On garbled circuits and recent improvements

Per Hallgren

1 Introduction to garbled circuits

Garbled Circuits is a technique to compute a function based on inputs from multiple parties without disclosing any data that correlated to the user's input, except for the output of the function. It has been around and under development for about 30 years, but has only recently seen enough performance improvements to be feasible for real applications. We detail the main ideas of the technique, and highlight two recent developments that lead to performance improvements. The first such improvement, TinyGarble, shows how to better utilize existing optimization tools used in computer architecture. The second improvement, called GraphSC, lends support to secure computation of graph algorithms.

Before going further into the details about garbled circuits, let us take a moment to understand the concept of secure function evaluation. We focus mainly on the two-party case, and happily denote the two parties *Alice* and *Bob*. Alice and Bob both have some secret data, that they under no circumstances want to disclose to the other party. A common example is the *Millionaires Problem*, where Alice and Bob want to know which of them is the wealthiest. However, they do not want to tell the other the exact amount they have. Let a and b be the amount in Alice's and Bob's bank account (the amount is positive and without decimals, $a, b \in \mathbb{N}$), respectively. What they want to achieve is that they both learn the output of the function $f : \mathbb{N} \times \mathbb{N} \to \{true, false\}$ applied to their inputs, where $f(x, y) = x < y$. They want to securely evaluate this function, such that their inputs are kept private, and such that it is not possible to cheat.

A solution to secure function evaluation (SFE) was first proposed by Andrew Yao [9], in what is today known as garbled circuits. There are currently three main tracks that achieve SFE, homomorphic encryption [3], secret sharing [7], and garbled circuits.

1.1 Yao's construction

The construction by Andrew Yao [9] was the first solution to SFE, and works for two parties in the presence of semi-honest attackers. However, the initial paper is very theoretical, and it lacks a lot of the more descriptive terminology that makes garbled circuits easier to reason about in modern literature; the first garbled circuit was implemented 18 years after Yao's paper [5]. A good introduction to the problem is given by Lindell and Pinkas [4], we will follow their notation closely. In essence, a garbled

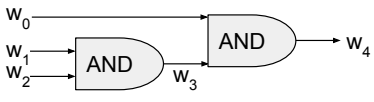

Fig. 1. Combinatorial circuit

circuit is a normal boolean circuit, but with an "encrypted truth table". To show what this means, we will walk through the construction using the example shown in Figure 1, which is a combinatorial circuit for the boolean formula $f(w_0, w_1, w_2) = w_0 \wedge w_1 \wedge w_2$. Normally, the truth table for the respective wires would be as shown in Table 1 (with w_0, w_1, and w_2 determined from the input).

Table 1. Truth table for w_3 and w_4

w_1	w_2	w_3
0	0	0
0	1	0
1	0	0
1	1	1

w_3	w_0	w_4
0	0	0
0	1	0
1	0	0
1	1	1

Table 2. Gabled truth table for w_3 and w_4

Label	w_1	w_2	w_3
G_0^0	$= E(\ w_1^0\ , E(\ w_2^0\ ,\ w_3^0\ ))$		
G_1^0	$= E(\ w_1^0\ , E(\ w_2^1\ ,\ w_3^0\ ))$		
G_2^0	$= E(\ w_1^1\ , E(\ w_2^0\ ,\ w_3^0\ ))$		
G_3^0	$= E(\ w_1^1\ , E(\ w_2^1\ ,\ w_3^1\ ))$		

Label	w_0	w_3	w_4
G_0^1	$= E(\ w_0^0\ , E(\ w_3^0\ ,\ w_4^0\ ))$		
G_1^1	$= E(\ w_0^0\ , E(\ w_3^1\ ,\ w_4^0\ ))$		
G_2^1	$= E(\ w_0^1\ , E(\ w_3^0\ ,\ w_4^0\ ))$		
G_3^1	$= E(\ w_0^1\ , E(\ w_3^1\ ,\ w_4^1\ ))$		

Instead, a "garbled" truth table is used, as shown in Table 2. The output of a gate in the garbled circuit is calculated by decrypting a *gate label* using the value on the two input wires as decryption keys (where superscript references a gate, and a subscript an output of that gate, G_2^1 is the third output of the second gate). The value of a wire should be viewed symbolically such that the *wire label* w_1^0 represents a 0 on the wire w_1, and similarly w_1^1 represents a 1. Concretely, the wires of the garbled circuit will hold decryption keys.

Now, let's work through an instance of Yao's protocol for when Alice and Bob wants to compute $f(y_0, y_1, x_0)$, when Alice has x_0 and Bob holds y_0, y_1 (in Figure 1, $w_0 = y_0$, $w_1 = y_1$ and $w_2 = x_0$). Alice will create the garbled circuit represented as the two gate labels and all wire labels, send the circuit to Bob, who will evaluate it. Recall that the goal is that Alice learns $y_0 \wedge y_1 \wedge x_0$, which means that she will learn that all values are true or if one is $false$. Note that the result being $false$ does not leak which of Bob's inputs was $false$.

To evaluate the circuit, Bob needs to know the values of the input wires. Given the gate labels and the wire labels of the input wires, Bob can first decrypt w_3 using w_1 and w_2, and then use w_0 and w_3 to find w_4, which is the output. However, he will be doing this without knowing whether the value of w_2 is w_2^0 or w_2^1 (as this is Alice's input). Thus, he will need to decrypt all of the gate labels of the gate (e.g. $D(w_1, D(w_2, G_i^0))$ for $i \in \{0, 1, 2, 3\}$), and see which one produces a "correct decryption" – there should be only one such.

Alice generates all of the keys used in the garbled circuit, and as such she can send the correct wire label for her inputs (if $x_0 = 0$, she sends w_2^0, etc.). She also generates the wire labels for Bob's inputs, which is more troublesome. If she sends both w_i^0 and w_i^1 for each of Bob's input i, bob can compute the formula for any input he chooses, which violates the privacy of Alice. Likewise, Bob can not tell Alice which one to use as this compromises the privacy of Bob. Thus, to select the correct input wire for Bob's inputs, they use an interactive protocol for *oblivious transfer*.

In short, Yao's construction consists of two core building blocks; it requires a semantically secure encryption scheme using which you can detect invalid

encryptions and an oblivious transfer protocol. Oblivious transfer, is a protocol that let's one party choose one out of two of the other parties inputs. More precisely, Alice holds x_0 and x_1, and Bob holds a bit b. The goal is that Bob chooses either x_0 or x_1 without learning the other, and Alice doesn't learn b. After the protocol, Bob learns x_b, and nobody learns anything about the other inputs. Concretely for us, Bob's input wire is w_2 for which he knows the correct bit $b \in \{0, 1\}$, and Alice thus has two wire labels w_2^0 and w_2^1, and using OT allows Bob to learn w_2^b without disclosing any additional information.

Encryption Scheme and Oblivious Transfer There are a lot of clever solutions to find a good encryption scheme to use for garbled circuits. Most efficient solutions use AES to be able to use the on-chip operations exposed through the Intel NI instruction set. One of the most straight-forward solutions is to use private keys for all wires, and to publish the public keys. Then, after decrypting, Bob can check if the resulting bit string is a private-key matching any of the publicized public keys.

The last piece we need to construct a garbled circuit is now a concrete oblivious transfer protocol. "*The Simplest Protocol for Oblivious Transfer*" was published in LatinCrypt 2015 [2], and it is indeed very simple. The protocol builds upon the Diffie-Hellman key exchange, recall that this works using a generator of a group g, where Alice sends to Bob $X = g^x$, for which Bob replies with $Y = g^y$. x and y becomes Alice's and Bob's private keys, respectively, and X, Y their public keys. Now, they can both compute g^{xy}, which commonly is hashed and used as the key in a block cipher, where Alice computes $k = H(Y^x)$ and Bob $k = H(X^y)$ to arrive at the same key. We now tweak this protocol to use it for OT, where Alice has two secrets and Bob has a choice bit b. Alice sends X to Bob as in the traditional protocol. Bob now replies with Y if $b = 0$, and XY otherwise (this gives no additional data to Alice as compared to normal DH). Alice now computes two keys to use in a symmetric encryption scheme, $k_0 = H(Y^x)$ and $k_1 = H((YX^{-1})^x)$, and she sends two ciphertexts to Bob $E(k_0, M_0), E(k_1, M_1)$. Only one of k_0 or k_1 can be computed by Bob, but not both, and thus he learns either M_0 or M_1, without learning the other.

2 Frontier Paper One – TinyGarble

TinyGable[8] is a compiler to enables garbled circuits to be represented as *sequential circuits* as well as combinatorial circuits as described above. A combinatorial circuit has the shape of a tree, where a sequential circuit can contain cycles. The traditional combinatorial garbled circuits are stateless, whereas a sequential circuit can make use of memory elements.

One of the key contribution of this work is that it enables garbled circuits to make use of the well-matured tooling that exists for hardware developers. Languages used by the hardware industry such as Verilog and VHDL enjoy many optimizations by tool support, rather than forcing the programmer to tweak the code for performance. Prior to TinyGarble, garbled circuits could not

make use of the full range of such hardware synthesis tools, as they may optimize a combinatorial circuit into a more compact sequential version.

The next step taken in this work, going from a garbled combinatorial circuit, to a garbled sequential circuit, is of course a garbled ALU from which they construct a garbled processor. One may ask why we would need garbled processor. Such a construction is called for in the case that the function to be evaluated needs to remain private, e.g. if Alice holds a proprietary algorithm, which she will allow Bob to run using his inputs. In essence, Bob will be allowed to set the initial state of the RAM, and read the final state of (some part of) the memory, while Alice controls the instruction ROM.

3 Frontier Paper Two – GraphSC

The authors of GraphSC [6] present a framework for parallel secure computations compatible with the ObliVM-GC programming language. The idea behind GraphSC is to support graph algorithms, which are well-suited for data mining and machine learning. The GraphSC framework uses data-augmented directed graphs (where both vertices and edges may store data). GraphSC uses the same programming paradigm as Pregel/GraphLab, where an algorithm is split into the three components *scatter*, *gather*, and *apply*, which serve to split an algorithm into well-defined and independent "chunks", similarly to the MapReduce methods *map* and *reduce*.

The model is that of two cloud providers with large datasets that want to jointly compute something on their data, while of course the data of each provider must remain secret to the other. One example could be two movie databases who want to compute joint rankings of reviews, another where a single cloud provider partitions it's data in order to be resilient towards advanced persistent threats.

The authors define data-oblivious algorithms for the three methods scatter, gather and apply. A data-oblivious algorithm is an algorithm where data access patterns are independent of the input data, such that the algorithm doesn't branch based on input data, doesn't index an array using input data, etc. Showing that an oblivious algorithm is easily converted to a secure algorithm (i.e. can be run in a GC framework), they are able to parallelize secure computations to a very large degree, and are able to show that the framework scales to large computing clusters both on a local area network, and across distant data centers.

Towards secure computations, the main contribution of the paper is the realization that oblivious algorithms can be turned into secure algorithms. For a data-oblivious algorithm, each processor would be assigned a "chunk" of work according to the scatter-gather-apply paradigm. Since data access patterns are independent of the input data, an attacker may learn nothing from observing what parts of the memory is accessed. However, an adversary can of course read the data, both in registers in the processor and from memory. To prevent this, garbled circuits is used together with a secret-shared memory. One processor in the oblivious model becomes a garbler-evaluator pair in SFE. The memory of the oblivious model is replaced by a secret-shared memory in the secure model.

References

1. *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015.
2. T. Chou and C. Orlandi. The simplest protocol for oblivious transfer. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2015.
3. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.
4. Y. Lindell and B. Pinkas. A proof of yao's protocol for secure two-party computation. *Electronic Colloquium on Computational Complexity (ECCC)*, (063), 2004.
5. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302. USENIX, 2004.
6. K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* [1], pages 377–394.
7. A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
8. E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* [1], pages 411–428.
9. A. C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.