On Garbled Circuits

Ignacio Navarro

Imperial College London

Department of Computing

Supervisor: Mahdi Cheraghchi

June 2018

Abstract

A very useful cryptographic tool is to allow distrusting parties to jointly compute a function revealing the output while keeping the input private. This tool is commonly known as secure multi-party computation (SMC), and was originally posed by Andrew Yao in 1982.

Since then, many solutions have been proposed including using secret sharing schemes, homomorphic encryption, or garbled circuits. In this thesis we look deeper into garbled circuits, an approach that breaks down a function into a boolean circuit to allow a finer grained manipulation of the function, yielding an elegant solution to the original problem. We look at the history, theory, and optimizations in the last few years of garbled circuits while also proposing a practical implementation.

Contents

Chapter 1

Introduction

1.1 Motivation

In 1982 Andrew Yao posed a relatively simple problem that informally introduced secure two-party computation. Two millionaires, Alice and Bob, wish to know who is richer without revealing their wealth to each other. This could be easily solved using a trusted third party Charlie, whereby Alice and Bob send their personal wealth's amount to Charlie and Charlie sends back the result. The goal that Yao posed, however, was to accomplish this without the use of any trusted third party. In his seminal works [13, 14] he not only posed this problem but gave an elegant solution that inspired what later became known as *garbled circuits*. The secure two-party case was soon generalized to the multi-party case by Goldreich, Micali and Widgerson [5], and a new subfield of cryptography was born.

More formally, the initial motivation Yao sought was letting two parties reveal a function's output while keeping the input private. In other words, if we have party P_X, party P_Y, and a function $f : X \times Y \to Z$ for some sets X, Y and Z, then we would like to know for any $x \in X$ supplied by P_X and any $y \in Y$ supplied by P_Y the value of $f(x, y)$ without P_X knowing y and without P_Y knowing x. For the Millionaire's Problem, $X = Y = \mathbb{R}_{\geq 0}$, $Z = \{0, 1\}$, P_X is Alice, P_Y is Bob, x and y are their respective wealth, and f is the function that outputs true if $x > y$, and false otherwise.

Up until recently, all advancements in the area were theoretical in nature; that is, focusing on feasibility instead of practicality. This changed in the last few years, however, with the first practical implementation of garbled circuits given by Malkhi et al. [10] in their Fairplay system, capable of solving the millionaire's problem in approximately 1 second or finding the median of two sorted arrays in 7 seconds. With computers and CPUs getting faster and with each refinement proposed on garbled circuits, secure two party computation is close to being a widely used cryptographic tool in our everyday lives.

1.2 Goals

The following goals given in [9] apply to any secure multi-party computation scheme, but apply for garbling schemes.

- **Privacy**: No party should learn anything more than its prescribed output. Note that any deduction the parties derive from the output is fine. For instance, as Lindell et al. explain, in an auction where the only bid revealed is that of the highest bidder, we can immediately learn that the other bids were lower than the winning bid. However, this should be the only information revealed about the losing bids.

- **Correctness**: No party can change the output, guaranteeing the final output is correct. This is less specific to garbled circuits, as we assume a semi-honest model where the parties follow the protocol, but it is worth mentioning nevertheless.

- **Independence of Inputs**: Corrupted parties must choose their inputs independently from the inputs of the honest parties.

- **Guaranteed Output Delivery**: Corrupt parties can't perform a denial of service attack to honest parties and deny them of learning the output.

- **Fairness**: A corrupt party may learn the output if and only if an honest party learns the output.

1.3 Structure of Thesis

Chapter 1 introduces garbled circuits and the ultimate goal they try to achieve. Chapter 2 will go into more detail on the original construction of garbled circuits, formally defining garbled circuits and the underlying security. Chapter 3 explains all the optimizations proposed in the last few years on garbled circuits, focusing primarily on size of the garbled table as a benchmark to optimize. Chapter 4 will finally focus on our implementation, detailing past known implementations and what our new implementation offers. Chapter 5 concludes the work with some final thoughts and remarks.

Chapter 2

Yao's Garbled Circuits

2.1 Yao's Solution the Millionaire's Problem

Yao not only posed the Millionaire's problem in 1982, but he also gave a simple solution that became a precursor to garbled circuits. Suppose Alice has i millions and Bob has j millions, where $1 < i, j < 10$ for simplicity. Let $M = \{0,1\}^N$, and let $Q_N = \{f : M \to M |\, f \text{ is a bijection}\}$, i.e., the set of bijections from M to itself. Let $E_A \in Q_N$ be Alice's public key, and let $E_A^{-1} = D_A \in Q_N$ be the private key (one must exist, as E_A is a bijection).

1. Bob picks a random $x \in M$, computes privately the value $k = E_A(x)$, and sends Alice $t = k - j + 1$.
2. Alice computes privately the values $y_u = D_A(t + u)$ for $u \in \{1, 2, ..., 10\}$.
3. Alice generates a random prime p of $N/2$ bits, and then computes $z_u \equiv y_u \mod p$. If all z_u differ by at least two, then she stops. Otherwise, she repeats the process with another p.
4. Alice sends the pair $(p, \{z_1, z_2, ..., z_i, z_{i+1} + 1, ..., z_{10} + 1\})$. In other words, she adds 1 to every element after the i'th place.
5. Bob looks at the j'th number in the list z_j, and concludes that he is wealthier if $x \not\equiv z_j \mod p$, and Alice is wealthier if $x \equiv z_j \mod p$.
6. He communicates the final result to Alice.

2.2 Oblivious Transfer

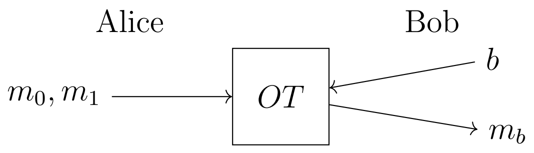


Figure 2.1: 1-2 Oblivious transfer

Before delving into garbled circuits, we will need to use 1 out of 2 oblivious transfer (OT), a cryptographic protocol that exchanges selective information between two parties. That is, say Alice has two secret messages m_0 and m_1, and Bob has a secret bit b. Then 1-2 oblivious transfer allows Bob to learn m_b without Bob knowing both messages and without Alice knowing which message Bob chose (see Figure 2.1). A simple OT protocol would for example be:

1. Alice generates two asymmetric key pairs, $(pub_0, priv_0)$ and $(pub_1, priv_1)$.
2. Bob generates a symmetric key K and with his bit b encrypts K as $c = \text{Enc}_{pub_b}(K)$.
3. Alice decrypts both $c_0 = \text{Dec}_{priv_0}(c)$ and $c_1 = \text{Dec}_{priv_1}(c)$. Notice one of the two will correctly decrypt K, but she doesn't know which. Say for example, $c_0 = K$.
4. Alice sends $c_0' = \text{Enc}_{c_0=K}(m_0)$ and $c_1' = \text{Enc}_{c_1}(m_1)$.
5. Bob just decrypts both messages $\text{Dec}_K(c_0') = \text{Dec}_K(\text{Enc}_K(m_0)) = m_0$ and $\text{Dec}_K(c_1')$ which will just be rubbish.

Although an unfit choice for our implementation (what happens if Alice cheats and chooses $m_0 = m_1$?), it is simple enough to illustrate the purposes of oblivious transfer. However, in our implementation we will choose a stronger 1-2 oblivious transfer.

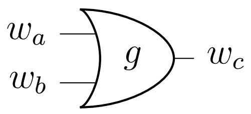


Figure 2.2: An OR gate

w_a	w_b	w_c
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.1: OR truth table

2.3 Boolean Circuits

A *gate* g is essentially a function $g : \{0,1\} \times \{0,1\} \to \{0,1\}$ that has two input wires and one output wire. A *wire* is a formal way of stating the set $\{0,1\}$, i.e., a wire can take the value 0 or 1. In the case of Figure 2.2, we have that the wires are w_a, w_b and w_c, and that g is an OR gate, the function represented in Table 2.1.

More formally, if we take the definition from Bellare *et al.* from [4], a circuit is a 6-tuple $f = (n, m, q, A, B, K)$, where $n \geq 2$ is the number of inputs to the circuit, $m \geq 1$ is the number of outputs to the circuit, and $q \geq 1$ is the number of gates. Therefore, if we have n inputs and q gates, we have $r = n+q$ number of wires. Now we label each input from the set of inputs $I = \{1, ..., n\}$, each gate from the set of gates $G = \{n+1, ..., r\}$ and note that the the set of wires $W = \{1, ..., r\}$ can be partitioned into the set of input wires $I = \{1, ..., r-m\}$ and output wires $O = \{r-m+1, ..., r\}$ to form $W = I \cup O$. Then the functions $A, B : G \to I$ identify each gate's first and second incoming wires respectively. Finally, $K : G \times \{0,1\}^2 \to \{0,1\}$ determines the functionality of each gate. Note that by definition we require $A(g) < B(g) < g$ for any gate $g \in G$, as otherwise we could have a cyclic graph.

2.4 Yao's Protocol

We are now in a position to define garbled circuits. In its core, a *garbled circuit* is a boolean circuit that has its truth table obfuscated. The elegance resides on *how* this

w_a	w_b	w_c
0	0	0
0	1	1
1	0	1
1	1	1

Before encryption

w_a	w_b	w_c
k_a^0	k_b^0	k_c^0
k_a^0	k_b^1	k_c^1
k_a^1	k_b^0	k_c^1
k_a^1	k_b^1	k_c^1

After encryption

Table 2.2: Second step of garbling

circuit is obfuscated to allow the parties to compute the output while keeping the input private. The protocol has two roles: a garbler and an evaluator.

- **Garbler**:

  1. The garbler, Alice, will first convert a function into a boolean circuit.
  2. For each possible `true` or `false` value in each wire of the circuit, she will encrypt this value and call the encrypted value a *label* or *key*. For instance, for a circuit consisting of a single `OR` gate, the garbler will do what's depicted on Table 2.2.
  3. Now she needs the evaluator, Bob, to have the power to reach $k_c^{g(j,k)}$ from having only the two labels k_a^j and k_b^k, where g is the gate function. If all keys were created at uniformly and at random, then there would be no way for the evaluator to deduce k_c^0 from k_a^0 and k_b^0 in the case of Table 2.2. So we need a "bridge" function f that goes from the input labels to the output labels of the form

     $$f : k_a^i \times k_b^j \to k_c^{g(i,j)}$$

     with the strict condition that f is a bijection and that furthermore, no other output label can be learned other than the one intended by f. In the original protocol, we set f to be

     $$f(k_a^i, k_b^j) = \text{Enc}_{(k_a^i, k_b^j)}(k_c^{g(i,j)})$$

     We will see in later chapters that optimizations on garbled circuits optimize precisely f to achieve faster results.

4. Alice computes f for every possible combination, scrambles the order she will present f to the evaluator and sends this to Bob. Again, if we refer to the example of an `OR` gate, Alice will send for instance the following to Bob:

$$\begin{aligned}
c_{10} &= f(k_a^1, k_b^0) = \text{Enc}_{(k_a^1, k_b^0)}(k_c^1) \\
c_{11} &= f(k_a^1, k_b^1) = \text{Enc}_{(k_a^1, k_b^1)}(k_c^1) \\
c_{01} &= f(k_a^0, k_b^1) = \text{Enc}_{(k_a^0, k_b^1)}(k_c^1) \\
c_{00} &= f(k_a^0, k_b^0) = \text{Enc}_{(k_a^0, k_b^0)}(k_c^0)
\end{aligned}$$

- **Evaluator**:

1. Bob, the evaluator, will request via Oblivious Transfer (see section 2.2) either k_b^0 or k_b^1, but Alice will not know which one he will receive due to the properties of OT.
2. Bob will also receive, along the four ciphertexts c_{ij}, Alice's choice k_a^i. If we recap what Bob possesses, he has, say, k_b^1, k_a^0 (of course he will not know that k_a^0 actually represents `false`), and four ciphertexts **in some order** c_{ij}.
3. To recover $k_c^{g(0,1)}$, he simply needs to apply the inverse of f. In the original protocol, this would be

$$f^{-1}(c_{ij}) = \text{Dec}_{(k_a^0, k_b^1)}(c_{ij})$$

Notice that there are some caveats with this approach. The first is that the evaluator doesn't know which ciphertext corresponds to c_{01}. He therefore needs to decrypt the four ciphertexts. Therefore, he performs

$$\begin{aligned}
f^{-1}(c_{10}) &= \text{Dec}_{(k_a^0, k_b^1)}(c_{10}) = \text{rubbish} \\
f^{-1}(c_{11}) &= \text{Dec}_{(k_a^0, k_b^1)}(c_{11}) = \text{rubbish} \\
f^{-1}(c_{01}) &= \text{Dec}_{(k_a^0, k_b^1)}(c_{01}) = k_c^1 \\
f^{-1}(c_{00}) &= \text{Dec}_{(k_a^0, k_b^1)}(c_{00}) = \text{rubbish}
\end{aligned}$$

This leads to the second caveat: our encryption scheme needs a way to know decryption has succeeded. How will Bob know otherwise that he has decrypted the correct c_{01}? In other words, how can Bob tell apart rubbish from k_c^1?

4. Bob, once he learns k_c^1, sends it back to Alice, who knows if k_c^1 represents `true` or `false`, and communicates the result back to Bob. In the case of a circuit, Bob reuses k_c^1 as the input to the next gate in the circuit, and so on. He only sends back the final output of the circuit to Alice.

The communication between Alice and Bob in the protocol can be summarized in Figure 2.3.

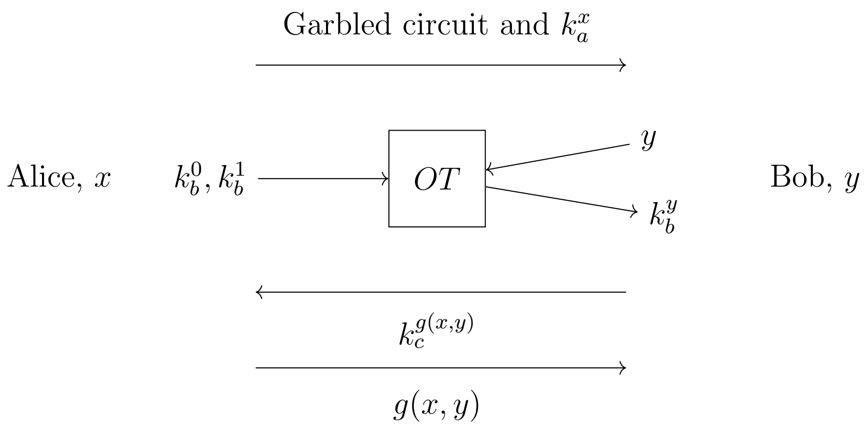


Figure 2.3: Communication summary

2.5 Formalizing Yao's Protocol

To begin proving properties of garbled circuits or to simply improve the scheme, we need to formalize the protocol in section 2.4. The first paper to tie in all the literature concerning garbled circuits was Foundations of Garbled Circuits, written by Bellare *et al.* [4]. They abstract garbled circuits and formalize what is later known as a *garbling scheme*. We now detail their definition.

2.5.1 Garbling Schemes

A garbling scheme is a five tuple of algorithms $\mathcal{G} = (\texttt{Gb}, \texttt{En}, \texttt{De}, \texttt{Ev}, \texttt{ev})$:

- $\texttt{Gb}$ (Garble): This algorithm is the only one probabilistic in nature (the remaining algorithms are deterministic). It takes as inputs a security parameter k and a function f and returns a triple (F, e, d). F is the function (or circuit) representing f, e is an encoding function, and d is a decoding function. The key part of the definition is that we have for any $x \in \{0,1\}^n$:

  $$f(x) = d(F(e(x)))$$

  For the input x, $X = e(x)$ transforms the input into a garbled input, $Y = F(X)$ evaluates the garbled input on the garbled function F, and $y = d(Y)$ turns the garbled output into the final output y. Of course, y must equal $f(x)$.

- $\texttt{En}$ (Encode): In fact, we have already mentioned the encoding function, so this algorithm simply takes the function e and the input x and outputs $X = e(x)$, transforming any input into a garbled input:

  $$\texttt{En}(e, x) = e(x) = X$$

- $\texttt{Ev}$ (Evaluate): This algorithm takes the garbled function F and a garbled input X and outputs a garbled output Y:

  $$\texttt{Ev}(F, X) = F(X) = Y$$

- $\texttt{De}$ (Decode): Similarly, this algorithm takes a decoding function d and a garbled output Y and returns the final output y:

  $$\texttt{De}(d, Y) = d(Y) = y = f(x)$$

- $\texttt{ev}$ (evaluate): This algorithm is the one that evaluates $f(x)$:

  $$\texttt{ev}(f, x) = f(x)$$

  With all of these algorithms we reach the correctness condition: For any $x \in \{0,1\}^n$

  $$\texttt{De}(d, \texttt{Ev}(F, \texttt{En}(e, x))) = \texttt{ev}(f, x)$$

Bellare *et al.* give an illustrative diagram of a garbling scheme in Figure 2.4.

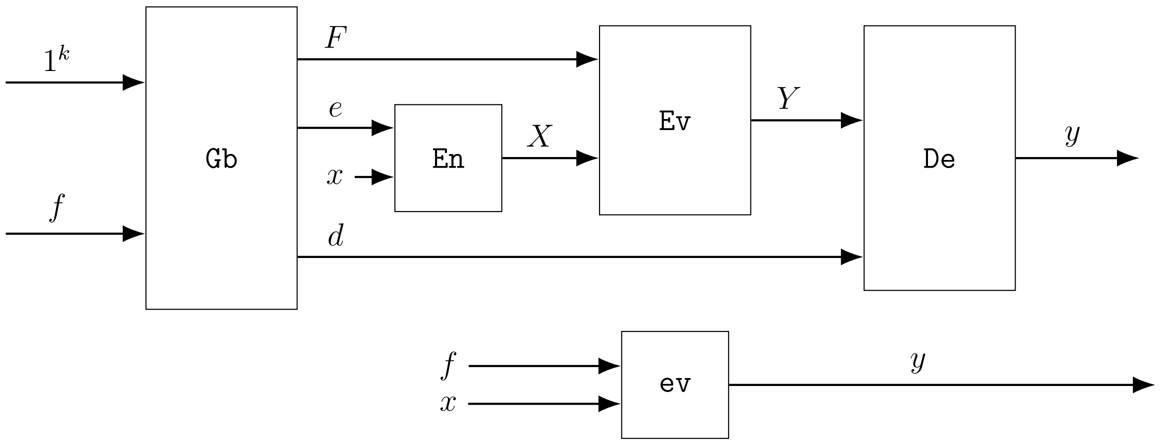


Figure 2.4: Components of a garbling scheme

2.5.2 Security of garbling schemes

There are three main notions of security in garbling schemes proposed by Bellare *et al.* [4] and summarized in [16].

- **Privacy**: privacy needs to ensure that (F, X, d) doesn't reveal any more information about x than $f(x)$. Precisely, there must exist a simulator S that takes input $(1^k, f, f(x))$ and whose output is indistinguishable from (F, X, d).
- **Obliviousness**: (F, X) should reveal no information about x. Precisely, there must exist a simulator S that takes input $(1^k, f)$ and whose output is indistinguishable from (F, X).
- **Authenticity**: Given only (F, X), no adversary should be able to produce $Y' = \texttt{Ev}(F, X)$ such that $\texttt{De}(d, Y') = y$, except with negligible probability.

Chapter 3

Optimizations

3.1 Parameters to optimize

3.1.1 Size

The first parameter to optimize is the size of the garbled table Alice sends Bob. In the classical Yao's protocol, this number was four, one entry in the table for each possible boolean pair. We will later see how various methods decrease the number of entries needed to send over the communication channel.

Most of the research in recent years has focused on optimizing this parameter, and we will therefore go deeper into the optimizations that have reduced the size of the garbled table.

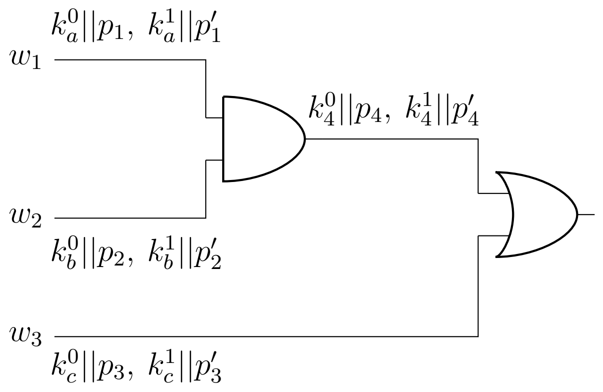
3.1.2 Computation

While the size of the garbled table can be reduced, this can come at a cost of having the evaluator Bob compute CPU-intensive calculations. The aim of this parameter is to reduce the computation needed in order for Bob to evaluate the circuit.

3.1.3 Hardness assumption

This parameter focuses on improving the hardness assumptions of the garbling scheme in order to guarantee stronger security.

3.2 Point-and-permute (1990)



$k_a^0||p_1,\ k_a^1||p_1'$
w_1
$k_4^0||p_4,\ k_4^1||p_4'$
w_2
$k_b^0||p_2,\ k_b^1||p_2'$
w_3
$k_c^0||p_3,\ k_c^1||p_3'$

Figure 3.1: The permutation bit appended to each label

Notice that for each gate Bob has to decrypt four ciphertexts when in reality he only needs to decrypt one. Point-and-permute, introduced by Beaver, Micali and Rogaway in [2] greatly improves the evaluation process from having to decrypt the four ciphertexts to having to decrypt only the necessary one per gate.

The way they accomplish this is by associating for the two labels in each wire w_i two random permutation bits p_i and p_i' such that one of the two is 1 and the other is 0, i.e., $p_i = r$ and $p_i' = 1 - r$, where $r \in \{0, 1\}$ is chosen randomly. Then, they append this permutation bit to each label in the gate as in Figure 3.1. The elegant reason for this is that now the four possible labels per gate have four different *orderings*. For example, for input wires w_a and w_b say Alice creates as usual the four labels k_a^0, k_a^1, k_b^0, and k_b^1. Let's say she appends the random permutation bit to each label to give $k_a^0||0$, $k_a^1||1$, $k_b^0||1$, and $k_b^1||0$ (notice that the two labels for each wire must have a different permutation bit). The key feature now is that we can establish a canonical ordering for the labels Bob receives. Alice states that if Bob sees two 0s in the two labels, then he must decrypt the first ciphertext he gets. If he receives a 0 and a 1 he must decrypt the second ciphertext, and so on. More generally, when Bob receives the labels $k_a^i||\alpha$ and $k_b^j||\beta$ he will **only** decrypt the ciphertext in position $2\alpha + \beta$.

In his talk at the Simons institute, Mike Rosulek from Oregon State University gives a less dense demonstration of how one can think about point-and-permute which

Optimization	Size per gate		Calls to H per gate			
			Alice		Bob	
	XOR	AND	XOR	AND	XOR	AND
Classical	4	4	4	4	4	4
Point-and-permute	4	4	4	4	1	1

Table 3.1: Comparison of all optimizations so far

is worth showing. Instead of thinking it as permutation bits, give the two labels in each wire two different colors: blue and red (in a random order) like in Figure 3.2.

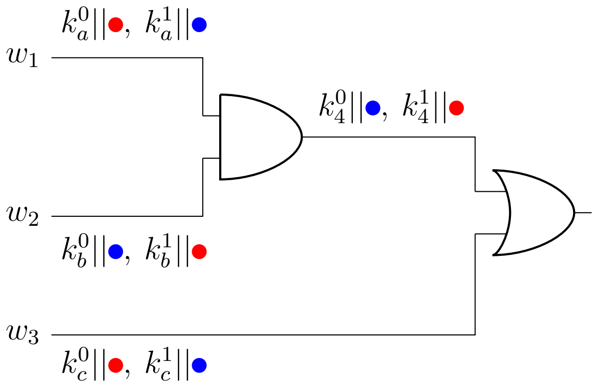


Figure 3.2: Coloring the circuit

In Alice's world, blue always comes before red, so her natural ordering would be {🔵🔵, 🔵🔴, 🔴🔵, 🔴🔴}. Now for example, if Bob receives two labels where the colors are (in order) blue and red, he will **only** evaluate the second ciphertext.

So far, we can see in Table 3.1 the size per gate of each method and the number of ciphertexts to send/evaluate.

3.3 Garbled Row Reduction 3 (1999)

Up until now, every label k_a^i was chosen as a random key. But Naor, Pinkas and Sumner in [11] propose to choose the labels in a different manner. We can illustrate their optimization with an example. Take the gate to be Figure 3.3 and take its corresponding ordered encrypted truth table in Table 3.2. Now take the first element in this table, i.e., $\text{Enc}_{(k_a^1, k_b^0)}(k_c^0||$🔵$)$. Instead of sending this ciphertext to Bob, we can

Order	Encrypted Output
🔵🔵	$\text{Enc}_{(k_a^1,k_b^0)}(k_c^0 \Vert$ 🔵 $) = 0^n$
🔵🔴	$\text{Enc}_{(k_a^1,k_b^1)}(k_c^1 \Vert$ 🔴 $)$
🔴🔵	$\text{Enc}_{(k_a^0,k_b^0)}(k_c^0 \Vert$ 🔵 $)$
🔴🔴	$\text{Enc}_{(k_a^0,k_b^1)}(k_c^0 \Vert$ 🔵 $)$

Table 3.2: Ordering of Garbled Table for Figure 3.3

just choose k_c^0 cleverly to be the label such that when encrypted under the key (k_a^1, k_b^0) will give 0^n, the zero N-bit integer. In other words, we set $k_c^0 = \text{Enc}^{-1}_{(k_a^1,k_b^0)}(0^n)$. When Bob and Alice agree to this, then it is not necessary for Alice to transmit the first row in the table, because the only thing Bob needs to do is to conclude that if he gets 🔵🔵, then the ciphertext is 0^n.

We have reduced the problem to sending 3 rows instead of 4, hence its name Garbled Row Reduction 3 (GRR3). We will see later on that one construction sends only 2 rows, GRR2.

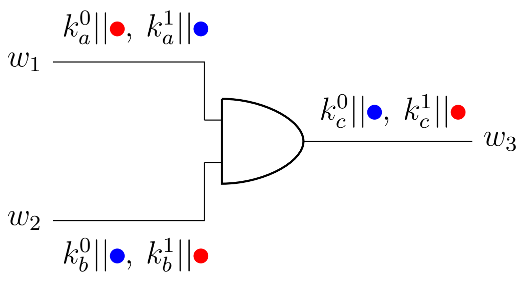
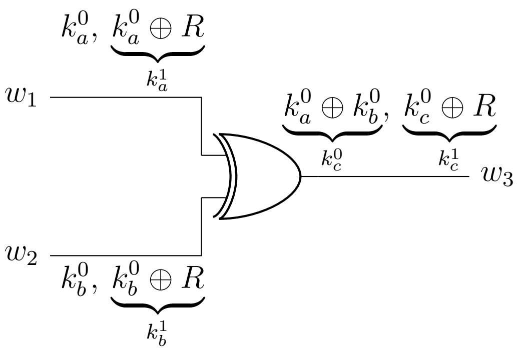

$k_a^0 \Vert$ 🔴, $k_a^1 \Vert$ 🔵
w_1
$k_c^0 \Vert$ 🔵, $k_c^1 \Vert$ 🔴
w_3
w_2
$k_b^0 \Vert$ 🔵, $k_b^1 \Vert$ 🔴

Figure 3.3: Example gate to illustrate Garbled Row Reduction 3

3.4 Free XOR (2008)

In the spirit of GRR3, instead of choosing purely random labels for each gate, Kolesnikov and Schneider in [8] propose to relate the labels for each gate in a different manner. The main optimization they achieve is the ability to compute XOR gates "for free", i.e., without the need for Bob to decrypt anything. The way they accomplish is the following:

Fix a random $R \in \{0,1\}^n$, and as usual choose the input labels k_α^0 randomly, i.e., choose the label corresponding to 0 randomly for each input wire in every gate. Now, however, **don't** choose k_α^1 randomly, but set it to be $k_\alpha^1 = k_\alpha^0 \oplus R$. Finally, set the output label for 0 to be $k_c^0 = k_a^0 \oplus k_b^0$. Graphically, we have something like in Figure 3.4:



$k_a^0, \underbrace{k_a^0 \oplus R}_{k_a^1}$
w_1
$\underbrace{k_a^0 \oplus k_b^0}_{k_c^0}, \underbrace{k_c^0 \oplus R}_{k_c^1}$
w_3
w_2
$k_b^0, \underbrace{k_b^0 \oplus R}_{k_b^1}$

Figure 3.4: XOR gates for free

The beauty of this is that the two labels Bob receives from Alice, say k_a^i and k_b^j, result in the correct output label when XORed together. In other words, Bob doesn't need to decrypt anything, he just needs to XOR the two labels he receives to get the output label. This internally works by the semantics of XOR. We can try to XOR each pair and see what happens:

$$\begin{aligned}
&k_a^0 \oplus k_b^0 = k_c^0 \quad \text{(by definition)}\\
&k_a^0 \oplus k_b^1 = k_a^0 \oplus (k_b^0 \oplus R) = (k_a^0 \oplus k_b^0) \oplus R = k_c^0 \oplus R = k_c^1\\
&k_a^1 \oplus k_b^0 = (k_a^0 \oplus R) \oplus k_b^0 = (k_a^0 \oplus k_b^0) \oplus R = k_c^0 \oplus R = k_c^1\\
&k_a^1 \oplus k_b^1 = (k_a^0 \oplus R) \oplus (k_b^0 \oplus R) = (k_a^0 \oplus k_b^0) \oplus (R \oplus R) = k_a^0 \oplus k_b^0 = k_c^0
\end{aligned}$$

Note this optimization invites the boolean circuit to have as many XOR gates as possible.

3.4.1 Formalizing FreeXOR

We now follow Kolesnikov and Schneider's formal definition of the optimization.

- **Algorithm 1 (Construction)**:
  1. Randomly choose a global offset $R \in \{0,1\}^N$.
  2. For each input wire W_i of the circuit C
     a) Choose the label to be $w_i^0 = k_i^0 || p_i^0 \in \{0,1\}^{N+1}$, where both k_i^0, the key, and p_i^0, the permutation bit, are chosen at random.
     b) Set the other label to be $w_i^1 = k_i^1 || p_i^1 = k_i^0 \oplus R || p_i^0 \oplus 1$.
  3. For each gate G_i of C in topological order
     a) if G_i is an XOR gate with output label $w_c = XOR(w_a, w_b)$ and with input labels $w_a^0 = k_a^0 || p_a^0$, $w_b^0 = k_b^0 || p_b^0$, $w_a^1 = k_a^1 || p_a^1$, $w_b^1 = k_b^1 || p_b^1$:
        i. Set the output label $w_c^0 = k_a^0 \oplus k_b^0 || p_a \oplus p_b$.
        ii. Set the output label $w_c^1 = k_a^0 \oplus k_b^0 \oplus R || p_a \oplus p_b \oplus 1$.
     b) Otherwise, if the gate is not XOR:
        i. Randomly choose the output label $w_c^0 = k_c^0 || p_c \in \{0,1\}^{N+1}$.
        ii. Set the output label $w_c^1 = k_c^0 \oplus R || p_c \oplus 1$.
        iii. Create G_i's garbled table. That is, for each of the 4 possible combinations of input values $v_a, v_b \in \{0,1\}$, set

$$e_{v_a,v_b} = H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a,v_b)}$$

        Sort the entries in the garbled table according to the point-and-permute technique, i.e., entry e_{v_a,v_b} in position $2p_a + p_b$.

- **Algorithm 2 (Evaluation)**:
  1. For each input wire W_i of C receive the corresponding garbled value $w_i = k_i || p_i$.
  2. For each gate G_i in the topological order given by the labels
     a) If G_i is an XOR gate with garbled input values $w_a = k_a || p_a$ and $w_b = k_b || p_b$, then compute $w_c = k_a \oplus k_b || p_a \oplus p_b$.
     b) Otherwise, decrypt garbled output value from the garbled table entry e in position $2p_a + p_b$ as

$$w_c = k_c || p_c = H(k_a || k_b || i) \oplus e$$

With these two algorithms we are in a position to define the protocol.

Protocol:

- **Inputs**: Party P_1 has private input $x = (x_1, x_2, ..., x_{u_1}) \in \{0,1\}^{u_1}$ and party P_2 has private input $y = (y_1, y_2, ..., y_{u_2}) \in \{0,1\}^{u_2}$.

- **Auxiliary input**: A boolean acyclic circuit C that evaluates the desired function f.

- **Steps of the protocol**:

  1. P_1 constructs the garbled circuit using Algorithm 1 and sends the garbled tables to P_2.
  2. Let $W_1, ..., W_{u_1}$ be the circuit input wires for x, and similarly let $W_{u_1} + 1, ..., W_{u_1+u_2}$ be the circuit input wires for y. Then:
     a) P_1 sends P_2 the garbled values $w_1^{x_1}, ..., w_{u_1}^{x_{u_1}}$.
     b) For every $i \in \{1, ..., u_2\}$, P_1 and P_2 execute a 1-2 oblivious transfer protocol where P_1's input is $(k^0_{u_1+i}, k^1_{u_1+i})$ and P_2's input is y_i. Note that all the instances of 1-2 oblivious transfer can be run in parallel, a useful optimization when implementing garbled circuits.
  3. P_2 now has the garbled tables and the labels, so he can run Algorithm 2 and output $f(x, y)$.

3.4.2 Intuition of Security

The main idea is that Algorithm 1 uses H as a one-time pad to encrypt the garbled output values in the garbled tables (this corresponds to step 3.b.iii). Since any combination of H's inputs is used for encryption of at most one table entry, then this indeed acts as a one-time pad. Lastly, since the evaluator of the garbled circuit only knows one garbled label per wire, he can decrypt exactly one entry of G_i's garbled table. All other entries are encrypted with a key he can not evaluate. Therefore, as Kolesnikov explains, one of the two garbled values of **every** wire will look random to him.

w_a	w_b	w_c
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.3: An example of an odd gate (`AND`)

w_a	w_b	w_c
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.4: An example of an even gate (`XOR`)

3.5 Garbled Row Reduction 2 (2009)

The main idea proposed in [12] is to use polynomial interpolation by Alice and Bob to identify using **only** two ciphertexts the proper output labels. We need to divide two cases for this optimization: odd gates and even gates. Odd gates are those that have as possible output three same truth values (e.g. Table 3.3), while even gates are those that have two true values and two false values (e.g. Table 3.4).

3.5.1 Odd Gates

The idea works as follows. We first create four keys corresponding to labels that will output 0^n when encrypted under each pair of input labels. In other words, we set

$$\begin{aligned}
K_1 &= \mathrm{Dec}_{(k_a^0,k_b^0)}(0^n)\\
K_2 &= \mathrm{Dec}_{(k_a^0,k_b^1)}(0^n)\\
K_3 &= \mathrm{Dec}_{(k_a^1,k_b^0)}(0^n)\\
K_4 &= \mathrm{Dec}_{(k_a^1,k_b^1)}(0^n)
\end{aligned}$$

For an AND gate, Alice will decide that K_4 corresponds to the label k_c^1, while the other three keys correspond to k_c^0 (for an OR gate we would set K_1 to correspond

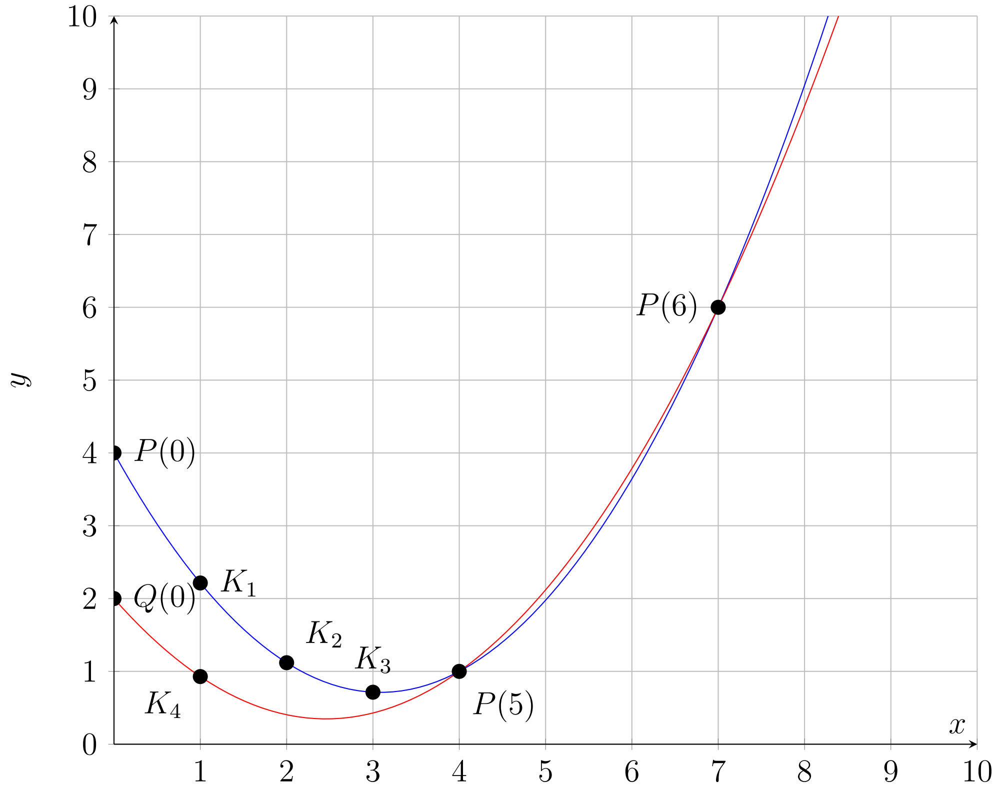


Figure 3.5: GRR2 for `AND` gate. Note that although the graph uses real numbers, GRR2 uses a finite field.

with k_c^0, and the other three with k_c^1). If we continue the example of an AND gate, Alice will take K_1, K_2 and K_3 and perform polynomial interpolation over $\mathbb{F}_{2^n}$ with the points $(1, K_1), (2, K_2)$ and $(3, K_3)$. This will yield a unique quadratic polynomial $P(X)$ that passes through these points. We now interpolate again another polynomial $Q(X)$ that passes through the points $(4, K_4), (5, P(5))$ and $(6, P(6))$. Finally, Alice chooses the output labels as $k_c^0 = P(0)$, and $k_c^1 = Q(0)$ and sends $P(5)$ and $P(6)$ as part of the garbled gate to Bob.

As for Bob, he will as usual receive k_a^i and k_b^j. He can hence calculate using these two labels one of the four K_i. Say for instance he receives k_a^0 and k_b^1, then he will calculate K_2. Now that he has K_2, $P(5)$ and $P(6)$, he will perform *his* polynomial interpolation $R(X)$ with the points $(2, K_2), (5, P(5))$ and $(6, P(6))$. But notice that if he evaluates his polynomial at 0 he will get $P(0)$ as $R(0) = P(0) = k_c^0$. Similarly, if Bob receives k_a^1 and k_b^1, he will get K_4 and then $R(0) = Q(0) = k_c^1$ (see Figure 3.5

for a graphical representation).

This method reduces the evaluation of an AND gate to two ciphertexts (we only have to send $P(5)$ and $P(6)$).

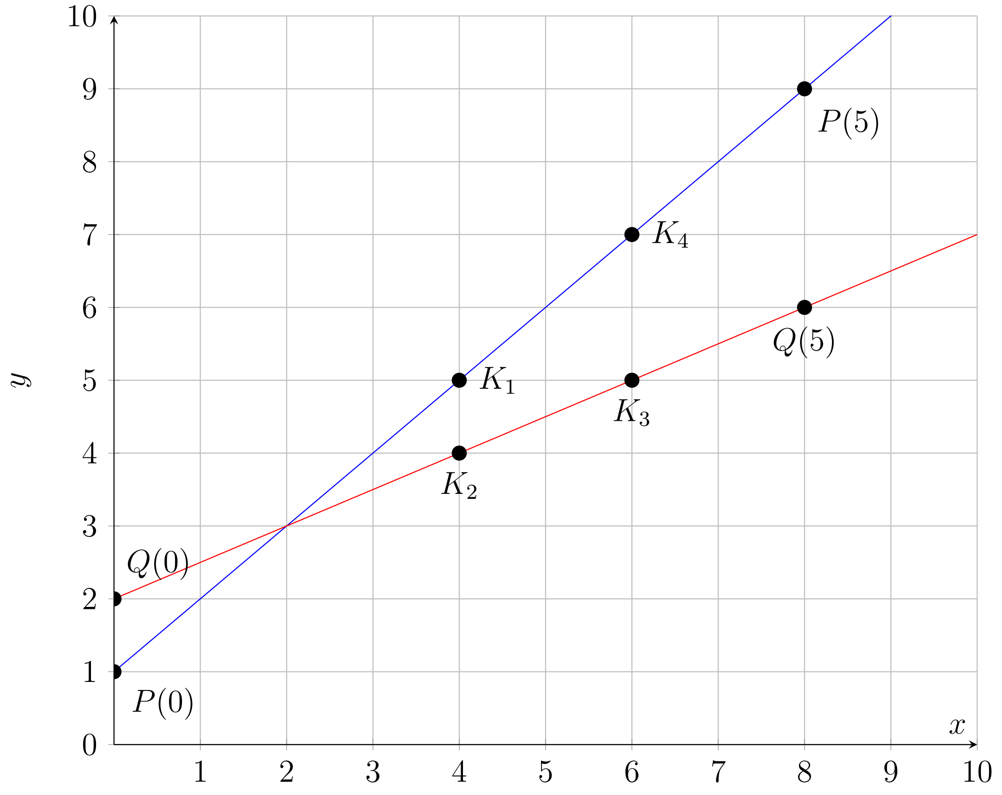


Figure 3.6: GRR2 for an `XOR` gate.

3.5.2 Even Gates

Suppose now the gate is an `XOR` gate, so its parity is even. In this case, Alice would create a linear polynomial $P(X)$ that passes through the points $(1, K_1)$ and $(4, K_4)$, and another linear polynomial $Q(X)$ that passes through the points $(2, K_2)$ and $(3, K_3)$. She would then set $k_c^0 = P(0)$ and $k_c^1 = Q(0)$. Finally, she would send Bob $P(5)$ and $Q(5)$, making sure that $P(5)$ is the first entry of the table, while $Q(5)$ is the second. This will aid Bob in knowing which point to choose to interpolate his polynomial.

Bob, now in possession of say k_a^0, k_b^1, $P(5)$, and $Q(5)$, first recovers K_2 and interpolates the linear polynomial $Q(X)$ at $(2, K_2)$ and $(5, Q(5))$. Once he finds out

$Q(X)$, he simply evaluates it at $X = 0$, i.e., $k_c^1 = Q(0)$.

3.5.3 Experimental Results

Before GRR2, all optimizations were theoretical in nature. In this paper the authors not only propose the optimization, but implement it using their *Fairplay* system. While we will go into the details of the implementation on the next chapter, it is worth highlighting the speed at which garbled circuits can evaluate secure two-party computation.

- **Example 1 - Evaluation of a Simple Circuit**: The output chosen for this circuit is the number of 1s in the string obtained from performing a bitwise `AND` between two 32 bit strings; one supplied by Alice and one by Bob. This circuit has 689 gates, and the calculations were performed on two machines with Intel Core 2 Duos 3.0 GHz, 4GB of RAM connected by a 1GB Ethernet. The hash function used was SHA-256.

Method	Total Time	Total KBytes
Without GRR2	2s	46
With GRR2	1s	34

Table 3.5: Time to evaluate a simple circuit

- **Example 2 - Evaluating AES**: The circuit of this example computes an AES encryption of a 128-bit block given a 128-bit key. Alice's input is the key, while Bob's input is the message block.

Method	Precomp. Time	Send Time	OT Time	Calc. Time	Total	KBytes
No GRR2	5s	2s	4s	3s	14s	3162
GRR2	5s	1s	3s	3s	12s	1752

Table 3.6: Time to evaluate AES

We now have an updated Table 3.7 showing a comparison between the optimizations mentioned.

Optimization	Size per gate		Calls to H per gate			
			Alice		Bob	
	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
FreeXOR	0	4	0	4	0	1
GRR2	2	2	4	4	1	1

Table 3.7: Comparison of all optimizations so far

3.6 FleXOR (2014)

FleXOR, or *flexible* XOR, was introduced by Kolesnikov, Mohassel and Rosulek in [7] in 2014. Consider the following XOR gate in Figure 3.7.

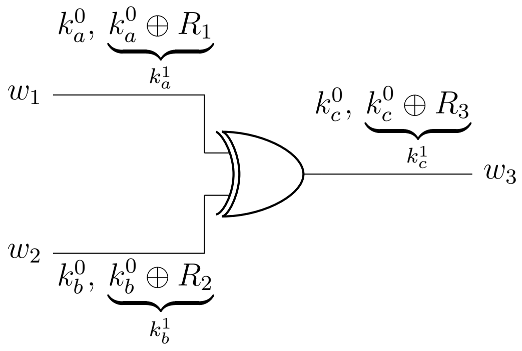

$k_a^0, \underbrace{k_a^0 \oplus R_1}_{k_a^1}$
w_1
$k_c^0, \underbrace{k_c^0 \oplus R_3}_{k_c^1}$
w_3
w_2
$k_b^0, \underbrace{k_b^0 \oplus R_2}_{k_b^1}$

Figure 3.7: XOR gate with different offsets

Notice that the true label for wire i is an offset R_i of the false label. Ideally, we would want the offset of the three wires to be the same to apply FreeXOR. In other words, we would want in Figure 3.7 for $R_1 = R_2 = R_3$. Unfortunately, most of the times these offsets will be distinct. What Kolesnikov, Mohassel and Rosulek propose is to **transform** R_1 and R_2 into R_3.

The way they accomplish this is by first selecting random "translated" labels $\tilde{k}_a^0$

and $\tilde{k}_b^0$. Then, they garble the gate with the following four ciphertexts:

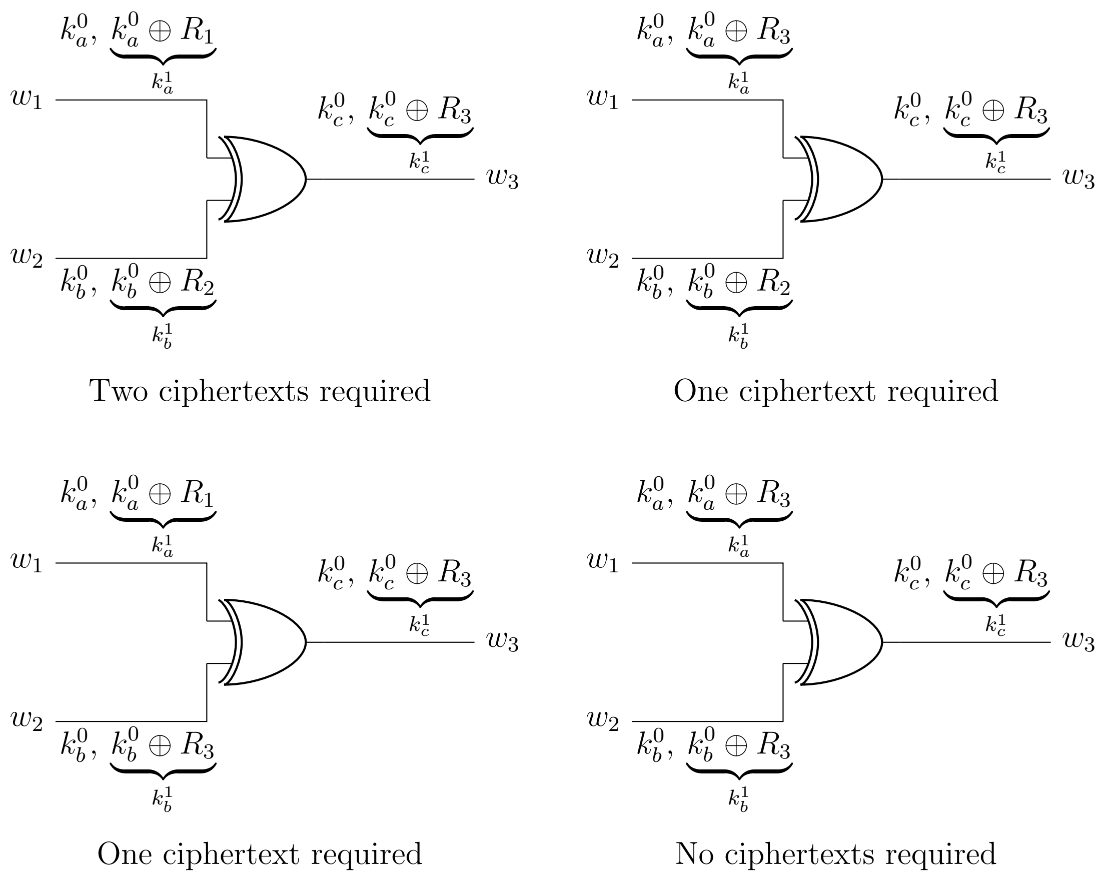
$$\begin{aligned}
&\text{Enc}_{k_a^0}(\tilde{k}_a^0)\\
&\text{Enc}_{k_a^0 \oplus R_1}(\tilde{k}_a^0 \oplus R_3)\\
&\text{Enc}_{k_b^0}(\tilde{k}_b^0)\\
&\text{Enc}_{k_b^0 \oplus R_2}(\tilde{k}_b^0 \oplus R_3)
\end{aligned}$$

Clearly the first two ciphertexts allow Bob to translate the true/false labels for wire a with an undesired offset into true/false labels with a desired offset. In other words, he converts R_1 into R_3 and as a byproduct he therefore changes k_a^0 into $\tilde{k}_a^0$. Similarly, the last two ciphertexts change R_2 into R_3 and k_b^0 into $\tilde{k}_b^0$. While this does indeed work, notice that Alice still needs to send four ciphertexts to Bob per XOR gate, much worse than the previous optimization (GRR2) which sent only two. What the authors propose is to apply the GRR3 trick and set both the first ciphertext $\text{Enc}_{k_a^0}(\tilde{k}_a^0)$ and the third ciphertext $\text{Enc}_{k_b^0}(\tilde{k}_b^0)$ to be the 0^n ciphertexts. In other words, $\tilde{k}_a^0 = \text{Enc}_{k_a^0}^{-1}(0^n)$ and $\tilde{k}_b^0 = \text{Enc}_{k_b^0}^{-1}(0^n)$. We therefore improve the garbled table as follows:

$$\begin{aligned}
&\sout{\text{Enc}_{k_a^0}(\tilde{k}_a^0)}\\
&\text{Enc}_{k_a^0 \oplus R_1}(\tilde{k}_a^0 \oplus R_3)\\
&\sout{\text{Enc}_{k_b^0}(\tilde{k}_b^0)}\\
&\text{Enc}_{k_b^0 \oplus R_2}(\tilde{k}_b^0 \oplus R_3)
\end{aligned}$$

Furthermore, the optimization doesn't stop there, since it might be possible that $R_1 = R_3$ from the beginning, or $R_2 = R_3$, or even better, $R_1 = R_2 = R_3$, which would correspond to the FreeXOR case. In fact, if $R_1 = R_3$, there is no need to send the first two ciphertexts, allowing the evaluation of an XOR gate with only one ciphertext (similarly if $R_2 = R_3$). This is precisely why the authors call their method flexible XOR; an `XOR` gate can be evaluated with 0, 1, or 2 ciphertexts depending on the nature of R_1, R_2 and R_3.

The novelty of this optimization was not particularly on the low size of the garbled table (after all, FreeXOR requires zero ciphertexts), but rather on the compatibility it gives with other optimizations. GRR2 is incompatible with FreeXOR as the offsets of the labels in GRR2 are unpredictable while FreeXOR needs a constant offset. However, GRR2 **is** compatible with FleXOR, as FleXOR doesn't need a constant offset, it can adapt.



$k_a^0,\ \underbrace{k_a^0 \oplus R_1}_{k_a^1}$
w_1
$k_c^0,\ \underbrace{k_c^0 \oplus R_3}_{k_c^1}$
w_3
w_2
$k_b^0,\ \underbrace{k_b^0 \oplus R_2}_{k_b^1}$
Two ciphertexts required
$k_a^0,\ \underbrace{k_a^0 \oplus R_3}_{k_a^1}$
w_1
$k_c^0,\ \underbrace{k_c^0 \oplus R_3}_{k_c^1}$
w_3
w_2
$k_b^0,\ \underbrace{k_b^0 \oplus R_2}_{k_b^1}$
One ciphertext required
$k_a^0,\ \underbrace{k_a^0 \oplus R_1}_{k_a^1}$
w_1
$k_c^0,\ \underbrace{k_c^0 \oplus R_3}_{k_c^1}$
w_3
w_2
$k_b^0,\ \underbrace{k_b^0 \oplus R_3}_{k_b^1}$
One ciphertext required
$k_a^0,\ \underbrace{k_a^0 \oplus R_3}_{k_a^1}$
w_1
$k_c^0,\ \underbrace{k_c^0 \oplus R_3}_{k_c^1}$
w_3
w_2
$k_b^0,\ \underbrace{k_b^0 \oplus R_3}_{k_b^1}$
No ciphertexts required

Figure 3.8: Possible scenarios for FleXOR

3.6.1 Experimental Results

The authors test their optimization with various functions. In particular, they calculate the Hamming distance between two bit strings, AES, and SHA-256, and the results can be seen in Table 3.8.

circuit	GRR2	FreeXOR	FleXOR
AES	2	0.64	0.72
SHA-256	2	2.05	1.56
Hamming distance	2	0.50	0.50

Table 3.8: Comparison of GRR2, FreeXOR, and FleXOR. The number in each cell represents the average number of ciphertexts per gate.

Clearly FreeXOR performs best when the number of `XOR` gates in the circuit is very high, suggesting AES is a heavily `XOR` oriented circuit. What would be interesting to find is what is the optimal ratio of `XOR` to `AND` gates that makes FleXOR works best. This isn't trivial to find, as after all the labels of the circuit are not deterministic, making it difficult to calculate how many times the garbled table will require zero, one, or two ciphertexts. The authors claim, based on their experimental results, that the average ciphertexts per `XOR` gate is one ciphertext. Although not an exhaustive result, it might be indicative of how many ciphertexts are needed on average for `XOR` gates using FleXOR.

3.7 Half Gates (2015)

We are now in a position to introduce the state-of-the-art in garbled circuits. This technique was introduced by Zahur, Rosulek, and Evans in [16] in 2015. The last technique made `XOR` gates cost either 0, 1, or 2 ciphertexts, while this one makes `AND` gates cost two ciphertexts and `XOR` gates zero ciphertexts. It does this by eliminating the need for GRR2 and creating a new method to evaluate `AND` gates with two ciphertexts in a way that is compatible with FreeXOR.

3.7.1 Half Gates

Before explaining the optimization we need some terminology. We define a *half gate* as an `AND` gate for which one of the parties knows one of the inputs (not privately, but out in the public). Say we want to compute $c = a \wedge b$. Let $k_a^0, k_a^0 \oplus R$, k_b^0 and $k_b^0 \oplus R$ be the input labels for the `AND` gate to evaluate, and let k_c^0 and $k_c^0 \oplus R$ correspond to the output labels. We distinguish two cases:

- **Case 1**: Alice, the garbler, knows one of the inputs. If the input Alice knows is false, then she will simply garble a unary gate that always outputs false (indeed, $F \wedge * = F$). If, however, the input Alice knows is true, then she will simply garble a unary identity gate. Hence, Alice will produce two ciphertexts:

$$c_1 = H(k_b^0) \oplus k_c^0$$
$$c_2 = H(k_b^1) \oplus k_c^0 \oplus aR$$

  where H is a suitable hash function and a is the input bit Alice knows. These two ciphertexts are then permuted according to k_b^0's point-and-permute bit. Bob, on receiving the two ciphertexts, takes a hash of the secret label he wants and decrypts the appropriate ciphertext.

  - $a = 0$: then he will be able to recover C. This is trivial; if he receives $k_b^1 = k_b^0 \oplus R$ and along with it c_2, he will perform $c_2 \oplus H(k_b^1) = H(k_b^1) \oplus k_c^0 \oplus 0 \cdot R \oplus H(k_b^1) = k_c^0$. If he receives k_b^0 and along with it c_1, he will perform $c_1 \oplus H(k_b^0) = H(k_b^0) \oplus k_c^0 \oplus H(k_b^0) = k_c^0$.
  - $a = 1$: Bob will be able to recover either k_c^0 or k_c^1 depending on his secret bit b. Recall that Alice garbled a unary identity gate, so if $b = 0$, then Bob will recover k_c^0, and if $b = 1$, Bob will recover k_c^1. The reason is similar to the previous case. If Bob has $b = 0$ and therefore k_b^0, then $c_1 \oplus H(k_b^0) = H(k_b^0) \oplus k_c^0 \oplus H(k_b^0) = k_c^0$. If Bob has $b = 1$, then he has k_b^1, and we have $c_2 \oplus H(k_b^1) = H(k_b^1) \oplus k_c^0 \oplus 1 \cdot R \oplus H(k_b^1) = k_c^0 \oplus R = k_c^1$.

  Alice still sends two ciphertexts, but she can drop the first one by applying the GRR standard trick, making the first ciphertext be the all zeros ciphertext. In total, this half gate was evaluated with only one ciphertext. On the downside, Alice needed to call H twice and Bob H once.

- **Case 2**: Bob, the evaluator, knows one of the inputs. Let's say we want to compute $c = a \wedge b$ and Bob knows the value of a at the time of evaluation. If $a = 0$, Bob should somehow be able to derive k_c^0, and if $a = 1$, then Bob should be able to derive k_c^0 if $b = 0$, and k_c^1 if $b = 1$. However, what the authors note is that in the case $a = 1$, then it is sufficient for Bob to receive $\Delta = k_c^0 \oplus k_b^0$. The reason is that once Bob receives the secret label for wire b (either k_b^0 or k_b^1), he can `XOR` Δ with b's secret label to obtain either k_c^0 or k_c^1. More precisely, if Bob obtains via OT k_b^0, then $\Delta \oplus k_b^0 = k_c^0 \oplus k_b^0 \oplus k_b^0 = k_c^0$. Similarly, if Bob obtains k_b^1, then $\Delta \oplus k_b^1 = k_c^0 \oplus k_b^0 \oplus k_b^1 = k_c^0 \oplus k_b^0 \oplus k_b^0 \oplus R = k_c^0 \oplus R = k_c^1$.

Therefore, Alice will provide the following two ciphertexts:

$$\begin{aligned} c_1 &= H(k_a^0) \oplus k_c^0 \\ c_2 &= H(k_a^1) \oplus k_c^0 \oplus k_b^0 \end{aligned}$$

- $a = 0$: Then Bob uses k_a^0 to decrypt the first ciphertext. He will take the hash of k_a^0 and `XOR` it with the first ciphertext. We thus have $H(k_a^0) \oplus c_1 = H(k_a^0) \oplus H(k_a^0) \oplus k_c^0 = k_c^0$.
- $a = 1$: Then Bob uses $k_a^1 = k_a^0 \oplus R$ to decrypt the second ciphertext. He will take the hash of k_a^1 and `XOR` it with the second ciphertext. We thus have $H(k_a^1) \oplus c_2 = H(k_a^1) \oplus H(k_a^1) \oplus k_c^0 \oplus k_b^0 = k_c^0 \oplus k_b^0 = \Delta$. Once we have Δ, we refer to the previous discussion on how to obtain the output labels.

As in the previous case, there is no need to send two ciphertexts. Using GRR, we can set c_1 to be the all zeros ciphertext. By the properties of `XOR`, this means we set $k_c^0 = H(k_a^0)$. The cost of garbling this half gate is the same as above: Alice calls H twice and Bob once.

3.7.2 Putting it all together

Or, as the authors subtly state, *two halves make a whole.* Consider the general case where Alice wants to garble an `AND` gate $c = a \wedge b$ and no inputs are public to anybody. Essentially this optimization relies on the following identity:

$$\begin{aligned} a \wedge b &= a \wedge (0 \oplus b) \\ &= a \wedge ((r \oplus r) \oplus b) \\ &= (a \wedge r) \oplus (a \wedge (r \oplus b)) \end{aligned}$$

Suppose Alice chooses a uniformly random bit r. We are thus in Case 1 for the half gate $(a \wedge r)$. If now we set up the scheme so that Bob learns $r \oplus b$, we are in Case 2 for the half gate $(a \wedge (r \oplus b))$. As the authors state, there is no need to worry about letting Bob know $r \oplus b$, as it doesn't detail any information about b. The final XOR between the two half gates is free, so the total cost of the two half gates is two ciphertexts.

Furthermore, it is possible for Bob to learn $r \oplus b$ without any overhead. The way to do so is by encoding the wire label for b to have as a point-and-permute bit $r \oplus b$. In other words, send via OT $k_b^i || b \oplus r$.

In total, an AND gate was garbled with two ciphertexts, one XOR call, Alice calling H four times, and Bob calling H twice. Of course, this optimization doesn't just apply to AND gates but rather any odd gate such as OR, NAND, etc.

3.7.3 Experimental Results

As in FleXOR, the authors test their optimization with various functions. In particular, they calculate again the Hamming distance between two bit strings, AES, and SHA-256, and the results can be seen in Table 3.9.

circuit	GRR2	FreeXOR	FleXOR	Half Gates	% decrease
AES	2	0.64	0.72	0.42	33%
SHA-256	2	2.05	1.56	1.37	12%
Hamming distance	2	0.50	0.50	0.33	33%

Table 3.9: Comparison of GRR2, FreeXOR, FleXOR, and Half Gates. The number in each cell represents the average number of ciphertexts per gate.

We can clearly see that this optimization performs better in all cases, as it only requires two ciphertexts for AND and zero for XOR. We can update the comparison table to include the latest optimization in Table 3.10.

3.7.4 Lower Bounds on Garbled Circuits

A common pattern in all the optimizations seen is the reduction of the size of the garbled table Alice has to send to Bob. A natural question that follows is whether

Optimization	Size per gate		Calls to H per gate			
			Alice		Bob	
	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
FreeXOR	0	4	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	0	2	0	4	0	2

Table 3.10: Comparison of all optimizations so far

this can go on until everything is "free", i.e., achieving a compatible FreeXOR and FreeAND, so to speak. The authors investigate this question, and come up with interesting conclusions.

3.7.4.1 Linear Garbling Schemes

The paper states that all garbling schemes proposed in the literature share certain features, in particular the fact that both the `Gb` and `Ev` procedures use only *linear* operations. For instance, let's take a few optimizations to see their linearity:

- **GRR2**: This optimization requires interpolating polynomials on a finite field. Since the number of points to interpolate is fixed, interpolation is a linear operation.
- **FreeXOR**: The labels are chosen using `XOR` relationships, so clearly linear.
- **Half Gates**: Same idea as FreeXOR, the labels chosen are of the form $H(k_a^0) \oplus k_c^0$, so `XOR` is used and therefore linear.

Once this relationship has been established, the authors propose one of the major theorems of the paper.

Theorem 1. *Every ideally secure garbling scheme for AND gates that is linear must have a garbled table of at least two ciphertexts.*

This theorem has big consequences for the research of optimizations of garbled circuits. In fact, it proves that their method is optimal for linear schemes. An improvement in the size of the garbled table will have to be achieved through non-linear methods.

3.8 Garbled Gadgets

The problem with all the optimizations explained so far is that they are very tied in to fan-in 2 gates, i.e., two input wires and one output wire. While that is fine for a number of applications (e.g. calculating Hamming distance, AES, etc.), other applications that are cumbersome and expensive to express with fan-in 2 gates are poorly suited for garbled circuits. Some examples of applications not suited with current optimizations are for instance a neural-network-based classifier, or arithmetic computations. In fact, arithmetic computations are done for free in secret sharing based schemes, so there is no practical reason to choose garbled circuits over the secret sharing scheme.

Therefore, the paper written by Ball, Malkin, and Rosulek titled *Garbling Gadgets for Boolean and Arithmetic Circuits* [1] addresses these problems and proposes new solutions supporting high fan-in gates.

3.8.1 Improvements

Let λ be the security parameter for the scheme, that is, how many bits each label has (typically this will be 128-256 bits). The paper offers the following improvements:

- **Linear operations in arithmetic circuits**: The scheme they propose allows addition and multiplication by a public constant for free. This is typical of secret sharing schemes, but now with the ability of garbled circuits to offer arithmetic operations and the advantage they have over secret sharing schemes in the fact that they have constant-round protocols makes it an arguably better rounded option for secure computation.

  Previously, other garbled circuit schemes would represent integers in binary and would need $O(\lambda l)$ to add two l-bit numbers. Now it is essentially for free.

- **Other arithmetic operations**: The scheme not only supports addition and multiplication, but also exponentiation by a public power with a cost independent of the exponent.

- **High fan-in boolean threshold gates**: For fan-in b gates, this scheme requires $O(\lambda \log^3 b)$ bits, while current schemes are exponentially worse.

3.8.2 Generalizing FreeXOR

3.8.2.1 Addition

The main idea is to provide free addition mod m for any fixed m (when $m = 2$, this will just be FreeXOR). Assume the wire labels we have been working on as bits are now vectors of $\mathbb{Z}_m$. Previously there was only two labels per gate, but now there is m labels per gate. Therefore, the label k_a^x encoding $x \in \{0, 1, ..., m\}$ will equal

$$k_a^x = k_a^0 + xR$$

where now addition refers to component-wise addition, R is a random vector in $\mathbb{Z}_m$, and xR is multiplication by a constant as in vector spaces. With this it now follows that we can get addition mod m for free. Indeed, we have that

$$k_a^x + k_b^y = k_a^0 + xR + k_b^0 + yR = (k_a^0 + k_b^0) + (x + y)R$$

3.8.2.2 Multiplication by a constant

This is a similar approach to addition. Indeed, let $c \in \mathbb{Z}_m$ be a known constant, then:

$$ck_a^x = c(k_a^0 + xR) = ck_a^0 + (cx)R$$

where again the operations are component-wise mod m. Note this works provided c is prime to m. While we won't go into details as to why this is the case, the intuition is primarily that when c is prime to m the wire label preserves its uniform distribution. Simply, c and m not being coprime can lead to security issues.

3.8.3 Generalizing Point-and-Permute

As in the original point-and-permute, instead of appending a bit to each wire label we now append an element $x \in \mathbb{Z}_m$ and we append $1 \in \mathbb{Z}_m$ to R. Let $\tau_m(k_a)$ denote the last component of a wire label. Then we have:

$$\tau_m(k_j^x) = \tau_m(k_j^0) + x \cdot \tau_m(R) = \tau_m(k_j^0) + x \cdot 1 = \tau_m(k_j^0) + x$$

As the authors state, each wire label out of the m possible wire labels is a assigned a random cyclic shift of the set $\mathbb{Z}_m$, where the shift is determined by where $\tau_m(k_j^0)$ goes in $\mathbb{Z}_m$.

3.9 Focusing on optimizing AND gates

The quest for a FreeAND has been highly sought after following Kolesnikov and Schneider's FreeXOR in 2008. GRR3 required Alice to send 3 ciphertexts, GRR2 required two (but was incompatible with FreeXOR), and finally half gates required two ciphertexts *and* was compatible with FreeXOR.

Hence, unfortunately, each optimization breaks FreeXOR (i.e. they are incompatible), or the optimization only reduces the problem to sending 2 ciphertexts. The natural question is then, is there an optimization that requires 0 ciphertexts and is compatible with FreeXOR? From the last section we know that in their *Two Halves make a Whole*, Rosulek et al. state that this is impossible in a linear sense. They give a lower bound of two as to the number of ciphertexts required to send to Bob to evaluate an AND gate if the scheme uses are using linear techniques.

There are two main problems with finding FreeAND. The first is that since group operations are linear (e.g. FreeXOR is simply $C_2 \times C_2 \times ... \times C_2$ where C_2 is the cyclic group of order two) then finding FreeAND must combine **more** than a group operation. The second problem, and a big one at that, is that it must be compatible with FreeXOR. That's a big rock researchers have tripped over in the last few years. For instance, GRR2 gave optimal conditions to evaluate AND gates, but in the process broke XOR gates to two ciphertexts per gate.

With all of these constraints, we are essentially looking for a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ such that

$$
\begin{aligned}
f(k_a^0, k_b^0) &= k_c^0 \\
f(k_a^0, k_b^0) &= k_c^0 \\
f(k_a^1, k_b^1) &= k_c^0 \\
f(k_a^1, k_b^1) &= k_c^1
\end{aligned}
$$

for $k_c^0, k_c^1 \in \{0,1\}^n$ where $k_a^i \neq k_b^j \neq k_c^l$. Furthermore, we need it to be compatible with FreeXOR, so we impose an additional constriction that

$$
\begin{aligned}
k_a^1 &= k_a^0 \oplus S \\
k_b^1 &= k_b^0 \oplus S \\
k_c^1 &= k_c^0 \oplus S
\end{aligned}
$$

for some random element $S \in \{0,1\}^n$.

3.9.1 An (almost) successful approach

The first issue to overcome before thinking of compatibility is finding a scheme that is not linear. If the group C_2 (addition modulo 2) is not enough, then why don't we try to take the commutative ring $\mathbb{Z}_2$? We now have two operations, addition and multiplication, but more importantly, it wouldn't be linear in the strict sense, since not all elements of a ring have inverses (in fact, we will use that to our advantage).

Procedure:

Let f be bitwise AND, or in other words, multiplication in the commutative ring $\mathbb{Z}_2 \times \mathbb{Z}_2 \times ... \times \mathbb{Z}_2 = (\mathbb{Z}_2)^n = \{0,1\}^n$. Now, randomly choose k_a^0, k_b^0 and some S like in FreeXOR with the restriction on S that both $k_a^0 S = k_b^0 S = 0$ (notation: from now on $k_a^0 k_b^0$ means bitwise AND or multiplication on the Cartesian product of $\mathbb{Z}_2$, and 0 is the n bit 0 element in $(\mathbb{Z}_2)^n$).

Set $k_a^1 = k_a^0 \oplus S$, and $k_b^1 = k_b^0 \oplus S$ (note the similarities with FreeXOR; this is what's going to make it compatible later on). Now with all of this we have that:

$$\begin{aligned}
f(k_a^0, k_b^0) &= k_a^0 k_b^0 = k_c^0 \\
f(k_a^0, k_b^1) &= k_a^0(k_b^0 \oplus S) = k_a^0 k_b^0 \oplus k_a^0 S = k_a^0 k_b^0 \oplus 0 = k_c^0 \\
f(k_a^1, k_b^0) &= (k_a^0 \oplus S) k_b^0 = k_a^0 k_b^0 \oplus k_b^0 S = k_a^0 k_b^0 \oplus 0 = k_c^0
\end{aligned}$$

And finally, since we need $k_c^0 \neq k_c^1$, we have that

$$\begin{aligned}
f(k_a^1, k_b^1) &= (k_a^0 \oplus S)(k_b^0 \oplus S) \\
&= k_a^0 k_b^0 \oplus k_a^0 S \oplus k_b^0 S \oplus S^2 \\
&= k_a^0 k_b^0 \oplus 0 \oplus 0 \oplus S \quad \text{since } S^2 = S \\
&= k_a^0 k_b^0 \oplus S \\
&= k_c^0 \oplus S \\
&= k_c^1
\end{aligned}$$

So as long as $S \neq 0$, then $k_c^0 \neq k_c^1$. But even more than that, it is compatible with FreeXOR! Indeed, notice that like in Equation 1, we have set

$$\begin{aligned}
k_a^1 &= k_a^0 \oplus S \\
k_b^1 &= k_b^0 \oplus S \\
k_c^1 &= k_c^0 \oplus S
\end{aligned} \tag{3.1}$$

so the output of an `AND` gate can be pipelined with the input of an `XOR` gate to perform FreeXOR.

3.9.2 Problems

Unfortunately, this approach has some challenges it needs to overcome.

3.9.2.1 On finding the right S

Unlike FreeXOR, S is not totally random, as it has to be an element such that when multiplied by the zero labels will yield 0. Furthermore, there is an inherent danger with S and multiplication. On the one side, we need S to be different from 0. Without this, we would have that $k_c^0 = k_c^1$ and then Bob doesn't know the output of the boolean gate. On the other side, if we require $k_a^0 S = k_b^0 S = 0$, a bit in S in position i can be 1 only if both k_a^0 and k_b^0 have a 1 in position i. For example, say $k_a^0 = 0101010$ and $k_b^0 = 1100101$. Then S is forced to be $S = 010000$. This is an unfortunate consequence of multiplication, as 3/4 of the times multiplication yields 0.

A way to overcome this, however, is to have k_a^0 and k_b^0 agree upon at least having m 1s in m different positions in the bit string. Back to our previous example, when Alice generates k_a^0 and k_b^0, she can privately say that on positions 2, 4, and 7 k_a^0 and k_b^0 must have a 1. After that, they can be random, as in the equation below:

$$\begin{aligned} k_a^0 &= *1*1**1 \rightarrow 0111011 \\ k_b^0 &= *1*1**1 \rightarrow 1101001 \\ S &= 0101001 \end{aligned}$$

making S more balanced.

3.9.2.2 Generalizing to a Circuit

Up until now, we have been posing the problem and solution with only one gate in mind. However, we need to generalize to a circuit. Suppose we have the following circuit as in Figure 3.9 where we have set up everything accordingly so that $k_a^0 S_1 = k_b^0 S_1 = 0$ and $k_d^0 S_2 = k_e^0 S_2 = 0$. Then, the final gate will not work. The problem is that we don't have necessarily that $k_c^0(k_f^0 \oplus S_2) = k_c^0 k_f^0 \oplus k_c^0 S_2 = k_c^0 k_f^0$.

How do we solve this? We don't create our S_i per gate, but rather globally, as Kolesnikov and Schneider do in FreeXOR. In other words, we find an S such that $k_a^0 S = k_b^0 S = k_d^0 S = k_e^0 S = 0$. With this approach, the previous problem is solved, but the constraints posed on S gets bigger.

3.9.3 Questions

With all of this, the two big questions to investigate are:

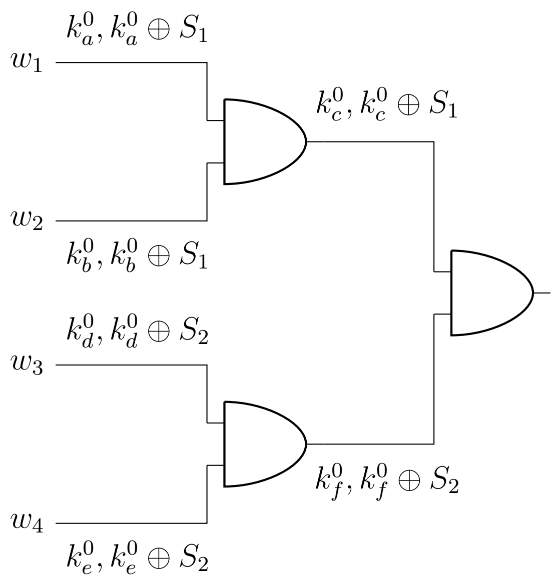

$k_a^0, k_a^0 \oplus S_1$
w_1
$k_c^0, k_c^0 \oplus S_1$
w_2
$k_b^0, k_b^0 \oplus S_1$
$k_d^0, k_d^0 \oplus S_2$
w_3
$k_f^0, k_f^0 \oplus S_2$
w_4
$k_e^0, k_e^0 \oplus S_2$

Figure 3.9: Different S in a circuit

1. Will imposing 1s on the input false labels make it insecure? In other words, if Alice secretly puts a 1 in m different positions on $k_a^0, k_b^0, ...$, can Bob use this to his advantage to leak Alice's input?

2. What happens when a circuit becomes **really** big? In their Fairplay System, Nigel Smart et al. build a compiler that has 30,000 gates. That means that there is 60,000 false input labels (i.e. $k_a^0, k_b^0, k_c^0, ...$). Assuming that we need a global S such that $k_a^0 S = k_b^0 S = ... = 0$, then it's practically impossible for S to be non-zero. As we mentioned before, can we globally agree to have 1s in certain positions in the labels? Is this secure? How many agreements do we need for it to be secure?

3.9.4 Answers

1. Unfortunately, posing the false labels to have m different 1s leaks information about Alice's input. Indeed, Bob can break the security of the protocol as follows:

   i) Bob simply chooses to receive via OT the true label k_b^1.

   ii) We now distinguish the two possible cases:

- Alice sends k_a^0: In which case, Bob XORs the two input labels $k_a^0 \oplus k_b^1 = k_a^0 \oplus k_b^0 \oplus S$. As both the two input labels have 1s agreed upon on m different positions, then XORing $k_a^0 \oplus k_b^0$ will yield m different zeros. This in turn implies $k_a^0 \oplus k_b^0 \oplus S$ will have less 0s in the bit string than $k_a^0 \oplus k_b^0$. Therefore, he can conclude based on the number of 0s in the bit string whether Alice chose k_a^0 or k_a^1.
- Alice sends k_a^1: In which case, Bob XORs the two input labels $k_a^1 \oplus k_b^1 = k_a^0 \oplus S \oplus k_b^0 \oplus S = k_a^0 \oplus k_b^0$. By the explanation above, if he sees much more zeros than ones, he concludes Alice sent k_a^1.

2. If the circuit becomes really big, then since we need $k_a^0 S = k_b^0 S = ... = 0$, inevitably $S \to 0$. As we can't have $S = 0$ (otherwise $k_c^0 = k_c^1$), and as we can't use what we proposed about imposing 1s on the false input labels, then this scheme is not fit for a FreeAND.

3.9.5 Impossibility of FreeAND in a Ring

We now show a result we have found that demonstrates FreeAND is impossible to achieve not only in a linear sense, but also on rings **if** we follow a FreeXOR approach.

Theorem 2. *Let R be a non-commutative ring. Let every true label $k_\alpha^1 \in R$ be a function g of the false label $k_\alpha^0 \in R$ as in FreeXOR, i.e., $k_\alpha^1 = g(k_\alpha^0)$ for some $g : R \to R$. Then there does not exist a function $f : R \times R \to R$ such that*

$$\begin{aligned} f(k_a^0, k_b^0) &= k_c^0 \\ f(k_a^0, k_b^1) &= k_c^0 \\ f(k_a^1, k_b^0) &= k_c^0 \\ f(k_a^1, k_b^1) &= k_c^1 \end{aligned}$$

for $k_\alpha^0, k_\alpha^1 \in R$ with $k_\alpha^0 \neq k_\alpha^1$ for $\alpha \in \{a, b, c\}$.

First of all, notice that for FreeXOR the function f chosen was $f = \oplus$, and the function g chosen was $k_\alpha^1 = g(k_\alpha^0) = k_\alpha^0 \oplus S$. In our unsuccessful approach, we chose $f =$ bitwise-AND, and we chose g as in FreeXOR.

Proof. Suppose for a contradiction such a function f exists. For simplicity, denote $f(k_a^i, k_b^j) = k_a^i k_b^j$. Then we have

$$k_a^0 k_b^0 = k_a^0 k_b^1 = k_a^1 k_b^0 \neq k_a^1 k_b^1$$

Since each true label is a function of the false label, we then have that

$$k_a^0 k_b^0 = k_a^0 g(k_b^0) = g(k_a^0) k_b^0 \neq g(k_a^0) g(k_b^0)$$

In particular, we have that

$$(g(k_a^0) - k_a^0) k_b^0 = 0 \quad \forall k_a^0, k_b^0 \in R$$

Since this holds for any label in R, set $k_b^0 \equiv g(k_b^0) + k_b^0$. We then have that

$$\begin{aligned}
&(g(k_a^0) - k_a^0) k_b^0 = 0 \\
&(g(k_a^0) - k_a^0)(g(k_b^0) + k_b^0) = 0 \quad \text{by substitution} \\
&g(k_a^0) g(k_b^0) - k_a^0 g(k_b^0) + g(k_a^0) k_b^0 - k_a^0 k_b^0 = 0 \\
&g(k_a^0) g(k_b^0) \underbrace{- k_a^0 g(k_b^0) + g(k_a^0) k_b^0}_{=0} - k_a^0 k_b^0 = 0 \\
&g(k_a^0) g(k_b^0) - k_a^0 k_b^0 = 0
\end{aligned}$$

But then $g(k_a^0) g(k_b^0) = k_a^0 k_b^0$, and therefore $k_c^0 = k_c^1$, breaking the definition of f. Therefore, such f doesn't exist. □

Chapter 4

Implementation

In this chapter we propose **Gabes**, a new implementation to garbled circuits that focuses on usability and ease-of-use instead of speed. Current implementations focus on speed instead of ease-of-use and simplicity. Offering an implementation that concentrates on the latter has more pedagogical benefits to the user, as every part of the implementation is out in the clear. The language of choice is `Python` instead of current choices such as `C` or `C++`, making the code easier to understand and reason about. Before we explain our implementation, we highlight some implementations already published.

4.1 Other Implementations

- `Fairplay`: This was the first practical implementation of garbled circuits written in [12]. Nowadays this implementation is obsolete, but it paved the way for current implementations.

- `FastGC`: This library was implemented in `Java` as part of the authors' paper *Faster Secure Two-Party Computation Using Garbled Circuits* [6]. The main contribution they give to the implementation of garbled circuits is the concept of *Pipelined Circuit Execution*. What they note is that previous implementations such as *Fairplay* loaded the whole garbled circuit into memory. However, this is not strictly necessary as the circuit can be evaluated per topographical level. Therefore, each gate is sent to the evaluator as soon as it is garbled, eliminating the bottleneck of having to wait for the whole circuit to be garbled before sending it to the evaluator.

- `JustGarble`: This implementation developed at the University of California, San Diego along the paper [3] offers general optimizations to both garble and evaluate the circuit. They introduce a circuit representation called Simple Circuit Description (SCD) to handle the circuit. The name for the project comes from the idea that garbled circuits should not be a component for secure multiparty computation, but rather a goal on its own, the idea of *just* garbling.
- `Obliv-C`: Another implementation developed at the University of Virginia in the paper [15] that acts as an extension to the `C` programming language.

4.2 Gabes

We now introduce the various aspects that compose **Gabes**. For a more detailed explanation and usage of the library, see the Appendix for the whole documentation.

4.2.1 Oblivious Transfer

One of the bottlenecks of garbled circuits is the 1-out-of-2 oblivious transfer protocol that must occur for each input wire the evaluator supplies. The OT protocol we implement in **Gabes** is based on RSA and works as follows:

1. The garbler is owner of the two possible messages (or labels) m_0 and m_1. He will send one of the two to the evaluator, but he will not know which one.
2. The garbler generates an RSA key-pair with public key (e, N) and private key (p, q, d).
3. The garbler generates two random numbers in the range $(0, N)$ denoted by x_0 and x_1.
4. The garbler sends to the evaluator both x_0 and x_1 as well as his public key.
5. The evaluator chooses which message he will like to receive based on his bit b.

6. He generates a random $k \in (0, N)$ and blinds x_b by sending to the garbler the value $v = (x_b + k^e) \mod N$.

7. The garbler will calculate both $k_0 = (v - x_0)^d \mod N$ and $k_1 = (v - x_1)^d \mod N$. One of them will equal k, but he doesn't know which.

8. The garbler finally sends $m'_0 = m_0 - k_0$ and $m'_1 = m_1 - k_1$, and the evaluator will simply add k to the message he wants to receive.

4.2.2 Circuit

4.2.2.1 Internal Representation of the Circuit

One of the main challenges that was faced making **Gabes** was the design and implementation of the circuit. Internally, the circuit is a `Python` object that has a pointer to the last gate of the circuit. This is the minimum requirement needed to define the circuit, as now each gate in the circuit can be reached by traversing the circuit through the last gate of the circuit.

A simpler way to think of it is that the circuit keeps a reference to the root of the tree, and any node (gate) in the tree can be reached by traversing the tree. In fact, we use an external library called `anytree` to implement the graph of the circuit.

4.2.2.2 Parsing

Ideally, the program would take any function and convert it into the circuit that is later used by the garbler and evaluator. However, we found out this was not trivial by any means and out of the scope of this project. Therefore, we decided that the user would be the one that provides the circuit as a logic expression. For instance, the user might provide the following expression:

$$((A \wedge B) \wedge (C \veebar D)) \wedge (E \veebar F)$$

and the program will create the circuit by parsing that expression as shown on Code Listing 4.1.

```
>>> from gabes.circuit import Circuit
>>> c = Circuit("sample1.circuit")
>>> c.draw_circuit()
AND
|--- AND
|    |-- AND
|    |-- XOR
|--- XOR
```

Code Listing 4.1: Drawing a Circuit

4.2.3 Gates

All gates are represented via the `Gate` object, where each `Gate` is a node in the tree. Each `Gate` contains a left, right, and output `Wire` object (more on this object on the Wire section).

It is in this class that all the garbling of the gates are performed. The main function is `garble()` (Code Listing 4.2), which delegates based on the flags set by the user to the function that garbles a specific optimization.

```
def garble(self):
    """
        Garbles the gate. Delegates to the correct
        optimization depending on the user's choice.
    """
    if settings.CLASSICAL:
        self.classical_garble()
    elif settings.POINT_AND_PERMUTE:
        self.point_and_permute_garble()
    elif settings.FREE_XOR:
        self.free_xor_garble()
    elif settings.FLEXOR:
        self.flexor_garble()
    elif settings.GRR3:
        self.grr3_garble()
    elif settings.GRR2:
        self.grr2_garble()
    elif settings.HALF_GATES:
        self.half_gates_garble()
```

Code Listing 4.2: Main function of `Gate`

For instance, for the classical garble, the function that will garble the table is as follows on Code Listing 4.3.

```
def classical_garble(self):
    """
        The most simple type of garbling.
        In classical garbled circuits,
        the whole boolean table is obfuscated
        by encrypting the output label using
        the input labels as keys.
        After this the table is shuffled
        (or *garbled*) so that the evaluator
        can't know more than one output label. For
        more information see `the paper
        <https://dl.acm.org/citation.cfm?id=1382944>`_.

        Note that a *Fernet* scheme is used
        since this method relies on knowing
        whether decryption was successful or not,
        as the evaluator needs to try and decrypt
        the four possible entries
        in the boolean table.
    """
    for left_label in self.left_wire.labels():
        for right_label in self.right_wire.labels():
            key1 = Fernet(left_label.to_base64())
            key2 = Fernet(right_label.to_base64())
            in1, in2 = left_label.represents, right_label.
                                             represents
            logic_value = self.evaluate_gate(in1, in2)
            output_label = self.output_wire.get_label(
                                             logic_value)
            pickled = pickle.dumps(output_label)
            table_entry = key1.encrypt(key2.encrypt(
                                             pickled))
            self.table.append(table_entry)
    shuffle(self.table)
```

Code Listing 4.3: Classical Garbling

4.2.4 Wires

The main function of the `Wire` object is to coordinate the two labels running through that wire. In particular, it must make sure that the labels have different point-and-permute bits, and in the case of FreeXOR or Half Gates, that the true label is an offset R of the false label. This can be perfectly seen in the initialization of the object (Code Listing 4.4).

```
class Wire(object):
    """
        The :class:`Wire` object holds two labels
        representing *True* and *False*. In
        classical garbled circuits, there is no
        need for a point-and-permute bit. In all
        the other cases, a pp_bit is associated
        to each label. The two labels
        in the same wire must have opposing pp_bits.

        If the optimization chosen is FreeXOR or
        Half Gates then the true label is the
        false label xored with the global
        parameter `R` defined in
        :class:`gabes.circuit.Circuit`.
    """
    def __init__(self, identifier=None):
        self.identifier = identifier
        if settings.CLASSICAL:
            self.false_label = Label(False)
            self.true_label = Label(True)
        else:
            b = random.choice([True, False])
            self.false_label = Label(False, pp_bit=b)
            self.true_label = Label(True, pp_bit=not b)
        if settings.FREE_XOR or settings.HALF_GATES:
            self.true_label.label = xor(self.false_label.
                                             label, settings.R
```

```
                                              )
```

Code Listing 4.4: Wire Object

4.2.5 Labels

The `Label` object stores both the truth value it represents and the actual label as a 256 random bit string. Labels will be sent through the network using the `pickle` library which allows `Python` objects to be sent through the network.

4.2.6 Use of Cryptography

The module `crypto.py` handles all the cryptography involved with garbled circuits. The external module `cryptography` offers a Fernet encryption scheme that suits well for classical garbled circuits as it shows if decryption was successful or not. However, for the majority of optimizations decrypting the zero ciphertext is necessary. Therefore, the encryption/decryption scheme used is AES. While probably an unfit choice for a secure application, AES suffices for simple applications.

For instance, on Code Listing 4.5 we can see how labels are set on GRR3.

```
def generate_zero_ciphertext(left_label, right_label):
    k1 = AESKey(left_label.to_base64())
    k2 = AESKey(right_label.to_base64())
    enc = k2.decrypt(k1.decrypt(bytes(settings.NUM_BYTES),
                                 unpad=False), unpad=
                                False)
    return enc
```

Code Listing 4.5: Decrypting the Zero Ciphertext

And we can check this really works:

```
>>> from gabes.crypto import AESKey,
                             generate_zero_ciphertext
>>> from gabes.label import Label
>>> left_label, right_label = Label(0), Label(1)
>>> key1 = AESKey(left_label.to_base64())
>>> key2 = AESKey(right_label.to_base64())
>>> enc = generate_zero_ciphertext(left_label,right_label)
>>> enc
b'\\\\\\x07\\x08\\xd8\\x05\\x8bX\\x1dE\\x05\\x83D...'
>>> key1.encrypt(key2.encrypt(enc, pad=False), pad=False)
b'\\x00\\x00\\x00\\x00\\x00\\x00...'
```

Code Listing 4.6: Testing the Zero Ciphertext

4.2.7 Garbler and Evaluator

The modules `garbler.py` and `evaluator.py` provide the communication protocol that the two parties undertake. First, the garbler and the evaluator establish a connection through a socket. Then the garbler creates the circuit and garbles all the gates. He then sends to the evaluator the wire identifiers so that the evaluator can choose which truth values to supply to each wire he controls. After this, the input labels are transferred to the evaluator. Following the garbled circuits protocol, the garbler's labels can be sent *as is*, as they are obfuscated so the evaluator can not learn anything. The evaluator's labels however are trickier, so a 1-out-of-2 oblivious transfer protocol must be followed for each input label the evaluator supplies.

Once the evaluator is in possession of all the input labels, he can reconstruct the circuit and send the final output label to the garbler. The garbler can then compare the label in his circuit and decide which truth value it corresponds. Finally, the garbler sends the evaluator the final truth value.

The whole process can be seen on the functions on Code Listing 4.7.

```
# from garbler.py
def garbler(args):
    """
    The main function of the application for the garbler.
    """
    print("Welcome, garbler.Waiting for the evaluator...")
    sock, client = net.connect_garbler(args.address)
    circ = Circuit(args.circuit)
    print("Circuit created...")
    identifiers = hand_over_wire_identifiers(client, circ)
    inputs = ask_for_inputs(identifiers)
    hand_over_labels(client, circ, inputs)
    hand_over_cleaned_circuit(client, circ)
    final_output = learn_output(client, circ)
    print("The final output of the circuit is: {}".format(
                                             final_output))
    sock.close()
    return final_output

# from evaluator.py
def evaluator(args):
    """
    The main function of the application for the
    evaluator.
    """
    print("Welcome, evaluator. Waiting for the garbler..."
                                             )
    sock = net.connect_evaluator(args.address)
    idents = request_wire_identifiers(sock)
    inputs = ask_for_inputs(idents)
    labels = request_labels(sock, idents, inputs)
    circ = request_cleaned_circuit(sock)
    print("Reconstructing circuit...")
    secret_output = circ.reconstruct(labels)
    final_output = learn_output(sock, secret_output)
    print("The final output of the circuit is: {}".format(
                                             final_output))
    sock.close()
    return final_output
```

Code Listing 4.7: Garbling and Evaluating

4.2.8 Running Gabes

Gabes runs as a command line interface application. The garbler and the evaluator both run on their terminals their respective roles. The garbler needs to tell the evaluator the address and port number to establish the connection. For instance, the garbler might run the command `gabes -g -grr3 -a localhost:5000 -c sample1.circuit`, while the evaluator would run on his terminal the command `gabes -e -grr3 -a localhost:5000`.

Chapter 5

Conclusion

In this project we studied the history, theory, and implementation of garbled circuits, an exciting, active subfield of cryptography. The last few years have seen many optimizations that have reduced the size of the table, and it seems, unless there's drastically new ways to approach this problem, that optimal solutions have been reached (in the linear sense).

We divided this project into two parts: theory and implementation of garbled circuits. However, the original idea was to focus on a theoretical, research project, leaving the implementation as a proof-of-concept rather than full-on library supporting all the optimizations. However, the practical aspect of the project turned out to be more difficult and challenging than expected, and the decision to make a full library supporting all the optimizations was made.

After spending the last two months solely working on the project, we can say that it has been an intense, rewarding journey. In fact, we believe it is the perfect type of project for the final year project at Imperial. The theory is challenging but understandable if enough hours are put in, and the implementation is tough but in the spirit of all the labs done throughout the four years at Imperial. Furthermore, the subfield of garbled circuits is pretty self-contained, and there is no external dependencies on other fields that might take away time on a project like this.

5.1 Summary of Work

As stated previously, the project was divided into a theoretical and practical side. We started out with the theoretical side by studying all the papers related to the field, starting out with Yao's construction in the late 1980's. After the last paper was read, we researched non-linear ways to optimize `AND` gates, failing to propose a new method but discovering along the way a new result on rings.

After completing and writing up the theoretical side, we proceeded in implementing the practical side by first researching on Github current implementations and extracting common themes among the libraries to include in our own. We then spent the remaining time coding and documenting the library we created.

5.2 Evaluation of the Project

We evaluate each aspect of the project, detailing the strengths and weaknesses encountered along the way.

5.2.1 Theory

- **Strengths**:
  - One of the questions when the project started was the amount of optimizations we could research and fully understand before the project was due. Originally, it was planned that all the optimizations except garbled gadgets were going to be talked about. Not only did we have time for that, but we also included garbled gadgets in the report, giving a full picture of the theory of garbled circuits since its inception until present day without leaving any key part.
  - We were able to understand what failed in the proposed optimization we discussed and were able to establish a new result that builds on top of the result mentioned in Half Gates. That is, we established there could be no FreeAND in a ring R that followed the technique used by FreeXOR.
- **Weaknesses**:

  - There were however some weaknesses to this side of the project. In particular, we didn't touch upon proofs of security and privacy in any of the optimizations discussed. This would have been a nice addition to the report to make it more complete. However, the difficulty of understanding the proofs in this field and the lack of time made it impossible to include all the proofs.
  - Although an ambitious aspect to begin with and probably not classifiable as a weakness, we couldn't come up with an optimization for `AND` gates that lowered the size of the table from two downwards. However, this is a very exciting part to research and can perfectly lead to research in a programme for an MPhil or a PhD.

5.2.2 Implementation

- **Strengths**:
  - One of the goals of the library was to be as clear and easy to use as possible. We believe the goal was reached. Using `Python` eased the process, as the code is written in a clean, commented way. Also, the OOP design makes it intuitive to understand to the developer (e.g. there is `Circuit`, `Wire`, `Label` objects resembling the theoretical aspect).
  - The library has a thorough documentation that details all the functions written for the library with examples of how to use it.
  - Most of the optimizations are included in the library, and the addition of future optimizations is very simple to include by attaching the optimization to the `garble()` function.
- **Weaknesses**:
  - The library is slower than other `C/C++` implementations. This is primarily because `Python` was used, but other practical optimizations (such as pipelining) weren't implemented. In a future release it would be nice to include practical optimizations to speed up the library.

- There was no library that supported polynomial interpolation over general fields. Of course if the field was of size p for any prime then we could've implemented ourselves. However, for a general p^a, we needed to create the field, a polynomial ring, and find an irreducible polynomial to perform division. We tried to implement this optimization of garbled circuits but found out that the process of finding an irreducible polynomial is not trivial at all. We also tried to fix some irreducible polynomials by predefining them for various field orders, but this is not flexible for the user, and therefore we have not included GRR2 in our library. Whenever a library comes up that supports general fields we can easily include GRR2 in our library.
- The circuit has to be written by the user, instead of providing the function to garble and then the library building the circuit from the function.

5.2.3 Comparison to Existing Solutions

Gabes has its advantages and disadvantages. While already talked about, this library focuses on ease-of-use while other existing libraries offer speed sacrificing legibility. In our case, this was intentional by design as we saw the practical side of this project as a proof of concept rather than a real life library to use on critical applications. Another advantage to our library is the fact that (almost) all the optimizations are included, and using one or the other is as easy as setting a flag. Other existing solutions either implement one of the optimizations or some of them. Finally, a big disadvantage of **Gabes** at the moment is the fact that the user has to supply the circuit. Ideally, the user would provide a `Python` function that takes two parameters and outputs one parameter, and the library would convert that into a circuit. However, we researched and found out that this was out of the scope of the project, but it would be a good addition to the future of the library.

5.3 Future Work

A distinction should be made between the future work of this project and the subfield of garbled circuits as a whole. Nevertheless, both of these have clear future work.

In terms of the project, an improvement of **Gabes** would go through speeding up the library via practical optimizations performed over the last few years. Mainly, the idea of pipelining so that the evaluator doesn't have to wait for the whole circuit to be created, but rather he can evaluate gates as soon as the garbler garbles them. Also, including GRR2 would be a nice extension so that the user has all the optimizations at hand. Finally, the fact that the user needs to write the circuit can be improved greatly by building the circuit from the function the user chooses.

In terms of garbled circuits, the future work runs by lowering that bound on the size of the table of each gate. The paper that introduced Half Gates also gave a lower bound for linear schemes, suggesting that any optimizations that lowered the number of ciphertexts per gate would have to be implemented much differently to current techniques. We'll see where we are at in a few years.

Bibliography

[1] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 565–577. ACM, 2016.

[2] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[3] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[4] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[5] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[6] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, pages 331–335. USENIX, 2011.

[7] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for xor gates that beats free-xor. In *International Cryptology Conference*, pages 440–457. Springer, 2014.

[8] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[9] Yehida Lindell. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*, pages 1005–1009. IGI Global, 2005.

[10] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4, page 9. USENIX, 2004.

[11] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[12] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *Asiacrypt*, volume 9, pages 250–267. Springer, 2009.

[13] A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[14] A. C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[15] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015.

[16] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

gabes Documentation

Release 0.1.0

Ignacio Navarro

Jun 15, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

Garbled Circuits allow two distrusting parties to compute a joint function while keeping their inputs private. More precisely, it allows Alice with input *x* and Bob with input *y* to compute a function *f(x, y)* without Alice ever knowing *y* and without Bob knowing *x*. The way it does so is by first translating *f* to a boolean circuit from which it will cleverly obfuscate or **garble** the circuit to allow the computation of *f* while keeping the inputs private.

The classical example is that of two millionaires who wish to find out who is richer without revealing their wealth. In that case, *f* becomes the ">" (greater than) function, and *x* and *y* are their wealth.

Gabes implements garbled circuits in Python. The application runs as a command line interface but the functions required to run garbled circuits can be used without the command line (see gabes).

CHAPTER TWO

INSTALLATION

At the command line either via easy_install or pip:

```
$ easy_install gabes
$ pip install gabes
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv gabes
$ pip install gabes
```

CHAPTER

THREE

USAGE

Each party will run their own instance of the program on their computer as a CLI app. The garbler will provide the IP and port number to establish the connection with the evaluator.

Note: Make sure to open the port when connecting between two different networks.

Garbler's Side:

```
gabes -g -grr3 -c Desktop/my-circuit.circuit -a localhost:5000
```

Evaluator's Side:

```
gabes -e -grr3 -a localhost:5000
```

CHAPTER FOUR

FLAGS

```
usage: gabes [-h] [-g] [-e] [-b bits] [-i identifier [identifier ...]]
                   [-c file] -a ip:port [-cl] [-pp] [-grr3] [-free] [-grr2]
                   [-fle] [-half]

Program to garble and evaluate a circuit.

optional arguments:
  -h, --help            show this help message and exit
  -g, --garbler         Set this flag to become the garbler
  -e, --evaluator       Set this flag to become the evaluator
  -b bits, --bits bits  Include your private input bitstring to the circuit
                        (e.g. 001011)
  -i identifier [identifier ...], --identifiers identifier [identifier ...]
                        Indicate which input wires you supply to the circuit
                        (e.g. -i A C D)
  -c file, --circuit file
                        Path of the file representing the circuit. Only the
                        garbler needs to supply the file
  -a ip:port, --address ip:port
                        IP address followed by the port number
  -cl, --classical      Set this flag for classical garbled circuits
  -pp, --point-and-permute
                        Set this flag to include point-and-permute
  -grr3, --grr3         Set this flag for GRR3 garbled circuits
  -free, --free-xor     Set this flag for free-xor garbled circuits
  -fle, --flexor        Set this flag for flexor garbled circuits
  -half, --half-gates   Set this flag for half gates garbled circuits
```

CHAPTER FIVE

DOCUMENTATION

All the documentation can be found in https://gabes.readthedocs.io/en/latest/

CHAPTER

SIX

INTERFACE

Details of functions and classes are given in this section.

6.1 API

This part of the documentation explains each function or class in detail to better understand the internal details behind garbled circuits.

6.1.1 Circuit

This module implements the `Circuit` object which is in charge of maintaining the tree hierarchy for the circuit. This includes handling all the gates in the circuit as well as keeping track of global parameters (for instance R in FreeXOR). It also parses the .circuit file into the tree via the function `gabes.circuit.build_tree()`.

At the moment, the structure of the .circuit file must be like the one shown in examples; that is, each child gate of a gate (except the root) must be surrounded by parenthesis.

class `gabes.circuit.`**`Circuit`**(*file*)

The `Circuit` object holds all the gates and wires composing the circuit. Internally, it is represented as a binary tree. It also sets *R* for FreeXOR or Half Gates if necessary.

Parameters **`file`** (`str`) – path of the file describing the circuit

`build_tree`(*file*)

Builds the tree recursively by parsing and breaking up the circuit file into smaller expressions until the expression left is the wire's identifier. On each gate it visits the function also garbles it in preparation for the evaluator.

Parameters **`file`** (`str`) – path of the file describing the circuit

Returns the root node as used in the package *anytree*

Return type `anytree.Node`

`clean`()

Cleans the circuit from any private labels in preparation for the evaluator. Before cleaning, it creates a copy of itself so that the garbler still has a reference of the mapping between labels and truth values.

`draw_circuit`()

Draws the tree structure of the circuit.

```
>>> from gabes.circuit import Circuit
>>> c = Circuit('gabes/circuits/simple-2.circuit')
>>> c.draw_circuit()
AND
├── AND
│   ├── AND
│   └── XOR
└── XOR
```

reconstruct(*labels*)

Function used by the evaluator to reconstruct the circuit given only the input labels by the garbler. The reconstruction needs to be done in a bottom-up approach since the output labels of input gates will serve as input labels for parent gates. The function traverses the tree by level starting at the leaves. If the nodes are leaves, then the labels will be provided through the network by the garbler. For all other nodes, the evaluator will have the necessary labels as part of the node's children by a process of ungarbling (see `gabes.gate.Gate.ungarble()`).

Parameters **labels** (list(`Label`)) – the list of input labels supplied by the garbler

Returns the final output label at the end of the circuit

Return type `Label`

6.1.2 Gate

This module implements the `Gate` object. The bulk of **gabes** resides on this module. In it, both garbling and ungarbling (or evaluating) techniques are implemented.

class gabes.gate.**Gate**(*gate_type*, *create_left=True*, *create_right=True*)

The `Gate` object contains three wires: a left wire, a right wire, and an output wire, each having a false label and a true label. Depending on the settings, different optimizations will be used to garble and ungarble.

Parameters

- **gate_type** (*str*) – type of gate (AND, OR, etc)
- **create_left** (*bool*) – whether to create the left wire on the gate's initialization
- **create_right** (*bool*) – whether to create the right wire on the gate's initialization

classical_garble()

The most simple type of garbling. In classical garbled circuits, the whole boolean table is obfuscated by encrypting the output label using the input labels as keys. After this the table is shuffled (or *garbled*) so that the evaluator can't know more than one output label. For more information see the paper.

Note that a *Fernet* scheme is used since this method relies on knowing whether decryption was successful or not, as the evaluator needs to try and decrypt the four possible entries in the boolean table.

classical_ungarble(*garblers_label*, *evaluators_label*)

The classical evaluation, in which the evaluator tries the four possible table entries until one of them decrypts the cipher.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type `Label`

evaluate_gate(*input1*, *input2*)

Evaluates a gate given two inputs.

Parameters

- **input1** (*bool*) – the first input
- **input2** (*bool*) – the second input

Returns the output of the gate

Return type bool

flexor_garble()

In this optimizationn *XOR* are garbled with a table size of 0, 1, or 2 (hence its name flexible XORs). The innovation at the time was that this method is compatible with GRR2. The way it accomplishes this is by changing the input wires' labels to have the same offset as the output wire's labels. For more information see the paper.

flexor_ungarble(*garblers_label*, *evaluators_label*)

Transforms the two input labels to have the same offset as the output's true label.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type `Label`

free_xor_garble()

In this optimization *XOR* gates are garbled for free, that is, the table corresponding to this gate is empty. The way this optimization accomplishes this is by setting the true label of each wire as an offset *R* of the false label. This offset is global to the whole circuit, so by the properties of *XOR*, everything works out nicely. For more information see the paper.

Note that FreeXOR is not compatible with GRR2.

free_xor_ungarble(*garblers_label*, *evaluators_label*)

Evaluates *XOR* gates for free by XORing the two labels he receives.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type `Label`

garble()

Garbles the gate. Delegates to the correct optimization depending on the user's choice.

grr3_garble()

In this optimization the entry corresponding to the two labels that have a false point-and-permute bit is not sent over the network. Instead, the output label corresponding to this entry is set to be equal to the decryption of the zero ciphertext. Therefore, there is no need to send the entry because it is simply the zero ciphertext. The only thing the evaluator needs to do is to conclude that if he receives two false point-and-permute bits, the ciphertext will be the all zeros. For more information see the paper.

Note that now *Fernet* schemes can not be used since there is no way to decrypt the zero ciphertext. Instead, AES is used. See the Cryptography section for more details.

grr3_ungarble(*garblers_label*, *evaluators_label*)
If the point-and-permute bits are false, then imagine the ciphertext was the all zero ciphertext. Otherwise, proceed as in the point-and-permute optimization.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type Label

half_gates_garble()
In this optimization, the most current one to date, the authors propose a method to garble *AND* gates with a table size of two ciphertexts in a way that is compatible with FreeXOR. The way they accomplish this is by breaking up an *AND* gate into two *half gates*. For more information see the paper.

half_gates_ungarble(*garblers_label*, *evaluators_label*)
Evaluates the gate by decrypting each half gate and XORing the result.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type Label

modify_pp_bits(*A0_*, *B0_*, *C0_*)
Modifies the point-and-permute bits according to the last bit of the label.

point_and_permute_garble()
In this optimization each label has a point-and-permute bit associated to it, with the only rule that labels running in the same wire must have opposing point-and-permute bits. The garbler will insert the encrypted output labels according to the point-and-permute bit of the input labels. Therefore, now the evaluator does not need to try and decrypt all the four ciphers but rather the one indicated by the two point-and-permute bits he has. For more information see the paper.

point_and_permute_ungarble(*garblers_label*, *evaluators_label*)
Evaluates the gate by indexing the table according to the point-and-permute bits given.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type Label

set_zero_ciphertext()
Generates the zero ciphertext by taking the two labels with false point-and-permute bits and setting the output labels accordingly. This function is used for GRR3.

transform_label(*label*, *garbler=True*)
Transforms the label accordingly.

Parameters

- **label** – the label to transform

- **garbler** (*bool*) – the type of label supplied

ungarble (*garblers_label, evaluators_label*)
Ungarbles the gate. Delegates to the correct optimization depending on the user's choice.

Parameters

- **garblers_label** – the chosen label by the garbler
- **evaluators_label** – the chosen label by the evaluator

Returns the correct output label

Return type `Label`

update_output_wire (*false_label, true_label*)
Updates the output wire's labels and point-and-permute bits.

Parameters

- **false_label** – the false label
- **true_label** – the true label

wires ()
Returns the three wires related to the gate.

Returns the three wires

Return type list(`Wire`)

6.1.3 (*The*) Wire

Note: *"The king stay the king"*

- D'Angelo Barksdale

This module implements the *Wire* object. Each gate will have a left wire (in the case of input gates, this will be probably be supplied by the garbler), a right wire (evaluator), and an output wire. Each wire holds the two possible for labels that run through it.

class gabes.wire.**Wire** (*identifier=None*)
The *Wire* object holds two labels representing *True* and *False*. In classical garbled circuits, there is no need for a point-and-permute bit. In all the other cases, a pp_bit is associated to each label. The two labels in the same wire must have opposing pp_bits.

If the optimization chosen is FreeXOR or Half Gates then the true label is the false label xored with the global parameter *R* defined in *gabes.circuit.Circuit*

Parameters **identifier** (*str*) – (optional) wire's unique identifier

```
>>> from gabes.wire import Wire
>>> w = Wire(identifier='A')
>>> str(w)
'Wire A'
>>> w.false_label
b'dnE2Gsvhx84HgwrLRm8L9aFtI_aBYxzEDaOBRK2qkP0='
>>> w.true_label
b'eBRWiJzYL65gU8nBFvXRZ8NK4_Cf9GlrYtNGZNEZOSs='
>>> w.get_label(True) == w.true_label.represents
True
```

get_label(*representing*)
Gets the label according to which truth value it represents.

Parameters **representing** (*bool*) – *True* for the true label and *False* for the false label

Returns the corresponding label

Return type Label

labels()
A getter method to get the two labels (*False* and *True*) going through the wire.

Returns a tuple of labels

Return type Generator[*Wire*]

6.1.4 Label

This module implements the *Label* object. A label represents an obfuscated truth value. By default, the label is represented as a random 256 bitstring, but the label is encoded in base64 for the user. To change the length of the bitstring, head to gabes.settings.

class gabes.label.**Label**(*represents*, *pp_bit=None*)
The *Label* object, which contains the label that will represent either the boolean *False* or *True* for a particular gate.

Parameters

- **represents** (*bool*) – (optional) the boolean value this label represents
- **pp_bit** (*bool*) – (optional) the point-and-permute bit

```
>>> from gabes.label import Label
>>> label = Label(0, pp_bit=True)
>>> label.label
b'y\x8c\xc4C\x99\x9c\x1d&\xa3R\xdbB\xcep-\xc5
\xe9R=\xc1\xd8\xaeq}\xe0c\x80\xd8g\xac_\x96'
>>> label.to_base64()
b'eYzEQ5mcHSajUttCznAtxelSPcHYrnF94GOA2GesX5Y='
```

to_base32()
Returns the label encoded in base32.

Returns the label in base32

Return type str

to_base64()
Returns the label encoded in base64.

Returns the label in base64

Return type str

6.1.5 Garbler

This module provides the communication protocol seen from the point of view of the garbler. First, the garbler and the evaluator establish a connection through a socket. Then the garbler creates the circuit and garbles all the gates. He then sends to the evaluator the wire identifiers so that the evaluator can choose which truth values to supply to each wire he controls. After this, the input labels are transferred to the evaluator. Following the garbled circuits protocol, the garbler's labels can be sent *as is*, as they are obfuscated so the evaluator can not learn anything. The evaluator's labels however are trickier, so a 1-out-of-2 oblivious transfer protocol must be followed for each input label the evaluator supplies.

Once the evaluator is in possesion of all the input labels, he can reconstruct the circuit and send the final output label to the garbler. The garbler can then compare the label in his circuit and decide which truth value it corresponds. Finally, the garbler sends the evaluator the final truth value.

`gabes.garbler.`**`garbler`**`(`*args*`)`
The main function of the application for the garbler. For more information on the process, see above.

Parameters **`args`** – the arguments from the command line interface

Returns the output of the circuit

Return type bool

`gabes.garbler.`**`hand_over_wire_identifiers`**`(`*client*, *circ*`)`
Sends the wire identifiers to the evaluator.

Parameters

- **`client`** – the client is the evaluator
- **`circ`** – the circuit to which the wires belong

Returns the identifiers of the input wires

Return type list(str)

`gabes.garbler.`**`hand_over_cleaned_circuit`**`(`*client*, *circ*`)`
Sends a clean circuit (in which every label's *represents* flag has been deleted) to the evaluator.

Parameters

- **`client`** – the client is the evaluator
- **`circ`** – the circuit in question

`gabes.garbler.`**`hand_over_labels`**`(`*client*, *circ*, *garbler_inputs*`)`
Sends the input labels of the circuit to the evaluator. The labels that belong to the garbler can be sent without any modification. In order for the evaluator to learn his labels, he must acquire them through the oblivious transfer protocol, in which the garbler inputs the two possible labels, the evaluator inputs his choice of truth value, and the evaluator learns which label corresponds to his truth value without the garbler learning his choice and without the evaluator learning both labels.

Parameters

- **`client`** – the client is the evaluator
- **`circ`** – the circuit in question
- **`garbler_inputs`** – the inputs the garbler provides

`gabes.garbler.`**`learn_output`**`(`*client*, *circ*`)`
Learns the final truth value of the circuit by comparing the label that was handed to him by the evaluator to the two labels in the root of the tree (i.e. the final gate).

Parameters

- **`client`** – the client is the evaluator
- **`circ`** – the circuit in question

Returns the output of the circuit

Return type bool

6.1.6 Evaluator

This module provides the communication protocol seen from the point of view of the evaluator. To learn the whole process, see the Garbler's section.

`gabes.evaluator.`**`evaluator`**`(`*args*`)`

The main function of the application for the evaluator. For more information on the process, see the introduction to the Garbler's section.

Parameters **`args`** – the arguments from the command line interface

Returns the output of the circuit

Return type bool

`gabes.evaluator.`**`request_cleaned_circuit`**`(`*sock*`)`

Receives a clean circuit (in which every label's *represents* flag has been deleted) from the garbler.

Parameters **`sock`** – the socket from which it will receive the data

Returns the cleaned circuit

Return type `Circuit`

`gabes.evaluator.`**`request_wire_identifiers`**`(`*sock*`)`

Receives the wire identifiers from the garbler.

Parameters **`sock`** – the socket from which it will receive the data

Returns the identifiers of the input wires

Return type list(str)

`gabes.evaluator.`**`request_labels`**`(`*sock*, *identifiers*, *evaluator_inputs*`)`

Receives the input labels of the circuit from the garbler. The labels that belong to the garbler can be sent without any modification. In order for the evaluator to learn his labels, he must acquire them through the oblivious transfer protocol, in which the garbler inputs the two possible labels, the evaluator inputs his choice of truth value, and the evaluator learns which label corresponds to his truth value without the garbler learning his choice and without the evaluator learning both labels.

Parameters

- **`sock`** – the socket from which it will receive the data
- **`identifiers`** – the identifiers for all the input wires
- **`evaluator_inputs`** – the inputs the evaluator provides

Returns the input labels

Return type list(`Label`)

`gabes.evaluator.`**`learn_output`**`(`*sock*, *secret_output*`)`

Sends the final label and learns the final truth value from the garbler.

Parameters

- **`sock`** – the socket from which it will receive the data
- **`secret_output`** – the final label of the circuit

Returns the output of the circuit

Return type bool

6.1.7 Cryptography

This module handles all the cryptography involved with garbled circuits. The external module `cryptography` offers a Fernet encryption scheme that suits well for classical garbled circuits as it shows if decryption was successful or not. However, for the majority of optimizations decrypting the zero ciphertext is necessary. Therefore, the encryption/decryption scheme used is AES. While probably an unfit choice for a secure application, AES suffices for simple applications. If more security is needed, the recommendation is to change AES for a stronger cryptographic encryption scheme such as AES256.

class `gabes.crypto.`**`AESKey`**(*key*)

The *`AESKey`* object handles the key to AES and the encryption/decryption routines. To ensure that the key can be fed into AES, the input to the object is hashed with SHA256 to a 32 bytestring (AES only allows 16/32/64 bytes inputs).

Parameters **`key`** (*`bytes`*) – parameter to be hashed and used as a key

```
>>> from gabes.crypto import AESKey
>>> from gabes.label import Label
>>> label = Label(1)
>>> key = AESKey(label.to_base64())
>>> enc = key.encrypt(b"The winner is...")
>>> key.decrypt(enc)
b'The winner is...'
```

`decrypt`(*msg*, *from_base64=False*, *unpad=True*)

Decrypts the message `msg` by first unpadding it or decoding it from base64 if necessary.

Parameters

- **`msg`** (*`bytes`*) – the message to be decrypted
- **`from_base64`** (*`bool`*) – (optional) whether to decode the cipher from base64
- **`unpad`** (*`bool`*) – (optional) whether to unpad the message

Returns decrypted message

Return type bytes

`encrypt`(*msg*, *to_base64=False*, *pad=True*)

Encrypts the message `msg` by first padding it if necessary since AES requires prespecified input sizes. It then converts the cipher into base64 if needed.

Parameters

- **`msg`** (*`bytes`*) – the message to be encrypted
- **`to_base64`** (*`bool`*) – (optional) whether to convert the cipher to base64
- **`pad`** (*`bool`*) – (optional) whether to pad the message

Returns encrypted message

Return type bytes

`pad`(*msg*, *size=16*)

Takes a bytestring and pads it to be a multiple of `size`. To keep track of the padding, the first four bytes store the size of the padded bytestring.

Parameters

- **`msg`** (*`bytes`*) – the bytestring to pad
- **`size`** (*`int`*) – (optional) padded result must be a multiple of this number

Returns padded bytestring

Return type bytes

unpad (*msg*)
Takes a bytestring and unpads it to the original bytestring. Since the first four bytes store the original unpadded *size* of the bytestring, we extract those four bytes and return the bytestring from position 4 to `size + 4`.

Parameters **msg** (*bytes*) – the bytestring to unpad

Returns unpadded bytestring

Return type bytes

gabes.crypto.**generate_zero_ciphertext** (*left_label*, *right_label*)
Generates the label *c* that when decrypted using the `left_label` and `right_label` keys will yield the zero ciphertext.

Parameters

- **left_label** (Label) – left label to use as key
- **right_label** (Label) – right label to use as key

Returns encrypted text

Return type bytes

```
>>> from gabes.crypto import AESKey, generate_zero_ciphertext
>>> from gabes.label import Label
>>> left_label, right_label = Label(0), Label(1)
>>> key1 = AESKey(left_label.to_base64())
>>> key2 = AESKey(right_label.to_base64())
>>> enc = generate_zero_ciphertext(left_label, right_label)
>>> enc
b'\\\x07\x08\xd8\x05\x8bX\x1dE\x05\x83D ?\xe6
\x10\\\x07\x08\xd8\x05\x8bX\x1dE\x05\x83D ?\xe6\x10'
>>> key1.encrypt(key2.encrypt(enc, pad=False), pad=False)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

6.1.8 Network

This module is in charge of handling all the communication between the garbler and the evaluator, providing an easy API to hide the lower-level sockets.

gabes.network.**connect_garbler** (*address*)
Connects the garbler to the socket. The garbler will act as the server, and the evaluator as the client.

Parameters **address** (*str*) – the address of the socket (IP and the port number in the format IP:port)

Returns the socket and the client (the evaluator)

gabes.network.**connect_evaluator** (*address*)
Connects the evaluator to the socket. The garbler will act as the server, and the evaluator as the client.

Parameters **address** (*str*) – the address of the socket (IP and the port number in the format IP:port)

Returns the socket

`gabes.network.`**`send_data`**(*sock*, *data*)

Sends `data` through the socket. The data is pickled so that objects can be sent through the socket. As the socket can accept a fixed size number of bytes, the function sends the size of the data to know how many bytes to receive through the network.

Parameters

- **`sock`** – the socket or the client
- **`data`** (*`bytes`*) – the data to send through the socket

`gabes.network.`**`receive_data`**(*from_whom*)

Receives data through the socket.

Parameters **`from_whom`** – either the client (evaluator) or the socket (the garbler)

Returns the unpickled data

`gabes.network.`**`send_ack`**(*sock*)

Sends an *ACK* through the socket. This will be useful for the OT protocol.

Parameters **`sock`** – either the client (evaluator) or the socket (the garbler)

`gabes.network.`**`wait_for_ack`**(*sock*)

Waits until it receives an *ACK* through the socket. This will be useful for the OT protocol.

Parameters **`sock`** – either the client (evaluator) or the socket (the garbler)

6.1.9 Oblivious Transfer

This module implements 1-out-of-2 oblivious transfer. Essentially, the garbler inputs to the protocol two messages `m0` and `m1`, while the evaluator inputs a single bit `b`. The garbler learns nothing from this protocol and the evaluator learns either `m0` or `m1` depending on his bit `b`, but not both. The OT protocol followed in this module is the following:

1. The garbler generates an RSA public/private key pair and sends the public portion `(e, N)` to the evaluator along with two random messages `x0` and `x1`.
2. The evaluator generates a random `k` and depending on his bit `b` sends to the garbler `v = (xb + k ^ e) mod N`.
3. The garbler computes both `k0 = (v - x0) ^ d mod N` and `k1 = (v - x1) ^ d mod N`. One of these will equal `k`, but he doesn't know which.
4. The garbler sends `m0_ = m0 + k0` and `m1_ = m1 + k1` to the evaluator.
5. The evaluator decrypts depending on his bit `mb_ = mb - k`, learning only `m0` or `m1`.

`gabes.ot.`**`garbler_ot`**(*client*, *m0*, *m1*)

The OT protocol seen from the point of view of the garbler. This includes creating the RSA key pair, generating `x0` and `x1`, computing `k0` and `k1`, and sending `m0_` and `m1_`. Note that pickling of m0 and m1 is done beforehand for it to be possible to send `Label` objects.

Parameters

- **`client`** – the evaluator's address
- **`m0`** (*`bytes`*) – the first bytes object (in this case, a label)
- **`m1`** (*`bytes`*) – the second bytes object (in this case, a label)

`gabes.ot.`**`evaluator_ot`**(*sock*, *b*)

The OT protocol seen from the point of view of the evaluator. This includes choosing the random `k`, sending `v`, and learning either `m0` or `m1`.

Parameters

- **`sock`** – the garbler's address
- **`b`** (*`bool`*) – the evaluator's bit

6.1.10 Utils

This module includes utility functions used throughout the package.

`gabes.utils.`**`ask_for_inputs`**(*identifiers*)

CLI helper function that queries the user to indicate which identifier he supplies and the his choice for each identifier.

Parameters **`identifiers`** (*`list(str)`*) – the identifiers of the input wires

Returns the identifiers the user supplies

Return type dict

`gabes.utils.`**`get_last_bit`**(*label*)

Gets the last bit from a bytestring.

Parameters **`label`** (*`bytes`*) – any bytes object

Returns the last bit

Return type bool

```
>>> import os
>>> from gabes.utils import get_last_bit
>>> b1 = os.urandom(10)
>>> b1
b'\xf3\x9e\xb0w,|\xd9\xa8\xd73'
>>> get_last_bit(b1)
False
```

`gabes.utils.`**`xor`**(*b1*, *b2*)

XORs two bytestrings.

Parameters

- **`b1`** (*`bytes`*) – first argument
- **`b2`** (*`bytes`*) – second argument

Returns the XORed result

Return type bytes

```
>>> import os
>>> from gabes.utils import get_last_bit
>>> b1 = os.urandom(10)
>>> b2 = os.urandom(10)
>>> b1, b2
(b'\xf8\x00r\xaf\x9a\x06!68\x83', b'\x88
\xee\x1c,a\xd0^\x8a\xb4\xf2')
>>> xor(b1, b2)
```

(continues on next page)

(continued from previous page)

```
b'p\xeen\x83\xfb\xd6\x7f\xbc\x8cq'
>>> xor(b1, xor(b1, b2)) == b2
True
```

`gabes.utils.`**`adjust_wire_offset`**`(`*wire*`)`

Adjusts the wire's offset so that the two labels have a distinct last bit.

Parameters **`wire`** – the wire in question

CHAPTER SEVEN

FEEDBACK

If you have any suggestions or questions about **gabes** feel free to email me at nachonavarroasv@gmail.com.

If you encounter any errors or problems with **gabes**, please let me know! Open an Issue at the GitHub http://github.com/nachonavarro/gabes main repository.

PYTHON MODULE INDEX

INDEX