Implementing Improvements for Secure Function Evaluation

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

I want to thank a few people.

Preface

This is an example of a thesis setup to use the reed thesis document class.

List of Abbreviations

You can always change the way your abbreviations are formatted. Play around with it yourself, use tables, or come to CUS if you'd like to change the way it looks. You can also completely remove this chapter if you have no need for a list of abbreviations. Here is an example of what this could look like:

{AL-0: Do I need this?}

ABC	American Broadcasting Company
CBS	Columbia Broadcasting System
CDC	Center for Disease Control
CIA	Central Intelligence Agency
CLBR	Center for Life Beyond Reed
CUS	Computer User Services
FBI	Federal Bureau of Investigation
NBC	National Broadcasting Corporation

Table of Contents

Introduction **1**

Chapter 1: Cryptographic Primitives **3**
1.1 Introducing Cryptographic Security 3
1.2 Encryption 6
1.3 Computational Indistinguishability 9
1.4 Boolean Circuit 10
1.5 Oblivious Transfer 11

Chapter 2: Classic 2PC **13**
2.1 2PC Security Motivation 14
2.2 2PC Security Definition 16
2.3 Yao's Garbled Circuit 19
2.3.1 Security of Garbled Circuits 23
2.3.2 Notes about complexity 25
2.4 GMW 26

Chapter 3: Improving MPC **27**
3.1 Point and Permute 29
3.2 Garbled Row Reduction 3 31
3.3 Free XOR 32
3.4 Garbled Row Reduction 2 34
3.5 FleXOR 34
3.6 Half Gates 36
3.7 Improving Oblivious Transfer 39

Chapter 4: Chaining Garbled Circuits **41**
4.1 The Random Oracle Model and Random Permutation Model 41
4.2 Chaining 41
4.3 Security of Chaining 41
4.4 Single Communication Multiple Connections 41
4.5 Security of SCMC 41

Chapter 5: Implementation **43**
5.1 JustGarble 43

5.2 Our Implementation: CompGC 43
5.3 Adding SCMC 43

Chapter 6: Experiments and Results **45**
6.1 Experimental Setup 45
6.2 Experiments 45
6.3 Results 45

Chapter 7: Future and Related Work **47**

Conclusion **49**

Appendix A: The First Appendix **51**

Appendix B: The Second Appendix, for Fun **53**

References **55**

List of Tables

1.1 The mapping of an XOR gate. 10
1.2 The truth table of the less than circuit. 12

2.1 A garbled table for an AND gate with input wires W_0 and W_1 and output wire W_5. 21

3.1 Summary of Garbled Circuit Improvements. GRR3 stands for garbled row reduction 3 and GRR2 stands for garbled row reduction 2 29
3.2 Garbled Gate for Point and Permute 30
3.3 Example garbled gate using point and permute and garbled row reduction 3. The gate being computed is given in figure **Make it 23:30 in mike's talk** 31
3.4 Generator's Garbled Half Gate for $a = 0$, $a = 1$, and written more succinctly with $a\Delta$ for $a \in \{0, 1\}$. If $a = 0$, then $a\Delta = 0$. Otherwise if $a = 1$, then $a\Delta = \Delta$. 36
3.5 Evaluator's half gate garbled table. 37

List of Figures

1.1 An AND gate. 11
1.2 An XOR gate. 11
1.3 A NOT gate. 11
1.4 A circuit that computes the less or equal to function, equivalent to the less than function for input of two one-bit values.{AL-7: make with tikz} 12

2.1 A high level overview of the garbled circuit protocol 20
2.2 A simple boolean circuit. {AL-13: Draw gate and wire number notation in here} 20

Abstract

The preface pretty much says it all.

Dedication

You can have a dedication here if you wish.

Introduction

{AL-3: in primitives, should I clarify what I mean by "break an algorithm"?} {AL-4: Needs to be redone to emphasize chaining, and the work that I did. Right now, it just generally motivates SFE}

Secure Function Evaluation (SFE) is the study and creation of protocols for securely computing a function between multiple parties. Secure, in this context, means that the protocol prevents each party from learning anything about the other parties.

The idea is best communicated through an example: suppose Alice and Bob are millionaires and wish to determine who is wealthier. But Alice and Bob are also secretive, and do not want to disclose their exact amount of wealth. Is there some method by which they can determine who has more money?

{AL-5: check if using greater than or less than function} Alice and Bob can solve their problem by specifying a function f; in this case f takes two inputs x and y and outputs 1 if $x < y$, and 0 otherwise (f is the less than function). If Alice and Bob can secretly give f their amount of wealth and then retrieve the answer from f, then their problem is solved: they have securely computed who is wealthier.

The goal of SFE broadly is to give Alice, Bob and their friends the ability to securely compute an arbitrary function at their will. For the protocl to be secure, it needs to have the following informal properties:

- **Privacy:** Each party's input is kept secret.

- **Correctness:** The correct answer to the computation is computed.

Originally, the goal of the research community was to develop a secure protocol, define security, and prove the protocol is secure. In more recent times, the focus has shifted to making SFE protocols faster.

If SFE could be made fast enough, it could serve a wide range of applications. For example, imagine that two companies who operate in a similar industry want to work together, but they don't want to disclose any company research which the other doesn't know. These companies could a run set intersection function (a function that given two inputs finds their intersection, or overlap), to determine what information they can disclose without giving away important information.

Another example is running algorithms on confidential medical data. Consider the case where various hosptitals want to joinly run algorithms on their data for medical research. The hosptical need guarantees, for both legal and ethical reasons, that the data will not be disclosed as the algorithm is being run, as medical data can be very sensitive. The hopsitals use could use SFE to run computation on all of their data, mainintaining the privacy of their patients.

Since research into SFE has focused on creating a method by which an arbitrary function can be computed securely, the application of SFE may be beyond what we can presently conceive of. It's not unlikely that SFE will become a standard in the internet, where when you access the internet, behind the scenes your access is being is plugged into an SFE protocol, sent off to another computer to do some processing. For this future to be realized, work need to be done to make SFE faster and more flexible.

Chapter 1

Cryptographic Primitives

Secure Compuation is the study of computing functions in a secure fashion. SC is split 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 will focus primarily 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 outward. 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 Katz & Lindell (2007).

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.

The goal 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 makes 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 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 an 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 a decade ago.

Changing computational power makes it important to scale how hard a cryptographic scheme is to break. 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 correlates to the input-size

of the problem underlying the cryptographic scheme. For example, if the underlying problem is factoring large number N, then $N = \lambda$. As N and λ scale up, the factoring problem becomes more difficult and the scheme becomes hard 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. Formally, a negligible function is:

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 $x > N_p$,

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

Goldreich (1995) ♢

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 is to guess the answer to P. Hence the probability that the adversary that finds the answer the

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. 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, that is k is randomly sampled from $\{0,1\}^{\lambda}$ where λ is the security parameter of the

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

encryption scheme. As per the discussion on security parameters, as λ increases and k grows in length, an 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 say that an encryption scheme is secure if an adversary cannot tell the difference between two messages. We define security using a thought experiment. In the thought experiment, the adversary has access to the encryption algorithm with 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 which they did not give to their ecnryption algorithm and see how they encrypt. The adversay 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 in. Evenutally the adversary must output either 0 or

[3]We select the message uniformly at random.

1 indicating that they think we selected m_0 or m_1 respectively. If the adversary picks correctly, then we say that the adversary wins; otherwise, we say that the adversary loses.

Finall and informally, the encryption scheme is considered secure if the probability that the adversary wins is $\frac{1}{2} + \mu(\lambda)$, 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 neglible 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 $\mathsf{Enc}_k.$, and outputs a pair of messages m_0 and m_1 of the same length.

3. A uniform bit $b \in \{0, 1\}$ is selected, and then a ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$.

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. In the former case, we say that $\mathcal{A}$ succeeds.

♢

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 cipher* (DKC) Bellare et al. (2012). 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 encrytion scheme used to create the DKC is secure, then it is easy to see to that the DKC is also secure. We are formally considering the statement: Enc secure $\implies$ DKC secure. Consider the contrapositive: DKC insecure $\implies$ Enc insecure. If the DKC is insecure, then an adversary can find two messages, m_0 and m_1 such that their ciphertexts are distinguishable. {AL-6: Figure this out, need to break the old intro crypto work}

1.3 Computational Indistinguishability

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

Informally, two probability distribution are indistinguishable if no polynomial-time algorithm can tell them apart. The thought experiment is like this: an algorithm is that there are two distributions, it is given one of the distributions. If the algorithm correctly determines which distribution it was given, then it wins, 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:

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

Katz & Lindell (2007). ♢

We quickly 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 will correspond to the value 1, and False will corresond 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 simpler 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: The mapping of an XOR gate.

A circuit is a combination of gates that are stringed 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 **?**. In other words, if there's some algorithm that can do it, then there is some circuit that can do it as well. Figure 1.4 shows the circuit representation of the less than function, f as specified in equation **??**.

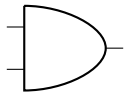

Figure 1.1: An AND gate.

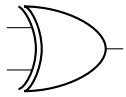

Figure 1.2: An XOR gate.

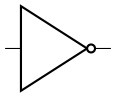

Figure 1.3: A NOT gate.

1.5 Oblivious Transfer

Oblivious Transfer (OT) is a communicating protocol that underlies many more complicated cryptosystems. At the highest level, Alice potentially sends two messages to Bob, and Bob only recieves one of the messages.

Consider the following thought experiment: Alice and Bob want to exchange a message. Alice has two messages m_0 and m_1, and she Alice wants to send one of them Bob but does not care which one. Bob knows that he wants message m_b where $b \in \{0, 1\}$. Alice gives both messages to the trusted post-office. Bob also goes to the trusted post office, and says that he wants m_b. The post office gives m_b to Bob, throws the other message (m_{1-b}) in the trash, and does not tell Alice which message

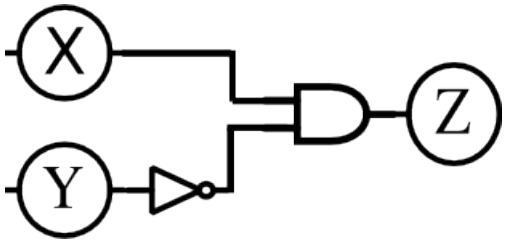

Figure 1.4: A circuit that computes the less or equal to function, equivalent to the less than function for input of two one-bit values.{AL-8: make with tikz}

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

Table 1.2: The truth table of the less than circuit.

they gave to Bob. Now, Alice has sent a message to Bob, but is oblivious as to which message was sent.

Oblivious Transfer is a useful protocol and is a fundamental component of 2PC protocols. Oblivious Transfer protocols guarantee the following security properties:

1. Alice does not know whether Bob recieved m_0 or m_1.

2. Bob does not know anything about m_{1-b}, the message that he did not receive.

Chapter 2

Classic 2PC

{AL-9: this paragraph is a little weak} Secure Computation (SC) was first proposed in an oral presesntation by Andrew Yao **?**. Since Yao's presentation, multiple methods for performing SC were developed. One method was developed by Yao himself and is called garbled circuit. Another was developed by a group of researchers, Goldreich, Micali and Widgerson {AL-10: cite}. The two methods are premised on a similar idea: encrypt a circuit by encrypting its gates, which has since been termed garbled circuit. At this point, it is unclear which method is better, so researchers continue to study both methods. {AL-11: get source}

This chapter explains the two most prominent methods of SC for the two party case, referred to as Two-Party Protocol (2PC). The first part of the chapter will motivate and describe desirable properties of a 2PC protocol, culminating in a definition. The second part of the chapter will describe in Yao's Garbled Circuits, a method for performing 2PC, and discuss security of 2PC. The third part will provide an overview of GMW's method.

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 $y < x$; however, this information about y is deducible from the output 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. In particular, correctness means that Alice and Bob *both* learn the output.

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

One reason is that an import security property that is only relevant in certain cases may be missed. There are many applications of 2PC, and in some cases, there may be certain properties that critical to security. 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. A definition based on properties in this respect as it becomes the burden of the prover of security to show that all relevant properties are covered Lindell & Pinkas (2009).

We must also think about the aims of each party involved in the protocol. Can we trust that parties are going to obey the protocol? It's relevant in the two party case, but if there are more than three parties, parties may either act independently or collude. 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, they do not send or withhold messages out of order, 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 much 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's simply easier to think through and create protocols for the semi-honest setting; at the very least, it's a valuable starting point for building complex cryptosystems. 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 that it does have use cases in the real world. 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 at a high level the definition of 2PC security, 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 2-party computation protocol, that is instructions that Alice and Bob follow, that 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 *essentally the same* information executing Π in the real world as they do computing f in the ideal world with Carlo. Specifically, Alice and Bob should not learn any more information in the real world than they do in the ideal world. If they do learn more information in the ideal world, then the protocol Π is leaking information that cannot be deduced from their individual inputs and the output of f. This is formaly 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 confidently 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 the idea will become clearer as the probability distrubtions are explained

To construct the probability distribution of the ideal world, we introduce *simulators* S_A and S_B. 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 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 input: 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 deduced from all possible x and $f(x, y)$. We think of $\{S_B(y, f(x, y)\}_{y\in\{0,1\}^*}$ for Bob's input similarly.

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 also maintain a record of all intermediate computations. We call Alice's record of intermediate computations Alice's *view*, $\mathsf{view}_A(x, y)$, which depends on inputs x and y And now we create the probability distribution for Alice: $\{\mathsf{view}_A(x, y)\}_{x,y\in\{0,1\}^*}$. This distribution represents the Alice's information from the intermediate computation 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 $\{\mathsf{view}_B(x, y)\}_{x,y\in\{0,1\}^*}\}$

To wrap it up, if Π in the real world is the same as the Alice, Bob and Carlo 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 indstinguishable.

With this intuition in mind, we give Goldreich's definition of 2PC security from his textbook *Foundations of Cryptography Volume II* Goldreich (1995).

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. Also note that for two party computation.

We say that Π securely computes f in the presence of static[1] 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:

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

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

♢

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 **?**.

2.3 Yao's Garbled Circuit

We now discuss a popular 2PC scheme called garbled circuits. At a high level, garbled circuits work by having one party, Alice, design a circuit that computes f. Alice encrypts (garble) that circuit and sends the encrypted circuit to Bob, along with some values corresponding to her and Bob's inputs. Bob decrypts the circuit, acquring the output of $f(x,y)$ at the end.

{AL-15: figure numbering is off} We now walk through how Alice garbles a circuit. She starts with a typical boolean circuit, like the one shown in figure 2.3. In figure 2.3 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 the inputs and gates in the figure. Because Alice is working with

[1] {AL-12: TODO} Mention what static means

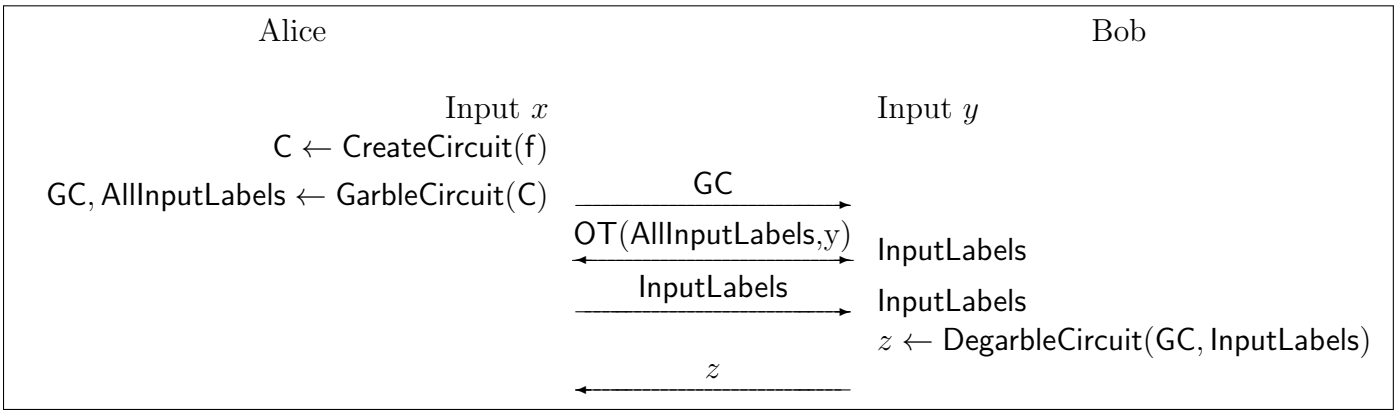

Figure 2.1: A high level overview of the garbled circuit protocol

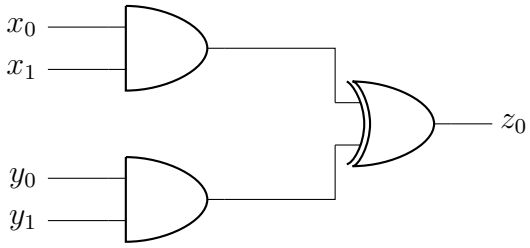

Figure 2.2: A simple boolean circuit. {AL-14: Draw gate and wire number notation in here}

a boolean circuit, each wire can be represented by either a 0 or 1. For a given wire W_i, the zeroith wire label W_i^0 represents 0, and the first wire label W_i^1 represents 1. We use W_i to represent the ith wire, W_i^j to represent the i wire's j label, and later we use W_i^* to represent one of wire i's wire labels (without being specific about whether it is wire label 0 or wire label 1). A wire labels is a ciphertext, the output of the encryption algorithm[2]. Alice assigns the wire labels by randomly sampling from $\{0,1\}^\lambda$, where λ is the size of the output of the encryption algorithm.

After assigning wire labels, Alice first garbles gate G_0, the gate nearest to the inputs[3]. Alice garbles by creating a garbled table T_{G_0} for 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; in this case,

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

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

these are wires W_0 and W_1. The third column is the wire labels for the output wire; this case W_5 The wire label placed in the third column is based on logical operation of the AND gate. For example, the third column and third row has the wire label associated with 0, since AND outputs zero on input of 0 and 1. The fourth column is the dual-key cipher encryption[4] 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 2.3

W_0	W_1	W_5	Encryption
W_0^0	W_1^0	W_5^0	$\mathsf{Enc}_{W_0^0,W_1^0}(W_5^0)$
W_0^1	W_1^0	W_5^0	$\mathsf{Enc}_{W_0^1,W_1^0}(W_5^0)$
W_0^0	W_1^1	W_5^0	$\mathsf{Enc}_{W_0^0,W_1^1}(W_5^0)$
W_0^1	W_1^1	W_5^1	$\mathsf{Enc}_{W_0^1,W_1^1}(W_5^1)$

Table 2.1: A garbled table for an AND gate with input wires W_0 and W_1 and output wire W_5.

Alice creates garbled tables all remaining gates; in this case, G_1 and G_2 remain. She then sends the fourth column all of 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. More formally, say a gate was 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).

In order to decrypt each gate, Bob needs to first acquire the wire lables 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 sematic value of the wire labels.

[4]See section {AL-16: what section?} for information on dual-key ciphers

Recall that Bob's input to the 2PC protocol is $y = y_0 y_1$ 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 has two possible values that she wants to send to Bob: W_2^0 and W_2^1. Alice only wants Bob to acquire one of the values, becuase otherwise he can decrypt multiple rows garbled table. {AL-17: introduce htis OT notation in chapter 1} Bob wants to receive $W_2^{y_0}$, as that wire label corresponds to his input. So Alice and Bob do $\mathsf{OT}(\{W_2^0, W_2^1\}, y_0)$, so that Alice obvliously sends Bob the correct wire label. Alice and Bob also perform OT on wire W_3; in particular, they do $\mathsf{OT}(\{W_3^0, W_3^1\}, y_1)$.

Alice is finished garbling the circuit, and communicating information to Bob. Bob has everything he needs to ungarble, 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 0, which I denote W_0^* since Bob does not 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 zeroith 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 function should indicate whether the decryption was valid. A single additional bit, where 0 represents that decryption failed and 1 represents that decryption was successful. It is noteworth that such a property is common to encrypion schemes, and can be added to any existing encryption if necessary. With this property, as Bob tries decrtyping all four rows of the garbled table, only one should decrypt correctly. Because of the way Alice constructed the garbled table, Bob knows that this correclty decrypted value is one of the wire labels of W_5, the output wire of gate G_0. Bob then assigns this decrypted value to W_5^*, 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 acquired by ungarbling gates G_0 and G_1. Bob notifes Alice after acquiring W_7^*. Alice sends him values W_7^0 and W_7^1. If $W_7^* = W_7^0$, then the function output 0 and Bob notifes Alice that the output was 0. Otherwise if $W_7^* = W_7^0$, then the function output 1 and Bob notifes Alice that the output was 1.

Bob and Alice have now securely computed the function.

{AL-18: Match notation in this description with the algorithms}

Algorithm 1 Garble Circuit

Require: Circuit $f(x, \cdot)$
Ensure: Populate garbled tables $f(x, \cdot).tables$.
  for wire w_i in $f(x, \cdot).wires$ **do**
    Generate two encryption keys, called garbled values, W_i^0 and W_i^1.
    Assign (W_i^0, W_i^1) to w_i.
  end for

  for gate g in $f(x, \cdot).gates$ **do**
    Let w_i be g's first input wire.
    Let w_j be g's second input wire.
    Let w_k be g's output wire.
    for $(u, v) \in \{(0,0), (0,1), (1,0), (1,1)\}$ **do**
      $T_g[u, v] = \mathsf{Enc}_{W_i^u}(\mathsf{Enc}_{W_j^v}(W_k^{g(u,v)}))$ {AL-19: use DKC}
    end for
    $f(x, \cdot).tables[g] = T_g$.
  end for

2.3.1 Security of Garbled Circuits

We now discuss the security of garbled circuits without proof.

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 an ideal 2PC execution with a trusted third party Carlo. This definition is extremely useful; however, it is difficult to adapt it to intuiting the security of a given protocol. To think about the security of garbled circuits, we think about the information that Alice and Bob.

Algorithm 2 Evaluate Circuit

Require: $(input\_wires, tables, gates)$
  for Input wire w_i in $input\_wires$ **do** ▷ retrieve garbled values of input wires
    Perform OT(w_i, x_i) ▷ retrieve $W_i^{x_i}$ from Alice
    Save the value to w_i.
  end for
  for Gate g in $gates$ **do** ▷ compute the output of each gate.
    Let w_i be g's first input wire.
    Let w_j be g's second input wire.
    Let w_k be g's output wire.
    Require w_i and w_j have been assigned garbled values.
    Require w_k has not been assigned a garbled value.
    for $(u, v) \in \{(0,0),(0,1),(1,0),(1,1)\}$ **do** {AL-20: use DKC}
      $temp = \mathsf{Dec}_{w_j}(\mathsf{Dec}_{w_i}(tables[g][u, v]))$
      if $temp$ decrypted correctly **then**
        $w_k = temp$
      end if
    end for
  end for

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 the security of OT, whose security we describe in chapter 1. Hence, we can confidently say that Alice does not learn anything during garbled circuits.

The security of Bob is more complicated. Bob learns three sets of information: a single wire label for each wire (i.e. input labels) and the fourth column of the garbled table for each gate. The input labels are acquired in two ways: some are sent naiively by Alice and some are acquired via OT. We can be confident that Bob doesn't learn any extra informatin 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 on 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 with some thinking learn 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. And that 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.

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 decrypts 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 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!

2.4 GMW

Where Yao's protocol is premised on encrypting gates individually, GMW's protocol for garbling circuits is premised on secret sharing and performing operations on the shared secrets. In general, secret sharing is a class of methods for distributing a secret to a group of participants, where each participants is allocated a *share* of the secret. The secret can only be reconstructed when a sufficient number of the participants combine their shares, but any pool of insufficient shares yields no information about the secret.

GMW begins by having Alice and Bob secret share their inputs, so each party now has a collection of *shares*. Algorithm **??** describes this process in more detail. Then Alice and Bob perform a series of operations on their shares, which are dictated by the gates in the function they wish to compute. As with Yao's protocol, a gate may either compute XOR, AND or NOT. Each operation requires a different series of operations, which are described in Algorithm **??**. Finally, Alice and Bob publicize their shares to each other, at which point each party will have sufficient shares to compute the output of the function.

Chapter 3

Improving MPC

A number of improvements have been made to garbled circuits over the last two decades. The aim of this chapter is to outline the major improvements made to garbled circuits and discuss the costs and benfits of the improvments.

We consider 3 metrics that describe the costs of garbled circuits. First, we consider bandwidth per gate. In the classic scheme of garbled circuits, described in chapter 2, the garbler and evaluator communicated 2 pieces of information: wire labels for input wires and the fourth column of the garbled table for each gate [1]. Sending the input lables is an unavoid able costs, with the major caveat that using oblivious transfer to communicate the evaluator's input labels has substantial costs. Oblivious transfer has been improved to require a small, constant amount of communication and minimal computation requirements for garbler and evaluator, if they can perform some computation *before* executing the protocol. Improvements to oblivious transfer are discussed in more detail below.

The garbler and evaluator also commnicate the garbled table[2]. Classical garbled circuits require that all four rows of the garbled table be communicated, that is 4

[1] In chapter 2, Alice was the garbler and Bob was the evaluator. For clarify in chapter 3 and beyond, we use the term garbler to allude to the person who garbles the circuit, and we use the term evaluator to describe the person who evaluates, or decrypts, the garbled circuit.

[2] In this chapter, the garbled table will be used interchangeably with the fourth column of the garbled table.

ciphertexts. Recent research has reduce the number of ciphertexts to approximately 0.5 depending on the circuit being computed. We generally consider bandwidth to be the most important factor, as the high latency of the internet is generally the bottleneck of executing garbled circuits.

Second, we consider evaluator-side computation, which is determine by the number decryptions the evaluator performs per gate. Third, consider garbler-side computation. Many improvements to MPC increase the amount of garbler-side computation; having the garbler do more work can reduce the size of the garbled table which reduces bandwidth.

As we walk through the improvements to Yao's garbled circuit, we are going to think about the improvements affect the three metrics described above. It should also be noted before we begin that these are improvements to the the computation of a single gate. Speedups to the computation of a gate propagate through the computation of the entire garbled circuit. For example, the computation of AES now requires approximatley $40,000$ gates, so reducing the number of ciphertexts being transmitted by 1 per gate reduces the total ciphertexts communicated by $10,000$.

Table 3 is 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. Size is the size of the garbled table per gate. Eval cost is number of encryptions that the evaluator performs per gate, and decryptions is the number of decryptions that the garbler performs per gate. Each section is divided into two columns: AND and XOR. The AND columns show the requirements for AND gates, and the XOR columns show the cost for XOR gates. The columns are separate, because many of the improvements reduce the costs of one type of gate but not the other.

The goal of this chapter is primarily to explain each row in this table. How the improvement works, and the computational and bandwidth effects of the improvement. We start with the earliest improvement and move chronological to the most

Garbled Circuit Improvement	Size (xλ)		Eval Cost		Garble Cost		Assumption
	XOR	AND	XOR	AND	XOR	AND	
Classical	4	4	1024	μ s	a	b	a
Point and Permute	TBD	TBD	TBD	μ s	a	b	a
GRR3	TBD	TBD	TBD	μ s	a	b	a
Free XOR	TBD	TBD	TBD	μ	a	bs	a
GRR2 XOR	TBD	TBD	TBD	μ	a	bs	a
FleXOR	TBD	TBD	TBD	μ	a	bs	a
Half Gates	2	0	2	0	4	0	a

Table 3.1: Summary of Garbled Circuit Improvements. GRR3 stands for garbled row reduction 3 and GRR2 stands for garbled row reduction 2

recent improvement.

3.1 Point and Permute

{AL-21: add in that rows need to be permuted!}

Beaver, Micali and Rogaway contributed the first major improvement to garbled circuits in 1990. Recall that the garbled table is randomly permuted - that is, the rows of the garbled table are reordered randoml by the garbler before the table is sent to the evaluator [3]. Upon receiving the circuit, the garbler attempts to decrypt each row of the garbled table until a decryption succeeds [4].

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 ciphertext to decrypt. 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 lables of the same wire have opposite bits. That is, if the zeroith wire label has select bit 1, then the first wire label has select bit 0. The garbler permutes the garbled table based on the select bits, and sends the select bits to the evaluator.

[3] This is required for security. See chapter 2 for more information

[4] Recall that the decryption algorithm outputs a single bit indicating whether or not the decryption was successful. For more information, see chapter 1.

Select Bit	Wire Label
0	A_0
1	A_1
1	B_0
0	B_1

Select Bits	Encryption
(0,0)	$\mathsf{Enc}_{A_0,B_1}(C_1)$
(0,1)	$\mathsf{Enc}_{A_0,B_0}(C_0)$
(1,0)	$\mathsf{Enc}_{A_1,B_1}(C_0)$
(1,1)	$\mathsf{Enc}_{A_1,B_0}(C_0)$

Table 3.2: Garbled Gate for Point and Permute

Upon receving the garbled table, the evaluator knows exactly which ciphertext to decrypt based on the select bits of the input wires.

Tables 3.1 show an example of point and permute. A, B and C are wires, where A_0 and A_1 are the zeroith and first wire label of wire A respectively. The left table shows the select bits of the input wires A and B. The garbler gives A_0 select bit 0, determining that A_1 has select bit 1. Likewise, the garbler gives B_0 select bit 1, determining that B_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 A_* and B_* with select bits a and b. The evaluator decrypts the ciphertext in the row corresponding a and b.

Intuitively, point and permute is secure because the select bits are independent of the truth value (also known as semantic value) of a the wire. This allows the garbler to permute the table based on the select bits, and the garbler can send the evaluator the select bits.

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 per gate: a small constant increase. [5]

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

Select Bit	Wire Label
0	A_0
1	A_1
1	B_0
0	B_1

Select Bits	Encryption
(0,1)	$\mathsf{Enc}_{A_0,B_0}(C_0)$
(1,0)	$\mathsf{Enc}_{A_1,B_1}(C_0)$
(1,1)	$\mathsf{Enc}_{A_1,B_0}(C_0)$

$C_0 \leftarrow \{0,1\}^n$
$C_1 \leftarrow \mathsf{Enc}^{-1}_{A_0,B_1}(0^n)$

Table 3.3: Example garbled gate using point and permute and garbled row reduction 3. The gate being computed is given in figure **Make it 23:30 in mike's talk**

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. In classical garbled circuits, the wire labels for each wire are chosen prior to garbling any gates. In GRR3, the input wire labels are sampled in the beginning, and the other wire labels are chosen as each garbled table is created.

Suppose that a gate with input wires A and B and output wire C is being garbled. In GRR3, the garbler uses the point and permute method. After the garbler samples select bits and permutes the garbled table, they set the ciphertext in the top row of the garbled table equal to a value that decrypts to 0^n, the string of n zeros. That is, C_*, the wire label on the top row, is set to $\mathsf{Enc}^{-1}_{A_*,B_*}(0^n)$. 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 first row, the evaluator simply assumes the ciphertext be of value O^n. Otherwise, the evaluator decrypts the indicated row of the garbled table as per usual.

Tables 3.3 gives an example of garbling an XOR gate. The left table shows the select bits of wire labels A_0, A_1, B_0 and B_1. The right table shows the garbled table, in which the top row, the row associated with select bits $(0,0)$, is missing. The bottom table shows the values of C_0 and C_1. The value of C_0 is randomly sampled from $\{0,1\}^n$.

{AL-22: add more to this paragraph} In considering the security of GRR3, we consider the effect of always setting one the wire labels, C_*, to 0^n. Since the evaluator does not know whether C_0 or C_1 is set to 0^n, the evaluator does not learn any information.

GRR3 offers good performance benefits. It increases garbler side computation by requiring an additional decryption operation. However, evaluator perform slightly less computation: in the event that the first row is to be decrypted, the evaluator does not perform the decryption algorithm. This events occur with probabilty $\frac{1}{4}$, so we can extrapolate that the evaluator performs $\frac{1}{4}$ fewer decryptions. Finally, GRR3 reduces the size of the garbled table from 4 ciphertexts to 3 ciphertexts, a 25% reduction.

3.3 Free XOR

The Free XOR technique makes the computation of XOR gates free, in the sense that no garbled table need to be communicated. The evaluator can compute C_* from only A_* and B_*.

Like GRR3, the free xor techniques 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 A, let A_0 be randomly sampled from $\{0,1\}^n$ as before, and set $A_1 = A_0 \oplus \Delta$.

If the garbler is garbling an XOR gate, then the garbler does not constructed a garbled table or use select bits. The garlber sets $C_0 = A_0 \oplus B_0$ and sets $C_1 = C_0 \oplus \Delta$.

The evaluator, when evaluating an XOR gate, simply computes $C_* = A_* \oplus B_*$. As simple as it is, the evaluator will always acquire the correct value for C_* based on

the semantic value of A_* and B_*. The math for each of the four cases is shown:

$$A_0 \oplus B_0 = C_0$$
$$A_0 \oplus B_1 = A_0 \oplus (B_0 \oplus \Delta) = (A_0 \oplus B_0) \oplus \Delta = C_1$$
$$A_1 \oplus B_0 = (A_0 \oplus \Delta) \oplus B_0 = (A_0 \oplus B_0) \oplus \Delta = C_1$$
$$A_1 \oplus B_1 = (A_0 \oplus \Delta) \oplus (B_0 \oplus \Delta) = (A_0 \oplus B_0) = C_0$$

For any wire A, since A_1 is dependent on A_0, we often simplify notation such that the wire labels for A are A and $A \oplus \Delta$, omitting the subscript.

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.

{AL-23: discuss security. Why is free xor secure?} One interesting implication of using the Free XOR technique is that an added assumption must be made to our encryption algorithm. Since Δ is part of the key and part of the the payload[6] 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. But since XOR gates are much 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 (while minimizing the size of the entire circuit of course). The garbler-side computation is reduced: constructing the xor garbled table does not require 3 encryptions and 1 decryption. Evaluator-side computation is reduced likewise: xor gates do not requre any decryption.

[6]The payload is the value that is being encrypted

3.4 Garbled Row Reduction 2

Garbled row reduction 2 (GRR2) reduces the size of the garbled table of AND gates to 2 ciphertexts. Unfortunatley, GRR2 is not compatible with Free XOR, as it require that the zeroith and first wire labels of each wire bear a specific relationship. Because GRR2 is incompatible with FreeXOR, it is not often used in practice.

{AL-24: I don't think much more needs to be said, but this should be clearer} GRR2 is much different from GRR3. When garlbing a gate, GRR2 creates two third degree polynomials. One polynomial corresponds to C_0, and the other polynomial corresponds to C_1. As such, that C_0 polynomial is defined by the wire labels that result in C_0, and likewise the C_1 polynomial is defined by the wire labels that result in C_1. Since each polynomial is of degree three, they are generated by 3 points. The garbler sends over 2 points of each polynomial, and the final point is supplied by the input wire labels to the gate. Upon acquriring three points for one of the polynomials, the evaluator interpolates the polynomial, and plugs in values to recover C_*.

3.5 FleXOR

After the creation of GRR2, secure computation was at an awkward point. Circuits with many XOR gates were comptuted most quickly with Free XOR, but circuits with many AND gates were computed most quickly with GRR2. FleXOR reconciles GRR2 with Free XOR, giving researchers a technique that is universally faster.

GRR2 is incompatiable with FreeXOR because the wire labels on output wires of AND gates the result of plugging a value into a polynomial. The wire labels are solely dependent on the polynomial, so that $C_1 = C_0 \oplus \Delta_C$ for some random Δ_C not the global Δ used for FreeXOR. 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. [7]

Suppose an XOR gate has input wires A and B and output wire C. Input wires A and B each come from an AND gate, so their labels are the result of the polynomial interpolation of GRR2. A has labels A and $A \oplus \Delta_A$ and B has labels B and $B \oplus \Delta_B$. To perform Free XOR, A, B and C need to be using the same delta value. The garbler adds an extra gate between A and B and the XOR gate that adjusts their XOR value to the correct value. A and $A \oplus \Delta_A$ change to A' and $A' \oplus \Delta$. B and $B \oplus \Delta_B$ change to B' and $B' \oplus \Delta$. Since A, B and C have the same delta value, FreeXOR is used.

The unary gate maps $A, A \oplus \Delta_1 \rightarrow A', A' \oplus \Delta$, where Δ is the correct delta value for the XOR gate.

FreeXOR can be improved by not corrected every output wire of an AND gate. For example, if an output wire of an AND gate is immiedatley inputted into another AND gate, it does not need to fixed. Moreover, the wires do not need to be corrected to a global delta. Free XOR only requires that the three wires involved in the gate, A, B and C, use the same delta, so each XOR gate has its own Δ that is uses. For example, A and C may share a delta but B may have a different delta, so it is sufficient to only correct B's delta value.

One issue that FleXOR raises is that circuits can be optimized to perform quickly in it. FleXOR is fastest when AND gates are grouped together and XOR gates are grouped together, since fewer unary gates will be required. This optimization turns out to be NP-hard, but fortunately, FleXOR works well without much (or any) optimization to the circuit. Reserach reveals that FleXOR requires on average an extra 0 or 1 ciphertext per gate. That cost comes at the benefit of a Free XOR circuit, and 1 fewer ciphertext for each AND gate.

FleXOR requires slightly more garbler-side computation than FreeXOR and GRR2,

[7] 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.

since the garbler must create the unary gates. The size of the XOR garbled table is 0, and the size of the AND garbled table is 2, with the addition of the garbed table of the unary gate. The garbled table of the unary gate has 2 ciphertexts, and the number of unary gates depends on the circuit. The evaluator-side computation is the same as FreeXOR and GRR2, with the additional computation of the unary gates, which is small.

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 that an input is missing. However, FreeXOR requires the circularity assumption, The circularity assumption is discussed in the section on FreeXOR.

3.6 Half Gates

Half Gates is the most recent improvement to garbled circuits. The goal of Half Gates is to make AND gates cost two ciphertexts, while preserving properties necessary for Free XOR without adding unary gates.

We consider the case of an AND gate $c = a \wedge b$, where the generator knows the value of a. If $a = 0$, then the generator will create a unary gate that always outpost false, c. Otherwise if $a = 1$, then the generator create a unary identity gate that always outputs b. Table 3.6 shows the garbled gates for different values of a.

Garbled Table for $a = 0$	Garbled Table for $a = 1$		Garbled Table for any a
$H(B) \oplus C$	$H(B) \oplus C$	$\rightarrow$	$H(B) \oplus C$
$H(B) \oplus C$	$H(B) \oplus C \oplus \Delta$		$H(B) \oplus C \oplus a\Delta$

Table 3.4: Generator's Garbled Half Gate for $a = 0$, $a = 1$, and written more succinctly with $a\Delta$ for $a \in \{0, 1\}$. If $a = 0$, then $a\Delta = 0$. Otherwise if $a = 1$, then $a\Delta = \Delta$.

Since the evaluator has the wire label corresponding to b (either B or $B \oplus \Delta$), the evaluator can compute the label of the output wire of the AND gate by xoring the

rows in the garbled table by $H(B)$ or $H(B \oplus \Delta)$ to get C or $C \oplus \Delta$ respectively.

A further improvement can be garnered by using the garbled row-reduction trick. We choose C such that the first of the top row of the garbled table is the all zeros ciphertext. The top row may not necessarily be $H(B) \oplus C$, since for security the rows are permuted. Because the top row is all 0s, it does not need to be sent to the evaluator. If the evaluator should decrypt the cipher text on the top row (as directed by point and permute), then the evaluator assumes the cipher text to be all 0s. Overall, computing $a \wedge b = c$ requires two having operations by the generator, a single hash operation by the evaluator, and the communication of one ciphertext.

We now consider computing $a \wedge b = c$, where the evaluator somehow already knows the value of a. If $a = 0$, then the evaluator should acquire C. Otherwise if $a = 1$, then the evaluator should acquire $C \oplus b\Delta$, in which case it is sufficient for the evaluator to obtain $\Omega = C \oplus B$ (then xor Ω with the wire label corresponding to b). Table 3.6 shows cipher texts which the garbler gives to the evaluator.

This table is different from other garbled tables. First, the table does not need to be permuted. Secondly, evaluation is different. If $a = 0$, then the evaluator uses wire label A to decrypt the top row of the table and acquire C. If $a = 1$, then the evaluator uses $A \oplus \Delta$ to decrypt the second row, yielding $C \oplus B$. The evaluator then xors wire label $B + b\Delta$ by $C \oplus B$ to obtain $C \oplus b\Delta$. As before, the ciphertext on the top row does not need to be communicatd by using the garbled row-reduction trick. The generator sets C such that $H(A) \oplus C = 0$ (i.e. $C = H(A)$). The total cost of this half gate is the same as before: two hashes by the generator, one hash by the evaluator, and once cipher text.

Garbled Table for any A
$H(A) \oplus C$
$H(A \oplus \Delta) \oplus C \oplus B$

Table 3.5: Evaluator's 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 generator:

$$a \wedge b = a \wedge (r \oplus r \oplus b) = (a \wedge r) \oplus (a \wedge (r \oplus b)). \tag{3.1}$$

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

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

The overall cost of using Half Gates for AND gates is four calls to H for the generator, two calls to H for the evaluator, and the communication of 2 ciphertexts. Half Gates guarantees only two ciphertexts are needed per AND gate, but the tradeoff is the additional computation for both parties (i.e. computing H). With FleXOR, the number of ciphertexts that need to be communicated may vary, but there is less computation required.

When secure computation becomes used in real operations, the circuit will likely not be optimized for XOR gates (reducing the number of AND gates). In this case, where an arbitrary function is being computed, it is hypothesized that Half Gates will outperform FleXOR. But in many of the cases being examined now - most notably computing AES - the circuit has been optimized heavily to have very few AND gates, hence the benefits of Half Gates are less noticeable.[8]

[8] At present, the AES circuit is 80% XOR gates

3.7 Improving Oblivious Transfer

{AL-25: Fix this up} Oblivious transfer has been improved in two relevant ways for 2PC protocols. These improvements are called OT-extension and OT-preprocessing. In OT-extension a constant number of OT's can be run to generate a polynomial number of exchanged values. In OT-preprocessing the OT's occur before the actual protocol, during what is called the offline phase, and then when the OT's needed during the online phase, simpler and efficient communication realizes the exchanged values. This is useful because the OT step requires a large amount of communication to be sent between the sender and receiver, often resulting in OT being the bottleneck of 2PC protocols.

Chapter 4

Chaining Garbled Circuits

the "theory"

talk about our idea about chaining here.

4.1 The Random Oracle Model and Random Permutation Model

4.2 Chaining

4.3 Security of Chaining

4.4 Single Communication Multiple Connections

4.5 Security of SCMC

Chapter 5

Implementation

talk about my implementation here. be clear about what I did.

5.1 JustGarble

5.2 Our Implementation: CompGC

5.3 Adding SCMC

Chapter 6

Experiments and Results

talk about experiments and results here

6.1 Experimental Setup

talk about what computer was used, bandwidth, network simulation

6.2 Experiments

AES, CBC, Levenshtein

6.3 Results

Chapter 7

Future and Related Work

talk about future and related work here

Conclusion

Here's a conclusion, demonstrating the use of all that manual incrementing and table of contents adding that has to happen if you use the starred form of the chapter command. The deal is, the chapter command in LaTeX does a lot of things: it increments the chapter counter, it resets the section counter to zero, it puts the name of the chapter into the table of contents and the running headers, and probably some other stuff.

So, if you remove all that stuff because you don't like it to say "Chapter 4: Conclusion", then you have to manually add all the things LaTeX would normally do for you. Maybe someday we'll write a new chapter macro that doesn't add "Chapter X" to the beginning of every chapter title.

Appendix A

The First Appendix

Appendix B

The Second Appendix, for Fun

References

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

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

Boneh, D. (1998). *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, chap. The Decision Diffie-Hellman problem, (pp. 48–63). Berlin, Heidelberg: Springer Berlin Heidelberg. http://dx.doi.org/10.1007/BFb0054851

Goldreich, O. (1995). Foundations of cryptography.

Goldreich, O., Micali, S., & Wigderson, A. (1987). How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, (pp. 218–229). New York, NY, USA: ACM. http://doi.acm.org/10.1145/28395.28420

Katz, J., & Lindell, Y. (2007). *Introduction to Modern Cryptography: Principles and Protocols (Chapman & Hall/CRC Cryptography and Network Security Series)*. Chapman and Hall/CRC. http://www.amazon.com/Introduction-Modern-

Cryptography-Principles-Protocols/dp/1584885513%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D1584885513

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

Snyder, P. (????). Yaos garbled circuits: Recent directions and implementations.