Implementing Component-Based Garbled Circuits

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Alex Ledger

May 2016

Approved for the Division
(Mathematics)

Adam Groce

Acknowledgements

List of Notation

W_i	Wire i
W_i^j	Wire i's jth wire label. $j \in \{0, 1\}$
W_i^*	An unknown wire label of wire W_i
σ_i	The semantic value of wire W_i
C	A circuit
GC	A garbled circuit
c	A component-circuit
e_{C}	set of C's input wires
d_{C}	set of C's output wires
Δ	The delta value for the Free XOR technique
G_i	The ith gate
T_g	Garbled table for gate g
L_{ij}	Link label for mapping output wire W_i to input wire W_j.
x	Alice's input string
x_i	The ith bit of string x.
y	Bob's input string
z	he output string
$\gamma \leftarrow 3$	γ is assigned the value 3
$\gamma \leftharpoonup \{0,1\}^n$	γ is sampled uniformly at random from the set $\{0,1\}^n$.

Table of Contents

List of Tables

List of Figures

Abstract

Secure computation is a cryptographic method for securely computing a function between two parties while preserving the privacy of the parties' inputs. Some modern research into secure computation uses an offline/online scheme, where parties preprocess and exchange information in an offline phase prior to computing the function in an online phase. The offline/online setting has many advantages, namely that the online computation is very fast, but the setting is limiting, as the function that is being computed must be determined ahead of time.

This thesis is part of a larger project that created component-based garbled circuits to address some of the problems with the offline/online setting. Component-based garbled circuits observe that many real-world functions are composed of standard components such as arithmetic operations, matrix operations and other common operations. Component-based garbled circuits exchange many small, generic garbled components in an offline phase. Later in an online phase, the parties choose a function they wish to compute, and they stitch together the garbled components in order to create the function.

This thesis contributes two items to the larger project of component-based garbled circuits. The first item is *Single Communication Multiple Connections* (SCMC); SCMC is a technique that improves the online bandwidth of component-based garbled circuits. The second item is an implementation, CompGC, of component-based garbled circuits. CompGC is designed to be fast and secure; we run performance measurements on CompGC, and we find that component-based garbled circuits offer considerable savings in online time and online bandwidth over other implementations of garbled circuits.

Introduction

Suppose that millionaires Alice and Bob wish to determine who is wealthier, but they do not want to disclose their wealth to each other. Alice and Bob could tell a trusted third party their wealth, and then that trusted third party could tell Alice and Bob who is wealthier. However, that method has many disadvantages, one being that they need to trust a third party. What if there were a method that allows Alice and Bob to determine who is wealthier by only sending messages between themselves?

This classic problem is known as the *millionaire problem*, and it is a special case of *secure computation*. Secure computation gives multiple parties the ability to jointly to compute an arbitrary function, while keeping each party's input private from all other parties. In the millionaire problem, there are two parties, Alice and Bob, and they wish to compute the less than function.

We generalize secure computation to allow for an arbitrary number of parties to compute an arbitrary function amongst themselves. By generalizing secure computation, we open up many possible uses where secure computation can improve an existing process or allow for computations that are otherwise impossible or illegal. For example, we can run an election without a centralized body; we can perform an auction without using an auctioneer; and we can enable multiple hospitals with sensitive healthcare data to work together without sacrificing the privacy of their data. While secure computation works with arbitrarily many parties, this thesis focuses on the special case where there are two parties, called two-party computation (2PC).

The most common method for performing 2PC is *garbled circuits*. In a garbled circuit protocol, the parties transform their function f into a circuit. They then *garble* their circuit, such that with a bit more information they can *evaluate* the garbled circuit, recovering the answer to their function. Garbling the circuit involves obscuring the truth value of each wire inside the circuit. Once the truth values are obscured, the parties cannot learn any information as they process, or evaluate, the circuit: they only learn the answer to their function.

Garbled circuits have been heavily researched and optimized since their creation in the late 1980s. For most of that period, garbled circuit protocols were too slow and cumbersome for usage in the real world, but as of late, the protocols have achieved speeds comparable to that of loading a webpage.

In order to further increase the speed of garbled circuits, researchers split the garbled circuit protocol into two phases, an offline phase and an online phase. In the offline phase, before the two parties determine their inputs to the function, the parties exchange as much information as feasible. later, when the parties decide to compute the function, and they determine determine their inputs to the function, the parties engage in an online phase, where they exchange more information and finish the garbled circuit protocol, whereby they recover the output to their function. Using an offline/online protocol greatly reduces the latency of the computation - the time that it takes to go from determining inputs to acquiring an answer.

The offline/online system, while offering a number of benefits, is also limiting in that the function is determined ahead of time in the offline phase. Say that millionaires Alice and Bob originally agree to find out who is wealthier, so they exchange information for that function ahead of time in the offline phase, but then at the beginning of the online phase, Alice and Bob change their minds. Instead, they now wish to verify that they are indeed both millionaires, and verify that the other party is not going bankrupt. Since they exchanged information specific to the “who is wealthier" function during the offline phase, they are stuck. They cannot pivot and compute the new function without large computational sacrifices.

This thesis presents a new method called *component-based* garbled circuits to solve this problem. In the offline phase, instead of exchanging information corresponding to a single function, Alice and Bob exchange smaller pieces of information that can be stitched together to compute a class of functions. For example, instead of exchanging a garbled circuit that computes the "who is wealthier" function, Alice and Bob exchange many garbled circuits which are *components* or subparts of the less than function and other similar functions. Then, during the online phase, Alice and Bob select the function that they wish to compute from the class of available functions, and build their function by *chaining* pre-exchanged components. With component-based garbled circuits, Alice and Bob have more flexibility. They are no longer stuck using their original function - they can securely compute a host of functions at their whim.

This thesis is part of a larger project which created component-based garbled circuits; specifically, it contributes two items to the project. The first is a new method that improves the efficiency of component-based garbled circuits. The original component-based garbled circuits scheme requires that a ciphertext be communicated per wire chained - in other words, the communication scales linearly with the size of data. Our new method, called *Single Communication Multiple Connections* (SCMC), requires a single ciphertext be communicated per block of data instead of per wire. The communication requirement for chaining is now linear in the number of data objects but is constant in the size of the data object: chaining a 10 by 10 matrix has the same bandwidth requirement as chaining a $1,000$ by $1,000$ matrix. This allows for extremely fast computation of large statistical operations.

The second, and larger, contribution of this thesis is the implementation of component-based garbled circuits and SCMC into a program called CompGC. CompGC is a full-fledged secure computation system, where parties connect via TCP to agree on a function, exchange inputs, and securely compute the function. The program is fast, beating the best timings in the literature even for functions that do not benefit the most from SCMC.

For those that are not familiar with cryptography, it is best to read Chapters 1, 2 and 3 as they provide the necessary background. Chapter 1 introduces cryptographic primitives, and Chapter 2 discusses what it means for a secure computation protocol to be secure and introduces garbled circuits. Chapter 3 builds off of Chapter 2 by explaining a variety of improvements to garbled circuits. Chapter 4 and 5 focus on component-based garbled circuits. Chapter 4 explains the basic idea of component-based garbled circuits, and gives our improvement, SCMC. Chapter 5 discusses our implementation of component-based garbled circuits and SCMC, CompGC, and gives performance metrics of CompGC.

Chapter 1

Cryptographic Primitives

Secure computation is a cryptographic technique for securely computing a function between two parties. We split secure computation into two cases: cases where there are two parties involved, referred to as two-party computation (2PC), and cases where there are three or more parties, referred to as multiparty computation (MPC). This thesis focuses on 2PC protocols, but many of the methods are applicable to MPC as well.

2PC protocols are complex cryptographic protocols that rely on a number of cryptographic primitives. In order to understand 2PC, it is not crucial to understand how the cryptographic primitives work, but it is important to understand their inputs, outputs and security guarantees. This first chapter will give an overview of the cryptographic primitives used in 2PC protocols.

1.1 Introducing Cryptographic Security

The goal of this section is to explain cryptographic security, starting at an intuitive level and moving into more complex ideas. We do not present cryptographic security in a comprehensive fashion; rather, we explain cryptographic security with the goal of explaining 2PC protocols and their security. For more information on cryptographic security, we encourage the reader to peruse [11].

We define a few intuitive terms to get started. A *cryptographic scheme* is a series of instructions designed to perform a specific task. An *adversary* is an algorithm that tries to *break* the scheme. If an adversary *breaks* a scheme, then the adversary has learned information about inputs to the scheme that they shouldn't.

One aim of defining security in cryptography is to build a formal definition that matches real world needs and intuitions. A good starting place is to consider perfect security. A scheme is perfectly secure if no matter what the adversary does, they cannot break the scheme. Even if the adversary has unlimited computational power, in terms of time and space, an adversary cannot break a perfectly secure scheme.

However, perfect security is not the most useful way to think of security, because it forces schemes to be slow and communication-intensive. We relax the definition of security by requiring that the adversary run in polynomial time. This substantially reduces the power of the adversary, and it matches our intuition: we are really only concerned with what adversaries can reasonably achieve, as opposed to what is theoretically possible.

Because computers have improved drastically over the years, what was previously considered a reasonable adversary is not what is considered a reasonable adversary today. Modern computing advances have created easy access to faster computation, meaning that modern adversaries can solve harder problems than they could in previous years. As a concrete example, consider giving an adversary the following problem: find the factors of N. The average computer today can solve the problem for a larger N than the average computer could a decade ago.

The changing computational power highlights the importance of being able to scale how hard it is to break a cryptographic scheme. To this end, we introduce a *security parameter*, denoted as λ. A security parameter is a positive integer that represents how hard a scheme is to break; a larger security parameter should mean that the scheme is more difficult to break. More specifically, the security parameter is correlated to the input size of the problem underlying the cryptographic scheme. For example, if the underlying problem is factoring a large number N, then λ is the number of bits needed to express N in binary, that is, $\lambda = |N|$. As N and λ scale up, the factoring problem becomes more difficult and the scheme becomes harder to break.

Finally, we acknowledge that adversaries have access to some random values, hence we strengthen adversaries to be probabilistic algorithms. Probabilistic means that the algorithms have access to a string of uniform random bits, with the implication being that the algorithm is capable of guessing.

In order to reason about the security of cryptographic schemes, it is useful to think about breaking a scheme in terms of a probability. For example, we want to be able to say that the best adversary, that is best probabilistic polynomial-time algorithm, has some probability p of breaking the scheme. We note that p is nonzero, since the adversary can always guess and be right with some nonzero probability. To achieve a probability based formalism, we introduce a negligible function. Informally, a negligible function is smaller than the reciprocal of all polynomial functions.

Definition 1 A function $\mu : \mathbb{N} \to \mathbb{R}$ is negligible if for all positive polynomial $p(\cdot)$, there exists positive integer N_p such that for all $x > N_p$,

$$|\mu(x)| < \frac{1}{p(x)}. \tag{1.1}$$

[7] $\diamondsuit$

Examples of negligible functions include 2^{-n}, $2^{-\sqrt{2}}$ and $n^{-\log n}$.

To put a negligible function to use, say an adversary is attacking a cryptographic scheme that is equivalent to solving a problem P with input size λ and 2^λ possible answers. Moreover, say that P is known to be NP-hard such that there is no polynomial time algorithm to solve P. Then, the best that the adversary can do might be to guess the answer to P. Hence the probability that the adversary successfully answers P, or breaks the scheme, is

$$Pr[\text{A correctly answers } P] = 2^{-\lambda}$$

Since $2^{-\lambda}$ is a negligible function, we say that the adversary has a negligible probability of breaking the scheme.

In summary, we model an adversary as a probabilistic polynomial-time algorithm. This limits the computational power of the adversary to what is reasonably computable in reality. Moreover, we can scale the security of a scheme or problem by changing λ, the security parameter, where a higher security parameter makes the scheme more difficult to break.

1.2 Encryption

Encryption is the process of obfuscating a message, and then later un-obfuscating the message. Say Alice has a message that she wants to send to Bob, but somewhere between Alice and Bob sits Eve, who wants to learn about the message. An encryption scheme enables Alice to send her message to Bob with confidence that Eve cannot learn any information about the message.

An encryption scheme is composed of three algorithms: Enc, Enc^{-1} and Gen; formally, we say an encryption scheme is a tuple $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Enc}^{-1})$.[1] Enc the obfuscating algorithm, Enc^{-1} is the un-obfuscating algorithm and Gen generates a key. The key is extra information that Enc and Enc^{-1} use to obfuscate and un-obfuscate the message respectively. The key, denoted k, is a random[2] string of λ bits,

[1] We use Π here to denote the encryption scheme, because it is a protocol. Protocol starts with a p.

[2] The notion of randomness in cryptography is precisely defined, and in cases where λ is large, it

that is k is randomly sampled from the set $\{0,1\}^\lambda$ where λ is the security parameter of the encryption scheme. As per the discussion on security parameters, as λ increases and k grows in length, the encryption scheme should become harder to break.

Enc, the encryption algorithm, takes a message and the key as input and outputs an obfuscated message. Enc^{-1}, the decryption algorithm, takes the encrypted message and the key as input and outputs the original message. We refer to the original message as the plaintext or pt and the encrypted message as the ciphertext or ct.

$$\begin{aligned} \mathsf{Gen}(1^n) &\to k \\ \mathsf{Enc}_k(pt) &\to ct \\ \mathsf{Enc}_k^{-1}(ct) &\to pt \end{aligned} \tag{1.2}$$

We are not concerned with how encryption schemes are implemented or on what problems they rely; rather, we use encryption schemes as subroutines, so we are concerned with the security guarantees that they offer.

We define security using a thought experiment. In the thought experiment, the adversary has access to the encryption algorithm with the key hardcoded in. This means that the adversary can encrypt any message they want, and see how the message would encrypt. The goal of the adversary at this point in the thought experiment is to find a pattern or weakness in the encryption algorithm that they can exploit. The adversary eventually picks any two messages m_0 and m_1; the adversary shows us m_0 and m_1. We choose one of the messages[3], encrypt the message, and send the resulting ciphertext to the adversary. The adversary's goal now is to determine which message we encrypted. They still may use their encryption algorithm with the key hardcoded. Eventually the adversary must output either 0 or 1 indicating

is sufficient for k to be pseudorandom. Pseudorandomness is also precisely defined.

[3] We select the message uniformly at random.

that they think we selected m_0 or m_1 to encrypt, respectively. If the adversary picks correctly, then we say that the adversary wins; otherwise, we say that the adversary loses. We consider the encryption scheme to be secure if the probability that the adversary wins is $\frac{1}{2} + \mu(\lambda)$ where μ is a negligible function, i.e. the best the adversary can do is guess.

Definition 2 An encryption scheme is secure under a chosen-plaintext attack if for all probabilistic polynomial-time adversaries A, there exist a negligible function μ such that

$$Pr[E_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mu(n) \tag{1.3}$$

where E is the following experiment:

1. Generate key k by running $\mathsf{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given 1^n and oracle access to Enc_k, and outputs a pair of messages m_0 and m_1 of the same length.
3. A uniform bit $b \leftharpoondown \{0,1\}$ is sampled uniformly at random, and then ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$.[4]
4. $\mathcal{A}$ continues to have oracle access to Enc_k, and outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If at any point $\mathcal{A}$ encrypts m_0 or m_1 with their encryption oracle, the output is 0. 1 indicates that the adversary wins, and 0 indicates that the adversary loses.

$\diamondsuit$

It is useful for 2PC to create an encryption scheme that requires two keys to encryption and decrypt. An encryption scheme with two keys is called a *dual-key*

[4]Throughout this thesis, we use the notation $x \leftharpoondown X$ to mean that x is sampled uniformly at random from the set X. We use the arrow $\leftarrow$ to indicate that a value to being assigned to variable. For example, $x \leftarrow 3$, means that x is being assigned the value 3.

cipher (DKC) [3]. It is easy to instantiate a DKC if one has a secure single-key encryption scheme: let k_0 and k_1 be two keys and instantiate the DKC as follows:

$$\begin{aligned} \mathsf{EncDKC}_{k_0,k_1}(pt) &= \mathsf{Enc}_{k_1}(\mathsf{Enc}_{k_0}(pt)) \\ \mathsf{EncDKC}^{-1}_{k_0,k_1}(ct) &= \mathsf{Enc}^{-1}_{k_0}(\mathsf{Enc}^{-1}_{k_1}(ct)) \end{aligned} \tag{1.4}$$

If the encryption scheme used to create the DKC is secure, then it is easy to see that the DKC is also secure. We are formally considering the statement: if Enc is secure then the constructed DKC is secure.

Consider the contrapositive: if the constructed DKC is insecure then Enc is insecure. We sketch a proof of this claim. Suppose that we have an adversary $\mathcal{A}$ competing in the Enc security experiment, and $\mathcal{A}$ can break the DKC security experiment - this assumption is made since we assume that the DKC is insecure. Since $\mathcal{A}$ can beat the DKC experiment, $\mathcal{A}$ finds two messages, m_0 and m_1, such that they can determine b given $\mathsf{Enc}_k(\mathsf{Enc}_k(m_b))$.

In the Enc experiment, $\mathcal{A}$ sets $m'_0 \leftarrow \mathsf{Enc}_k(m_0)$ and $m'_1 \leftarrow \mathsf{Enc}_k(m_1)$ using their encryption oracle. The adversary submits m'_0 and m'_1 as their messages, and they recieve back $\mathsf{Enc}_k(m'_b)$. Since $\mathsf{Enc}_k(m'_b)$ is actually $\mathsf{Enc}_k(\mathsf{Enc}_k(m_b))$, the $\mathcal{A}$ can determine b. The adversary can do this because they chose m_0 and m_1 such that they could beat the DKC security experiment. Finally, $\mathcal{A}$ outputs b, which will be correct with probabilty greater than $\frac{1}{2} + \mu(\lambda)$ where μ is a negligible function. Therefore the adversary can break the Enc experiment if they can break the DKC experiment, and the contrapositive of this statement tells us that if the Enc scheme is secure, then the DKC scheme is also secure.

1.3 Computational Indistinguishability

This section introduces the idea of computational indistinguishability. We do not use computational indistinguishability immediately, but it will important later for defining security of a 2PC protocol.

Informally, two probability distributions are indistinguishable if no probabilistic polynomial-time algorithm can tell them apart. The thought experiment is like this: an algorithm knows that there are two distributions. It samples from one of the distributions. If the algorithm correctly determines which distribution it was given, then it wins, and otherwise the algorithm loses. The algorithm, since it must run in polynomial time, can only sample a polynomial number of values from the distributions.

Formally, computational indistinguishability is:

Definition 3 Let $\mathcal{X} = \{X_n\}_{n\in\mathbb{N}}$ and $\mathcal{Y} = \{Y_n\}_{n\in\mathbb{N}}$ be distribution ensembles. $\mathcal{X}$ and $\mathcal{Y}$ are computationally indistinguishable, denoted $\mathcal{X} \approx_C \mathcal{Y}$, if for all probabilistic polynomial-time algorithms D, there exists a negligible function μ such that:

$$|Pr_{x\leftarrow X_n}[D(1^n, x) = 1] - Pr_{y\leftarrow Y_n}[D(1^n, y) = 1]| < \mu(n) \tag{1.5}$$

[11]. $\diamondsuit$

We break down the definition. The unary input 1^n tells the algorithm D to run in polynomial time in n. The probability distributions X_n and Y_n are restricted by n, which in this context is the security parameter. The phrases $x \leftarrow X_n$ and $y \leftarrow Y_n$ mean that the probability is taken over samples from the distributions.

1.4 Boolean Circuit

A function in a 2PC protocol is represented as a boolean circuit. A boolean circuit takes as input $x \in \{0,1\}^n$, performs a series of small operations on the inputs, and outputs $y \in \{0,1\}^m$. You may have encountered circuits and logical operators in another context, where the inputs and outputs were True and False. For our usage, True corresponds to the value 1, and False corresponds to the value 0.

The small operations done inside of a circuit are performed by a *gate*. A gate is composed of three wires: two input wires and one output wire, where a *wire* can have a value either 0 or 1. A gate performs a simple operation on the two inputs, resulting in a single output bit. Table 1.4 gives the mapping of an XOR gate.

x	y	xor(x,y)
1	1	0
1	0	1
0	1	1
0	0	0

Table 1.1: Logical table of an XOR gate.

A circuit is a combination of gates that are strung together. It turns out that circuits are quite powerful: in fact, a circuit composed only of AND gates, XOR gates and NOT gates can compute any function or algorithm [7]. In other words, if there is some algorithm that can do it, then there is some circuit that can do it as well.

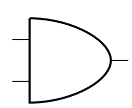

Figure 1.1: An AND gate.

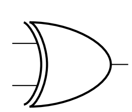

Figure 1.2: An XOR gate.

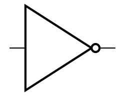

Figure 1.3: A NOT gate.

x	y	$x < y$
0	0	0
0	1	1
1	0	0
1	1	0

Table 1.2: Logical table of the less than function. See equation 2.1.

1.5 Oblivious Transfer

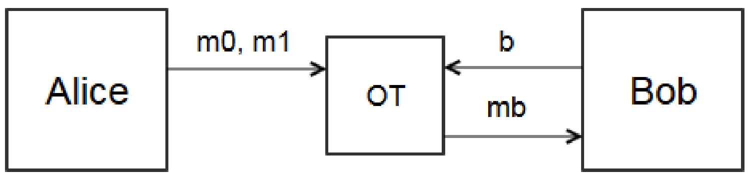

Figure 1.4: The high level idea of oblivious transfer. Image from [21].

Oblivious Transfer (OT) is a simple, useful protocol that underlies many more complicated crypto-systems [6, 23].

Figure 1.4 outlines the high level idea of oblivious transfer. The box labeled OT in figure 1.4 may be thought of as a trusted post office. Alice potentially sends two messages m_0 and m_1 to Bob, but instead of sending the messages to Bob, she sends the messages to the post office. Bob, without seeing either m_0 or m_1, knows that he wants m_b where $b \in \{0, 1\}$, so he notifies the post office that he wants message b. With this information, the post office gives Bob m_b. We want two secure properties to hold: (1) Alice does not know whether Bob received m_0 or m_1 and (2) Bob does not learn any information about m_{1-b}, the message that he did not receive.

We will not focus on the internal operations of oblivious transfer; for our purposes, it is a secure black box. When we use oblivious transfer in the next chapter, the two messages that Alice potentially sends will be encryption keys, and Bob will select a key that corresponds to his input - a value that he doesn't want Alice to know. In Chapter 3 we will discuss improvements to oblivious transfer, since oblivious transfer, despite its simple appearance, is computationally expensive.

Chapter 2

Classic 2PC

Using the tools of Chapter 1, this chapter presents a classic method of performing two-party computation: garbled circuits. There are other methods of performing secure computation, but this thesis focuses on garbled circuits. We begin this chapter by motivating and describing desirable properties of 2PC protocols, culminating in a definition of security. We then describe garbled circuits and discuss their security.

2.1 2PC Security Motivation

Think back to Alice and Bob from the introduction. Alice and Bob are millionaires who wish to determine who is wealthier without disclosing how much wealth they have. More formally, Alice has input x and Bob has input y (x and y are integers corresponding to the wealth of each party), and they wish to compute the less than function f, such that

$$f(x,y) = \begin{cases} 1 & \text{if } x < y; \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

We call the overarching interaction between Alice and Bob protocol Π, and Π consists of all messages exchanged and computations performed. Based on the setup of the problem, we can list a few properties that Alice and Bob wish Π to have.

Privacy Parties only learn their output. Any information learned by a single party during the execution of Π must be deduced from the output. For example, if Alice learns that she had more money after computing f, then she learns that $y < x$; this information about y is deducible from the output, and therefore it is reasonable. It would be unreasonable if Alice learns that $1,000,000 < y < 2,000,000$, as that information is not deducible from $f(x, y)$.

Correctness Each party receives the correct output. In the case of Alice and Bob, this simply means that they learn correctly who has more money.

One possible method for constructing a definition of security would be to list a number of properties a secure protocol must have; however, this approach is unsatisfactory for a number of reasons.

One reason is that is that if we use properties, we may miss certain security properties that are only relevant in certain cases. There are many applications of 2PC, and in some cases, an important security property may be lurking in the shadows. Ideally, a good definition of 2PC works for all applications, hence capturing all desirable properties. A second reason that the property based definition is unsatisfactory is that the definition should be simple. If the definition is simple, then it should be clear that *all* possible attacks against the protocol are prevented by the definition. With a definition based on properties, it becomes the burden of the prover of security to show that all relevant properties are covered [14].

We must also think about the aims of each party involved in the protocol. Can we trust that the parties are going to obey the protocol? Are the parties going to try to cheat? These considerations are called the *security setting*. There are two primary security settings: the semi-honest setting and the malicious setting.

The work presented in this thesis uses the semi-honest setting. In the semi-honest setting, we assume that each party obeys the protocol but tries to learn as much as possible from the information they are given. This means that parties do not lie about their information, they do not abort, do not send out of order, do not withhold messages or deviate in any way from what is specified in the protocol. In contrast, the malicious setting considers that each party is liable to lie and cheat; parties can take any action to learn more information.

The malicious setting is often more realistic. Parties that are involved in cryptographic protocols are liable to lie and cheat, for why else would they even be engaged in the cryptographic protocol in the first place? There are two main reasons why the semi-honest setting is useful. The first is that many protocols can be constructed for the semi-honest setting, and then improved to function in the malicious setting. There is a strong history of this occurring with protocols. It is simply easier to think through and create protocols for the semi-honest setting; at the very least, it is a valuable starting point for building complex crypto-systems. In the case of 2PC, there exist malicious protocols, and in fact, the primary 2PC protocol that this thesis uses, garbled circuits, can be improved to be malicious secure without too much difficulty.

The second reason that the semi-honest setting is useful is because it does have real world use cases. There are some scenarios where parties want to compute a function amongst themselves, and trust each other to act semi-honestly. One example is hospitals sharing medical data. Hospitals are legally, and arguably ethically, restricted from sharing medical data, but this data can have great value especially when aggregated with datasets from other hospitals. 2PC offers hospitals the means to "share" their data, perform statistics and other operations on it, while keeping the data entirely private. Other examples where semi-honesty is sufficient include mutually trusting companies and government agencies.

2.2 2PC Security Definition

In this section we discuss the definition of 2PC security at a high level, and then give the formal definition. The definition of security for 2PC protocols is the most complicated cryptographic theory that we have encountered thus far.

Recall our setup: Alice and Bob are semi-honest parties with inputs x and y respectively who wish to compute the function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^m \times \{0,1\}^m$. The protocol Π is a 2PC protocol, that is, Π is a series of instructions that Alice and Bob follow which enables them to compute f. To think about the security of Π, we imagine an ideal world where the protocol is computed securely, and compare the ideal world to the real world where Π is executed.

In the ideal world, we imagine that there is a third, trusted and honest party Carlo. Instead of Alice and Bob communicating amongst each other, Alice and Bob send their inputs to Carlo. Carlo computes $f(x, y)$ himself and sends the output to Alice and Bob. The only information that Alice and Bob have in the ideal world is their individual inputs and the output.

Informally, we say that Π is secure if Alice and Bob learn *essentially the same* information executing Π in the real world as they do computing f in the ideal world with Carlo. If either Alice or Bob manage to learn more information in the real world, then the protocol Π is leaking information that cannot be deduced from their individual inputs and the output of f. This idea of security is formally achieved using the concept of computational indistinguishability presented in Chapter 1. Recall that computational indistinguishability is the idea that two probability distributions are essentially the same - no polynomial time algorithm can distinguish them. To use computational indistinguishability, we think of the information that Alice and Bob learn in the real and ideal world as probability distributions. The idea may seem fuzzy now, but it will become clearer as the probability distributions are explained.

To construct the probability distribution of Alice and Bob's information in the ideal world, we introduce *simulators* S_A and S_B. Simulators S_A and S_B are probabilistic polynomial-time algorithms who are essentially adversaries, like the adversary in the definition of encryption from Chapter 1, that specifically attack the ideal world. Simulator S_A takes as input x, Alice's input, and $f(x, y)$, the output of the function because that is the information that Alice has access to in the ideal world. Likewise, S_B takes input y and $f(x, y)$, since that is the is information that Bob has access to in the ideal world.

To create a probability distribution, we consider what S_A does over all possible inputs: the distribution of the possible outputs is given by $\{S_A(x, f(x, y)\}_{x \in \{0,1\}^*}$. Let us break this distribution down: S_A is a fixed single algorithm, x is Alice's input and $f(x, y)$ is the output of the function. The set is indexed by all possible x, so all possible inputs that Alice could have. In summary, $\{S_A(x, f(x, y)\}_{x \in \{0,1\}^*}$ represents the possible information that an algorithm could deduce from all possible x and $f(x, y)$. Similarly, we think of $\{S_B(y, f(x, y)\}_{y \in \{0,1\}^*}$ for Bob's input.

In the real world, we need to consider what information Alice and Bob have at their disposal. Recall that Alice and Bob are semi-honest, which means that Alice and Bob follow all instructions of Π, but they use any information they receive along the way. More precisely, Alice and Bob obey the protocol, but they also maintain a record of all messages sent and received. We call Alice's record of communications Alice's *view*, denoted $\mathsf{view}_A(x, y)$, which depends on inputs x and y. We can now create the probability distribution for Alice: $\{\mathsf{view}_A(x, y)\}_{x,y \in \{0,1\}^*}$. This distribution represents Alice's information from the exchanged messages indexed over all possible inputs x and y. Likewise, we call Bob's record of intermediate computations Bob's view, $\mathsf{view}_B(x, y)$, and his probability distribution of intermediate computations is $\{\mathsf{view}_B(x, y)\}_{x,y \in \{0,1\}^*}$

In summary, if Π in the real world is the same as the protocol Alice, Bob and Carlo run in the ideal world, then the simulator S_A and S_B should only be able to learn what can be learned from the intermediate computations. That is, the probability distributions $\{S_A(x, f(x,y)\}_{x\in\{0,1\}^*}$ and $\{\mathsf{view}_A(x,y)\}_{x,y\in\{0,1\}^*}\}$ should be essentially the same, i.e., computationally indistinguishable.

With this intuition in mind, we give Goldreich's definition of 2PC security from his textbook *Foundations of Cryptography Volume II* [7].

Definition 4 Let $f = (f_1, f_2)$ be a probabilistic, polynomial time functionality where Alice and Bob compute $f_1, f_2 : \{0,1\}^n \to \{0,1\}^m$ respectively. Let Π be a two party protocol for computing f. Define $\mathsf{view}_i^\Pi(n, x, y)$ (for $i \in \{1, 2\}$) as the view of the ith party on input (x, y) and security parameter n. $\mathsf{view}_i^\Pi(n, x, y)$ equals the tuple $(1^n, x, r^i, m_1^i, \ldots, m_t^i)$, where r^i is the contents of the ith party's internal random tape, and m_j^i is the jth message that the ith party received. Define $output_i^\Pi(n, x, y)$ as the output of the ith party on input (x, y) and security parameter n. Also denote $\mathsf{output}^\Pi(n, x, y) = (\mathsf{output}_1^\Pi(n, x, y), \mathsf{output}_2^\Pi(n, x, y))$. Note that view_i^Π and output_i^Π are random variables whose probabilities are taken over the random tapes of the two parties.

We say that Π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial time algorithms S_1 and S_2 such that for all $x, y \in \{0,1\}^*$, where $|x| = |y|$, the following hold:

$$\{(S_1(x, f_1(x,y), f(x,y)))\}_{x,y} \approx_C \{(\mathsf{view}_1^\Pi(x,y), \mathsf{output}^\Pi(x,y))\}_{x,y} \tag{2.2}$$

$$\{(S_2(x, f_2(x,y), f(x,y)))\}_{x,y} \approx_C \{(\mathsf{view}_2^\Pi(x,y), \mathsf{output}^\Pi(x,y))\}_{x,y} \tag{2.3}$$

$\diamondsuit$

The definition requires that $|x| = |y|$; however, this constraint can be overcome by padding the shorter input.

A definition of 2PC security with malicious parties is substantially more complex. For more information on a malicious security definition, we refer the reader to [14].

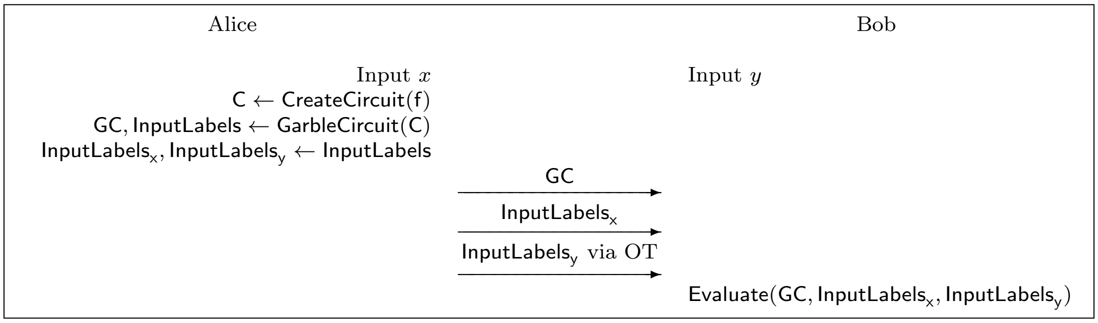

Figure 2.1: A high level overview of the garbled circuit protocol.

2.3 Yao's Garbled Circuit

We now discuss a popular 2PC scheme called garbled circuits [28]. In garbled circuits, one party, Alice, designs a circuit that computes f. Alice encrypts, or *garbles*, the circuit and sends the encrypted circuit to Bob, along with some values corresponding to her and Bob's inputs. Bob decrypts the circuit, acquiring the value $f(x, y)$.

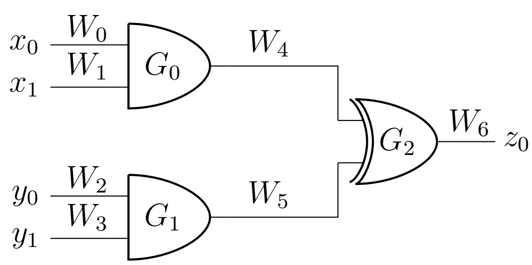

Figure 2.2: A simple boolean circuit with labeled inputs, output, gates and wires.

We now walk through how Alice garbles a circuit. She starts with a boolean circuit, like the one shown in figure 2.2. In the figure, the gates are ordered from 0 to 2, in order from nearest to the inputs to farthest from the inputs. Alice begins by assigning two wire labels to each wire - a wire is a line connecting inputs and gates in the figure. Because Alice is working with a boolean circuit, each wire can semantically represent either a 0 or 1. For a given wire W_i, the zeroth wire label W_i^0 represents 0, and the first wire label W_i^1 represents 1. A wire label is a ciphertext, the output of the encryption algorithm.[1] Alice assigns the wire labels by randomly

[1] A common encryption algorithm used now is AES-128, so the wire labels are 128 bit strings.

sampling from $\{0,1\}^\lambda$, where λ is the size of the output of the encryption algorithm.

In general, we use W_i to represent the ith wire, W_i^j to represent the ith wire's jth label, and later we use W_i^* to represent one of wire i's wire labels without specifying whether it is wire label 0 or wire label 1.

After assigning wire labels, Alice garbles gate G_0, the gate nearest to the inputs.[2] Alice garbles gate G_0 by creating a garbled table T_{G_0}. Gate G_0 is an AND gate, so the structure of the table resembles the logical table for an AND gate. The table has four columns. The first two columns are the input wire labels of the input wires W_0 and W_1. The third column is the wire labels for the output wire W_4

The wire label placed in the third column is based on logical operation of the AND gate. For example, the first three rows of the third column have the wire label associated with 0, since AND outputs 0 if any inputs are 0. The fourth column is the dual-key cipher encryption[3] of the value in the third column, using the values of the first two columns as keys. The garbled table for G_0 is shown in table 2.1.

W_0	W_1	W_4	Encryption
W_0^0	W_1^0	W_4^0	$\mathsf{Enc}_{W_0^0,W_1^0}(W_4^0)$
W_0^1	W_1^0	W_4^0	$\mathsf{Enc}_{W_0^1,W_1^0}(W_4^0)$
W_0^0	W_1^1	W_4^0	$\mathsf{Enc}_{W_0^0,W_1^1}(W_4^0)$
W_0^1	W_1^1	W_4^1	$\mathsf{Enc}_{W_0^1,W_1^1}(W_4^1)$

Table 2.1: A garbled table for an AND gate with input wires W_0 and W_1 and output wire W_4.

Alice creates garbled tables for each remaining gate; in this case, G_1 and G_2. She then sends the fourth column of all garbled tables, the encryption of the respective out-wires, to Bob. Now if Bob has a label for each input wire, then Bob can acquire one of the labels of the output wire. Say a gate has input wires W_i and W_j and output wire W_k. Bob can acquire a wire label of W_k (i.e., W_k^0 or W_k^1) if he has one wire label of W_i (i.e. W_i^0 or W_i^1) and one wire label of W_j (i.e. W_j^0 or W_j^1).

[2]Multiple gates at some point could be equidistant from the input. In these cases, the ordering of gates does not matter.

[3]See Chapter 1 for information on dual-key ciphers.

Alice cannot send the fourth column of the garbled table to Bob as it is. Since the table is ordered, Bob can interpret the semantic value of wire W_k based on which row successfully decrypts. The fix is simple: Alice randomly permutes the rows of the table prior to sending.

In order to decrypt each gate, Bob needs to first acquire the wire labels of the input wires, W_0, W_1, W_2 and W_3. Recall that Alice's input to the 2PC protocol are $x = x_0x_1$ where $x_1, x_0 \in \{0,1\}$. Alice communicates her input to Bob, without revealing values of x_i, by sending wire labels $W_0^{x_0}$ and $W_1^{x_1}$. Bob does not know the values of x_0 or x_1, because he is simply receiving two ciphertexts, and does not know which value the ciphertexts represent. The only information that Bob has is the fourth column of the garbled tables, and that does not provide any information about the semantic value of the wire labels.

Recall that Bob's input to the 2PC protocol is $y = y_0y_1$ where $y_1, y_0 \in \{0,1\}$. Alice now wants to send Bob $W_2^{y_0}$ and $W_3^{y_1}$, but she does not know and cannot know (for the sake of security) y_0 and y_1. Alice and Bob can achieve this by using Oblivious Transfer, as described in Chapter 1. As an example, we look at wire W_2. Alice potentially sends W_2^0 and W_2^1 to Bob. Alice only wants Bob to acquire one of the values, because otherwise he can decrypt multiple rows garbled table. Bob wants to receive $W_2^{y_0}$, as that wire label corresponds to his input. In terms of the post office metaphor from Chapter 1, Alice gives W_2^0 and W_2^1 to the post office, and Bob notifies the post office that he wants message y_0. The post office gives $W_2^{y_0}$ to Bob, but does not tell Bob anything about $W_2^{1-y_0}$ and does not tell Alice which message Bob requested. Alice and Bob also perform OT on wire W_3, such that Bob acquires wire label $W_3^{y_1}$.

Alice is finished garbling the circuit, and finished communicating information to Bob. Bob has everything he needs to evaluate, or decrypt, the circuit. Bob starts with gate G_0, for which he has the fourth column of T_{G_0}, a wire label for wire W_0, which

I denote W_0^* since Bob does not know which wire label it is, and W_1^* accordingly. Bob starts with the first row of T_{G_0} and tries decrypting the value using W_0^* and W_1^*. Formally, Bob tries $\mathsf{Enc}^{-1}_{W_0^*,W_1^*}(T_{G_0}[0])$ where $T_{G_0}[0]$ represents the value in the zeroth row of T_{G_0}. Bob tries decrypting the values in all four rows of the garbled table T_{G_0}, but only one should work, since the other decryptions using the incorrect keys.

For Bob to recognize that encryption is failing, we add an additional property to the encryption scheme: the output of the decryption algorithm should indicate whether the decryption was valid. The decryption algorithm outputs a single additional bit, where 1 indicates that decryption was successful and 0 indicates that decryption failed, i.e., that keys or ciphertext were invalid. It is noteworthy that such a property is common to encryption schemes, and can be added to any existing encryption if necessary. With this property, as Bob tries decrypting all four rows of the garbled table, only one should decrypt correctly. Because of the way that Alice constructed the garbled table, Bob knows that this correctly decrypted value is one of the wire labels of W_4, the output wire of gate G_0. Bob then assigns this decrypted value to W_4^*, and uses the value as the input wire when decrypting gate G_2.

Bob repeats this same process for gate G_1 and for gate G_2. For gate G_2, Bob uses wire labels W_5^* and W_6^* which he acquires by evaluating gates G_0 and G_1. Bob notifies Alice after acquiring W_6^*. Alice sends him values W_6^0 and W_6^1. If $W_6^* = W_6^0$, then the function output 0 and Bob notifies Alice that the output was 0. Otherwise if $W_6^* = W_6^1$, then the function outputs 1 and Bob notifies Alice that the output was 1. Alice and Bob have now securely computed the function.

2.3.1 Security of Garbled Circuits

The definition of security of a 2PC protocol Π given early in Chapter 2 asked us to compare the execution of Π in the real world to a protocol running in an ideal world with a trusted third party. To think about the security of garbled circuits, we think about the information that Alice and Bob acquire throughout the protocol.

The only information that Alice receives from Bob during the execution of garbled circuits is in the OT stage. Thereby the security on Alice's side is dependent on OT, which we assume to be secure. Hence, we can confidently say that Alice does not learn anything that undermines security during an execution of garbled circuits.

The security of Bob is more complicated. Bob learns two sets of information: input labels and the fourth column of the garbled table for each gate. The input labels are acquired in two ways: some are sent naively by Alice and some are acquired via OT. We can be confident that Bob doesn't learn any extra information in the process of receiving the labels, since we are confident that OT does not reveal information, and other labels are coldly sent by Alice. Moreover, knowing input labels associated with Alice's input doesn't give Bob any information, since he cannot tell if the label is associated with 0 or 1.

The fourth column of the garbled table by itself does not reveal any information to Bob, as it is simply the encryption of things. Bob has the keys to decrypt some of the values in the garbled table. We need to be sure that Bob only has the keys to decrypt the single, intended row of the table. Bob can only decrypt a row of the table if he has both of the labels used as keys. For gates that are connected to input wires, Bob only has a single label for each wire, therefore he must only be able to decrypt a single row. For subsequent gates, Bob will only ever have a single label for each wire, meaning he can only decrypt a single row.

For Bob to learn extra information, he needs to acquire two wire labels for a single wire. If he can acquire two labels, then he can decrypt the circuit using that wire label, and make deductions about Alice's input. Based on the information that Bob has, there is no way he can access two wire labels for a single wire. This is the intuition behind the security of garbled circuits. Of course without a formal proof using computational indistinguishability, we cannot be confident that garbled circuits are secure, as there may be an attack that eludes us. However, a formal proof of garbled circuits is long and beyond the scope of this paper.

2.3.2 Notes about Complexity

There are three things to think about when considering the complexity of garbled circuits. The first is the amount of information that needs to be communicated per gate. Garbled circuits require 4 ciphertexts, that is 4λ bits. On top of this communication is Alice sending her input labels, and the communication required to complete OT for Bob's input labels. OT, if used naively, is a significant contributor to overall bandwidth.

The second thing to consider is the amount of computation that Alice performs. Alice performs 4 encryptions with a dual-key cipher for each gate. The third cost to consider is the amount of computation that Bob performs. Bob performs 4 decryptions with the dual-key cipher for each gate.

These constraints are all important, but in practice the biggest bottlenecks are the communication per gate and Bob's computation, which are correlated as we will see in Chapter 3. A circuit that computes AES requires approximately 35,000 gates. If 4 ciphertexts per gate need to communicated, and AES-128 is the encryption scheme, then $4 * 128 * 35,000$ bits $= 2.24$ megabytes: a huge amount of communication for just encryption!

Chapter 3

Improving MPC

The aim of this chapter is to outline new techniques that improve the performance of garbled circuits and discuss their costs and benefits.

We consider three metrics to understand the costs benefits of the new techniques: size of the garbled table, garbler-side computation and evaluator-side computation.[1] In the classic scheme of garbled circuits, the garbler and evaluator communicate wire labels of input wires and the fourth column of the garbled table for each gate. Communicating the input labels is an unavoidable cost; this information needs to be exchanged. In particular, many input labels are communicated via oblivious transfer which in its naive form is computational expensive. We discuss improvements to oblivious transfer at the end of this chapter.

New techniques also reduce the size of the garbled table, the other information exchanged between the garbler and evaluator.[2] In the classical garbled circuit scheme, the evaluator sends all four rows of the garble table to the evaluator. New techniques, wherein the garbler cleverly chooses wire labels, reduce the number of rows of the garbled table that are communicated. Reducing the size of the garbled table is im-

[1] In this chapter, the term garbled table is used interchangeably with the fourth column of the garbled table.

[2] In Chapter 2, Alice was the garbler and Bob was the evaluator. For clarity in Chapter 3 and beyond, the term *garbler* is used to allude to the person who garbles the circuit, and the term *evaluator* is used to describe the person who evaluates, or decrypts, the garbled circuit.

portant, because it is a measure of the bandwidth requirement of the scheme. Most communication is spent exchanging the garbled tables, and the number of garbled tables scales with the size of the circuit - that is, a larger circuit means more gates means more garbled tables. Hence, any reduction is the size of the garbled circuit translates into a notable reduction in bandwidth. It turns out that bandwidth is the most important factor in assessing the performance of a garbled circuit scheme because communication over the internet is slower than local computation.

We also consider garbler-side computation, which is the number of encryptions that the garbler performs in preparing a garbled table. In the classical garbled circuit scheme, the garbler performs four encryptions; however, new techniques do not substantially reduce this number. Typically, the new techniques increase or alter the computation that the garbler performs.

Finally, we consider evaluator-side computation, which is the number of decryptions the evaluator performs in evaluating a gate - that is, finding the output wire label. In the classical garbled circuit scheme, the evaluator trial decrypts each row until one of the decryptions is successful. Hence, on average, the evaluator performs 2.5 decryptions. New techniques reduce the evaluator-side computation to require a single decryption.

Table 3.1 shows an overview of the cost of all improvements made to garble circuits. The table is split into three sections: size, eval cost and garble cost, corresponding to thee three metrics mentioned above. Size is the number of rows in the garbled table. Garble cost is the number of encryptions that the garbler performs per gate, and eval cost is the number of decryptions that the evaluator performs per gate. Because many of the techniques have different effects on XOR gates and AND gates, each section is divided into two columns showing the metrics for AND gates and XOR gates separately.

The goal of this chapter is primarily to explain each row in this table. We will

Garbled Circuit Improvement	**Table Size (xλ)**		**Garble Cost**		**Eval Cost**	
	XOR	**AND**	**XOR**	**AND**	**XOR**	**AND**
Classical	4	4	4	4	4	4
Point and Permute	4	4	4	4	1	1
GRR3	3	3	4	4	1	1
Free XOR	0	3	0	4	0	1
GRR2	2	2	4	4	1	1
FleXOR	{0,1,2}	2	{0,2,4}	4	{0,1,2}	1
Half Gates	2	0	2	0	0	2

Table 3.1: Summary of Garbled Circuit Improvements. GRR3 stands for garbled row reduction three and GRR2 stands for garbled row reduction two. The values shown for each improvement include benefits from point and permute and other compatible improvements: Free XOR uses GRR3; FleXOR uses GRR2 and Free XOR; and Half Gates use FreeXOR. This table is adapted from [29].

understand the new techniques, and their associated computational and bandwidth cost. We start with the earliest technique and move forward chronologically.[3]

3.1 Point and Permute

The *point and permute* technique speeds up the evaluator's computation of the garbled table by removing the need to trial decrypt the ciphertexts; instead, the garbler subtly communicates which row of the garbled table to decrypt, and the evaluator only decrypts that ciphertext [17].

In the classic garbled circuit scheme, the garbled table is randomly permuted - that is, the garbler randomly reorders the rows of the garbled table before sending the garbled table to the evaluator.[4] Upon receiving the garbled table, the evaluator trial decrypts each row of the garbled table until a decryption succeeds.[5]

Point and permute enables the evaluator to bypass the trial decryption step, so

[3]For further information and exposition, we encourage the reader to investigate [24].
[4]This is required for security. See Chapter 2 for more information.
[5]Recall that the decryption algorithm outputs a single bit indicating whether or not the decryption was successful. See Chapter 1 for more information.

that they decrypt the correct wire label the first time. In point and permute, the garbler randomly assigns a select bit 0 or 1 to each wire label of the gate's input wires, where wire labels of the same wire have opposite bits. More formally, the garble randomly samples b_i from $\{0, 1\}$, and then gives wire label W_i^0 select bit b_i and gives W_i^1 select bit $1 - b_i$. The garbler permutes the garbled table based on the select bits, and appends the select bits to each wire label. When the evaluator evaluates the gate, they use the select bits appended to each of the two input wire labels to determine which row to decrypt. For example, if the select bits on the end of the input wire labels are 1 and 0, then the evaluator decrypts the third row.

Tables 3.1 show an example of point and permute. W_i, W_j and W_k are wires, where W_i^0 and W_i^1 are the zeroth and first wire label of wire W_i respectively. The left table shows the select bits of the input wires W_i and W_j. The garbler gives W_i^0 select bit 0, determining that W_i^1 has select bit 1. Likewise, the garbler gives W_j^0 select bit 1, determining that W_j^1 has select bit 0.

The garbler then permutes the garbled table based on the select bits. The permuted table is shown on the left in table 3.1. When evaluating this gate, the evaluator has W_i^* and W_j^* with select bits b_i and b_j. The evaluator decrypts the ciphertext in the row corresponding to b_i and b_j.

Intuitively, point and permute is secure because the select bits are independent of the truth value (also known as semantic value) of the wires. Thus it is secure for the garbler to permute the table based on the select bits, and it secure for the garbler to send the select bits to the evaluator.

Point and permute slightly increases garbler-side computation to substantially decrease evaluator-side computation. The garbler samples 4 additional random bits, and the evaluator performs a single decryption. Without point and permute, the evaluator needs to decrypt 2.5 ciphertexts on average, hence the garbler performs roughly 1.5 fewer decryptions per gate. The overall bandwidth is increased by 4 bits

Select Bit	Wire Label
0	W_i^0
1	W_i^1
1	W_j^0
0	W_j^1

Select Bits	Encryption
(0,0)	$\mathsf{Enc}_{W_i^0,W_j^1}(W_k^0)$
(0,1)	$\mathsf{Enc}_{W_i^0,W_j^0}(W_k^0)$
(1,0)	$\mathsf{Enc}_{W_i^1,W_j^1}(W_k^1)$
(1,1)	$\mathsf{Enc}_{W_i^1,W_j^0}(W_k^0)$

Table 3.2: Garbled AND gate for Point and Permute

per gate: a tiny constant increase.[6]

3.2 Garbled Row Reduction 3

Garbled Row Reduction 3 (GRR3) reduces the size of the garbled table from 4 ciphertexts to three 3 ciphertexts [22]. In a classical garbled circuit scheme, the wire labels for each wire are chosen prior to garbling any gates. In GRR3, the garbler samples values for the input wire labels prior to garbling, and the garbler gives all other wire labels values as they generate each garbled table.

Suppose the garbler is garbling an XOR gate with input wires W_i and W_j and output wire W_k. The garbler begins by using the point and permute method, sampling select bits and permuting the garbled table. In GRR3, the garbler sets the ciphertext in the top row of the garbled table equal to a value that decrypts to 0^n, the string of n zeros. Specifically, W_k^*, the wire label on the top row, is set to $\mathsf{Enc}^{-1}_{W_i^*,W_j^*}(0^n)$. The garbler sends the bottom three rows of the garbled table to the evaluator. When evaluating the garbled gate, if the evaluator sees that the select bits of the input wires indicate to decrypt the top row, then the evaluator simply assumes the row to have value 0^n and decrypts it as usual. In that case, the garbler sets W_k^* to $\mathsf{Enc}^{-1}_{W_i^*,W_j^*}(0^n)$. Otherwise if the select bits indicate to decrypt the other rows, the garbler decrypts the indicted row as per usual.

Tables 3.3 gives an example of garbling an XOR gate. The left table shows the

[6]The value is constant in the sense that it is independent of the security parameter.

Select Bit	Wire Label
0	W_i^0
1	W_i^1
1	W_j^0
0	W_j^1

Select Bits	Encryption
(0,1)	$\mathsf{Enc}_{W_i^0,W_j^0}(W_k^0)$
(1,0)	$\mathsf{Enc}_{W_i^1,W_j^1}(W_k^0)$
(1,1)	$\mathsf{Enc}_{W_i^1,W_j^0}(W_k^1)$

$$W_k^0 \leftharpoondown \{0,1\}^n$$
$$W_k^1 \leftarrow \mathsf{Enc}^{-1}_{W_i^0,W_j^1}(0^n)$$

Table 3.3: A garbled AND gate using point and permute and garbled row reduction 3 (GRR3)

select bits of wire labels W_i^0, W_i^1, W_j^0 and W_j^1. The right table shows the garbled table, in which the top row, the row associated with select bits $(0,0)$, is missing, as the row is assumed to have value 0^n. The bottom table shows the values of W_k^0 and W_k^1. The value of W_k^0 is randomly sampled from $\{0,1\}^n$.

In considering the security of GRR3, we consider the effect of always setting one of the wire labels, W_k^*, to 0^n. Since the evaluator does not know whether W_k^0 or W_k^1 is set to 0^n, the evaluator does not learn any information about the semantic representation of W_k^*.

GRR3 offers good performance benefits. Garbler-side computation is the same, except that a decryption is performed in place of an encryption in constructing the first row of the garbled table. Evaluator-side computation is the same as point and permute: the evaluator performs a single decryption. Finally, GRR3 reduces the size of the garbled table from 4 ciphertexts to 3 ciphertexts, a 25% reduction in bandwidth.

3.3 Free XOR

The Free XOR technique makes the computation of XOR gates free, in the sense that no garbled table needs to be communicated [13]. The evaluator can compute W_k^* from only W_i^* and W_j^*. Like GRR3, the Free XOR technique takes advantage of carefully crafted wire labels, even input wires.

To start, the garbler randomly samples a ciphertext Δ from $\{0,1\}^n$. For each input wire W_i, let W_i^0 be randomly sampled from $\{0,1\}^n$ as before, and set $W_i^1 = W_i^0 \oplus \Delta$. If the garbler is garbling an XOR gate, then the garbler does not construct a garbled table. Instead, for each XOR gate, the garbler sets the output wire of the gate W_k to have labels $W_k^0 = W_i^0 \oplus W_j^0$ and sets $W_k^1 = W_k^0 \oplus \Delta$.

The evaluator, when evaluating an XOR gate, simply computes $W_k^* = W_i^* \oplus W_j^*$. As simple as it is, the evaluator will always acquire the correct value for W_k^* based on the semantic value of W_i^* and W_j^*. The math for each of the four cases is shown:

$$
\begin{aligned}
&W_i^0 \oplus W_j^0 = W_k^0 \\
&W_i^0 \oplus W_j^1 = W_i^0 \oplus (W_j^0 \oplus \Delta) = (W_i^0 \oplus W_j^0) \oplus \Delta = W_k^1 \\
&W_i^1 \oplus W_j^0 = (W_i^0 \oplus \Delta) \oplus W_j^0 = (W_i^0 \oplus W_j^0) \oplus \Delta = W_k^1 \\
&W_i^1 \oplus W_j^1 = (W_i^0 \oplus \Delta) \oplus (W_j^0 \oplus \Delta) = (W_i^0 \oplus W_j^0) = W_k^0
\end{aligned}
$$

At times, we may write the first wire label W_i^1 as $W_i^0 \oplus \Delta$, since the values are the same if Free XOR is being used. Moreover, if we do not know which wire label we are examining, we may write $W_i^0 \oplus \sigma_i \Delta$ in place of W_i^*. In this notation, σ_i is the semantic value of wire W_i, so if σ_i is 0, then we have written W_i^0 and if σ_i is 1, then we have written $W_i^0 \oplus \Delta$.

With this notation, we can rewrite the above equations succinctly as

$$
\begin{aligned}
W_i^0 \oplus \sigma_i \Delta \oplus W_j^0 \oplus \sigma_j \Delta &= W_i^0 \oplus W_j^0 \oplus (\sigma_i \oplus \sigma_j)\Delta \\
&= W_k^0 \oplus (\sigma_i \oplus \sigma_j)\Delta.
\end{aligned}
$$

Free XOR is compatible with point and permute and GRR3; however, since XOR does not require a garbled table, GRR3 is only used on AND gates.

One interesting implication of using the Free XOR technique is that an added assumption must be made in our encryption algorithm. Since Δ is part of the key and part of the the payload[7] of the encryption algorithm, the encryption algorithm must be secure under the circularity assumption. Fortunately, the popular encryption scheme AES-128 is presumed to be secure under the circularity assumption.

Free XOR dramatically reduces bandwidth, and because XOR gates are relatively cheaper than AND gates, circuits with more XOR gates perform faster. Many programs have been made to optimize the number of XOR gates and minimize the number of AND gates in a circuit (while minimizing the size of the entire circuit of course). The Free XOR technique reduces garbler-side computation: constructing the XOR garbled table does not require three encryptions and one decryption. It likewise reduces evaluator-side computation since XOR gates do not require any decryption. The biggest benefit is undoubtedly that XOR gates do not require a garbled table.

3.4 Garbled Row Reduction 2

Garbled row reduction 2 (GRR2) reduces the size of the garbled table of AND gates to 2 ciphertexts [22]. Unfortunately, GRR2 is not compatible with Free XOR, and GRR2 is less efficient than Free XOR combined with GRR3 for most circuits, so GRR2 is not often used by itself in practice.

Suppose the garbler is garbling an AND gate with input wires W_i and W_j and output wire W_k. For all $a, b \in \{0, 1\}$, let $V_{a,b} = H(W_i^a, W_j^b)$. When evaluating a gate, the evaluator acquires W_i^a and W_b^j for some a and b, so they can compute $V_{a,b}$. The garbler constructs a polynomial P to be the unique two-degree polynomial passing through the points $(1, V_{0,0}, (2, V_{0,1})$ and $(3, V_{=1,0})$ - these V's are selected because the garbler is garbling an AND gate. And they set W_k^0 to $P(0)$. The garbler next constructs a second polynomial Q to be the unique two-degree polynomial passing through points $(4, V_{1,1}), (5, P(5))$ and $(6, P(6))$. The garbler then sets W_k^1 to $Q(0)$.

[7] The payload is the value that is being encrypted.

The garbler sends $(5, P(5))$ and $(6, P(6))$ to the evaluator - these two values compose the garbled table.

When evaluating, the evaluator holds a, b, W_i^a, W_j^b, $(5, P(5))$ and $(6, P(5))$. With this information, the evaluator constructs polynomial R to be unique the degree-2 polynomial passing through points $(2a + b + 1, H(W_i^a, W_j^b), (5, P(5))$ and $(6, P(6))$. Polynomial R is either P or Q, depending on the values of a and b. The evaluator simply sets W_k^* to be $R(0)$.

GRR2 is incompatible with Free XOR since it sets W_k^0 and W_k^1 implicitly and unpredictably. Free XOR requires that $W_k^0 = W_k^1 \oplus \Delta$ for some global delta, which is not achievable with the randomly constructed polynomials.

GRR2 alters garbler-side computation by having the garbler construct two polynomials instead of encrypting ciphertexts. This is slightly more computation than required by FreeXOR and GRR3, but not an undermining amount. Evaluator side computation increases slightly, as the evaluator constructs a polynomial instead of decrypting two ciphertexts.

3.5 FleXOR

After the creation of GRR2, secure computation was at an awkward point. Circuits with many XOR gates were computed most quickly with Free XOR and GRR3, but circuits with many AND gates were computed most quickly with GRR2. FleXOR reconciles GRR2 with Free XOR, resulting in a scheme that is universally faster [12].

Recall that GRR2 is incompatible with Free XOR because the wire labels are uncontrollably created by random polynomials, where Free XOR requires that $W_k^0 = W_k^1 \oplus \Delta$ for some global delta. FleXOR solves this problem in a straightforward fashion: correct the delta value of output wires of AND gates such that the output wires use the global delta. To correct the value, FleXOR adds a unary gate after each

AND gate that corrects the wire label.[8]

Suppose an XOR gate has input wires W_i and W_j and output wire W_k. Input wires W_i and W_j each come from an AND gate, so their labels are the result of the polynomial interpolation of GRR2. W_i has labels W_i^0 and $W_i^0 \oplus \Delta_i$ and W_j has labels W_j^0 and $W_j \oplus \Delta_j^0$. To perform Free XOR, W_i, W_j and W_k need to be using the same delta value. The garbler adds an extra gate between W_i and W_j and the XOR gate that adjusts their XOR value to the correct value. W_i^0 and $W_i^0 \oplus \Delta_i$ change to $W_i^{0'}$ and $W_i^{0'} \oplus \Delta$. W_j^0 and $W_j^0 \oplus \Delta_j$ change to $W_j^{0'}$ and $W_j^{0'} \oplus \Delta$. Since W_i, W_j and W_k have the same delta value, we can use Free XOR.

The unary gate maps $W_i^0, W_i^0 \oplus \Delta_1 \to W_i^{0'}, W_i^{0'} \oplus \Delta$, where Δ is the correct delta value for the XOR gate.

FleXOR is made more efficient by not correcting the output wire of every AND gate. For example, if an output wire of an AND gate is immediately inputted into another AND gate, the wire label does not need to corrected. Moreover, the wire labels do not even need to be corrected to a global delta. Free XOR only requires that the three wires involved in the XOR gate, W_i, W_j and W_k, use the same delta, so each XOR gate has its own Δ. For example, W_i and W_k may share a delta but W_j may have a different delta, so it is sufficient to only correct W_j's delta value.

FleXOR is fastest when AND gates are grouped together and XOR gates are grouped together, since fewer unary gates will be required. Thereby, FleXOR creates an optimization problem: place XOR gates, AND gates and unary gates to minimize bandwidth requirements. However, FleXOR's optimization problem is computationally expensive, but on the up side, analysis reveals that FleXOR requires on average an extra 0 or 1 ciphertext per gate, a relatively small amount. That cost comes at the benefit of a Free XOR circuit, and 1 fewer ciphertext for each AND gate.

[8] A unary gate is a gate that takes a single input wire and outputs a single wire. The unary gate does not change the semantic meaning of a wire label - that is, whether it represents 0 or 1. The unary gate merely alters the actual value of the wire label or ciphertext.

FleXOR requires slightly more garbler-side computation than FreeXOR and GRR2, since the garbler must create the unary gates. The evaluator-side computation is the same as FreeXOR and GRR2, with the additional computation of the unary gates, which is small. The size of the XOR garbled table is zero, and the size of the AND garbled table is two. But there is the additional communication of the unary gates' garbled tables. The garbled table of the unary gate has 2 ciphertexts, and the number of unary gates depends on the circuit.

FleXOR is intuitively secure, since the only additional information beyond GRR2 and FreeXOR is the unary gates. The unary gate is secure, since it functions the same as a normal garbled gate except with a single input wire.

3.6 Half Gates

Half Gates is the most recent improvement to garbled circuits [29]. The goal of Half Gates is to make AND gates cost two ciphertexts, while preserving properties necessary for Free XOR without adding unary gates. Half gates work by splitting an AND gate into two half gates. Each half gate is an AND gate, where one of the inputs is *known* to a party. One half gate is the garbler-half-gate, where the garbler knows the semantic value of one of the input wires, and the other gate is the evaluator-half-gate, where the evaluator knows the semantic value of one of the input wires. By combining the garbler-half-gate and evaluator-half-gate, we can construct a generic AND gate.

We first examine the garbler-half-gate. Imagine an AND gate with input wires W_i and W_j and output wire W_k, where the garbler knows the semantic value of W_i. This means that the garbler knows whether W_i^* is W_i^0 or W_i^1. Let bits σ_i, σ_j and $\sigma_k \in \{0, 1\}$ represent the semantic value of W_i, W_j and W_k respectively. Since we are working with an AND gate, if $\sigma_i = 0$, then $\sigma_k = \sigma_i \wedge \sigma_j = 0$, so the garbler creates a

Garbled Table for $\sigma_i = 0$
$\mathsf{Enc}_{W_j^0}(W_k^0)$

Garbled Table for $\sigma_i = 1$
$\mathsf{Enc}_{W_j^0}(W_k^0)$
$\mathsf{Enc}_{W_j^1}(W_k^1)$

$\rightarrow$

Garbled Table for any σ_i
$\mathsf{Enc}_{W_j^0}(W_k^0)$
$\mathsf{Enc}_{W_j^1}(W_k^0 \oplus \sigma_k\Delta)$

Table 3.4: Left: garbler-half-gate garbled table for $\sigma_i = 0$. Middle: garbler-half-gate garbled table for $\sigma_i = 1$. Right: garbler-half-gate garbled table for arbitrary σ_i.

unary gate that always outputs W_k^0. If $\sigma_i = 1$, then $\sigma_k = \sigma_i \wedge \sigma_j = \sigma_j$, so the garbler creates a unary identity gate that outputs $W_k^{\sigma_j}$. We combine the unary gates into a single garbled table, shown in table 3.4. Since the evaluator has W_j^*, the evaluator can compute the label of the output wire of the AND gate by decrypting the rows in the garbled table to acquire $W_k^0 \oplus \sigma_j\Delta$.

Using point and permute and the GRR3 trick, we reduce the size of the garbler-half-gate garbled table to one ciphertext. The garbler chooses W_k^0 such that the top row of the garbled table is the all zeros ciphertext, and therefore does not need to be sent to the evaluator. If the evaluator should decrypt the ciphertext on the top row (as directed by point and permute), then the evaluator assumes the ciphertext to be all 0s.

The evaluator-half-gate is somewhat different from the garbler-half-gate. Consider an AND gate where the evaluator *knows* the semantic value of wire W_i, σ_i. If $\sigma_i = 0$, then the evaluator should acquire W_k^0. Otherwise if $\sigma_i = 1$, then the evaluator should acquire $W_k^0 \oplus \sigma_j\Delta$. In this case, it suffices for the evaluator to obtain $\Omega = W_k^0 \oplus W_j^0$, as the evaluator simply sets W_k^* to $W_j^* \oplus \Omega$ which sets W_k^* equal to $W_k^0 \oplus \sigma_\Delta$.

Table 3.5 shows the garbled table for the evaluator-half-gate. This table does not need to be permuted, since the evaluator knows σ_i. Again, we use the GRR3 trick, and set W_k^0 to $\mathsf{Enc}^{-1}_{W_i^0}(0^n)$, eliminating the need to send the top row of the garbled table.

Garbled Table for any σ_i
$\mathsf{Enc}_{W_i^0}(W_k)$
$\mathsf{Enc}_{W_i^1}(W_k \oplus W_j)$

Table 3.5: Evaluator-half-gate garbled table.

We now put the two half gates together to form an AND gate. Consider the following where r is a random bit generated by the garbler:

$$\sigma_i \wedge \sigma_j = \sigma_i \wedge (r \oplus r \oplus \sigma_j) \tag{3.1}$$

$$= (\sigma_i \wedge r) \oplus (\sigma_i \wedge (r \oplus \sigma_j)). \tag{3.2}$$

The first AND gate, $\sigma_i \wedge r$, can be computed with a garbler-half-gate - the garbler *knows* r. Furthermore, if we can let the evaluator know the value of $r \oplus \sigma_j$, then the second AND gate, $(\sigma_i \wedge (r \oplus \sigma_j)$, can be computed with an evaluator-half-gate - the evaluator *knows* $r \oplus \sigma_j$. And the XOR gate can be computed with free xor at the cost of no ciphertexts.

It is secure for the garbler to give $r \oplus \sigma_j$ to the evaluator, since r is random so it blinds the value of σ_j. The value of $r \oplus \sigma_j$ can be communicated to the evaluator for free: use the select bit (from the point and permute technique) of the false wire label of wire W_j (so r is the select bit on the true wire label of wire σ_j).

The overall cost of using Half Gates for AND gates is four encryptions for the garbler, two decryptions for the evaluator, and the communication of 2 ciphertexts. Half gates guarantees only two ciphertexts are needed per AND gate, but the trade-off is the additional computation for both parties. With FleXOR, the number of ciphertexts that need to be communicated may vary, but there is less computation required.

3.7 Improving Oblivious Transfer

This section discusses improvements to oblivious transfer, the method by which the garbler communicates the wire labels corresponding to the evaluator's inputs to the evaluator. At its most basic, oblivious transfer enables Alice to potentially send one of two messages to Bob. Bob selects which message he wants to receive. The desirable security properties are (1) Alice does not know which message she sent to Bob and (2) Bob cannot infer any information about the message that he did not receive.

Oblivious transfer is a key component in secure computation, and often consumes a substantial portion of the secure computation protocol's time. There are two major improvements to oblivious transfer. The first is called *OT-extension*. When using OT with a garbled circuit scheme, Alice and Bob do not exchange a single message; rather, they exchange a wire label for each of Bob's inputs. This means that Alice and Bob are performing a linear number of OTs based on the number of inputs. OT-extension reduces the entire OT phase to a constant number of OT operations [10]. In particular, Alice and Bob run a constant number of OT in order to generate a polynomial number of exchanged messages.

The second improvement to OT is called *OT-preprocessing* [1]. In OT-preprocessing, Alice and Bob performs the expensive OT operation ahead of time, in what is called the offline phase, on random values. Then, during the online phase, Alice and Bob quickly exchange values to correct the pre-shared messages. Alice and Bob only exchange a single message during the online phase, and there is no computationally expensive math required like there is during the real OT operation.

In our work with secure computation, OT-preprocessing substantially improved the performance of garbled circuits, and OT-extension was less useful. If a garbled circuit scheme uses an offline phase and thereby uses OT-preprocessing, then OT-extension is largely unnecessary. The parties take the extra time needed to send all of the wire labels in the offline phase, since the time does not matter; the parties are concerned with online time. OT-preprocessing offers huge improvements for online running time and online bandwidth.

Chapter 4

Component-Based Garbled Circuits

In this chapter we introduce component-based garbled circuits, a method that allows for most of the work involved in building and communicating a garbled circuit to occur in an offline phase before the inputs or function to compute are known. This offers significant improvements to the online performance of garbled circuits.

Component-based garbled circuits are the research of a larger project to which this thesis contributes. The first part of this chapter will describe naive component-based garbled circuits. Naive component-based garbled circuits are presented in an unpublished paper, and I am a coauthor on the paper. I contributed two items to the unpublished paper. The first is an improvement to component-based garbled circuits called single communication multiple connections (SCMC); we present SCMC at the end of this chapter. The second is CompGC, an implementation of component-based garbled circuits. We discuss CompGC in detail in Chapter 5.

4.1 History

The goal to reduce online computation by using an offline phase is not a new goal. In Chapter 3, we discuss oblivious transfer preprocessing, a method that moves most of the work involved in oblivious transfer to an offline stage [1]. In the research of Malicious garbled circuits, researchers improve the online efficiency of cut-and-choose by using an offline phase [9, 15, 16]. Moreover, research in non-garbled circuit two party computation uses similar ideas in [4, 20].

Most similar to component-based garbled circuits is a technique called *partial garbled circuits* proposed by Mood et al. [18]. The goal of partial garbled circuits was to retain state after a garbled circuit computation. To achieve this, Mood et al. use two link labels per connection, whereas we use fewer than one label per connection on average.

4.2 Naive Component-Based Garbled Circuits

Component-based garbled circuits begin with the observation that most functions are composed of many smaller components. For example, many statistical operations are a composition of matrix operations; Smith-Waterman and Levenshtein distance, two algorithms important to analyzing genomes, are dynamic algorithm that run a single procedure inside a for loop; and encrypting arbitrary length messages via modes of operation repeats the encryption algorithm a number of times. The gist of component-based garbled circuits is then to build and communicate many small, garbled components in the offline stage. Later, in the online stage, the garbler and evaluator combine the pre-communicated components to form a larger function. They compute each component individually, linking wire labels from one component to another as needed.

Imagine that two banks integrate secure computation into their daily transactions.

Figure 4.1: An example of linking component c_0 to component c_1. The output wires of c_1, U_0, U_1 and U_2, are linked to the input wires of c_1, W_0, W_1 and W_2.

At night when activity is low, the banks' servers exchange many garbled circuits, and then during the day, they use the pre-exchanged garbled circuits to quickly perform secure computation. The computational requirements for the banks, when they compute the function, is only to exchange input labels and for the evaluator-bank to evaluate the garbled circuit, while, importantly, the banks preserve the ability to choose their inputs and the function to be computed at the time of the computation.

At a coarse level, chaining looks like figure 4.1 where the input to some some garbled components is the output of other garbler components. To chain, or stitch together, two garbled components, the evaluator needs to transform an output wire label of one component into a valid input wire label of another component, while preserving the semantic value of the wires. In figure 4.1 the zeroth output wire of c_0, U_0, needs to transform into the zeroth input wire of c_1, W_0.

Chaining, or stitching together two garbled circuits, essentially takes the output wires of one garbled circuits and converts them into valid input wires of another garbled circuit while retaining the semantic value of the input wires. Consider figure 4.1 where three output wires of component c_0 are chained to three input wires of component c_1. We specifically desire that the evaluator transform U_0^0 into W_0^0 and U_0^1 into W_0^1.

To achieve this transformation, the garbler sends the evaluator a link label. A link label is a ciphertext that allows the evaluator to transform the output wire label into the appropriate wire label. Suppose that the garbling uses Free XOR where each

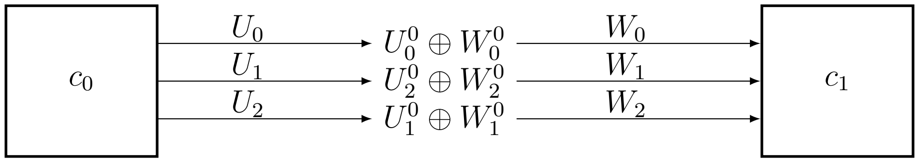

Figure 4.2: An example of linking component c_0 to component c_1. The link labels for linking the output wires of c_0 and the input wires of c_1 are shown between the wires.

component uses the same Δ, that is, $U_0^0 \oplus U_0^1 = W_0^0 \oplus W_0^1 = \Delta$, then it is sufficient for the link label L_{UW} to be $U_0^0 \oplus W_0^0$. The evaluator xors the output wire label U_0^* with L_{UW} to acquire W_0^*, a valid input label for wire W_0.

We show this computation below; the value $u_0 \in \{0, 1\}$ is the semantic value of wire U_0. For information on this notation, see section 3.5.

$$U_0^0 \oplus u_0\Delta \oplus L_{UW} = U_0^0 \oplus u_0\Delta \oplus U_0^0 \oplus W_0^0 \tag{4.1}$$

$$= W_0^0 \oplus u_0\Delta \tag{4.2}$$

We see from this equation that the evaluator acquires a valid wire label for wire W_0 which has the same semantic value as wire label U_0^*.

We formally describe component-based garbled circuits as a tuple of three algorithms $(\mathsf{Garble}, \mathsf{Link}, \mathsf{Eval})$. The Garble algorithm is the same algorithm that we used before to garble circuits; it takes as inputs a circuit C and outputs three pieces of information: a garbled circuit GC, a set of input wires e_{C}, and a set of output wires d_{C}. In the component-based setting, the Garble algorithm can also take a component-circuit c as input, to which the algorithm outputs a garbled component $\mathsf{GC_c}$, input wire set e_c and output wire set d_c.

The Link algorithm is unique to component-based garbled circuits, and it produces the link labels necessary for linking. Link takes as input two garbled components $c_0 = (\mathsf{GC_0}, \mathsf{e_0}, \mathsf{d_0})$ and $c_1 = (\mathsf{GC_1}, \mathsf{e_1}, \mathsf{d_1})$, and a mapping of output wires of c_0 to input wires of c_1. Link outputs the *link* labels needed to convert the output wires d_0 to input wires e_1. Suppose that output wire $U_i \in d_0$ has labels (U_i^0, U_i^1), and input wire $W_j \in e_1$ has wire labels (W_j^0, W_j^1), then Link outputs $L_{u_i w_j} = U_i^0 \oplus W_j^0$. The discussion above explains why link label $L_{u_i w_j}$ is sufficient for linking.

The Eval algorithm evaluates the garbled components and links garbled circuits where necessary. It takes three inputs: a list of a garbled components $\{c_i\}$, linking labels $\{L_{ij}\}$ and input labels $\{W_i\}$, and it outputs output labels $\{Z_i\}$. Eval starts with the inputs, and then proceeds component by component, evaluating each component in order to get the component output wire labels. Where necessary, Eval uses component output wire labels and link labels to compute the appropriate input label for later components. Once all components are evaluated, Eval recovers the garbled outputs $\{Z_i\}$ from the output components, and uses d for that component to recover the real output y.

Let us think about how the two banks who use secure computation in their daily operations employ the three algorithms. At night, during the offline stage, the garbler-bank generates many circuit components, garbles them with Garble, and sends the garbled circuits to the other evaluator-bank. The garbler-bank and evaluator-bank also perform the offline phase of OT-preprocessing. During the day, when the banks decide that they want to securely compute some function f, the garbler-bank runs Link on the components used in f to generate the link labels, which the garbler-bank subsequently sends to the evaluator-bank. The garbler and evaluator also exchange wire labels, some via online OT-preprocessing. Now that the evaluator-bank has the garbled components, link labels and input labels, they run Eval, recovering the output of the function.

4.3 Security of Component-Based Garbled Circuits

In this section, we show how to adapt the standard definition of privacy, presented in chapter 2, to component-based garbled circuits. Recall from Chapter 2 that we said a garbled circuit scheme is secure if for all probabilistic, polynomial-time simulators (i.e algorithms) S_1 and S_2 the following holds:

$$\{(S_1(x, f_1(x,y), f(x,y)))\}_{x,y} \approx_C \{(\mathsf{view}_1^{\Pi}(x,y), \mathsf{output}^{\Pi}(x,y))\}_{x,y} \tag{4.3}$$

$$\{(S_2(x, f_2(x,y), f(x,y)))\}_{x,y} \approx_C \{(\mathsf{view}_2^{\Pi}(x,y), \mathsf{output}^{\Pi}(x,y))\}_{x,y} \tag{4.4}$$

To consider the security of component-based garbled circuits, we add in the extra information that each party receives into $\mathsf{view}^{\Pi}(x,y)$. Party 1, the garbler, only receives additional information from OT-preprocessing, which we know to be secure, hence equation 4.3 remains true.

We think through the security of the evaluator, party 2, using *hybrids*, a common cryptographic technique in proofs. A hybrid argument begins by imagining the protocol in the real world. The aim is to show that the real world protocol is computationally indistinguishable from the ideal world. To achieve this, we rely on the fact that computational indistinguishability is transitive, that is if for worlds X, Y and Z such that $X \approx_C Y \approx_C Z$, we know that $X \approx_C Z$. The hybrid argument starts with the real world, makes a small incremental change to the world to make it more similar to the ideal world. Then we take the modified world, and make another small change to make the world more similar to the ideal world. Eventually, the world that we are operating with is the ideal world. If at every step, we prove that the previous world is computationally indistinguishable from the next world, then by the transitivity of computationally indistinguishability, we have shown that the real world is computationally indistinguishable from the ideal world.

We start with the real world protocol, in which the evaluator has the following information: pre-garbled components $\{GC_i\}_{i=0}^{|\mathsf{Components}|}$, input labels $\{W_j^{x_j}\}_{j\in\mathsf{Inputs}(C)}$, output map $d_{C_{out}}$ and link labels $\{L_{ij}\}_{i,j\in\mathsf{Components}}$. We first suppose that garbler, instead of sending correct link labels, sends garbage, random values for link labels. Yes, this will prevent Bob from recovering the correct output, but correctness is irrelevant when we are reasoning about security; we are only concerned with the information that Bob can access. We now ask the question: is the real world computationally indistinguishable from the world where Bob receives incorrect link labels? Or worded more accessibly: does Bob learn any more information from correct wire labels than he does from garbage wire labels? We argue, without proof, that Bob does not learn any more information because (1) the link labels are encrypted so they look like garbage to Bob anyways, and (2) Bob cannot use the link labels with other pieces of data, such as input wire labels of the garbled circuits, to gain information.

We now hybrid from the world in which Bob receives garbage link labels to a world in which Bob receives both garbage link labels and garbage input wire labels. Again, we note that Bob will not recover the correct answer, but correctness is irrelevant to our aim. We know that input wire labels do not give Bob any information from our proof in Chapter 3 that garbled circuits are secure. Therefore these two worlds are computationally indistinguishable.

By referencing the proof in Chapter 3 that garbled circuits are secure, we also know that turning the output map and garbled circuits into garbage values gives Bob no more information. This means a world in which all the information that Bob receives is garbage is computationally indistinguishable from the real world. This garbage infested world is identically the ideal world - the only non-garbage value Bob has is his input, just like the ideal world - hence we successfully argue that the real world is computationally indistinguishable from the ideal world, satisfying equation

4.4.

It should be noted that this not even close to a formal proof of security. A formal proof is long, many pages long, and beyond the scope of this project; rather, this section highlights the intuition behind why we believe a formal proof of security to be possible.

4.4 Single Communication Multiple Connections

The previous method of chaining garbled circuits was presented to me, and I was asked to implement the ideas in a program (see Chapter 5) as part of a larger research project. In this midst of writing the program, I discovered a method to improve the efficiency of chaining. In this section I discuss my theoretical contribution to component-based garbled circuits: *Single Communication Multiple Connections*.

Single Communication Multiple Connections (SCMC) is a technique that improves upon naive component-based garbled circuits for some functions by observing that large sequences of consecutive wires often represent a single element of data, like a string, number or matrix. When large sequences of consecutive wires are mapped between garbled components, the sequence of consecutive wires is mapped together - the order of the wires is preserved. SCMC takes advantage of this fact by requiring that only a single link label is communicated for each sequence of consecutive wires, as opposed to one link label per wire. This technique is analogous to SIMD-style (single instruction multiple data) computation that is used in homomorphic encryption [26] and GMW-based two party computation [5, 25] in that a single piece of communication, or instruction, is used for multiple links, or data.

As with most improvements to garbled circuits, SCMC operates by modifying the generation of wire labels. Wire labels that are part of a block representing a single piece of data are generated in a fixed, patterned way such that the differences between linked labels will be the same for all wires in a block.

SCMC modifies the Garble algorithm by choosing input wires and output wires in a particular way. For every component, we first choose three random labels A, B and T: A and B are the base values for input wires and output wires respectively, and T is a tweak value, ensuring security for the hash function. We next assume that all parties have access to a random oracle H. A random oracle is a theoretically perfect random function, which on input of any value $x \in \{0,1\}^*$, returns a value from $\{0,1\}^\lambda$ selected uniformly at random. In practice, we use a hash function for H. Then, for all input wire labels W_i^b, set wire label W_i^b to $A \oplus H(T \oplus (i||b))$. Likewise, for all output wires labels W_i^b, we do the same process, setting W_i^b to $B \oplus H(T \oplus (i||b))$.

Suppose that all output wires of component c_0 are being linked to the input wires component c_1, where the ith input wire is being mapped to the ith output wire. Then, it is sufficient for the garbler to send $B_{c_0} \oplus A_{c_1}$ to the garbler. This works since for any wire i and bit b,

$$(B_{c_0} \oplus H(T \oplus (i||b)) \oplus (B_{c_0} \oplus A_{c_1}) = A_{c_1} \oplus H(T \oplus (i||b)) \tag{4.5}$$

Evaluation is the same as naive component-based garbled circuits, wherein the evaluator links using the correct wire label.

4.4.1 Analysis

SCMC substantially reduces the amount of communication required for chaining. For example, if AES is being computed where the components are single AES rounds, then SCMC requires only 9 link labels to be communicated between the garbler and evaluator; naive component-based garbled circuits requires 1152 labels.

The milage of SCMC varies based on the function being computed and components being used. SCMC offers the most benefits to functions that are modular, and where large amounts of data are being passed around, like matrix-based functions and dynamic algorithms like Levenshtein distance. When using SCMC, online bandwidth does not scale with an increase in data size. SCMC requires that a single link label being communicated for a 2 by 2 matrix and for a 1,000 by 1,000 matrix. Thereby, SCMC greatly improves the speed of securely computing large statistical computations, such as those that might be performed by hospitals or on genomes.

The security of SCMC follows directly from the security of naive component-based garbled circuits. The wire labels again do not offer any information, so a similar hybrid proof works in the SCMC setting as well. There is a caveat: SCMC uses a new method of generating wire labels. This is secure since H is random function, hence $H(\cdot)$ blinds A and B; in particular, the wire labels are only distinguishable if the evaluator can determine T, which is not possible since H is a random oracle. Therefore, SCMC is secure.

Chapter 5

Implementation

My largest contribution to the project of component-based garbled circuits is the implementation of the theoretical ideas in a program called CompGC. The aim of CompGC is to run a two party garbled circuit protocol from start to finish. The garbler and evaluator simply select a function to compute, plug their inputs into the system, and CompGC securely computes the function. This chapter describes the creation of CompGC and presents experimental results.

5.1 CompGC

CompGC was constructed over the period of 5 months in the programming language C. It consists of approximately $5,000$ lines of code, includes a submodule, LibGarble, and includes python scripts to generate auxiliary files.

CompGC started with JustGarble as its basic building block. JustGarble is a C-library written by Bellare et al. [2] that creates a garbled circuit given an inputted circuit, and evaluates the garbled circuit given input labels. It is a tool, supporting only garbling and evaluating operations, but does not perform the many operations needed for a whole secure system, like networking, oblivious transfer, and secure recovery of final outputs.

I later replaced JustGarble with LibGarble, an improvement of JustGarble, which was implemented by my colleague Alex Malozemoff. LibGarble made a number of improvements to JustGarble, including cleaning up the API and improving the memory layout of data-structures. These improvements contributed to a substantial increase in performance: evaluating AES uses 17 cycles per gate in LibGarble versus 22 cycles per gate in JustGarble, a 22% improvement. LibGarble also adds the newest cryptographic technique, half-gates; however, CompGC does not use half-gates since half-gates reduces the size of the garbled table at the cost of a single call to the hash function during evaluation. In the component-based garbled circuit setting, the garbled tables are communicated during offline time, and we are most concerned with online time when the hash function would be called; hence half-gates actually reduce performance.

CompGC has an offline and an online phase. In the offline phase, CompGC takes as input a list of circuits and creates a specified number of each circuit. The list of circuits could be small and focused, designed for computing a single function such as AES, or the list could be diverse, allowing for the computation of a variety of functions. CompGC garbles the list of circuits the specified number of times, and then sends the garbled circuits with attached identification numbers from the garbler to the evaluator. The garbler and evaluator each save the garbled circuits to disk. Finally, the garbler and evaluator perform the offline phase of preprocessed oblivious transfer. CompGC uses the Naor-Pinkas semi-honest oblivious transfer protocol; the library for performing oblivious transfer was given to me by Alex Malozemoff [19]. The garbler finishes by saving the input labels and oblivious transfer data to disk, and similarly, the garbler saves the oblivious transfer data to disk. Algorithms 1 and 2 show the steps taken by each party in the offline phase.

In the online phase, the evaluator begins by loading the function to be computed from disk. We specify the function in a JSON format, in which the following information is laid out: components needed in the function, how components input and output wires are linked, where inputs should go, and what wires are outputs. We chose to use JSON because the file format is human readable, but a different format, such as a binary format, would be faster. As the functions become more complex, it becomes harder to write the function specification files. To overcome this, I wrote a python script that automatically generates the function specification file for various functions on arbitrary inputs.

After the garbler loads the function specification file from disk, it generates a set of instructions for the evaluator. The instructions specify what components are used by naming them with the unique identification number assigned in the offline phase. The instructions further specify in what order to evaluate components, and how to link components. This step requires specifying the output wires of one garbled circuit, the input wires of another garbled circuit, and the linking value. Finally, the instructions specify how to map the final output wire labels to bits; this works by sending unary gates, with a two-row garbled table, for each output wire.

After the garbler sends the instructions to the evaluator, they both perform the online phase of oblivious transfer, whereby the evaluator acquires their input labels. The garbler next sends the input labels corresponding to their inputs. At this point, the evaluator has all the data they need to finish the protocol. They follow the instructions, and acquire the output bits. Algorithms 3 and 4 summarize the online phase.

Algorithm 1 Garbler Offline

Input: List of circuits, and number of ciphertexts to be OT-preprocessed.

1. Build circuits
2. Garble circuits (using LibGarble)
3. Assign each garbled circuit an ID
4. Send garbled circuits and their IDs to evaluator
5. Save input labels and output labels of garbled circuit to disk
6. Perform offline phase of OT-preprocessing
7. Save data from OT-preprocessing to disk

Algorithm 2 Evaluator Offline

1. Receive garbled circuits from garbler
2. Save garbled circuits to disk
3. Perform offline phase of OT-preprocessing
4. Save OT-preprocessing data to disk

Algorithm 3 Garbler Online

1. Load input labels and output labels of garbled circuits from disk
2. Load OT-preprocessing data from disk
3. Load function specification from disk
4. Generate instructions from function specification
5. Compute chaining values, and add values to instructions
7. Perform online stage of OT-preprocessing
8. Send input wires correspond to garbler's input
9. Send instructions

Algorithm 4 Evaluator Online

1. Load garbled circuits from disk
2. Load OT-preprocessing data from disk
3. Perform online stage of oblivious transfer - acquire evaluator's input labels
4. Receive garbler's input labels
5. Receive instructions
6. Following instructions, chaining and evaluating as instructed

5.2 Experiments

CompGC experiments were run on an Intel Core i5-4210H CPU. They were conducted over two network settings. The first network setting ran both parties on the default localhost configuration, which was measured to have a latency of 0.012 ms and bandwidth of 25.2 Gb per second. The second network setting used the built in Linux network emulator netem to configure localhost to mimic the latency and bandwidth of the internet. This included setting latency to 33 ms and bandwidth to 50 Mbits per second. Finally, CompGC requires reading from disk; our experimental machine was measured to have cache reads speed of 6.7 GB per second and buffered disk reads speed of 96 MB per second.

We ran four experiments: AES, CBC mode, and Levenshtein distance with 30 symbols and with 60 symbols.

In the AES experiment, we treated each round of AES as a separate component. AES has 10 rounds, and hence required linking 10 components. Moreover, we used 128-bit AES, meaning that each component link required linking 128 wires.

CBC mode is an algorithm for encrypting messages of arbitrary length using a blockcipher, for which we used AES-128. In CBC mode we used two components: we again used a single round of AES as a component, and we also used an XOR component, which took two 128 bit string as input, and outputted the bitwise XOR of those strings. For our experiment, we ran CBC mode on a 10 block message, a 1,280 bit string, thus requiring 110 components - 100 AES round components and 10 XOR components.

Levenshtein distance is a measure of the difference between two strings which counts the minimum number of insertions, deletions or substitutions needed to transform one inputted string into the other inputted string. The most popular algorithm for computing Levenshtein distance is a dynamic program that runs in $\Theta(n^2)$. The algorithm is two nested for loops where we run a procedure called *Levenshtein-core*

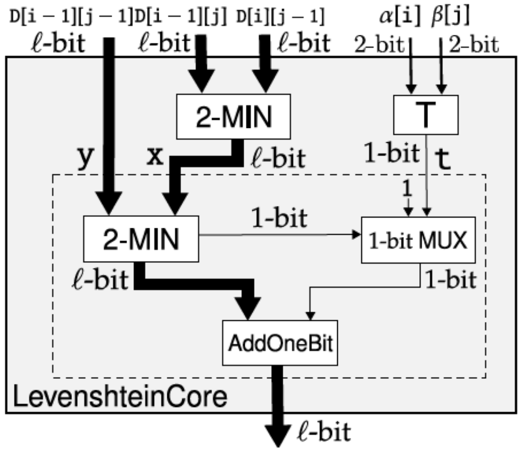

Figure 5.1: The Levenshtein-core component used in the Levenshtein distance algorithm. Levenshtein distance is a dynamic algorithm, and this is the only component used. This particular version of Levenshtein-core was designed by [8].

inside the inner for loop. Levenshtein-core takes five inputs - three distance values and two symbols - and it computes a new distance based on those five values. For more information on the Levenshtein distance algorithm, we refer the reader to [27].

For our Levenshtein experiment, we use the Levenshtein-core circuit as our only component; see figure 5.1 for a description of the component. If Levenshtein is being run on n-bit inputs, then we use n^2 Levenshtein-core components. Our experiment used an 8-bit alphabet and ran Levenshtein distance on strings of length 30 symbols and on strings of length 60 symbols, corresponding to 900 and 3600 components respectively.

5.3 Results

We ran experiments on the CompGC system to compare component-based garbled circuits to standard 2PC garbled circuits, and to compare standard component-based garbled circuits to SCMC. After I wrote CompGC and setup the code for experiments, I sent the code to Alex Malozemoff who ran the experiments.

Table 5.1 shows the online evaluator computation time, including time to load data from disk, of standard 2PC garbled circuits (implemented inside of the CompGC system) and component-based garbled circuits with SCMC. We give the average time of 100 trials and the 95% confidence interval. We see that larger circuits - Levenshtein 60 - offer more time savings than smaller circuits - AES; this is expected, since standard garbled circuits require that the entire circuit is communicated during online time, whereas the component-based garbled circuits sends the circuit during the offline phase.

Table 5.2 shows the same timing as table 5.1 except we remove the time spent loading from disk. Since CompGC employs an offline phase, data is loaded from disk in the online phase. Table 5.2 can be misleading - in what application do we assume that data is preloaded into RAM? We include table 5.2 because the literature often reports these times [16]. The times do have merit, as they highlight how CPU-bound the computation is, as opposed to measuring reading-from-disk speed, which can vary widely and is not something over which algorithm creators or programmers have much control.

Table 5.3 compares the speed of standard component based garbled circuits to SCMC. We see approximately a 2x improvement for all of the experiments. However, these experiments do not highlight the benefit of SCMC. SCMC offers the greatest benefits in a large circuit where a large number of wires are used to represent a section of data. Levenshtein distance, while a large circuit, has a small data-size: the data size of 60 symbol Levenshtein is only 6 bits. Consider circuits where an entire matrix is moved around between circuits; for example, linking a 10 by 10 matrix uses 800 link labels if entries in the matrix were between 0 and 256, but SCMC uses a single link label.

	Time (localhost)		Time (simulated network)		Communication	
	Naive	CompGC	Naive	CompGC	Naive	CompGC
AES	4.4 ± 0.0 ms	3.0 ± 0.2 ms	542.6 ± 0.7 ms	68.5 ± 0.2 ms	24 Mb	254 Kb
CBC, 10 blocks	45.8 ± 4.0 ms	22.7 ± 1.4 ms	4.8 ± 0.0 s	216.7 ± 0.2 ms	235 Mb	2.6 Mb
Leven, 30 symbols	28.9 ± 6.6 ms	24.3 ± 1.2 ms	2.2 ± 0.0 s	315.9 ± 0.5 ms	108 Mb	6.3 Mb
Leven, 60 symbols	109.8 ± 7.0 ms	62.2 ± 0.7 ms	10.6 ± 0.0 s	742.5 ± 2.0 ms	524 Mb	25 Mb

Table 5.1: Experimental results. Naive denotes standard semi-honest 2PC using garbled circuits and preprocessed OTs using LibGarble, whereas CompGC denotes our component-based implementation using SCMC. Leven is Levenshtein distance. Time is online computation time, not including the time to preprocess OTs, but including the time to load data from disk. All timings are of the evaluator's running time, and are the average of 100 runs, with the value after the ± denoting the 95% confidence interval. The communication reported is the number of bits received by the evaluator.

	Time (localhost)		Time (simulated network)	
	Naive	CompGC	Naive	CompGC
AES	4.4 ± 0.0 ms	1.3 ± 0.1 ms	542.6 ± 0.7 ms	66.9 ± 0.1 ms
CBC mode, 10 blocks	45.8 ± 4.0 ms	8.8 ± 0.5 ms	4.8 ± 0.0 s	204.3 ± 0.2 ms
Levenshtein, 30 symbols	28.9 ± 6.6 ms	14.1 ± 0.4 ms	2.2 ± 0.0 s	305.6 ± 0.2 ms
Levenshtein, 60 symbols	109.8 ± 7.0 ms	27.1 ± 0.4 ms	10.6 ± 0.0 s	703.4 ± 1.5 ms

Table 5.2: Experimental results without counting the evaluator time to load data from disk.

	Time (simulated network)		Communication	
	Standard	SCMC	Standard	SCMC
AES	134.4 ± 0.1 ms	68.5 ± 0.2 ms	656 Kb	254 Kb
CBC mode, 10 blocks	321.5 ± 0.9 ms	216.7 ± 0.2 ms	7.4 Mb	2.6 Mb
Levenshtein, 30 symbols	371.0 ± 0.9 ms	315.9 ± 0.5 ms	10.0 Mb	6.3 Mb
Levenshtein, 60 symbols	1119.6 ± 2.1 ms	742.5 ± 2.0 ms	44 Mb	25 Mb

Table 5.3: Comparison of the two approaches for component-based garbled circuits: the standard approach and the SCMC approach. The experiments are run on the simulated network.

Conclusion

Component-based garbled circuits offer a number of benefits in terms of flexibility and speed over other garbled circuits methods. Other garbled circuit methods require either that a function be selected ahead of time in an offline phase or that the garbled circuit be communicated during the online phase. In the former case, the parties performing secure computation lack flexibility, as they must choose their function ahead of time, and as a result, the selected function must be independent of their inputs. In the latter case, communicating a garbled circuit during the online phase is slow, as garbled circuits are quite large.

To solve this problem we propose component-based garbled circuits. Component-based garbled circuits break a large function up into its smaller components. In the offline phase, the parties exchange the smaller components, and then in the online phase, the parties stitch, or chain, together the pre-exchanged components into a function of their choosing. With component-based garbled circuits, the function to be computed may be selected in the online phase without the cost of cost of sending a garbled circuit.

Component-based garbled circuits is the the work of a larger project, in which this thesis contributes two specific items. The first contribution is Single Communication Multiple Connections (SCMC), a method for improving component-based garbled circuits. SCMC cleverly chooses input wire labels and output wire labels to have a predictable pattern, such that fewer link labels are exchanged during the online phase to stitch together the garbled components. SCMC offers the best improvements when large pieces of data are being chained between garbled components. Since SCMC uses a single link label per piece of data, a 10 by 10 matrix requires the same bandwidth as a 100 by 100 matrix. In other words, SCMC makes the online bandwidth of component-based garbled invariant to size of a piece of data but scale linearly with the number of pieces of data (linearly with a small constant).

The second contribution of this thesis is CompGC, an implementation of component-based garbled circuits in a full cryptographically secure system. CompGC performs the entire secure computation protocol from start to finish: the parties perform an offline phase, later select and function in their inputs, and CompGC returns the output to their function. We timed CompGC and found that it is the fastest implementation of garbled circuits in the literature.

Future work in the area of component-based garbled circuits should focus on adapting component-based methods to the malicious setting. In the malicious setting, we assume that the parties may lie, in which case the manner in which the parties exchange links needs to be made more secure. Another area is to expand component-based methods to work with more than two parties. The two party setting is a special case, so some theoretical work needs to be done to make component-based methods work with three or more parties.

The goal of secure computation research is to make methods that are sufficiently fast, flexible and secure. Component-based garbled circuits are a step in this direction: They are faster and generate more flexibility than past garbled circuit protocols. By continuing research in this direction, and extending component-based methods to work in stronger security settings, secure computation may soon be a standard part of the internet operations in the real world. As data becomes more prevalent and security becomes more important, the demand for secure computation will increase; fortunately, flexible, fast and secure protocols are near.

References

[1] Beaver, D. Precomputing oblivious transfer. In *Advances in Cryptology (CRYPT0)*. Springer, 1995, pp. 97–109.

[2] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy (SP)* (2013), IEEE, pp. 478–492.

[3] Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 784–796.

[4] Damgård, I., Pastro, V., Smart, N., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology (CRYPTO)*. Springer, 2012, pp. 643–662.

[5] Demmler, D., Schneider, T., and Zohner, M. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS* (2015).

[6] Even, S., Goldreich, O., and Lempel, A. A randomized protocol for signing contracts. *Communications of the ACM 28*, 6 (1985), 637–647.

[7] Goldreich, O. *Foundations of cryptography: volume 2, basic applications.* Cambridge university press, 2009.

[8] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium* (2011), vol. 201.

[9] Huang, Y., Katz, J., Kolesnikov, V., Kumaresan, R., and Malozemoff, A. J. Amortizing garbled circuits. In *Advances in Cryptology (CRYPTO)*. Springer, 2014, pp. 458–475.

[10] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology (CRYPTO)*. Springer, 2003, pp. 145–161.

[11] Katz, J., and Lindell, Y. *Introduction to Modern Cryptography: Principles and Protocols (Chapman & Hall/CRC Cryptography and Network Security Series)*. Chapman and Hall/CRC, 2007.

[12] Kolesnikov, V., Mohassel, P., and Rosulek, M. Flexor: Flexible garbling for XOR gates that beats free-XOR. In *Advances in Cryptology (CRYPTO)*. Springer, 2014, pp. 440–457.

[13] Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming*. Springer, 2008, pp. 486–498.

[14] Lindell, Y., and Pinkas, B. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality (JPC) 1*, 1 (2009), 5.

[15] Lindell, Y., and Riva, B. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology (CRYPTO)*. Springer, 2014, pp. 476–494.

[16] Lindell, Y., and Riva, B. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *Conference on Computer and Communications Security (SIGSAC)* (2015), ACM, pp. 579–590.

[17] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y., et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium* (2004), vol. 4, San Diego, CA, USA.

[18] Mood, B., Gupta, D., Butler, K., and Feigenbaum, J. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Conference on Computer and Communications Security (SIGSAC)* (2014), ACM, pp. 582–596.

[19] Naor, M., Pinkas, B., and Sumner, R. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce* (1999), ACM, pp. 129–139.

[20] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology (CRYPTO)*. Springer, 2012, pp. 681–700.

[21] Pan, A. Secure computation. http://www.alexirpan.com/2016/02/11/secure-computation.html, 2016.

[22] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *Advances in Cryptology (ASIACRYPT)*. Springer, 2009, pp. 250–267.

[23] Rabin, M. O. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive* (2005), 187.

[24] Rosulek, M. A brief history of practical garbled circuit optimizations. https://www.youtube.com/watch?v=FTxh908u9y8, June 2015.

[25] Schneider, T., and Zohner, M. Gmw vs. Yao? efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security*. Springer, 2013, pp. 275–292.

[26] Smart, N. P., and Vercauteren, F. Fully homomorphic SIMD operations. *Designs, codes and cryptography 71*, 1 (2014), 57–81.

[27] Wikipedia. Wikiepdia levenshtein distance, 2016.

[28] Yao, A. How to generate and exchange secrets. In *Foundations of Computer Science 1986* (1986), IEEE, pp. 162–167.

[29] Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole. In *Advances in Cryptology (EUROCRYPT)*. Springer, 2015, pp. 220–250.