A Survey of Garbled Circuit Techniques

Ben Terner

1 Introduction

Consider two billionaires eating dinner at a very expensive restaurant. When they finish their meal, the waiter delivers a very expensive bill and expects a large tip. The billionaires begin to quibble over who should pay, and they resolve that the richer one should pay the bill and the other cover the tip. Assume that neither billionaire wishes to reveal his true net worth to the other, for fear that the other may reveal his holdings to the IRS. Can the billionaires accurately compare their true net worths to discover who is richer, and more importantly, who should pay the bill?

Andrew Yao proposed a very similar problem in 1982 [1] (back when a million dollars was a lot of money), opening up a vast and fruitful field of research known as secure computation. Consider a situation in which two mutually distrustful parties wish to compute a joint function on their inputs, but do not wish to involve or do not have the luxury of knowing a trusted third party. Secure computation allows distrustful parties to jointly compute arbitrary functions while preventing any player or onlooker from learning more about another player's input than what she could deduce from her own input and output.

This survey paper will review the garbled circuit scheme for secure computation originally proposed by Yao and give several extended constructions that improve its efficiency. Where we have covered a primitive or security notion in class, this document will elide the definition and generally treat the primitive as a black box. Where new notions or constructions are necessary for understanding the garbled circuit construction, they will be given, although full proofs will usually not be given for the sake of space.

2 Yao's Garbled Circuits

The garbled circuit protocol is an interactive algorithm between two players, whom we will name *Gen* and *Eval*. *Gen* and *Eval* will jointly compute a function $f(x, y) = (f_x(x, y), f_y(x, y))$, where *Gen*'s input and output are given as $(x, f_x(x, y))$ and *Eval*'s input and output are given as $(y, f_y(x, y))$. We will assume for the sake of simplicity that $f_x(x, y) = f_y(x, y)$, *i.e.* *Gen* and *Eval* receive the same output from the protocol, and denote it simply as $f(x, y)$. For the sake of simplicity, also assume that $x, y, f(x, y) \in \{0, 1\}^n$ for some integer n.

Let $(G(1^n), E_k(m) = c, D_k(c) = m)$ be a private key encryption scheme such that $G(1^n)$ generates an encryption key k, $E_k(m)$ is an encryption algorithm using key k on message m that produces cipher text c, and D is a decryption algorithm that uses k to decrypt c and retrieve m.

Let OT be an oblivious transfer protocol between two players $P1$ and $P2$, where $P1$'s input is (m_0, m_1) and P_2's input is $b \in \{0, 1\}$. $P1$'s output from the protocol is $\perp$, and $P2$'s output is (m_b).

Yao's Garbled Circuit scheme is an interactive protocol in which *Gen* will *generate* a garbled circuit and *Eval* will *evaluate* it to discover the outputs. We will assume that *Eval* honestly relays the output to *Gen* when it has completed the output. This assumption can be removed for the final construction.

2.1 A Single Gate

We will show how to garble a single gate from the binary circuit which *Gen* and *Eval* compute, drawing heavily from the explanation given by Lindell and Pinkas in [2]. The construction of a full circuit follows inductively from the guarantees provided by the encryption of a single gate. Begin by considering, for example, an AND gate:

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: A Standard AND Gate

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

Table 2: A Standard XOR Gate

Garbling the gate will require obscuring the inputs to each gate and encrypting the outputs such that knowing the output of a gate betrays no information about its true value. We will replace the inputs and the outputs with keys that are mapped to their true values, and the mapping will be kept secret. Begin by generating six encryption keys by repeatedly invoking $G(1^n)$ and name them $k_x^0, k_x^1, k_y^0, k_y^1, k_z^0, k_z^1$. We replace the above standard gate with a new "garbled" gate and then define a new function for evaluating its output.

x	y	z
k_x^0	k_y^0	k_z^0
k_x^0	k_y^1	k_z^0
k_x^1	k_y^0	k_z^0
k_x^1	k_y^1	k_z^1

Table 3: A Garbled AND Gate

x	y	z
k_x^0	k_y^0	k_z^0
k_x^0	k_y^1	k_z^1
k_x^1	k_y^0	k_z^1
k_x^1	k_y^1	k_z^0

Table 4: A Garbled XOR Gate

Notice the convention that the subscript of k maps it to a specific input variable or *wire*, and the superscript describes the value of that input from the table above, or equivalently the *semantic* value of the wire. We now define a new function for evaluating a gate. Whereas for a standard gate, we define the function

$$g(x, y) : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$$

we now define the new function

$$garbledgate(k_x^\alpha, k_y^\beta) : k_z^\alpha \times k_y^\beta \to k_z^{g(\alpha,\beta)}$$

for $\alpha, \beta \in \{0, 1\}$, and $g(\alpha, \beta)$ defined by the truth table of the standard gate it computes. A single arbitrary garbled row is defined by

$$c_{x,y} = E_{k_x^\alpha}(E_{k_y^\beta}(k_z^{g(\alpha,\beta)}))$$

Or, more verbosely, a full garbled gate table is completely defined by the values

$$c_{0,0} = E_{k_x^0}(E_{k_y^0}(k_z^{g(0,0)}))$$

$$c_{0,1} = E_{k_x^0}(E_{k_y^1}(k_z^{g(0,1)}))$$

$$c_{1,0} = E_{k_x^1}(E_{k_y^0}(k_z^{g(1,0)}))$$

$$c_{1,1} = E_{k_x^1}(E_{k_y^1}(k_z^{g(1,1)}))$$

Given c, the function *garbledgate* is evaluated by decrypting c using both of the input keys to discover the output key.

2.2 Circuits Composed Of Gates

To garble a complete circuit, *Gen* will compute all of the cipher texts described in the construction above and send them to *Eval* for decryption using the input keys. *Gen* composes gates to construct a circuit by using the output key (denotes as k_2 above) of one gate as the input key to the next gate. Gates with fan-out of more than 1 will have their output keys used as input to multiple gates.

There are 2 glaring issues with the above scheme. First, if *Eval* knows all of the keys k, then *Eval* can decrypt every possible output key for every gate in the circuit. Second, *Eval* can learn the semantic value hidden by each key simply by the order of the cipher texts it receives for each gate.

To handle the first issue, we maintain the invariant that *Eval* knows only two of the input keys, k_0^α and k_1^β for each gate that it attempts to decrypt. If *Eval* attempts to decrypt a c for which she does not have both keys, she will receive $\perp$. Second, *Gen* can obscure which key *Eval* decrypts by randomly permuting the cipher texts for a gate before sending them to *Eval*. This hides the values of α and β from *Eval*, but it forces *Eval* to attempt to decrypt every cipher text. The scheme for both of these issues will be revisited and improved in section 4.

2.3 Input Privacy

In order to evaluate the circuit, *Eval* must learn the keys that correspond to both *Gen*'s inputs and its own inputs. *Gen* trivially sends the keys for its inputs to *Eval*, sending $k_{x_i}^\alpha$ for the value α of every one of its inputs x_i, revealing nothing about $x_i = \alpha$. To learn the keys $k_{y_i}^\beta$ for each of its own inputs y_i while preventing *Gen* from learning *Eval*'s inputs, *Eval* and *Gen* perform an OT where *Gen*'s inputs are $(k_{y_i}^0, k_{y_i}^1)$ and *Eval*'s input is β. All of the OTs for *Eval*'s inputs can be performed in parallel.

2.4 Correctness

A formal proof of correctness for this scheme is deferred to [2], but a couple of notions for correctness are important to give here. First, *Eval* should decrypt a cipher text for which she has both input keys with negligible probability of error. Second, *Eval*'s probability of decrypting a cipher text for which she does not have both necessary keys should be negligible. The probabilities of error will be dependent on the encryption scheme chosen, but the probability of error can always be reduced thanks to a result by Dwork, Naor, and Reingold [3].

2.5 Security

Rather than give a formal proof of security, we give a couple of notions of security that are important to this scheme.

Eval's input privacy is intuitively reduced to the security of the *OT*, since *Gen*'s view of the protocol is only of the *OT* messages passed.

Gen's privacy is proven by constructing a simulator that can generate the view of *Eval*. In this case, the simulator will not be able to correctly garble a circuit to evaluate $f(x, y)$, so it generates a "fake" garbled circuit that always evaluates to $f(x, y)$ which must be indistinguishable from a real garbled circuit in an actual execution of the protocol.

3 Free XOR

It is not necessary that all gate garblings be generated independently. In [4], Koleshnikov and Schneider showed that garbled XOR gates may in fact be completely dependent on their input wires, eliminating the necessity to transmit any information between *Gen* and *Eval* for their computation. In addition, the construction is incredibly cheap to compute, requiring only a single XOR operation per gate.

Begin by revisiting the construction of a garbled gate in Table 4. Notice that rather than generating k_z^0 and k_z^1 using $G(1^n)$, it is possible to make each k_z completely dependent on the input keys. Assume that there exists some $R \in \{0,1\}^n$, where n is the length of each key k. Now, let all $k_z^0 = k_x^0 \oplus k_y^0$ and, for all

$i \in \{x, y, z\}$, let $k_i^1 = k_i^0 \oplus R$. We can easily see that we have a new garbled XOR gate in Table 5.

x	y	z
k_x^0	k_y^0	$k_z^0 = k_0^0 \oplus k_y^0$
k_x^0	$k_y^0 \oplus R$	$k_z^1 = k_x^0 \oplus k_y^0 \oplus R$
$k_x^0 \oplus R$	k_y^0	$k_z^1 = k_x^0 \oplus k_y^0 \oplus R$
$k_x^0 \oplus R$	$k_y^0 \oplus R$	$k_z^0 = k_x^0 \oplus k_y^0$

Table 5: A Garbled Free XOR Gate

And now, in order for *Eval* to evaluate an XOR gate, she needs only to XOR her two input keys. R must be chosen once by *Gen* and held globally to be used for all gates, and *Gen* must keep R hidden from *Eval*, lest *Eval* learn extra information about the circuit. A complete argument for security is deferred to [4].

4 Point and Permute

We digress slightly from giving optimizations for gates in order to introduce new notation that will resolve some security concerns and reduce the decryption burden on *Eval*. We identified in section 2.2 that the truth table for a garbled circuit must be randomly permuted to prevent *Eval* from learning the semantics of the gate by the order of the cipher texts. We also noted in section 2.4 that *Eval* should have negligible probability of correctly decrypting a cipher text for which she does not have the keys, which is a concern because *Eval* needed to decrypt all possible cipher texts. We now give a construction that allows *Eval* to only decrypt one cipher text per gate.

Consider a wire w_i within the circuit to be composed of (k_i^α, p), where p is a random *permutation bit* and k is the key that the wire holds, as defined before. When *Gen* encrypts a wire, *Gen* appends p to the end of k. When *Eval* decrypts the wire to learn k, she also learns p, and with two values of p from two input wires she can deterministically select the value to decrypt from the garbled gate they enter. She does this by simply selecting the $2 * p_0 + p_1$ entry in the garbled table to decrypt. This scheme is compatible with Free XOR above by defining every p_z^1 for k_z^1 to be $p_z^0 \oplus 1$. When computing the new wires for a gate, we have that for wire i, $w_z^0 = (k_x^0 \oplus k_y^0, p_x^0 \oplus p_y^0)$ and $w_z^1 = (k_x^0 \oplus k_y^0 \oplus R, p_x^0 \oplus p_y^0 \oplus 1)$.

Intuitively, because the permutation bits are chosen independently of the keys, this permutation of garbled gate tables makes each table look randomly permuted, and in fact even gates of the same type are permuted independently of each other.

5 Garbled Row Reduction

5.1 GRR3

While the Free XOR trick allows us to eliminate all communication for XOR gates and evaluate them using a single XOR, it does not help us reduce the computation or communication necessary for other gates. Garbled Row Reduction (GRR3), suggested by [5] permits us to reduce the amount of information communicated and stored for every non-XOR garbled gate by one entry, transmitting only 3 table entries (hence the name GRR3). The intuition for this optimization is simple. Rather than generating the key for both output values (k_z^0 and k_z^1) from $G(1^n)$, we can directly determine the value of one k_z from the two input keys that are used to generate it, by computing $k = E_{k_x}(E_{k_y}(0^n))$. We define this k to be the first row sent by *Gen* for the gate, so *Gen* no longer needs to send that row. The other k_z is defined as determined by the Free XOR method.

We note here that in most current constructions, the double encryption over a key is computed as

$$E^s_{k_x,k_y}(m) = H(k_x||s) \oplus H(k_y||s) \oplus m$$

where $H(k||s)$ is some hash function keyed by k on input s, often implemented as SHA256 (originally SHA1 in FairPlay [6]), and s is some string completely unique to each invocation of the encryption. In some constructions, s is given as $Gid||c_0||c_1$, where Gid is the gate's number, also commonly referred to as a label, and the c values are the "external" values of the wire, given by the permutation bits concatenated onto the end of them. Here, rather than compute the encryption over some message k_z (where k_z takes the place of m above), k_z is simply defined to be the mask computed by XORing hashes of the input keys. In addition, because this key is defined to be the first row in the table, both c values are 0. (The value c for this output wire is also implicitly defined by this operation.)

Intuitively, this row reduction should maintain privacy because it reveals no new information to *Eval* about the keys. If the key in the first position of the table were defined to be the zero string, then *Eval* could always recover that row. However, *Eval* will only be able to compute the first row in this scheme if *Eval* has both input keys to that row of the gate, since she must use them to encrypt the zero string. A complete proof of security is deferred to [5].

5.2 GRR2

It is also possible to reduce the communication for a gate even further, although at the expense of no longer have a scheme compatible with Free XOR. This may be desirable when the number of XOR gates in a circuit is small compared to the total number of gates.

The intuition behind the scheme is that we can use a trick inspired by Shamir secret sharing to only send two values per gate, and use polynomial interpolation by *Gen* and *Eval* to identify and discover the proper output keys. Since we are not bounded by the Free XOR model, both output keys k_z can be chosen independently. The scheme for reducing the garbling of a truth table is dependent on whether the gate being garbled is *even*, *i.e.* both output keys appear twice in the truth table, or *odd*, *i.e.* one output key appears once and the other thrice.

Even Gates
An even gate is either an XOR gate or an XNOR gate. Without loss of generality, assume that the first row of the garbled table and the fourth row both encrypt k^0_z, and the second and third rows encrypt k^1_z. *Gen* computes all four cipher texts of the truth table, and arranges them appropriately in his own private table, which is given in Table 4. We briefly abuse notation and consider the four values $c^0_z, c^1_z, c^2_z, c^3_z$, where c is a cipher text, z denotes that it is used for output, and the superscript now identifies a row in the table, given in order. (We note that these will be permuted, but disregard that piece temporarily for the sake of clarity.)

Gen treats each cipher text c as a point in the finite field F_{2^n}, where n is the length of the cipher text, and constructs two linear polynomials as follows. The polynomial $P(X)$ is defined by setting $c_0 = P(1)$ and $c_3 = P(4)$, and interpolates $P(0)$ as the cipher text containing the key for k^0_z. The polynomial $Q(X)$ is defined by setting $c_1 = P(2)$ and $c_2 = P(3)$, and interpolates $Q(0)$ as the cipher text containing the key for k^1_z. *Gen* sends $P(5)$ and $Q(5)$ to *Eval* along with 4 bits that define the permutation used to randomly order the two keys he sent.

Eval computes a point on the polynomial using the private key encryption scheme on two keys and encrypting the message $s = Gid||c_0||c_1$. She uses the permutation bits of the input wires to determine which point she has computed and then interpolates either $P(0)$ or $Q(0)$ using the proper one of the two points that *Gen* sent her to discover the output key. This gives the intuition of the protocol; for a complete construction that accounts for how *Eval* uses the permutation bits (which themselves must be masked), refer to [5].

Odd Gates
Assume without loss of generality that *Gen* is garbling an OR gate (an AND gate is constructed analogously, as the polynomials are still defined but the values of k_z are switched). *Gen* computes the four cipher texts c as described in the previous section, defining the polynomial $P(X)$ with the points $P(2) = c_1$, $P(3) = c_2$, $P(4) = c_3$. k_z^1 is defined to be $P(0)$. *Gen* also evaluates $c_5 = P(5)$ and $c_6 = P(6)$. Then *Gen* creates a second polynomial $Q(X)$ by letting $Q(5) = c_5$, $Q(6) = c_6$, and $Q(1) = c_0$. *Gen* defines k_z^0 to be $Q(0)$.

Gen sends c_5 and c_6 to *Eval*, who uses the keys she has to determine if she is interpolating P or Q, and then generates the point determined by her input keys and permutation bits to interpolate the polynomial at 0, retrieving the output key. We again defer the complete construction to [5], which includes specifics about how the cipher texts are permuted. All of the permutation schemes, however, follow directly from the construction given in section 4.

6 Conclusion

We have given a generic construction of secure computation by garbled circuits, along with several optimizations for improving the protocol. Although the new techniques FleXOR [7] and HalfGates [8] are not covered here (I did not have time read them fully or space to describe them here, since the rest of these papers took longer than expected), this survey gives enough intuition to lay the foundation for them, and it brings the advances in garbled circuits constructions near to the state of the art.

References

[1] A. YAO, "Protocols for secure computation," *23rd FOCS, 1982*, 1982.

[2] Y. Lindell and B. Pinkas, "A proof of security of yao?s protocol for two-party computation," *Journal of Cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

[3] C. Dwork, M. Naor, and O. Reingold, "Immunizing encryption schemes from decryption errors," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 342–360.

[4] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *Automata, Languages and Programming*. Springer, 2008, pp. 486–498.

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

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

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

[8] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," 2014.