UNBOUND
(MATH OVER MATTER)

A Primer in Secure Multiparty Computation

Prof. Yehuda Lindell
Co-Founder & Chief Scientist of Unbound Tech

Introduction to Multiparty Computation

Secure multiparty computation (MPC) addresses the problem of jointly computing a function among a set of mutually distrusting parties. It has a long history in the cryptographic literature, with its origins being found in the literature in the mid 1980s. The basic scenario is that a group of parties wish to compute a given function on their private inputs, while still keeping their inputs private from each other. For example, suppose that there are three bankers Alice, Bob and Charlie, who wish to discover whose bonus was the largest that year, without revealing what their actual bonuses were to each other or to a third party. To do so they engage in an interactive protocol, exchanging messages, with the result being the output of the desired function. The goal is that the output of the protocol is just the value of the function, and nothing else is revealed. In particular, all that the parties can learn is what they can learn from the output and their own input. So in the above example, the only thing that the parties learn is that Charlie's bonus was the highest; they do not know anything about the actual bonus amounts, and do not know whether Alice's bonus was higher or lower than Bob's bonus.

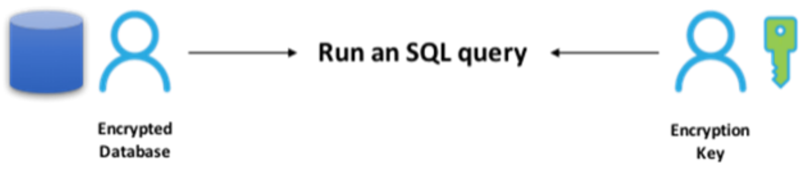
Informally speaking, the most basic properties that a multi-party computation protocol aims to ensure are:

- ***Input privacy***: The information derived from the execution of the protocol should not allow any inference of the private data held by the parties, except for what is revealed by the prescribed output of the function.
- ***Correctness***: Adversarially colluding parties willing to share information or deviate from the instructions during the protocol execution should not be able to force honest parties to output an incorrect result.

There are a wide range of practical applications for multi-party computation, varying from simple tasks such as coin tossing to more complex ones like electronic auctions (e.g. compute the market clearing price), electronic voting, private DNA matching, privacy-preserving data mining, and more.

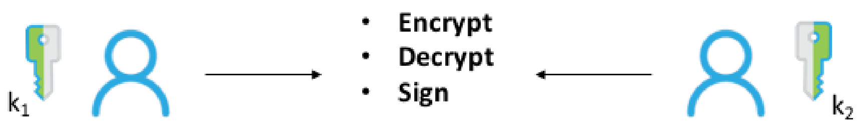
Secure multiparty computation can be leveraged to obtain a new paradigm of security: encryption of data **while in use**. We are all familiar with the two basic paradigms of encryption: data-at-rest and data-in-motion. However, encryption of data-in-use seems like an oxymoron: if data is encrypted, how is it possible to use it? With secure computation this can be achieved. Consider the case that Alice holds an encryption key and Bob holds an encrypted database, and the parties wish to run an SQL query on the database without ever decrypting it. This exact problem can be cast as a two-input function, and thus can be securely computed; in this case, input privacy means that the result of the SQL query is revealed and nothing else! Thus, SQL queries are computed while the database is encrypted, thereby keeping the database secure, even while it is being used.

ENCRYPTED DATA-IN-USE



Another novel application of secure multiparty computation is that it enables the use of cryptographic keys without ever having them in a single place, thereby eliminating the key as a single point of failure. Consider an AES key k that is used to encrypt and decrypt credit card information after a user logs in to their account on an eCommerce website. This key is very valuable since if stolen, an attacker can steal all the credit cards of all users. It is possible to randomly share the key into two parts k_1 and k_2 (where each part is completely random in isolation, but $k_1 \oplus k_2 = k$), and put the two parts on different machines with different administrators and so on, in order to make it hard to steal. However, if the key k is reconstructed upon use, then it is once again vulnerable. Secure MPC bridges that gap, as it can be used to compute AES encryption and decryption without ever bringing the key together. Furthermore, the input privacy property guarantees that neither party can learn anything about the key share of the other party (and thus about the actual key). Thus, even if one server is breached by an attacker, the attacker still cannot learn anything about the key.

USE OF CRYPTOGRAPHIC KEYS WITHOUT EVER HAVING THEM IN A SINGLE PLACE



In the academic literature, it was shown in the late 1980s that any function can be securely computed. However, the solutions proposed were not efficient enough to actually be used in practice. For the first 20 or so years, MPC research focused mainly on theoretical aspects. In recent years, significant algorithmic improvements have been made to MPC protocols, as a result of a major research effort to make MPC practical. Great progress has been made and secure computation can now be used to solve a wide range of problems with practical response times.

Mathematical Guarantees of Security

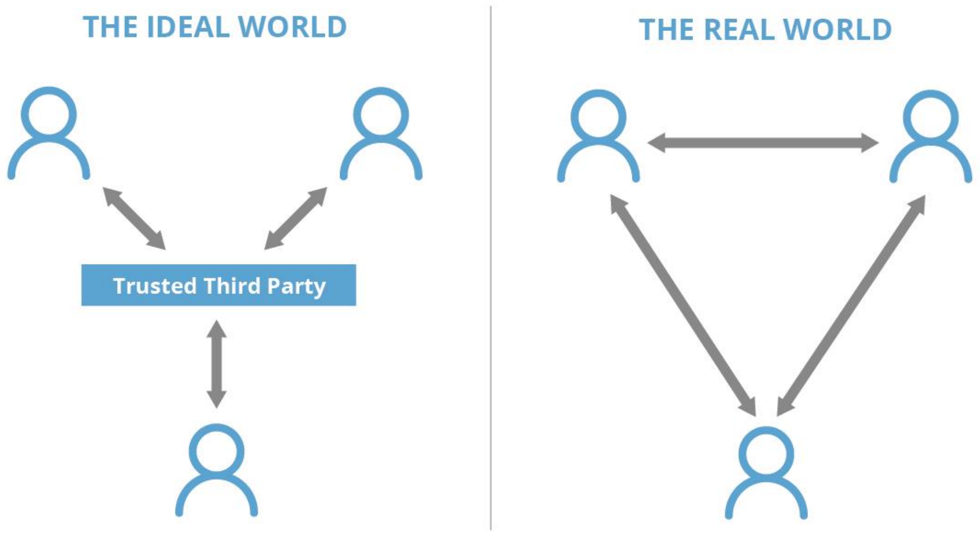
The ability to compute a function of joint inputs while preserving input privacy and correctness seems paradoxical; how can one compute on values without ever having them? To some, this notion even seems impossible. As such, an important question to ask is, “When is a multiparty computation protocol secure and how can we determine this?”

In modern cryptography, a protocol can only be deemed to be secure if it comes equipped with a ***rigorous security proof***. This is a mathematical proof that the security of the protocol follows from a well-established hard cryptographic problem (like RSA, discrete log over Elliptic curves, and so on). The fact that protocols come with such proofs differentiate MPC protocols from numerous other techniques that are merely heuristic in nature, and are all eventually broken. For example, whitebox crypto and heuristic obfuscation make it somewhat harder for an attacker, and are used. However, it’s almost impossible to know what level of security such a solution provides or how to compare two different solutions using such techniques. In addition, all such methods have a “cat and mouse” flavor: attackers are continually breaking the constructions and they are then changed to try to prevent the latest attack method. This endless repetitive process is highly unsatisfactory, and sophisticated attackers are able to break them very quickly.
In contrast, secure MPC protocols have a clear and exact guarantee. This provides a qualitative advantage to this approach over other merely heuristic ones.

Security Definitions

The Ideal-Real World Paradigm

Observe that in order to be able to provide a mathematical proof of security, we first need a definition of what it means for a protocol to be secure. This is hard to formalize in the case of MPC, since we cannot say that the parties should “learn nothing” since they need to learn the output and this depends on the inputs. In addition, we cannot just say that the output must be “correct” since the correct output depends on the parties’ inputs, and we do not know what inputs corrupted parties will use. The purpose of this paper excludes giving a full description, but a formal mathematical definition of security for MPC protocols follows the ideal-real-world paradigm, described below.



The ideal-real-world paradigm imagines two worlds. In the ideal world, there exists an incorruptible trusted party who helps the parties compute the function. Specifically, in this world, all the parties do is privately send their inputs to the trusted party, who computes the function on his own, and then sends back the appropriate output to each party. For example, let's go back to Alice, Bob and Charlie wishing to compute whose bonus was largest. In the ideal world, the parties can each send their bonus values securely to the trusted third party (i.e. each party sends an encryption of his/her bonus to the trusted party under a key known only to the trusted party). The trusted party then computes the function – here determining whose bonus was highest – and returns the identity of that party to all three parties. Observe that the ideal world is trivially secure: no party can do anything but send its input to the trusted party, and no party can learn anything but its prescribed output since this is the only value it sees in the entire execution. In contrast, in the real-world, there is no party that everyone trusts and all that the parties can do is to exchange messages with each other. The definition of security states that an MPC protocol is ***secure*** if a real-world protocol "behaves" like an ideal-world one, meaning that the only information revealed by the real-world protocol is that which is revealed by the ideal-world one, and that the output distribution is the same in both (guaranteeing correctness since this holds by definition in the ideal world).

Adversaries

Unlike in traditional cryptographic applications, such as encryption or signatures, the adversary in an MPC protocol can be one of the players engaged in the protocol. Different protocols can deal with different adversarial powers. We can categorize adversaries according to how willing they are to deviate from the protocol. There are essentially two types of adversaries, each giving rise to different forms of security:

- *Semi-honest (passive) security*: In this case, it is assumed that corrupted parties merely cooperate to gather information out of the protocol, but do not deviate from the protocol specification. This is a rather naive adversary model, that may yield weak security in real situations. However, protocols achieving this level of security prevent inadvertent leakage of information between parties, and are thus useful if this is the only concern. In addition, protocols in the semi-honest model can suffice in cases where it is possible to guarantee that the code being run cannot be replaced.

- *Malicious (active) security*: In this case, the adversary may arbitrarily deviate from the protocol execution in its attempt to cheat. Protocols that achieve security in this model provide a very high security guarantee. The only thing that an adversary can do in the case of dishonest majority is to cause the honest parties to "abort" having detected cheating. If the honest parties do obtain output, then they are guaranteed that it is correct. Of course, their privacy is always preserved.

Since security against active adversaries is often only possible at the cost of reducing efficiency one is led to consider a relaxed form of active security called covert security. Covert security captures situations where active adversaries are willing to cheat but only if they are not caught. For example, their reputation could be damaged, preventing future collaboration with other honest parties. Thus, protocols that are covertly secure provide mechanisms to ensure that, if some of the parties do not follow the instructions, then it will be noticed with high probability, say 75% or 90%. In a way, covert adversaries are active ones forced to act passively due to external non-cryptographic (e.g. business) concerns. This sets a bridge between the two models in the hope of finding protocols, which are efficient yet secure enough for practice.
The choice of which adversary to consider depends on the application and the setting. For example, in the application of using cryptographic keys without ever having them in a single place, the threat being considered is that of an attacker who breaches servers inside an organization's network (or in the cloud). In such a case, the attacker typically obtains root access to the machine and so can run any code that it wishes. Thus, security in the presence of malicious adversaries is required.

Two-Party Secure Computation of RSA

In an attempt to demystify the magic of secure MPC, we will first show how it is possible to securely compute the RSA signing and decryption operations without any single party holding the RSA secret key. The RSA public key is a pair (e, N) where $N = p \cdot q$ and p, q are two random primes, and the private key is a pair (d, N). The values e and d have the property that $e \cdot d = 1 \, mod \, \phi(N)$, where $\phi(N) = (p-1)(q-1)$. The RSA signing and decryption operation is defined by $y^d \bmod N$.
In this specific case, it is actually very easy to compute the RSA operation with a shared d, due to the very clean algebraic structure of RSA. Specifically, we can choose d_1 and d_2 uniformly at random[1] under the constraint that $d_1 + d_2 = d$. Then, consider the case that Alice has (d_1, N)

[1] Formally, one must specify the domain from which a value is chosen uniformly at random. In this case, since all operations in the exponent are actually modulo $\phi(N)$, we can choose d_1 completely at random between 0 and $\phi(N) - 1$, and then can set $d_2 = d - d_1 \bmod \phi(N)$.

and Bob has (d_2, N). Clearly, neither Alice nor Bob can carry out a signing or decryption operation by themselves, since they only hold a random share of the secret exponent d. Nevertheless, given a value y (that may be a ciphertext to decrypt, or the output of a padding method for RSA signatures), Alice can compute $x_1 = y^{d_1} \bmod N$ in isolation, and Bob can compute $x_2 = y^{d_2} \bmod N$ in isolation. Then, the product $x_1 \cdot x_2 \bmod N$ is the required result, since $x_1 \cdot x_2 = y^{d_1} \cdot y^{d_2} = y^{d_1+d_2} = y^d$ (where all operations are modulo N). The crucial point to observe here is that this operation was computed without d ever being revealed. (We remark that the full specification of how the RSA two-party protocol works along with its proof of security is beyond the scope of this primer.)

Proactive Security (Refreshing the Secrets)

A very important feature of this method is that it is actually possible for Alice and Bob to continually modify the value of their share of the key, without modifying the key itself. Specifically, the parties can run a secure coin-tossing protocol to obtain a random value r of the same length as N. Then, Alice can set $d_1' = d_1 + r$ and Bob can set $d_2' = d_2 - r$. The result is that $d_1' + d_2' = d_1 + d_2 = d$ and so everything still works as above. However, if an attacker successfully attacks Alice and steals d_1, and then later attacks Bob and steals d_2', then it will actually learn nothing about d (because $d_1 + d_2' = d - r$ and so the secret d is masked by the random r). *This means that the attacker has to successfully attack Alice and Bob at the same time in order to steal the private key*. In the literature, this type of security is known as "proactive security".

General Secure Two-Party Computation Using Garbled Circuits

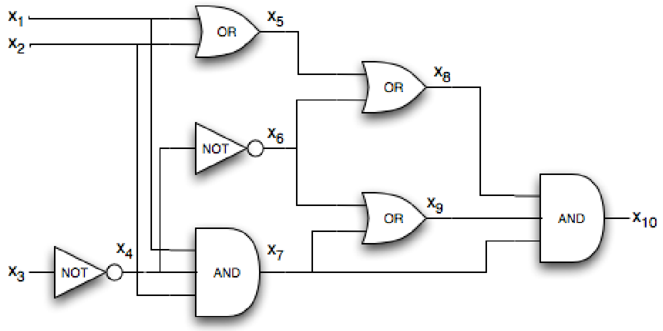
The example of how the RSA function can be securely computed shows that it is indeed possible to compute with shared inputs. However, one may be tempted to conclude that this is only possible because of the specific algebraic structure of the function. In particular, how is it possible to securely compute AES or HMAC-SHA256 on shared keys, since these functions have no such clean structure *by design*? We will therefore now describe a method that can be used to securely compute *any function whatsoever*; this method is called Yao's protocol.

Boolean Circuit Representation

Yao's basic protocol is secure against semi-honest adversaries and is extremely efficient in terms of number of rounds, which is constant, and independent of the target function being evaluated. The starting point of the protocol is the translation of the function to be computed into a Boolean

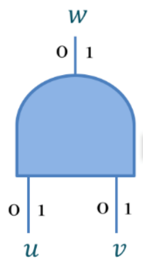
This will guarantee that neither one of d_1 or d_2 reveals anything about d; in fact one can view d_2 as a one-time pad encryption of d using the random key d_1.

circuit. A Boolean circuit is a collection of gates connected with three different types of wires: circuit-input wires, circuit-output wires and intermediate wires. Each gate has two input wires and a single output wire which is then input to *one or more* gates at the next level. Plain evaluation of the circuit is carried out by evaluating each gate in turn, according to a lexicographic ordering of the gates. Each gate is represented as a truth table such that for each possible pair of input bits (those coming on the input wires to the gate) the table assigns a unique output bit; which is the value of the output wire of the gate. The results of the evaluation are the bits obtained in the circuit-output wires. It is well known that *any function* can be represented as a Boolean circuit. For example, the AES encryption function can be converted into a Boolean circuit with approximately 32,000 gates.



Garbled Circuits and Yao's Protocol

Yao explained how to garble (or "encrypt") a circuit so that it can be evaluated without revealing anything but the output of the circuit. At a high level, Alice prepares the garbled circuit and sends it to Bob, who obliviously evaluates the circuit, learning the output corresponding to both his and Alice's input. The garbled circuit itself is computed as follows. The main ingredient is a double-keyed symmetric encryption scheme. Given a gate of the circuit, each possible value of its input wires and output wire (either 0 or 1) is encoded with a random encryption key (label). The garbled gate consists of *encryptions* of each output label using its inputs labels as keys. The position of these four encryptions in the truth table is randomized so no information on the gate is leaked. This process of constructing a garbled gate (without the randomization of the final ciphertexts) appears in Figure 1 below.



u	v	$w = u \wedge v$
0	0	0
0	1	0
1	0	0
1	1	1

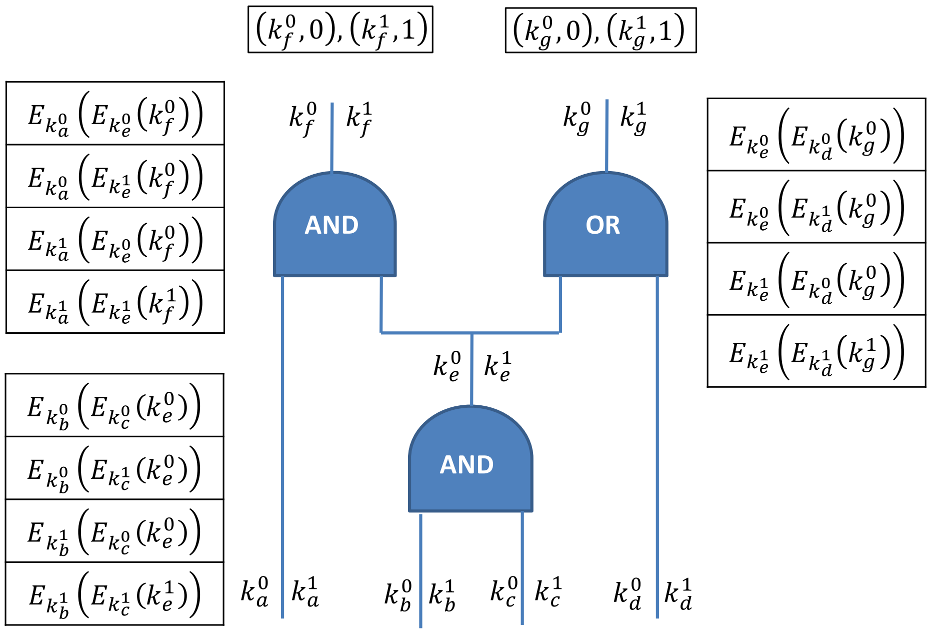
u	v	$w = u \wedge v$
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

$$E_{k_u^0}\left(E_{k_v^0}(k_w^0)\right)$$
$$E_{k_u^0}\left(E_{k_v^1}(k_w^0)\right)$$
$$E_{k_u^1}\left(E_{k_v^0}(k_w^0)\right)$$
$$E_{k_u^1}\left(E_{k_v^1}(k_w^1)\right)$$

Figure 1: Process of Garbling a Gate

Observe that given a *single* key on each input wire, it is possible to compute the correct key on the output wire, in a completely oblivious way. For example, assume that Bob is given k_u^0, k_v^1. Then, Bob can attempt to decrypt each of the four ciphertexts. However, Bob only has the valid keys for one of the ciphertexts and therefore only the ciphertext $E_{k_u^1}\left(E_{k_v^0}(k_w^0)\right)$ will be correctly decrypted (assume for now that one can detect when decrypting with an incorrect key). It follows that Bob will learn k_w^0. However, since the ciphertexts are in random order, all Bob sees are two random values on the input wire and one random value on the output wire. Since the 0 and 1 labels are just random and since Bob only sees one key per wire (note that Bob can only decrypt one of the four ciphertexts), Bob actually cannot know if he holds 0-values or 1-values. Thus, Bob correctly computed an AND gate without knowing what he computed.

A garbled circuit can be constructed by simply applying the above method to all the gates of a circuit, using the labels on a wire as plaintexts (when the wire is an output wire) and as encryption keys (when the wire is an input wire). In Figure 2, we show how a garbled circuit of three gates is constructed. We stress that the garbled circuit consists only of the topology of the circuit and the ciphertexts for each gate. The property of such a circuit is that given a single key on each input wire to the circuit, it is possible to compute the "correct" keys on the output wires of the circuit without learning anything else. For example, in Figure 2, consider the case that the input to the circuit is $a = 1,\ b = 0,\ c = 1$ and $d = 1$. The output of the circuit on these inputs is $f = 0$ and $g = 1$. Now, given $k_a^1, k_b^0, k_c^1, k_d^1$, Bob will compute k_e^0 and then k_f^0, k_g^1 and nothing else. In order to facilitate translating the output garbled labels into plain output, *output translation tables* are also provided. In Figure 2, observe that $(k_f^0, 0), (k_f^1, 1)$ are provided. Thus, when Bob receives k_f^0 for output, he will know that this means that the output on that wire is 0. At this point, it is worth noting that although Bob learns that the output of the circuit is 01, he has no way of knowing if this is due to the input being 1011 (as in the above example) or 0001 or 1100 or 1110 and so on, since all of these inputs yield the same output 01.



$(k_f^0,0),(k_f^1,1)$
$(k_g^0,0),(k_g^1,1)$
$E_{k_a^0}\left(E_{k_e^0}(k_f^0)\right)$
$E_{k_a^0}\left(E_{k_e^1}(k_f^0)\right)$
$E_{k_a^1}\left(E_{k_e^0}(k_f^0)\right)$
$E_{k_a^1}\left(E_{k_e^1}(k_f^1)\right)$
$E_{k_b^0}\left(E_{k_c^0}(k_e^0)\right)$
$E_{k_b^0}\left(E_{k_c^1}(k_e^0)\right)$
$E_{k_b^1}\left(E_{k_c^0}(k_e^0)\right)$
$E_{k_b^1}\left(E_{k_c^1}(k_e^1)\right)$
$E_{k_e^0}\left(E_{k_d^0}(k_g^0)\right)$
$E_{k_e^0}\left(E_{k_d^1}(k_g^0)\right)$
$E_{k_e^1}\left(E_{k_d^0}(k_g^0)\right)$
$E_{k_e^1}\left(E_{k_d^1}(k_g^1)\right)$
k_f^0 k_f^1
k_g^0 k_g^1
AND
OR
k_e^0 k_e^1
AND
k_a^0 k_a^1
k_b^0 k_b^1
k_c^0 k_c^1
k_d^0 k_d^1

Figure 2: A Garbled Circuit

We therefore conclude that Alice can construct the garbled circuit and send it to Bob, who can then evaluate the circuit and obtain the prescribed output (but nothing else). It remains to show

how Bob obtains a single key on each input wire, that is associated with his and Alice's input. Returning to the example of Figure 2, imagine that Alice's inputs are ab and that Bob's inputs are cd. (Formally, the function being computed is $f(a,b,c,d) = \big((b \wedge c) \wedge a, (b \wedge c) \vee d\big)$.) Now, since Alice is the one who constructed the garbled circuit in the first place, she can just send Bob the garbled values on wires a and b that are associated with her input; if she has input 10, then Alice just sends Bob the keys k_a^1, k_b^0. Since these are just random keys, Bob has no idea if they represent 0 or 1 values.

The challenge of Bob obtaining the garbled values associated with his own input is much more difficult. In particular, if Bob asks for the appropriate keys, then Alice will learn his input. Likewise, if Alice sends both keys on the wires for b and d, then Bob will learn more than allowed (recall that Bob is only allowed to have a *single* key on each wire). This problem is therefore solved using a cryptographic protocol called 1-out-of-2 Oblivious Transfer (OT). A 1-out-of-2 OT protocol enables Bob to obtain exactly one of two values held by Alice, without Alice learning which was received by Bob. We do not describe how OT is achieved in this paper, but remark that it can be achieved with very high efficiency.

In summary, Alice constructs a garbled circuit and sends it to Bob, along with the keys associated with her own input. Then, oblivious transfer is used for Bob to obtain the keys associated with his own input. Finally, Bob computes the entire circuit obliviously, and uses the output translation table to learn the real output. The result is a computation of an arbitrary function, without revealing *anything* but the output.

Malicious Adversaries

The above description informally shows how to achieve security in the presence of semi-honest adversaries. However, when considering malicious adversaries, further mechanisms are required to ensure correct behavior of both parties. For just one example, a malicious Alice may send an incorrect circuit to Bob; since the circuit is garbled, Bob will not be able to detect this cheating behavior (as long as the incorrect circuit has the same topology). This clearly means that correctness can be broken. However, such strategies can also breach the privacy property, since Alice may send a circuit computing a function that reveals all of Bob's input. Nevertheless, there exists efficient methods for preventing such cheating by Alice.

Securely Computing Complex Circuits

One of the main issues when working with Yao based protocols is that the function to be securely evaluated (which could be an arbitrary program) must be represented as a circuit, usually consisting of XOR and AND gates. Since most real-world programs contain loops and complex data structures, this is a highly non-trivial task. General compilers exist for translating code into Boolean circuits, for the purpose of MPC protocols. When considering specific functions (like AES and HMAC-SHA256 with shared keys), it is possible to compile the code once into a Boolean circuit and then (automatically and manually) optimize it to reduce the number of gates as much as possible.

As we have mentioned, the AES circuit has approximately 32,000 Boolean gates. It may seem that the work and communication involved in carrying out such a computation can never achieve response times that are suitable for practice. However, there are numerous optimizations to the basic Yao method that make it extremely fast. In particular, secure computation of the AES circuit can be achieved in mere milliseconds (and even less than a millisecond on a fast network). Thus, although Yao's protocol began as a theoretical construct, it has evolved into an extremely fast method that can be used to solve a wide variety of practical problems.

Summary

Secure multiparty computation enables parties to compute upon inputs without revealing them. Truly amazing constructions exist that enable parties to securely compute any function. Although these constructions work generically and involve translating the function to a Boolean circuit representation, they work extremely fast (for circuits that are not too large). In some case, like RSA, there are even faster protocols that work directly with the structure of the function and without needing to use a Boolean circuit representation. Importantly, all protocols for secure multiparty computation are mathematically proven secure and therefore provide very clear guarantees.