocol Completion. Finally, to conclude the informal description of the protocol, after P_2 successfully conducts the above checks, it proceeds by evaluating $\widehat{C}_{1-b}$ and obtaining the output which is then also sent to P_1. It now can be shown that malicious behavior of P_1 can be caught with probability $\epsilon = \frac{1}{2}$. We note that this probability can be increased simply by having P_1 generate and send more circuits, all but one of which are opened and checked. Aumann and Lindell (2007) provide a detailed treatment of the definitions and the formal protocol construction.

7.4 Publicly Verifiable Covert (PVC) Security

In the covert security model, a party can deviate arbitrarily from the protocol description but is caught with a fixed probability ϵ, called the *deterrence factor*. In many practical scenarios, this guaranteed risk of being caught (likely resulting in loss of business or embarrassment) is sufficient to deter would-be cheaters, and covert protocols are much more efficient and simpler than their malicious counterparts.

At the same time, the cheating deterrent introduced by the covert model is relatively weak. Indeed, an honest party catching a cheater certainly knows what happened and can respond accordingly (e.g., by taking their business elsewhere). However, the impact is largely limited to this, since the honest player cannot credibly *accuse* the cheater publicly. Doing so might require the honest player to reveal its private inputs (hence, violate its security), or the protocol may simply not authenticate messages as coming from a specific party. If, however, credible public accusation (i.e., a publicly-verifiable cryptographic proof of the cheating) were possible, the deterrent for the cheater would be much greater: suddenly, *all* the cheater's customers and regulators would be aware of the cheating and thus any cheating may affect the cheater's global customer base.

The addition of credible accusation greatly improves the covert model even in scenarios with a small number of players, such as those involving the government. Consider, for example, the setting where two agencies are engaged in secure computation on their respective classified data. The covert model may often be insufficient here. Indeed, consider the case where one of the two players deviates from the protocol, perhaps due to an insider attack. The honest player detects this, but non-participants are now faced with the problem

of identifying the culprit across two domains, where the communication is greatly restricted due to trust, policy, data privacy legislation, or all of the above. On the other hand, credible accusation immediately provides the ability to exclude the honest player from the suspect list, and focus on tracking the problem within the misbehaving organization, which is dramatically simpler.

PVC Definition. Asharov and Orlandi (2012) proposed a security model, *covert with public verifiability*, and an associated protocol, motivated by these concerns. At a high level, they proposed that when cheating is detected, the honest player can publish a "certificate of cheating" that can be checked by any third party. We will call this model PVC, following the notation of Kolesnikov and Malozemoff (2015), who proposed an improved protocol in this model.

Informally, the PVC definition requires the following three properties to hold with overwhelming probability:

1. Whenever cheating is detected, the honest party can produce a publicly verifiable proof of cheating.

2. Proof of cheating cannot be forged. That is, an honest player cannot be accused of cheating with a proof that verifies.

3. Proof of cheating does not reveal honest party's private data (including the data used in the execution where cheating occurred).

Asharov-Orlandi PVC Protocol. The Asharov-Orlandi protocol has performance similar to the original covert protocol of Aumann and Lindell (2007) on which it is based, with the exception of requiring signed-OT, a special form of oblivious transfer (OT). Their signed-OT construction, which we summarize next, is based on the OT of Peikert *et al.* (2008), and thus requires several expensive public-key operations per OT instance. After this, we describe several performance improvements to it proposed by Kolesnikov and Malozemoff (2015), the most important of which is a novel signed-OT extension protocol that eliminates per-instance public-key operations.

As usual, we make P_1 the circuit generator, P_2 the evaluator, and use C to represent the circuit to execute. Recall from Section 7.2 that in the standard Yao's garbled circuit construction in the semi-honest model, a malicious evaluator (P_2) cannot cheat during the GC evaluation. Hence, this protection

comes for free with natural GC protocols, and we only need consider a malicious generator (P_1).

Recall the selective failure attack on P_2's input wires, where P_1 sends P_2 (via OT) an invalid wire label for one of P_2's two possible inputs and learns which input bit P_2 selected based on whether or not P_2 aborts. To protect against this attack, the parties construct a new circuit C' that prepends an input XOR tree in C as discussed in Section 7.3. To elevate to the covert model, P_1 then constructs λ (the *GC replication factor*) garblings of C' and P_2 randomly selects $\lambda - 1$ of them and checks they are correctly constructed, and evaluates the remaining C' garbled circuit to derive the output.

We now adapt this protocol to the PVC setting by allowing P_2 to not only detect cheating, but also to obtain a publicly verifiable proof of cheating if cheating is detected. The basic idea is to require the generator P_1 to establish a public-private keypair, and to *sign* the messages it sends. The intent is that signed inconsistent messages (e.g., badly formed GCs) can be published and will serve as a convincing proof of cheating. The main difficulty of this approach is ensuring that neither party can improve its odds by selectively aborting. For example, if P_1 could abort whenever P_2's challenge would reveal that P_1 is cheating (and hence avoid sending a signed inconsistent transcript), this would enable P_1 to cheat without the risk of generating a proof of cheating.

Asharov and Orlandi address this by preventing P_1 from knowing P_2's challenge when producing the response. In their protocol, P_1 sends the GCs to P_2 and opens the checked circuits by responding to the challenge through a 1-out-of-λ OT. For this, P_1 first sends all (signed) GCs to P_2. Then the players run OT, where in the i-th input to the OT P_1 provides openings (seeds) for all the GCs except for the i-th, as well as the input wire labels needed to evaluate $\widehat{C}_i$. Party P_2 inputs a random $\gamma \in_R [\lambda]$, so receives the seeds for all circuits other than $\widehat{C}_\gamma$ from the OT and the wire labels for its input for $\widehat{C}_\gamma$. Then, P_2 checks that all GCs besides $\widehat{C}_\gamma$ are constructed correctly; if the check passes, P_2 evaluates $\widehat{C}_\gamma$. Thus, P_1 does not know which GC is being evaluated, and which ones are checked.

However, a more careful examination shows that this actually does not quite get us to the PVC goal. Indeed, a malicious P_1 simply can include invalid openings for the OT secrets which correspond to the undesirable choice of the challenge γ. The Asharov-Orlandi protocol addresses this by having P_1

sign all its messages as well as using a *signed*-OT in place of all standard OTs (including wire label transfers and GC openings). Informally, the signed-OT functionality proceeds as follows. Rather than the receiver $\mathcal{R}$ getting message $\mathbf{m}_b$ (which might include a signature that $\mathcal{S}$ produced) from the sender $\mathcal{S}$ for choice bit b, the signature component of signed-OT is explicit in the OT definition. Namely, we require that $\mathcal{R}$ receives $((b, \mathbf{m}_b), \mathsf{Sig})$, where Sig is $\mathcal{S}$'s valid signature of $(b, \mathbf{m}_b)$. This guarantees that $\mathcal{R}$ will always receive a valid signature on the OT output it receives. Thus, if $\mathcal{R}$'s challenge detects cheating, the (inconsistent) transcript will be signed by $\mathcal{S}$, so can be used as proof of this cheating. Asharov and Orlandi (2012) show that this construction is ϵ-PVC-secure for $\epsilon = (1 - 1/\lambda)(1 - 2^{-\nu+1})$, where ν is the replication factor of the employed XOR tree, discussed above.

We note that their signed-OT heavily relies on public-key operations, and cannot use the much more efficient OT extension.

Signed-OT Extension. Kolesnikov and Malozemoff (2015) proposed an efficient signed-OT extension protocol built on the malicious OT extension of Asharov *et al.* (2015b). Informally, signed-OT extension (similarly to the signed-OT of Asharov-Orlandi) ensures that (1) a cheating sender $\mathcal{S}$ is held accountable in the form of a "certificate of cheating" that the honest receiver $\mathcal{R}$ can generate, (2) the certificate of cheating *does not reveal* honest player's inputs and, (3) a malicious $\mathcal{R}$ *cannot defame* an honest $\mathcal{S}$ by fabricating a false "certificate of cheating."

Achieving the first goal is fairly straightforward and can be done by having $\mathcal{S}$ simply sign *all* its messages sent in the course of executing OT extension. Then, the view of $\mathcal{R}$, which will now include messages signed by $\mathcal{S}$, will exhibit inconsistency. The challenge is in simultaneously:

1. protecting the privacy of $\mathcal{R}$'s input—this is a concern since the view of $\mathcal{R}$ exhibiting inconsistency also may include $\mathcal{R}$'s input, and

2. preventing a malicious $\mathcal{R}$ from manipulating the part of the view which is not under $\mathcal{S}$'s signature to generate a false accusation of cheating.

Since the view of $\mathcal{R}$ plays the role of the proof of cheating, we must ensure certain non-malleability of the view of $\mathcal{R}$, to prevent it from defaming the honest $\mathcal{S}$. For this, we need to commit $\mathcal{R}$ to its particular choices throughout

the OT extension protocol. At the same time, we must maintain that those commitments do not leak any information about $\mathcal{R}$'s choices. Next, we sketch how this can be done, assuming familiarity with the details of the OT extension of Ishai *et al.* (2003) (IKNP from Section 3.7.2).

Recall that in the standard IKNP OT extension protocol, $\mathcal{R}$ constructs a random matrix M, and $\mathcal{S}$ obtains a matrix M' derived from the matrix M, $\mathcal{S}$'s random string $\mathbf{s}$, and $\mathcal{R}$'s vector of OT inputs $\mathbf{r}$. The matrix M is the main component of $\mathcal{R}$'s view which, together with $\mathcal{S}$'s signed messages will constitute a proof of cheating.

To reiterate, we must address two issues. First, because M' is obtained by applying $\mathcal{R}$'s private input $\mathbf{r}$ to M, and M' is known to $\mathcal{S}$, M is now sensitive and cannot be revealed. Second, we must prevent $\mathcal{R}$ from publishing a doctored M, which would enable a false cheating accusation. Kolesnikov and Malozemoff (2015) (KM) resolve both issues by observing that $\mathcal{S}$ does in fact learn some of the elements of M, since in the OT extension construction some of the columns of M and M' are the same (i.e., those corresponding to zero bits of $\mathcal{S}$'s string $\mathbf{s}$).

The KM signed-OT construction prevents $\mathcal{R}$ from cheating by having $\mathcal{S}$ include in its signature carefully selected information from the columns in M which $\mathcal{S}$ sees. Finally, the protocol requires that $\mathcal{R}$ generate each row of M from a seed, and that $\mathcal{R}$'s proof of cheating includes this seed such that the row rebuilt from the seed is consistent with the columns included in $\mathcal{S}$'s signature. Kolesnikov and Malozemoff (2015) show that this makes it infeasible for $\mathcal{R}$ to successfully present an invalid row of the OT matrix in the proof of cheating. The KM construction is in the random oracle model, a slight strengthening of the assumptions needed for standard OT extension and FreeXOR, two standard secure computation tools.

The KM construction is also interesting from a theoretical perspective in that it shows how to construct signed-OT from *any* maliciously secure OT protocol, whereas Asharov and Orlandi (2012) build a specific construction based on the Decisional Diffie-Hellman problem.

7.5 Reducing Communication in Cut-and-Choose Protocols

A basic technique in cut-and-choose used to achieve malicious-level security, and in the covert and PVC models, is for P_1 to send several garbled circuits,

of which several are opened and checked, and one (or more for malicious security) is evaluated. The opened garbled circuits have no further use, since they hold no secrets. They only serve as a commitment for purposes of the challenge protocol. Can *commitments* to these GCs be sent and verified instead, achieving the same utility?

Indeed, as formalized by Goyal *et al.* (2008), this is possible in covert and malicious cut-and-choose protocols. One must, of course, be careful with the exact construction. One suitable construction is provided by Goyal *et al.* (2008). Kolesnikov and Malozemoff (2015) formalize a specific variant of hashing, which works with their PVC protocol, thus resulting of the PVC protocol being of the same communication cost as the semi-honest Yao's GC protocol.

Free Hash. While GC hashing allows for significant communication savings, it may not provide much overall savings in total execution time since hashing is a relatively expensive operation. Motivated by this, Fan *et al.* (2017) proposed a way to compute a GC hash simply by xor-ing (slightly malleated) entries of garbled tables. Their main idea is to *greatly weaken* the security definition of the GC hash. Instead of requiring that it is infeasible to find a hash collision, they *allow* an adversary to generate a garbled circuit $\widehat{C}'$ whose hash collides with an honestly generated $\widehat{C}$, as long as such a $\widehat{C}'$ with high probability will fail evaluation and cheating will be discovered. Fan *et al.* (2017) then show how to intertwine hash generation and verification with GC generation and evaluation, such that the resulting hash is generated at no extra computational cost and meets the above definition.

7.6 Trading Off Leakage for Efficiency

Maliciously secure MPC provides extremely strong security guarantees, at a cost. In many cases, the security needs may allow for less than absolute inability of a malicious attacker to learn even a single bit of information. At the same time, a large majority of the cost of MPC comes from the "last mile" of entirely protecting all private information. Given this, useful trade-offs between MPC security and efficiency have been explored. In this section we discuss the dual-execution approach of Mohassel and Franklin (2006), as well as several private database systems.

Dual Execution. The *dual-execution* 2PC protocol of Mohassel and Franklin (2006) capitalizes on the fact that only the circuit generator is able to cheat in 2PC GC, assuming malicious-secure OT. Indeed, the evaluator simply performs a sequence of decryptions, and deviation of the prescribed protocol results in the evaluator being stuck, which is equivalent to abort. The idea behind dual execution is to allow *both* players to play a role of the generator (hence forcing the opponent to play the role of the evaluator where it will not be able to cheat). As a result, in each honest player's view, the execution where it was the generator must be the correct one. However, the honest player additionally must ensure that the other execution is not detrimental to security, for example, by leaking private data.

To achieve this, Mohassel and Franklin (2006) propose that the two executions must produce the same candidate output—a difference in candidate outputs in the two executions implies cheating, and the computation must be aborted without output to avoid leaking information from an invalid circuit. In the protocol, the two parties run two separate instances of Yao's semi-honest protocol, so that for one instance P_1 is the generator and P_2 is the evaluator, and for the other instance P_2 is the generator and P_1 is the evaluator. Each party evaluates the garbled circuit they received to obtain a (still garbled) output. Then the two parties run a *fully maliciously secure* equality test protocol to check whether their outputs are semantically equal (before they are decoded). Each party inputs both the garbled output they evaluated and the output wire labels of the garbled circuit they generated. If the outputs don't match, then the parties abort. Of course, abort is an exception, and the aborting honest party will know that that other player cheated in the execution. While this equality check subprotocol is evaluated using a malicious-secure protocol, its cost is small since the computed comparison function is fixed and only depends on the size of the output.

The dual-execution protocol is *not* fully secure in the malicious model. Indeed, the honest party executes an adversarially-crafted garbled circuit and uses the garbled output in the equality-test subprotocol. A malicious party can generate a garbled circuit that produces the correct output for some of the other party's inputs, but an incorrect one for others. Then, the attacker learns whether the true function output is equal to an arbitrary predicate on honest party input by observing the abort behavior of the protocol. However, since the

equality test has only one bit of output, it can be shown that the dual-execution protocol leaks at most one (adversarially-chosen) bit describing the honest party's input.

In a follow-up work, Huang *et al.* (2012b) formalize the dual-execution definition of Mohassel and Franklin and propose several optimizations including showing how the two executions can be interleaved to minimize the overall latency of dual execution overhead over semi-honest single execution. In another follow-up work, Kolesnikov *et al.* (2015) show how the leakage function of the dual-execution 2PC can be greatly restricted and the probability of leakage occurring reduced.

Memory Access Leakage: Private DB Querying. Another strategy for trading-off some leakage for substantial efficiency gains is to incorporate custom protocols into an MPC protocol that leak some information for large performance gains. This can be particularly effective (and important) for applications involving large-scale data analysis. Large-scale data collection and use has become essential to operation of many fields, such as security, analysis, optimization, and others. Much of the data collected and analyzed is private. As a result, a natural question arises regarding the feasibility of MPC application to today's data sets.

One particular application of interest is private database querying. The goal is to enable a database server (DB) to respond to client's *encrypted* queries, so as to protect the sensitive information possibly contained in the query. Private queries is a special case of MPC, and can be instantiated by using generic MPC techniques.

One immediate constraint, critical to our discussion is the necessity of *sublinear* data access. A fully secure approach would involve scanning the *entire* DB for a single query to achieve semi-honest security. This is because omitting even a single DB entry reveals to the players that this entry was not in the result set. Sublinear execution immediately implies loss of security of MPC even in the semi-honest model. However, as covered in Chapter 5, when linear-time preprocessing is allowed, amortized costs of data access can be sublinear while achieving full formal cryptographic guarantees.

Known algorithms for fully secure sublinear access, however, are still very expensive, bringing 3–4 orders of magnitude performance penalties.

A promising research direction looks into allowing certain (hopefully, well-understood) leakage of information about private inputs in the pursuit of increased efficiency in protocols for large-scale data search.

Several systems designed to provide encrypted database functionalities use MPC as an underlying technology (Pappas *et al.*, 2014; Poddar *et al.*, 2016). In the Blind Seer project (Pappas *et al.*, 2014; Fisch *et al.*, 2015), two- and three-party MPC was used as an underlying primitive for implementing encrypted search tree traversal, performed jointly by the client and the server.

Both areas are active research areas, and several commercial systems are deployed that provide searchable encryption and encrypted databases. We note that today we don't have precise understanding of the impact of the leaked information (or the information contained in the authorized query results sets, for that matter). Several attacks have shown that data leaked for the sake of efficiency can often be exploited (Islam *et al.*, 2012; Naveed *et al.*, 2015; Cash *et al.*, 2015). Understanding how to make appropriate trade-offs between leakage and efficiency remains an important open problem, but one that many practical systems must face.

7.7 Further Reading

MPC today is already fast enough to be applied to many practical problems, yet larger-scale MPC applications are often still impractical, especially in the fully-malicious model. In this chapter, we discussed several approaches that aim to strike a balance between security and performance. Reducing the adversary power (i.e., moving from malicious to covert and PVC models) enables approximately a factor of 10 cost reduction compared to authenticated garbling (Section 6.7) and a factor of 40 compared to the best cut-and-choose methods (Section 6.1). Allowing for additional information to be revealed promises even more significant performance improvements, up to several orders of magnitude in some cases (as seen in the private database work in Section 7.6). Of course, it is important to understand the effect of the additional leakage. We stress, however, that there is no fundamental difference in the *extra* leakage during the protocol execution, and the allowed information obtained from the output of the computation. Both can be too damaging and both should be similarly analyzed to understand if (securely) computing the proposed function is safe.

Another approach that can produce dramatic performance improvements is employing secure hardware such as Intel SGX or secure smartcards (Ohrimenko *et al.*, 2016; Gupta *et al.*, 2016; Bahmani *et al.*, 2017; Zheng *et al.*, 2017). This offers a different kind of a trade-off: should we trust the manufacturer of the hardware (we must consider both competence and possible prior or future malicious intent) to greatly improve performance of secure computing? There is no clear answer here, as hardware security seems to be a cat-and-mouse game, with significant attacks steadily appearing, and manufacturers catching up in their revisions cycle. Examples of recent attacks on SGX include devastating software-only key recovery (Bulck *et al.*, 2018), which does not make any assumption on victim's enclave code and does not necessarily require kernel-level access. Other attacks include exploits of software vulnerabilities, e.g., (Lee *et al.*, 2017a), or side channels, e.g., (Xu *et al.*, 2015; Lee *et al.*, 2017b). Ultimately, delivering high-performance in a CPU requires very complex software and hardware designs, which are therefore likely to include subtle errors and vulnerabilities. Secure enclaves present an attractive and high-value attack target, while their vulnerabilities hard to detect in the design cycle. As such, they may be suitable for computing on lower-value data in larger instances where MPC is too slow, but not where high assurance is needed.

Theoretical cryptography explored hardware security from a formal perspective, with the goal of achieving an ideal leak-free hardware implementation of cryptographic primitives. The motivation for this work is today's extensive capabilities to learn hardware-protected secrets by observing the many side channels of hardware computation including power consumption, timing, electromagnetic radiation, and acoustic noise. Leakage-resilient cryptography was a very popular research direction in late-2000's, whose intensity since subsided. Works in this area typically assumed either a small secure component in hardware (small in chip area and computing only a minimal functionality, such as a pseudorandom function), or that leakage collected by an adversarial observer at any one time slice/epoch is *a priori* bounded. The goal was then to build provably-secure (i.e., leakage-free) hardware for computing a specific function based on these assumptions. A foundational paper by Micali and Reyzin (2004) introduced the notion of *physically observable cryptography* and proposed an influential theoretical framework to model side-channel attacks. In particular, they state and motivate the "only computation leaks information"

axiom used in much of leakage-resilient cryptography work. Juma and Vahlis (2010) show how to compute any function in a leak-free manner by using fully-homomorphic encryption and a single "leak-free" hardware token that samples from a distribution that does not depend on the protected key or the function that is evaluated on it. A corresponding line of research in the more practical hardware space acknowledges that real-world leakage functions are much stronger and more complex than what is often assumed by theoretical cryptography. In this line of work, heuristic, experimental and mitigation approaches are typical. Several MPC works have also taken advantage of (assumed) secure hardware tokens computing basic functions, such as PRF. Goals here include eliminating assumptions (e.g., Katz (2007) and Goyal *et al.* (2010)) or improving performance (Kolesnikov, 2010).

8

Conclusion

In the past decade or so, MPC made dramatic strides, developing from a theoretical curiosity to a versatile tool for building privacy-preserving applications. For most uses, the key metric is cost, and the cost of deploying MPC has declined by 3–9 orders of magnitude in the past decade.

The first reported 2PC system, Fairplay (Malkhi *et al.*, 2004), executed a 4383-gate circuit in the semi-honest model, taking over 7 seconds on a local area network at a rate of about 625 gates per second. Modern 2PC frameworks can execute about 3 million gates per second on a 1Gbps LAN, and scale to circuits with hundreds of billions of gates.

Cost improvements for malicious secure MPC have been even more dramatic. The first substantial attempt to implement malicious secure generic MPC was Lindell *et al.* (2008), intriguingly titled "Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries". It reports malicious evaluation of the 16-bit comparison circuit, consisting of fifteen 3-to-1 gates and one 2-to-1 gate, in between 135 to 362 seconds depending on the security parameter settings. This evaluation rate corresponds to about 0.13 gates per second. An implementation of the authenticated garbling scheme (Section 6.7) reports malicious security 2PC at over 0.8 million gates per second on a 10Gbps LAN (Wang *et al.*, 2017b). This corresponds to over a

6 million *factor* performance improvement in under a decade! With honest-majority malicious 3PC the performance that can be achieved is even more remarkable (Section 7.1.2). Araki *et al.* (2017) report a 3-machine 20-core cluster that can evaluate a *billion* gates per second over a 10Gbps channel.

It is fair to say that the progress in the field of applied MPC is truly astounding, and things that would have been considered impossible a few years ago, and now considered routine.

Despite this progress, and many emerging real world uses, MPC is not yet widespread in practice. There remain several challenges to overcome before MPC can be deployed for a wide range of privacy-preserving applications. We discuss a few, and possible approaches for overcoming them next.

Cost. Despite dramatic advances in the 2PC and MPC technology in the past decade, secure function evaluation still may incur several orders of magnitude cost penalty over standard (non-private) execution, especially when protecting against malicious players. The exact overhead varies greatly from virtually non-existent to unacceptable, and mostly depends on the computed function.

In particular, for generic MPC, the protocols described in this book (which are, arguably, the ones that are most scalable today in typical settings) all require bandwidth that scales linearly in the size of the circuit. Bandwidth within a data center is inexpensive (indeed, many cloud providers do not charge customers anything for bandwidth between nodes within the same data center), but it requires a strong trust model to assume all participants in an MPC would be willing to outsource their computation to the same cloud provider. In some use cases, this linear bandwidth cost may be prohibitively expensive. Making bandwidth cost sublinear in circuit size requires a very different paradigm. Although it has been shown to be possible with threshold-based FHE schemes (Asharov *et al.*, 2012), such schemes are a long way from being practical. Recent results have shown that function-sharing schemes can be used to build lower-bandwidth MPC protocols for certain classes of functions (Boyle *et al.*, 2016a; Boyle *et al.*, 2018).

The solution to this bandwidth cost seems to require hybrid protocols that combining MPC with custom protocols or homomorphic encryption to enable secure computation without linear bandwidth cost. We have covered several approaches that incorporate these strategies in MPC including private

set intersection Section 3.8.1, RAM-MPC Section 6 and ABY Section 4.3, but the limits of this approach are not known. Custom protocols can offer much lower costs, but developing custom protocols is tedious and only cost-effective for the most important and performance-critical operations. Future work may find more principled and automated was to combine generic MPC with homomorphic encryption and custom protocols to enable very efficient solutions without compromising security properties.

Another direction for dramatically reducing the cost of MPC is to employ secure hardware such as Intel's SGX. This assumes a somewhat awkward (but often realistic in practice) threat model where a vendor is trusted to implement a secure enclave and manage its keys, but not trusted fully to be an trusted third party. Signal recently released a private contact discovery service using SGX (Marlinspike, 2017), and many researchers are pursuing MPC designs that take advantage of SGX (Bahmani *et al.*, 2017; Shaon *et al.*, 2017; Priebe *et al.*, 2018; Mishra *et al.*, 2018).

Leakage trade-offs. Clearly, the MPC performance race will continue, although specific improvement areas may shift. For example, we have already achieved several MPC milestones, such as being able to fully utilize a very cable 10Gbps communication channel. Further, there exist several barriers for algorithmic performance improvement, such as the need for two ciphertexts for AND gate encryption, shown by Zahur *et al.* (2015). This may limit expected performance gains in basic generic Yao-based MPC.

Indicative of the recognition of these barriers and the data processing demands, a line of work emerged addressing the trade-off between MPC efficiency and achieved security guarantees, which we partially covered in Chapter 7. Another possible direction for leakage trade-offs is within the core protocols, where large efficiency gains may be achieved by giving up the strong security guarantees targeted by current protocols for more flexible information disclosure limits.

In terms of which MPC operation is in most need of improvement, achieving secure access to random memory locations stands out. Despite all the improvements described in Chapter 5, today's ORAM protocols are still concretely inefficient and are the main limit in scaling MPC to large data applications. An alternative to ORAM, in the absence of a satisfactory

fully secure solution, could be security-aware leakage of data access patterns, accompanied by a formal analysis of how this information satisfies security and privacy requirements. It is plausible that in many scenarios it can be formally shown, e.g., using tools from programming languages among others, that revealing some information about the access pattern is acceptable.

Output leakage. The goal of MPC is to protect the privacy of inputs and intermediate results, but at the end of the protocol the output of the function is revealed. A separate research field has developed around the complementary problem where there is no need to protect the input data, but the output must be controlled to limit what an adversary can infer about the private data from the output. The dominant model for controlling output leakage is *differential privacy* (Dwork and Roth, 2014) which adds privacy-preserving noise to outputs before they are revealed. A few works have explored combining MPC with differential privacy to provide end-too-end privacy for computations with distributed data (Pettai and Laud, 2015; Kairouz *et al.*, 2015; He *et al.*, 2017), but this area is in its infancy and many challenging problems need to be solved before the impacts of different types of leakage are well understood.

Meaningful trust. When cryptographers analyze protocols, they assume each party has a way to executed their part of the protocol that they fully trust. In reality, protocols are executed by complex software and hardware, incorporating thousands of components provided by different vendors and programmers. The malicious secure protocols we discussed in this book provide a user with strong guarantees that if their own system executes its role in the protocol correctly, no matter what the other participants do the security guarantees are established. But, in order to fully trust an MPC application, a user also needs to fully trust that the system executing the protocol on its behalf, which is given full access to their private data in cleartext, will execute the protocol correctly. This includes issues like ensuring the implementation has no side channels that might leak the user's data, as well as knowing that it performs all protocol steps correctly and does not attempt to leak the user's private input.

Providing full confidence in a computing system is a longstanding and challenging problem that applies to all secure uses of computing, but is

particularly sharp in considering the security of MPC protocols. In many deployments, all participants end up running the same software, provided by a single trusted party, and without any auditing. In practice, this provides better security that just giving all the data to the software provider; but in theory, it is little different from just handing over all the plaintext data to the software (or hardware) provider.

One appealing aspect of MPC techniques is they can be used to alleviate the need to place any trust in a computing system—if private data is split into shares and computed on using two systems using MPC, the user no longer needs to worry about implementation bugs in either systems exposing the data since the MPC protocol ensures that, regardless of the type of bug in the system even including hardware side channels, there is no risk of data exposure since the private data is always protected by cryptography withing the MPC protocol and is not visible to the system at all.

Finding ways to meaningfully convey to end users how their data will be exposed, and what they are trusting in providing it, is a major challenge for future computing systems. Making progress on this is essential for enabling privacy-enhancing technologies like MPC to provide meaningful and understandable benefits to the vast majority of computing users, rather than just to sophisticated organizations with teams of cryptographers and program analysts.

We wish to conclude this book on an optimistic note. The awareness that personal data can be compromised in a data breach, or can be abused by companies whose interests do not align with those of their users, is increasing. New regulations, including the European Union's *General Data Protection Regulation*, are making holding personal data a liability risk for companies. MPC has emerged as a powerful and versatile primitive that can provide organizations and individuals with a range of options for designing privacy-preserving applications. Many challenges remain, but MPC (and secure computation broadly) is a young and vibrant field, with much opportunity to create, develop and apply.

Acknowledgements

The authors thank Jeanette Wing for instigating this project; our editors at Now Publishers, James Finlay and Mike Casey, for help and flexibility throughout the writing process; and Alet Heezemans for help with the final editing.

Vladimir Kolesnikov was supported in part by Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Mike Rosulek was supported in part by the National Science Foundation (award #1617197), a Google Research award, and a Visa Research award.

David Evans was supported in part by National Science Foundation awards #1717950 and #1111781, and research awards from Google, Intel, and Amazon.

References

Afshar, A., Z. Hu, P. Mohassel, and M. Rosulek. 2015. "How to Efficiently Evaluate RAM Programs with Malicious Security". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. *Lecture Notes in Computer Science*. Springer, Heidelberg. 702–729. DOI: 10.1007/978-3-662-46800-5_27.

Aly, A., M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. Smart, and T. Wood. 2018. "SCALE and MAMBA Documentation". https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf.

Ames, S., C. Hazay, Y. Ishai, and M. Venkitasubramaniam. 2017. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 2087–2104.

Araki, T., A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. 2017. "Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 843–862.

Araki, T., J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. 2016. "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 805–817.

Asharov, G., A. Beimel, N. Makriyannis, and E. Omri. 2015a. "Complete Characterization of Fairness in Secure Two-Party Computation of Boolean Functions". In: *TCC 2015: 12th Theory of Cryptography Conference, Part I*. Ed. by Y. Dodis and J. B. Nielsen. Vol. 9014. *Lecture Notes in Computer Science*. Springer, Heidelberg. 199–228. DOI: 10.1007/978-3-662-46494-6_10.

Asharov, G., A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. 2012. "Multiparty computation with low communication, computation and interaction via threshold FHE". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt)*. Springer. 483–501.

Asharov, G., Y. Lindell, T. Schneider, and M. Zohner. 2015b. "More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. *Lecture Notes in Computer Science*. Springer, Heidelberg. 673–701. DOI: 10.1007/978-3-662-46800-5_26.

Asharov, G. and C. Orlandi. 2012. "Calling Out Cheaters: Covert Security with Public Verifiability". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by X. Wang and K. Sako. Vol. 7658. *Lecture Notes in Computer Science*. Springer, Heidelberg. 681–698. DOI: 10.1007/978-3-642-34961-4_41.

Aumann, Y. and Y. Lindell. 2007. "Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries". In: *TCC 2007: 4th Theory of Cryptography Conference*. Ed. by S. P. Vadhan. Vol. 4392. *Lecture Notes in Computer Science*. Springer, Heidelberg. 137–156.

Bahmani, R., M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. 2017. "Secure multiparty computation from SGX". In: *International Conference on Financial Cryptography and Data Security*. 477–497.

Ball, M., T. Malkin, and M. Rosulek. 2016. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 565–577.

Bar-Ilan Center for Research in Applied Cryptography and Cyber Security. 2014. "SCAPI: Secure Computation API". https://cyber.biu.ac.il/scapi/.

Beaver, D. 1992. "Efficient Multiparty Protocols Using Circuit Randomization". In: *Advances in Cryptology – CRYPTO'91*. Ed. by J. Feigenbaum. Vol. 576. *Lecture Notes in Computer Science*. Springer, Heidelberg. 420–432.

Beaver, D. 1995. "Precomputing Oblivious Transfer". In: *Advances in Cryptology – CRYPTO'95*. Ed. by D. Coppersmith. Vol. 963. *Lecture Notes in Computer Science*. Springer, Heidelberg. 97–109.

Beaver, D. 1996. "Correlated Pseudorandomness and the Complexity of Private Computations". In: *28th Annual ACM Symposium on Theory of Computing*. ACM Press. 479–488.

Beaver, D., S. Micali, and P. Rogaway. 1990. "The Round Complexity of Secure Protocols (Extended Abstract)". In: *22nd Annual ACM Symposium on Theory of Computing*. ACM Press. 503–513.

Beerliová-Trubíniová, Z. and M. Hirt. 2008. "Perfectly-Secure MPC with Linear Communication Complexity". In: *TCC 2008: 5th Theory of Cryptography Conference*. Ed. by R. Canetti. Vol. 4948. *Lecture Notes in Computer Science*. Springer, Heidelberg. 213–230.

Beimel, A. and B. Chor. 1993. "Universally Ideal Secret Sharing Schemes (Preliminary Version)". In: *Advances in Cryptology – CRYPTO'92*. Ed. by E. F. Brickell. Vol. 740. *Lecture Notes in Computer Science*. Springer, Heidelberg. 183–195.

Bellare, M., V. T. Hoang, S. Keelveedhi, and P. Rogaway. 2013. "Efficient Garbling from a Fixed-Key Blockcipher". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 478–492.

Bellare, M., V. T. Hoang, and P. Rogaway. 2012. "Foundations of garbled circuits". In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM Press. 784–796.

Bellare, M. and P. Rogaway. 1993. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93: 1st Conference on Computer and Communications Security*. Ed. by V. Ashby. ACM Press. 62–73.

Bendlin, R., I. Damgård, C. Orlandi, and S. Zakarias. 2011. "Semi-homomorphic Encryption and Multiparty Computation". In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by K. G. Paterson. Vol. 6632. *Lecture Notes in Computer Science*. Springer, Heidelberg. 169–188.

Ben-Or, M., S. Goldwasser, and A. Wigderson. 1988. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th Annual ACM Symposium on Theory of Computing*. ACM Press. 1–10.

Bestavros, A., A. Lapets, and M. Varia. 2017. "User-centric Distributed Solutions for Privacy-preserving Analytics". *Communications of the ACM*. 60(2): 37–39. ISSN: 0001-0782. DOI: 10.1145/3029603.

Biryukov, A., D. Khovratovich, and I. Nikolic. 2009. "Distinguisher and Related-Key Attack on the Full AES-256". In: *Advances in Cryptology – CRYPTO 2009*. Ed. by S. Halevi. Vol. 5677. *Lecture Notes in Computer Science*. Springer, Heidelberg. 231–249.

Bogdanov, D. 2015. "Smarter decisions with no privacy breaches - practical secure computation for governments and companies". https://rwc.iacr.org/2015/Slides/RWC-2015-Bogdanov-final.pdf, retrieved March 9, 2018.

Bogdanov, D., S. Laur, and J. Willemson. 2008a. "Sharemind: A Framework for Fast Privacy-Preserving Computations". In: *ESORICS 2008: 13th European Symposium on Research in Computer Security*. Ed. by S. Jajodia and J. López. Vol. 5283. *Lecture Notes in Computer Science*. Springer, Heidelberg. 192–206.

Bogdanov, D., S. Laur, and J. Willemson. 2008b. "Sharemind: A framework for fast privacy-preserving computations". In: *European Symposium on Research in Computer Security*. Springer. 192–206.

Bogetoft, P., D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. 2009. "Secure Multiparty Computation Goes Live". In: *FC 2009: 13th International Conference on Financial Cryptography and Data Security*. Ed. by R. Dingledine and P. Golle. Vol. 5628. *Lecture Notes in Computer Science*. Springer, Heidelberg. 325–343.

Boneh, D., E.-J. Goh, and K. Nissim. 2005. "Evaluating 2-DNF formulas on ciphertexts". In: *Theory of Cryptography Conference*. Springer. 325–341.

Boyle, E., N. Gilboa, and Y. Ishai. 2016a. "Breaking the circuit size barrier for secure computation under DDH". In: *Annual Cryptology Conference*. Springer. 509–539.

Boyle, E., N. Gilboa, and Y. Ishai. 2016b. "Function Secret Sharing: Improvements and Extensions". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 1292–1303.

Boyle, E., Y. Ishai, and A. Polychroniadou. 2018. "Limits of Practical Sublinear Secure Computation". In: *Annual Cryptology Conference*. Springer.

Brandão, L. T. A. N. 2013. "Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique - (Extended Abstract)". In: *Advances in Cryptology – ASIACRYPT 2013, Part II*. Ed. by K. Sako and P. Sarkar. Vol. 8270. *Lecture Notes in Computer Science*. Springer, Heidelberg. 441–463. DOI: 10.1007/978-3-642-42045-0_23.

Brickell, J., D. E. Porter, V. Shmatikov, and E. Witchel. 2007. "Privacy-preserving remote diagnostics". In: *ACM CCS 07: 14th Conference on Computer and Communications Security*. Ed. by P. Ning, S. D. C. di Vimercati, and P. F. Syverson. ACM Press. 498–507.

Buescher, N. and S. Katzenbeisser. 2015. "Faster Secure Computation through Automatic Parallelization." In: *USENIX Security Symposium*. 531–546.

Buescher, N., A. Weber, and S. Katzenbeisser. 2018. "Towards Practical RAM Based Secure Computation". In: *European Symposium on Research in Computer Security*. Springer. 416–437.

Bulck, J. V., M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium*. Baltimore, MD: USENIX Association. 991–1008.

Burkhart, M., M. Strasser, D. Many, and X. Dimitropoulos. 2010. "SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics". In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, USA: USENIX Association.

Calctopia, Inc. 2017. "SECCOMP — The Secure Spreadsheet". https://www.calctopia.com/.

Canetti, R. 2001. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press. 136–145.

Canetti, R., A. Cohen, and Y. Lindell. 2015. "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation". In: *Advances in Cryptology – CRYPTO 2015, Part II*. Ed. by R. Gennaro and M. J. B. Robshaw. Vol. 9216. *Lecture Notes in Computer Science*. Springer, Heidelberg. 3–22. DOI: 10.1007/978-3-662-48000-7_1.

Canetti, R., O. Goldreich, and S. Halevi. 1998. "The Random Oracle Methodology, Revisited (Preliminary Version)". In: *30th Annual ACM Symposium on Theory of Computing*. ACM Press. 209–218.

Canetti, R., A. Jain, and A. Scafuro. 2014. "Practical UC security with a Global Random Oracle". In: *ACM CCS 14: 21st Conference on Computer and Communications Security*. Ed. by G.-J. Ahn, M. Yung, and N. Li. ACM Press. 597–608.

Carter, H., B. Mood, P. Traynor, and K. Butler. 2013. "Secure outsourced Garbled Circuit Evaluation for Mobile Devices". In: *22nd USENIX Security Symposium*. USENIX Association.

Carter, H., B. Mood, P. Traynor, and K. Butler. 2016. "Secure outsourced garbled circuit evaluation for mobile devices". *Journal of Computer Security*. 24(2): 137–180.

Cash, D., P. Grubbs, J. Perry, and T. Ristenpart. 2015. “Leakage-Abuse Attacks Against Searchable Encryption”. In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 668–679.

Chan, T.-H. H., K.-M. Chung, B. Maggs, and E. Shi. 2017. “Foundations of Differentially Oblivious Algorithms”. Cryptology ePrint Archive, Report 2017/1033. https://eprint.iacr.org/2017/1033.

Chandran, N., J. A. Garay, P. Mohassel, and S. Vusirikala. 2017. “Efficient, Constant-Round and Actively Secure MPC: Beyond the Three-Party Case”. In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 277–294.

Chase, M., D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha. 2017. “Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives”. In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 1825–1842.

Chaum, D. 1983. “Blind Signature System”. In: *Advances in Cryptology – CRYPTO’83*. Ed. by D. Chaum. Plenum Press, New York, USA. 153.

Chaum, D., C. Crépeau, and I. Damgård. 1988. “Multiparty Unconditionally Secure Protocols (Extended Abstract)”. In: *20th Annual ACM Symposium on Theory of Computing*. ACM Press. 11–19.

Chillotti, I., N. Gama, M. Georgieva, and M. Izabachène. 2016. “Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds”. In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by J. H. Cheon and T. Takagi. Vol. 10031. *Lecture Notes in Computer Science*. Springer, Heidelberg. 3–33. DOI: 10.1007/978-3-662-53887-6_1.

Chillotti, I., N. Gama, M. Georgieva, and M. Izabachène. 2017. “Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE”. In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by T. Takagi and T. Peyrin. Vol. 10624. *Lecture Notes in Computer Science*. Springer, Heidelberg. 377–408.

Choi, S. G., K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. 2012a. "Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces". In: *Topics in Cryptology – CT-RSA 2012*. Ed. by O. Dunkelman. Vol. 7178. *Lecture Notes in Computer Science*. Springer, Heidelberg. 416–432.

Choi, S. G., J. Katz, R. Kumaresan, and H.-S. Zhou. 2012b. "On the Security of the "Free-XOR" Technique". In: *TCC 2012: 9th Theory of Cryptography Conference*. Ed. by R. Cramer. Vol. 7194. *Lecture Notes in Computer Science*. Springer, Heidelberg. 39–53.

Chor, B., O. Goldreich, E. Kushilevitz, and M. Sudan. 1995. "Private Information Retrieval". In: *36th Symposium on Foundations of Computer Science*. IEEE. 41–50.

Clarke, E., D. Kroening, and F. Lerda. 2004. "A tool for checking ANSI-C programs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 168–176.

Cleve, R. 1986. "Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract)". In: *18th Annual ACM Symposium on Theory of Computing*. ACM Press. 364–369.

Cybernetica. 2015. "Track Big Data Between Government and Education". https://sharemind.cyber.ee/big-data-analytics-protection/, retrieved March 9, 2018.

Damgård, I. and M. Jurik. 2001. "A generalisation, a simpli. cation and some applications of Paillier's probabilistic public-key system". In: *International Workshop on Public Key Cryptography*. 119–136.

Damgård, I., M. Keller, E. Larraia, C. Miles, and N. P. Smart. 2012a. "Implementing AES via an actively/covertly secure dishonest-majority MPC protocol". In: *International Conference on Security and Cryptography for Networks*. Springer. 241–263.

Damgård, I., J. B. Nielsen, M. Nielsen, and S. Ranellucci. 2017. "The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited". In: *Advances in Cryptology – CRYPTO 2017, Part I*. Ed. by J. Katz and H. Shacham. Vol. 10401. *Lecture Notes in Computer Science*. Springer, Heidelberg. 167–187.

Damgård, I., V. Pastro, N. P. Smart, and S. Zakarias. 2012b. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by R. Safavi-Naini and R. Canetti. Vol. 7417. *Lecture Notes in Computer Science*. Springer, Heidelberg. 643–662.

Damgård, I. and S. Zakarias. 2013. "Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing". In: *TCC 2013: 10th Theory of Cryptography Conference*. Ed. by A. Sahai. Vol. 7785. *Lecture Notes in Computer Science*. Springer, Heidelberg. 621–641. DOI: 10.1007/978-3-642-36594-2_35.

D'Arco, P. and R. De Prisco. 2014. "Secure Two-Party Computation: A Visual Way". In: *ICITS 13: 7th International Conference on Information Theoretic Security*. Ed. by C. Padró. Vol. 8317. *Lecture Notes in Computer Science*. Springer, Heidelberg. 18–38. DOI: 10.1007/978-3-319-04268-8_2.

D'Arco, P. and R. De Prisco. 2016. "Secure computation without computers". 651(Sept.).

De Cristofaro, E., M. Manulis, and B. Poettering. 2013. "Private Discovery of Common Social Contacts". *International Journal of Information Security*. 12(1): 49–65.

Demmler, D., T. Schneider, and M. Zohner. 2015. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *ISOC Network and Distributed System Security Symposium – NDSS 2015*. The Internet Society.

Dessouky, G., F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner. 2017. "Pushing the communication barrier in secure computation using lookup tables". In: *Network and Distributed System Security Symposium*.

Doerner, J., D. Evans, and A. Shelat. 2016. "Secure Stable Matching at Scale". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 1602–1613.

Doerner, J. and A. Shelat. 2017. "Scaling ORAM for Secure Computation". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 523–535.

Dwork, C. and A. Roth. 2014. "The algorithmic foundations of differential privacy". *Foundations and Trends in Theoretical Computer Science*. 9(3–4): 211–407.

Ejgenberg, Y., M. Farbstein, M. Levy, and Y. Lindell. 2012. "SCAPI: The Secure Computation Application Programming Interface". Cryptology ePrint Archive, Report 2012/629. https://eprint.iacr.org/2012/629.

Faber, S., S. Jarecki, S. Kentros, and B. Wei. 2015. "Three-party ORAM for secure computation". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 360–385.

Fan, X., C. Ganesh, and V. Kolesnikov. 2017. "Hashing Garbled Circuits for Free". In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Ed. by J. Coron and J. B. Nielsen. Vol. 10211. *Lecture Notes in Computer Science*. Springer, Heidelberg. 456–485.

Fisch, B. A., B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. 2015. "Malicious-Client Security in Blind Seer: A Scalable Private DBMS". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 395–410. DOI: 10.1109/SP.2015.31.

Fraser, C. W. and D. R. Hanson. 1995. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc.

Frederiksen, T. K., T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti. 2015. "TinyLEGO: An Interactive Garbling Scheme for Maliciously Secure Two-Party Computation". Cryptology ePrint Archive, Report 2015/309. https://eprint.iacr.org/2015/309.

Frederiksen, T. K., T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. 2013. "MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by T. Johansson and P. Q. Nguyen. Vol. 7881. *Lecture Notes in Computer Science*. Springer, Heidelberg. 537–556. DOI: 10.1007/978-3-642-38348-9_32.

Furukawa, J., Y. Lindell, A. Nof, and O. Weinstein. 2017. "High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority". In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Ed. by J. Coron and J. B. Nielsen. Vol. 10211. *Lecture Notes in Computer Science*. Springer, Heidelberg. 225–255.

Gallagher, B., D. Lo, P. F. Frandsen, J. B. Nielsen, and K. Nielsen. 2017. "Insights Network – A Blockchain Data Exchange". https://s3.amazonaws.com/insightsnetwork/InsightsNetworkWhitepaperV0.5.pdf.

Gascón, A., P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. 2017. "Privacy-Preserving Distributed Linear Regression on High-Dimensional Data". *Proceedings on Privacy Enhancing Technologies*. 2017(4): 248–267.

Gentry, C. 2009. "Fully homomorphic encryption using ideal lattices". In: *41st ACM Symposium on Theory of Computing*.

Gentry, C. and S. Halevi. 2011. "Implementing Gentry's Fully-Homomorphic Encryption Scheme". In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by K. G. Paterson. Vol. 6632. *Lecture Notes in Computer Science*. Springer, Heidelberg. 129–148.

Giacomelli, I., J. Madsen, and C. Orlandi. 2016. "ZKBoo: Faster Zero-Knowledge for Boolean Circuits". In: *25th USENIX Security Symposium*. Austin, TX: USENIX Association. 1069–1083.

Gilboa, N. and Y. Ishai. 2014. "Distributed Point Functions and Their Applications". In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by P. Q. Nguyen and E. Oswald. Vol. 8441. *Lecture Notes in Computer Science*. Springer, Heidelberg. 640–658. DOI: 10.1007/978-3-642-55220-5_35.

Goldreich, O. 2004. *Foundations of Cryptography: Volume 2*. Cambridge University Press.

Goldreich, O., S. Micali, and A. Wigderson. 1987. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th Annual ACM Symposium on Theory of Computing*. Ed. by A. Aho. ACM Press. 218–229.

Goldreich, O. and R. Ostrovsky. 1996. "Software Protection and Simulation on Oblivious RAMs". *Journal of the ACM*. 43(3).

Goldwasser, S. and S. Micali. 1984. "Probabilistic Encryption". *Journal of Computer and System Sciences*. 28(2): 270–299.

Goldwasser, S., S. Micali, and C. Rackoff. 1985. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)". In: *17th Annual ACM Symposium on Theory of Computing*. ACM Press. 291–304.

Gordon, S. D., C. Hazay, J. Katz, and Y. Lindell. 2008. "Complete fairness in secure two-party computation". In: *40th Annual ACM Symposium on Theory of Computing*. Ed. by R. E. Ladner and C. Dwork. ACM Press. 413–422.

Gordon, S. D., J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. 2012. "Secure two-party computation in sublinear (amortized) time". In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM Press. 513–524.

Goyal, V., Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. 2010. "Founding Cryptography on Tamper-Proof Hardware Tokens". In: *TCC 2010: 7th Theory of Cryptography Conference*. Ed. by D. Micciancio. Vol. 5978. *Lecture Notes in Computer Science*. Springer, Heidelberg. 308–326.

Goyal, V., P. Mohassel, and A. Smith. 2008. "Efficient Two Party and Multi Party Computation Against Covert Adversaries". In: *Advances in Cryptology – EUROCRYPT 2008*. Ed. by N. P. Smart. Vol. 4965. *Lecture Notes in Computer Science*. Springer, Heidelberg. 289–306.

Gueron, S., Y. Lindell, A. Nof, and B. Pinkas. 2015. "Fast Garbling of Circuits Under Standard Assumptions". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 567–578.

Gupta, D., B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. 2016. "Using Intel Software Guard Extensions for efficient two-party secure function evaluation". In: *International Conference on Financial Cryptography and Data Security*. Springer. 302–318.

Gupta, T., H. Fingler, L. Alvisi, and M. Walfish. 2017. "Pretzel: Email encryption and provider-supplied functions are compatible". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM. 169–182.

Halevi, S. and V. Shoup. 2014. "Bootstrapping for HElib". Cryptology ePrint Archive, Report 2014/873. https://eprint.iacr.org/2014/873.

Hazay, C. and Y. Lindell. 2008. "Constructions of truly practical secure protocols using standardsmartcards". In: *ACM CCS 08: 15th Conference on Computer and Communications Security*. Ed. by P. Ning, P. F. Syverson, and S. Jha. ACM Press. 491–500.

He, X., A. Machanavajjhala, C. J. Flynn, and D. Srivastava. 2017. "Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 1389–1406.

Henecka, W., S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. 2010. "TASTY: tool for automating secure two-party computations". In: *ACM CCS 10: 17th Conference on Computer and Communications Security*. Ed. by E. Al-Shaer, A. D. Keromytis, and V. Shmatikov. ACM Press. 451–462.

Hofheinz, D. and V. Shoup. 2011. "GNUC: A New Universal Composability Framework". Cryptology ePrint Archive, Report 2011/303. http://eprint.iacr.org/2011/303.

Holzer, A., M. Franz, S. Katzenbeisser, and H. Veith. 2012. "Secure two-party computations in ANSI C". In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM Press. 772–783.

Huang, Y., P. Chapman, and D. Evans. 2011a. "Privacy-Preserving Applications on Smartphones". In: *6th USENIX Workshop on Hot Topics in Security*.

Huang, Y., D. Evans, and J. Katz. 2012a. "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society.

Huang, Y., D. Evans, J. Katz, and L. Malka. 2011b. "Faster Secure Two-Party Computation Using Garbled Circuits". In: *20th USENIX Security Symposium*.

Huang, Y., J. Katz, and D. Evans. 2012b. "Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution". In: *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 272–284.

Huang, Y., J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff. 2014. "Amortizing Garbled Circuits". In: *Advances in Cryptology – CRYPTO 2014, Part II*. Ed. by J. A. Garay and R. Gennaro. Vol. 8617. *Lecture Notes in Computer Science*. Springer, Heidelberg. 458–475. DOI: 10.1007/978-3-662-44381-1_26.

Huang, Y., L. Malka, D. Evans, and J. Katz. 2011c. "Efficient Privacy-Preserving Biometric Identification". In: *ISOC Network and Distributed System Security Symposium – NDSS 2011*. The Internet Society.

Husted, N., S. Myers, A. Shelat, and P. Grubbs. 2013. "GPU and CPU parallelization of honest-but-curious secure two-party computation". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM. 169–178.

Impagliazzo, R. and S. Rudich. 1989. "Limits on the Provable Consequences of One-Way Permutations". In: *21st Annual ACM Symposium on Theory of Computing*. ACM Press. 44–61.

Ishai, Y., J. Kilian, K. Nissim, and E. Petrank. 2003. "Extending Oblivious Transfers Efficiently". In: *Advances in Cryptology – CRYPTO 2003*. Ed. by D. Boneh. Vol. 2729. *Lecture Notes in Computer Science*. Springer, Heidelberg. 145–161.

Ishai, Y., E. Kushilevitz, R. Ostrovsky, and A. Sahai. 2007. "Zero-knowledge from secure multiparty computation". In: *39th Annual ACM Symposium on Theory of Computing*. Ed. by D. S. Johnson and U. Feige. ACM Press. 21–30.

Ishai, Y., M. Prabhakaran, and A. Sahai. 2008. "Founding Cryptography on Oblivious Transfer - Efficiently". In: *Advances in Cryptology – CRYPTO 2008*. Ed. by D. Wagner. Vol. 5157. *Lecture Notes in Computer Science*. Springer, Heidelberg. 572–591.

Islam, M. S., M. Kuzu, and M. Kantarcioglu. 2012. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation". In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society.

Jagadeesh, K., D. Wu, J. Birgmeier, D. Boneh, and G. Bejerano. 2017. "Deriving Genomic Diagnoses Without Revealing Patient Genomes". *Science*. 357(6352): 692–695.

Jakobsen, T. P., J. B. Nielsen, and C. Orlandi. 2016. "A Framework for Outsourcing of Secure Computation". Cryptology ePrint Archive, Report 2016/037. https://eprint.iacr.org/2016/037 (subsumes earlier version published in 6th ACM Workshop on Cloud Computing Security).

Jarecki, S. and V. Shmatikov. 2007. "Efficient Two-Party Secure Computation on Committed Inputs". In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by M. Naor. Vol. 4515. *Lecture Notes in Computer Science*. Springer, Heidelberg. 97–114.

Jawurek, M., F. Kerschbaum, and C. Orlandi. 2013. "Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press. 955–966.

Juma, A. and Y. Vahlis. 2010. "Protecting Cryptographic Keys against Continual Leakage". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by T. Rabin. Vol. 6223. *Lecture Notes in Computer Science*. Springer, Heidelberg. 41–58.

Kairouz, P., S. Oh, and P. Viswanath. 2015. "Secure multi-party differential privacy". In: *Advances in Neural Information Processing Systems*. 2008–2016.

Kamara, S., P. Mohassel, M. Raykova, and S. S. Sadeghian. 2014. "Scaling Private Set Intersection to Billion-Element Sets". In: *FC 2014: 18th International Conference on Financial Cryptography and Data Security*. Ed. by N. Christin and R. Safavi-Naini. Vol. 8437. *Lecture Notes in Computer Science*. Springer, Heidelberg. 195–215. DOI: 10.1007/978-3-662-45472-5_13.

Kamara, S., P. Mohassel, and B. Riva. 2012. "Salus: a system for server-aided secure function evaluation". In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM Press. 797–808.

Katz, J. 2007. "Universally Composable Multi-party Computation Using Tamper-Proof Hardware". In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by M. Naor. Vol. 4515. *Lecture Notes in Computer Science*. Springer, Heidelberg. 115–128.

Katz, J., V. Kolesnikov, and X. Wang. 2018. "Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures". Cryptology ePrint Archive, Report 2018/475. https://eprint.iacr.org/2018/475.

Keller, M., E. Orsini, and P. Scholl. 2015. "Actively Secure OT Extension with Optimal Overhead". In: *Advances in Cryptology – CRYPTO 2015, Part I*. Ed. by R. Gennaro and M. J. B. Robshaw. Vol. 9215. *Lecture Notes in Computer Science*. Springer, Heidelberg. 724–741. DOI: 10.1007/978-3-662-47989-6_35.

Keller, M., E. Orsini, and P. Scholl. 2016. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 830–842.

Keller, M., V. Pastro, and D. Rotaru. 2018. "Overdrive: making SPDZ great again". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 158–189.

Keller, M. and P. Scholl. 2014. "Efficient, oblivious data structures for MPC". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 506–525.

Kempka, C., R. Kikuchi, and K. Suzuki. 2016. "How to circumvent the two-ciphertext lower bound for linear garbling schemes". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 967–997.

Kennedy, W. S., V. Kolesnikov, and G. T. Wilfong. 2017. "Overlaying Conditional Circuit Clauses for Secure Computation". In: *Advances in Cryptology – ASIACRYPT 2017, Part II*. Ed. by T. Takagi and T. Peyrin. Vol. 10625. *Lecture Notes in Computer Science*. Springer, Heidelberg. 499–528.

Kerschbaum, F., T. Schneider, and A. Schröpfer. 2014. "Automatic Protocol Selection in Secure Two-Party Computations". In: *ACNS 14: 12th International Conference on Applied Cryptography and Network Security*. Ed. by I. Boureanu, P. Owesarski, and S. Vaudenay. Vol. 8479. *Lecture Notes in Computer Science*. Springer, Heidelberg. 566–584. DOI: 10.1007/978-3-319-07536-5_33.

Kilian, J. 1988. "Founding Cryptography on Oblivious Transfer". In: *20th Annual ACM Symposium on Theory of Computing*. ACM Press. 20–31.

Kiraz, M. and B. Schoenmakers. 2006. "A protocol issue for the malicious case of Yao's garbled circuit construction". In: *27th Symposium on Information Theory in the Benelux*. 283–290.

Knudsen, L. R. and V. Rijmen. 2007. "Known-Key Distinguishers for Some Block Ciphers". In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by K. Kurosawa. Vol. 4833. *Lecture Notes in Computer Science*. Springer, Heidelberg. 315–324.

Kolesnikov, V. 2005. "Gate Evaluation Secret Sharing and Secure One-Round Two-Party Computation". In: *Advances in Cryptology – ASIACRYPT 2005*. Ed. by B. K. Roy. Vol. 3788. *Lecture Notes in Computer Science*. Springer, Heidelberg. 136–155.

Kolesnikov, V. 2006. "Secure Two-party Computation and Communication". University of Toronto Ph.D. Thesis.

Kolesnikov, V. 2010. "Truly Efficient String Oblivious Transfer Using Resettable Tamper-Proof Tokens". In: *TCC 2010: 7th Theory of Cryptography Conference*. Ed. by D. Micciancio. Vol. 5978. *Lecture Notes in Computer Science*. Springer, Heidelberg. 327–342.

Kolesnikov, V. and R. Kumaresan. 2013. "Improved OT Extension for Transferring Short Secrets". In: *Advances in Cryptology – CRYPTO 2013, Part II*. Ed. by R. Canetti and J. A. Garay. Vol. 8043. *Lecture Notes in Computer Science*. Springer, Heidelberg. 54–70. DOI: 10.1007/978-3-642-40084-1_4.

Kolesnikov, V., R. Kumaresan, M. Rosulek, and N. Trieu. 2016. "Efficient Batched Oblivious PRF with Applications to Private Set Intersection". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press. 818–829.

Kolesnikov, V. and A. J. Malozemoff. 2015. "Public Verifiability in the Covert Model (Almost) for Free". In: *Advances in Cryptology – ASIACRYPT 2015, Part II*. Ed. by T. Iwata and J. H. Cheon. Vol. 9453. *Lecture Notes in Computer Science*. Springer, Heidelberg. 210–235. DOI: 10.1007/978-3-662-48800-3_9.

Kolesnikov, V., N. Matania, B. Pinkas, M. Rosulek, and N. Trieu. 2017a. "Practical Multi-party Private Set Intersection from Symmetric-Key Techniques". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 1257–1272.

Kolesnikov, V., P. Mohassel, B. Riva, and M. Rosulek. 2015. "Richer Efficiency/Security Trade-offs in 2PC". In: *TCC 2015: 12th Theory of Cryptography Conference, Part I*. Ed. by Y. Dodis and J. B. Nielsen. Vol. 9014. *Lecture Notes in Computer Science*. Springer, Heidelberg. 229–259. DOI: 10.1007/978-3-662-46494-6_11.

Kolesnikov, V., P. Mohassel, and M. Rosulek. 2014. "FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR". In: *Advances in Cryptology – CRYPTO 2014, Part II*. Ed. by J. A. Garay and R. Gennaro. Vol. 8617. *Lecture Notes in Computer Science*. Springer, Heidelberg. 440–457. DOI: 10.1007/978-3-662-44381-1_25.

Kolesnikov, V., J. B. Nielsen, M. Rosulek, N. Trieu, and R. Trifiletti. 2017b. "DUPLO: Unifying Cut-and-Choose for Garbled Circuits". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 3–20.

Kolesnikov, V., A.-R. Sadeghi, and T. Schneider. 2009. "Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima". In: *CANS 09: 8th International Conference on Cryptology and Network Security*. Ed. by J. A. Garay, A. Miyaji, and A. Otsuka. Vol. 5888. *Lecture Notes in Computer Science*. Springer, Heidelberg. 1–20.

Kolesnikov, V., A.-R. Sadeghi, and T. Schneider. 2010. "From Dust to Dawn: Practically Efficient Two-Party Secure Function Evaluation Protocols and their Modular Design". https://eprint.iacr.org/2010/079.

Kolesnikov, V., A.-R. Sadeghi, and T. Schneider. 2013. "A Systematic Approach to Practically Efficient General Two-party Secure Function Evaluation Protocols and Their Modular Design". *J. Comput. Secur.* 21(2): 283–315. ISSN: 0926-227X. URL: http://dl.acm.org/citation.cfm?id=2590614.2590617.

Kolesnikov, V. and T. Schneider. 2008a. "A Practical Universal Circuit Construction and Secure Evaluation of Private Functions". In: *FC 2008: 12th International Conference on Financial Cryptography and Data Security*. Ed. by G. Tsudik. Vol. 5143. *Lecture Notes in Computer Science*. Springer, Heidelberg. 83–97.

Kolesnikov, V. and T. Schneider. 2008b. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*. Ed. by L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz. Vol. 5126. *Lecture Notes in Computer Science*. Springer, Heidelberg. 486–498.

Kreuter, B. 2017. “Secure MPC at Google”. Real World Crypto.

Kreuter, B., a. shelat abhi, B. Mood, and K. Butler. 2013. “PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation.” In: *USENIX Security Symposium*. 321–336.

Küsters, R., T. Truderung, and A. Vogt. 2012. “A game-based definition of coercion resistance and its applications”. *Journal of Computer Security*. 20(6): 709–764.

Launchbury, J., I. S. Diatchki, T. DuBuisson, and A. Adams-Moran. 2012. “Efficient lookup-table protocol in secure multiparty computation”. In: *ACM SIGPLAN Notices*. Vol. 47. No. 9. ACM. 189–200.

Lee, J., J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. 2017a. “Hacking in Darkness: Return-oriented Programming Against Secure Enclaves”. In: *Proceedings of the 26th USENIX Conference on Security Symposium*. *SEC’17*. Vancouver, BC, Canada: USENIX Association. 523–539. ISBN: 978-1-931971-40-9. URL: http://dl.acm.org/citation.cfm?id=3241189.3241231.

Lee, S., M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. 2017b. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *Proceedings of the 26th USENIX Conference on Security Symposium*. *SEC’17*. Vancouver, BC, Canada: USENIX Association. 557–574. ISBN: 978-1-931971-40-9. URL: http://dl.acm.org/citation.cfm?id=3241189.3241233.

Li, M., S. Yu, N. Cao, and W. Lou. 2013. “Privacy-preserving distributed profile matching in proximity-based mobile social networks”. *IEEE Transactions on Wireless Communications*. 12(5): 2024–2033.

Lindell, Y. 2013. “Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries”. In: *Advances in Cryptology – CRYPTO 2013, Part II*. Ed. by R. Canetti and J. A. Garay. Vol. 8043. *Lecture Notes in Computer Science*. Springer, Heidelberg. 1–17. DOI: 10.1007/978-3-642-40084-1_1.

Lindell, Y. and B. Pinkas. 2007. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by M. Naor. Vol. 4515. *Lecture Notes in Computer Science*. Springer, Heidelberg. 52–78.

Lindell, Y. and B. Pinkas. 2009. "A Proof of Security of Yao's Protocol for Two-Party Computation". *Journal of Cryptology*. 22(2): 161–188.

Lindell, Y. and B. Pinkas. 2011. "Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer". In: *TCC 2011: 8th Theory of Cryptography Conference*. Ed. by Y. Ishai. Vol. 6597. *Lecture Notes in Computer Science*. Springer, Heidelberg. 329–346.

Lindell, Y., B. Pinkas, and N. P. Smart. 2008. "Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries". In: *SCN 08: 6th International Conference on Security in Communication Networks*. Ed. by R. Ostrovsky, R. D. Prisco, and I. Visconti. Vol. 5229. *Lecture Notes in Computer Science*. Springer, Heidelberg. 2–20.

Lindell, Y. and B. Riva. 2014. "Cut-and-Choose Yao-Based Secure Computation in the Online/Offline and Batch Settings". In: *Advances in Cryptology – CRYPTO 2014, Part II*. Ed. by J. A. Garay and R. Gennaro. Vol. 8617. *Lecture Notes in Computer Science*. Springer, Heidelberg. 476–494. DOI: 10.1007/978-3-662-44381-1_27.

Lindell, Y. and B. Riva. 2015. "Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 579–590.

Liu, J., M. Juuti, Y. Lu, and N. Asokan. 2017. "Oblivious Neural Network Predictions via MiniONN Transformations". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 619–631.

López-Alt, A., E. Tromer, and V. Vaikuntanathan. 2012. "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption". In: *44th Annual ACM Symposium on Theory of Computing*. ACM. 1219–1234.

Lu, S. and R. Ostrovsky. 2013. "How to Garble RAM Programs". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by T. Johansson and P. Q. Nguyen. Vol. 7881. *Lecture Notes in Computer Science*. Springer, Heidelberg. 719–734. DOI: 10.1007/978-3-642-38348-9_42.

Malkhi, D., N. Nisan, B. Pinkas, and Y. Sella. 2004. "Fairplay-Secure Two-Party Computation System". In: *USENIX Security Symposium*.

Marlinspike, M. 2017. "Technology preview: Private contact discovery for Signal". https://signal.org/blog/private-contact-discovery/.

Micali, S. and L. Reyzin. 2004. "Physically Observable Cryptography (Extended Abstract)". In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by M. Naor. Vol. 2951. *Lecture Notes in Computer Science*. Springer, Heidelberg. 278–296.

Mishra, P., R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. 2018. "Oblix: An Efficient Oblivious Search Index". In: *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.

Mohassel, P. and M. Franklin. 2006. "Efficiency Tradeoffs for Malicious Two-Party Computation". In: *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by M. Yung, Y. Dodis, A. Kiayias, and T. Malkin. Vol. 3958. *Lecture Notes in Computer Science*. Springer, Heidelberg. 458–473.

Mohassel, P. and B. Riva. 2013. "Garbled Circuits Checking Garbled Circuits: More Efficient and Secure Two-Party Computation". In: *Advances in Cryptology – CRYPTO 2013, Part II*. Ed. by R. Canetti and J. A. Garay. Vol. 8043. *Lecture Notes in Computer Science*. Springer, Heidelberg. 36–53. DOI: 10.1007/978-3-642-40084-1_3.

Mohassel, P., M. Rosulek, and Y. Zhang. 2015. "Fast and Secure Three-party Computation: The Garbled Circuit Approach". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 591–602.

Mohassel, P. and Y. Zhang. 2017. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 19–38.

Mood, B., D. Gupta, H. Carter, K. Butler, and P. Traynor. 2016. "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 112–127.

Mukherjee, P. and D. Wichs. 2016. "Two round multiparty computation via multi-key FHE". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt)*. Springer. 735–763.

Naccache, D. and J. Stern. 1998. "A New Public Key Cryptosystem Based on Higher Residues". In: *ACM CCS 98: 5th Conference on Computer and Communications Security*. ACM Press. 59–66.

Naor, M., B. Pinkas, and R. Sumner. 1999. "Privacy Preserving Auctions and Mechanism Design". In: *1st ACM Conference on Electronic Commerce*.

Naveed, M., S. Kamara, and C. V. Wright. 2015. "Inference Attacks on Property-Preserving Encrypted Databases". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 644–655.

Nielsen, J. B., P. S. Nordholt, C. Orlandi, and S. S. Burra. 2012. "A New Approach to Practical Active-Secure Two-Party Computation". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by R. Safavi-Naini and R. Canetti. Vol. 7417. *Lecture Notes in Computer Science*. Springer, Heidelberg. 681–700.

Nielsen, J. B. and C. Orlandi. 2009. "LEGO for Two-Party Secure Computation". In: *TCC 2009: 6th Theory of Cryptography Conference*. Ed. by O. Reingold. Vol. 5444. *Lecture Notes in Computer Science*. Springer, Heidelberg. 368–386.

Nikolaenko, V., S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. 2013a. "Privacy-preserving matrix factorization". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press. 801–812.

Nikolaenko, V., U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. 2013b. "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 334–348.

Ohrimenko, O., F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. 2016. "Oblivious Multi-Party Machine Learning on Trusted Processors." In: *USENIX Security Symposium*. 619–636.

Ostrovsky, R. and V. Shoup. 1997. "Private Information Storage". In: *ACM Symposium on Theory of Computing*.

Pagh, R. and F. F. Rodler. 2004. "Cuckoo hashing". *J. Algorithms*. 51(2): 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

Paillier, P. 1999. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology – EUROCRYPT'99*. Ed. by J. Stern. Vol. 1592. *Lecture Notes in Computer Science*. Springer, Heidelberg. 223–238.

Pappas, V., F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. Bellovin. 2014. "Blind Seer: A Scalable Private DBMS". In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 359–374. DOI: 10.1109/SP.2014.30.

Patra, A. and D. Ravi. 2018. "On the Exact Round Complexity of Secure Three-Party Computation". In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference*. Ed. by H. Shacham and A. Boldyreva. Vol. 10992. *Lecture Notes in Computer Science*. Springer. 425–458. DOI: 10.1007/978-3-319-96881-0.

Peikert, C., V. Vaikuntanathan, and B. Waters. 2008. "A Framework for Efficient and Composable Oblivious Transfer". In: *Advances in Cryptology – CRYPTO 2008*. Ed. by D. Wagner. Vol. 5157. *Lecture Notes in Computer Science*. Springer, Heidelberg. 554–571.

Pettai, M. and P. Laud. 2015. "Combining differential privacy and secure multiparty computation". In: *31st Annual Computer Security Applications Conference*. ACM. 421–430.

Pfitzmann, B. and M. Waidner. 2000. "Composition and Integrity Preservation of Secure Reactive Systems". In: *ACM CCS 00: 7th Conference on Computer and Communications Security*. Ed. by S. Jajodia and P. Samarati. ACM Press. 245–254.

Pinkas, B., T. Schneider, G. Segev, and M. Zohner. 2015. "Phasing: Private Set Intersection Using Permutation-based Hashing". In: *24th USENIX Security Symposium*. Ed. by J. Jung and T. Holz. USENIX Association. 515–530. URL: https://www.usenix.org/conference/usenixsecurity15.

Pinkas, B., T. Schneider, N. P. Smart, and S. C. Williams. 2009. "Secure Two-Party Computation Is Practical". In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by M. Matsui. Vol. 5912. *Lecture Notes in Computer Science*. Springer, Heidelberg. 250–267.

Pippenger, N. and M. J. Fischer. 1979. "Relations among Complexity Measures". *Journal of the ACM*. 26(2).

Poddar, R., T. Boelter, and R. A. Popa. 2016. "Arx: A strongly encrypted database system." *IACR Cryptology ePrint Archive*. 2016: 591.

Priebe, C., K. Vaswani, and M. Costa. 2018. "EnclaveDB: A Secure Database using SGX". In: *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.

Rastogi, A., M. A. Hammer, and M. Hicks. 2014. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations". In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 655–670. DOI: 10.1109/SP.2014.48.

Rivest, R. L., L. Adleman, and M. L. Dertouzos. 1978. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation*.

Rogaway, P. 1991. "The Round Complexity of Secure Protocols". Massachusetts Institute of Technology Ph.D. Thesis.

Sadeghi, A.-R., T. Schneider, and I. Wehrenberg. 2010. "Efficient Privacy-Preserving Face Recognition". In: *ICISC 09: 12th International Conference on Information Security and Cryptology*. Ed. by D. Lee and S. Hong. Vol. 5984. *Lecture Notes in Computer Science*. Springer, Heidelberg. 229–244.

Schneider, T. and M. Zohner. 2013. "GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits". In: *FC 2013: 17th International Conference on Financial Cryptography and Data Security*. Ed. by A.-R. Sadeghi. Vol. 7859. *Lecture Notes in Computer Science*. Springer, Heidelberg. 275–292. DOI: 10.1007/978-3-642-39884-1_23.

Shamir, A. 1979. "How to share a secret". *Communications of the ACM*. 22(11): 612–613.

Shan, Z., K. Ren, M. Blanton, and C. Wang. 2017. "Practical Secure Computation Outsourcing: A Survey". *ACM Computing Surveys*.

Shannon, C. E. 1937. "A symbolic analysis of relay and switching circuits". Massachusetts Institute of Technology Master's Thesis.

Shaon, F., M. Kantarcioglu, Z. Lin, and L. Khan. 2017. "SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 1211–1228.

shelat, a. and C.-H. Shen. 2011. "Two-Output Secure Computation with Malicious Adversaries". In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by K. G. Paterson. Vol. 6632. *Lecture Notes in Computer Science*. Springer, Heidelberg. 386–405.

shelat, a. and C.-H. Shen. 2013. "Fast two-party secure computation with minimal assumptions". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press. 523–534.

Shi, E., T.-H. H. Chan, E. Stefanov, and M. Li. 2011. "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by D. H. Lee and X. Wang. Vol. 7073. *Lecture Notes in Computer Science*. Springer, Heidelberg. 197–214.

Songhori, E. M., S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. 2015. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 411–428. DOI: 10.1109/SP.2015.32.

Stefanov, E., M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. 2013. "Path ORAM: an extremely simple oblivious RAM protocol". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press. 299–310.

Unbound Tech. 2018. "How to Control Your Own Keys (CYOK) in the Cloud". White Paper available from https://www.unboundtech.com.

Wagh, S., P. Cuff, and P. Mittal. 2018. "Differentially Private Oblivious RAM". *Proceedings on Privacy Enhancing Technologies*. 2018(4): 64–84.

Waksman, A. 1968. "A Permutation Network". *Journal of the ACM*. 15(1).

Wang, X. S., Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. 2014a. "SCORAM: Oblivious RAM for Secure Computation". In: *ACM CCS 14: 21st Conference on Computer and Communications Security*. Ed. by G.-J. Ahn, M. Yung, and N. Li. ACM Press. 191–202.

Wang, X. S., Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu. 2015a. "Efficient Genome-Wide, Privacy-Preserving Similar Patient Query based on Private Edit Distance". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 492–503.

Wang, X. S., K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. 2014b. "Oblivious Data Structures". In: *ACM CCS 14: 21st Conference on Computer and Communications Security*. Ed. by G.-J. Ahn, M. Yung, and N. Li. ACM Press. 215–226.

Wang, X., T.-H. H. Chan, and E. Shi. 2015b. "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by I. Ray, N. Li, and C. Kruegel: ACM Press. 850–861.

Wang, X., A. J. Malozemoff, and J. Katz. 2017a. "EMP-toolkit: Efficient MultiParty computation toolkit". https://github.com/emp-toolkit.

Wang, X., S. Ranellucci, and J. Katz. 2017b. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 21–37.

Wang, X., S. Ranellucci, and J. Katz. 2017c. "Global-Scale Secure Multiparty Computation". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 39–56.

Winternitz, R. S. 1984. "A Secure One-Way Hash Function Built from DES". In: *IEEE Symposium on Security and Privacy*. 88–88.

Wyden, R. 2017. "S.2169 — Student Right to Know Before You Go Act of 2017". https://www.congress.gov/bill/115th-congress/senate-bill/2169/.

Xu, Y., W. Cui, and M. Peinado. 2015. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 640–656. DOI: 10.1109/SP.2015.45.

Yao, A. C.-C. 1982. "Protocols for Secure Computations (Extended Abstract)". In: *23rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press. 160–164.

Zahur, S. and D. Evans. 2013. "Circuit Structures for Improving Efficiency of Security and Privacy Tools". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 493–507.

Zahur, S. and D. Evans. 2015. "Obliv-C: A Lightweight Compiler for Data-Oblivious Computation". Cryptology ePrint Archive, Report 2015/1153. http://oblivc.org.

Zahur, S., M. Rosulek, and D. Evans. 2015. "Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates". In: *Advances in Cryptology – EUROCRYPT 2015, Part II*. Ed. by E. Oswald and M. Fischlin. Vol. 9057. *Lecture Notes in Computer Science*. Springer, Heidelberg. 220–250. DOI: 10.1007/978-3-662-46803-6_8.

Zahur, S., X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. 2016. "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. 218–234. DOI: 10.1109/SP.2016.21.

Zhang, Y., A. Steele, and M. Blanton. 2013. "PICCO: a general-purpose compiler for private distributed computation". In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press. 813–826.

Zheng, W., A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. 2017. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform". In: *NSDI*. 283–298.

Zhu, R. and Y. Huang. 2017. "JIMU: Faster LEGO-Based Secure Computation Using Additive Homomorphic Hashes". In: *Advances in Cryptology – ASIACRYPT 2017, Part II*. Ed. by T. Takagi and T. Peyrin. Vol. 10625. *Lecture Notes in Computer Science*. Springer, Heidelberg. 529–572.

Zhu, R., Y. Huang, and D. Cassel. 2017. "Pool: Scalable On-Demand Secure Computation Service Against Malicious Adversaries". In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press. 245–257.

Zhu, R., Y. Huang, J. Katz, and A. Shelat. 2016. "The Cut-and-Choose Game and Its Application to Cryptographic Protocols." In: *USENIX Security Symposium*. 1085–1100.