Garbled Circuits

Frederic Naumann
Betreuer: Marcel von Maltitz
Seminar Innovative Internet-Technologien und Mobilkommunikation SS2016
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: naumann@in.tum.de

ABSTRACT

In 1982, Andrew Yao published a paper which described possible ways of handling Secure Multi-Party Computation, but only in a very theoretical manner. In the following years, Yao developed a conceptual implementation approach to this subject which he titled as "Garbled Circuits", although he never actually published any of his work on Garbled Circuits but only mentioned and explained the idea behind the algorithm in several talks. At the time this concept was presented, it was deemed more of a theoretical concept than an actual implementation due to the limited computation power. But over the years, the computational possibilities grew and actual implementations became feasible.
This paper is set out to explain the function of Yao's original algorithm in detail and also evaluate it under various aspects such as performance and resistance to certain attacks. We will also talk about improvements to Yao's algorithm that have been proposed during the last nearly thirty years, and finally get into some actual implementation of these algorithms.

Keywords

Secure Multi-Party Computation, Secure Function Evaluation, Garbled Circuit Protocol

1. INTRODUCTION

The general *Secure Multi-Party Computation* (SMC) problem is defined as the situation where N parties wish to securely compute the value of a function $f(x_1, ..., x_N)$, where each party i delivers exactly one input x_i. During execution, no information about the x_i must be leaked to any party $j \neq i$, at least no information that can not be derived from the computation result [20, 4].
A simple example for a two-party application of this problem is the so called "Millionaires' Problem"[20]. In this problem, two millionaires want to find out who of them is richer, but neither of them wants the other one to know their exact wealth. So this problem can be formulated using the terminology introduced above with $N = 2$ and $f(x_1, x_2) = 1$, if $x_1 > x_2$ and 0 else. Yao presented a simple protocol solving this specific problem in [20], along with the theoretical foundations for the development of an extended protocol.
A more practical example where SMC can be applied is *secret voting*[20], where N parties wish to secretly and securely host a voting. The result is to be computed in private without a third party that handles the voting evaluation, and without any party learning how some other party voted.
Yao's approach to a functional protocol for Two Party Computation is the *Garbled Circuit Protocol* (GCP), where a function is transformed into a boolean circuit modelling the same function, which is then altered in a way that no information can be extracted from the resulting circuit. Note that this protocol has been developed explicitly for Two Party Computation, i.e. a SMC with $N = 2$. In the following sections, we will define the basic terms and concepts that are needed for the protocol, in section 4 we will present and evaluate Yao's original GCP, and in section 5 we will discuss a few modern extensions to the GCP that seek to enhance performance and security. In section 6, we will briefly present a couple of different working implementations for the GCP, and in section 7 we conclude.

2. DEFINITIONS

This section defines the security terminology we will use in this paper, it will mostly rely on the terms introduced by [17].

2.1 Security requirements

When one wishes to evaluate some function f using SMC, one needs to cover some requirements in order to make the securely computed function f_s a correct secure computation for f. In [19], Yao introduced the concept of comparing a protocol to an "ideal-oracle" that fulfills the three requirements listed below, and that a *Secure Function Evaluation* (SFE) is correct if it performs exactly like this ideal-oracle[17].
The ideal-oracle evaluates a function f with inputs x_1, x_2 that are delivered by two parties and outputs the value to both parties without revealing the inputs.

2.1.1 Validity

The most obvious requirement is that the evaluation of f_s must always deliver a correct result just as f would. So $f(x) = f_s(x)$ must hold true for all x for f_s to be a valid evaluation function for f.

2.1.2 Privacy

The *privacy* definition for the ideal-oracle forces a SFE system to prevent any party from learning the other party's input like the ideal-oracle would, provided that the protocol is carried out correctly. It is interesting to note that this does only guarantee that there are no unwanted values leaked during protocol execution and that this privacy definition does not account for any party trying reverse engineering methods on the result, e.g. an addition could be

doi: 10.2313/NET-2016-09-1_10

computed privately complying with this definition of privacy, but a participant would still be able to learn the other participant's input by subtracting his own input.

2.1.3 Fairness

A protocol is called *fair* when it securely computes the function value and then correctly transmits the output to all parties that participated in the computation. In contrary, an *unfair* protocol is one that refrains from actually sending the output to all parties, but holds back the information for certain (or all) parties.

2.2 Adversary Models

In the evaluation section, we will evaluate our protocol with respect to different *adversary models*. In SMC, we do not deal with classic "Man-in-the-Middle" or side channel attacks. In SMC, we are communicating and cooperating with a possible attacker, so we need to take into account different levels of protocol obedience for our possible attacker.

2.2.1 Semi-Honest Adversaries

An adversary is said to be *semi-honest*, or *honest-but-curious*, when it is not willing to deviate from the protocol at any time but tries to gather as much information about the other parties as possible by using data that is leaked during protocol execution and by the output [17]. For example, in a SFE protocol a *semi-honest* participant might try to deduce the other participants' inputs from the output, e.g. by assuming a uniform distribution of values and then guessing the right value with a certain probability. Also, an adversary that is *semi-honest* will take any protocol conforming step that it can take profit from, as long as it doesn't make the adversary's position any worse.

2.2.2 Malicious Adversaries

In contrary to *semi-honest adversaries*, a *malicious* adversary will violate the protocol in an arbitrary manner, which means that the adversary might deviate from the protocol at any point of the execution to gather information about the other parties [3]. For example, the corrupt party might send incorrect values or even no values at all, or simply abort the protocol at any time [17].

3. BASIC CONCEPTS

3.1 Oblivious Transfer

An important concept needed for the execution of Yao's Garbled Circuit protocol is *Oblivious Transfer* (OT). In general, OT is the problem of sending a single value from a set of values, without either the sender learning which exact value from the set was received or the receiver finding out any other value than the one he actually intended to receive [17]. Formally speaking, we have a set of N values on the sender side and an index i with $0 \leq i < N$ on the receiver side. After execution of the protocol, the receiver has learned only the value of N_i and no N_j where $j \neq i$, and the sender has not learned i. This version is known as *1-out-of-N Oblivious Transfer* [17]. A less general version which will be used in Yao's protocol is the one for the case $N = 2$, known as *1-out-of-2 Oblivious Transfer*. A simple protocol for this version will be presented below.

3.1.1 1-out-of-2 Oblivious Transfer

1-out-of-2 Oblivious Transfer (1-2 OT) is a special case for the concept described above, where $N = 2$ and the receiver may only choose $i \in \{0, 1\}$. An original protocol version for 1-2-OT has been proposed by Rabin [16] in 1981, the protocol presented here has originally been introduced by Lindell[10]. It is secure against *semi-honest* adversaries, and provides an easy understanding which will be needed in the upcoming protocol execution. In the following section, we will call the sending party S and the receiving party R.
Assume that S holds a pair of strings (s_0, s_1) one of which is to be sent to R. R selects $i \in \{0, 1\}$, depending on whether she wants to learn s_0 or s_1. She then generates a pair of asymmetric cryptography keys (k^{priv}, k^{pub}), and in addition to that another value $k^{\perp}$ that looks like a public key to S, but to which R has no private key. Then, R chooses the working public key to be k_i^{pub} and $k^{\perp}$ as k_{i-1}^{pub}, and advertises them to S as keys for s_0 and s_1, respectively. S then encrypts s_0 with the received k_0^{pub} and s_1 with k_1^{pub} and transmits the resulting c_0 and c_1 to R, who will then decrypt her desired value c_i with the corresponding k_i^{priv}, which then results in the correct s_i. R will not be able to decrypt the c_{i-1} because she has not generated a corresponding private key for $k^{\perp}$, and S will not know which value R has actually seen. Therefore, the proposed protocol guarantees a working 1-2-OT.
This protocol is only secure against *semi-honest* adversaries. It is easy to observe that one party could obtain additional information by deviating from the protocol, e.g. R could just generate two public/private key pairs and advertise both public keys, then she would be able to receive both s_0 and s_1. An OT protocol that is secure against *malicious* adversaries will be discussed later.

3.2 Cryptographic Hash Functions

A *cryptographic hash function* is a one-way function[1] where, in addition to the one-way property, there is also no possibility to draw conclusions from the way the values are distributed, i.e. the values of the function are uniformly distributed over the function's image space. Also, a *cryptographic hash function* prevents collisions of hash values. All hash values have the same length and appear completely random[2].

4. YAO'S GARBLED CIRCUIT PROTOCOL

Until now, we have only defined the problem we want to solve with the GCP and several concepts we need for the execution. Yao's GCP assumes that every function can be represented as a boolean circuit that only consists of binary gates (AND, OR, XOR, ...). This assumption has been proven correct for polynomial functions.[4] The general idea of the protocol is to transform the function into a boolean circuit and then disguising the circuit (garbling) so that one can not derive any values from the execution (e.g. intermediate function values). We will give a short description of the protocol before describing the protocol step by step.

[1] A *one-way function* is a function that is easy to compute, but hard to invert, so $f(x)$ is easy, whereas $f^{-1}(y)$ is not feasible with polynomial time effort

[2] This property is in literature referred to as "performing like a random oracle"[17]

doi: 10.2313/NET-2016-09-1_10

4.1 Short description

Let the two communicating parties be P_1 and P_2 and $f(x_1, x_2)$ the function the two parties want to compute, where P_1 delivers x_1 and P_2 delivers x_2. First, $P1$ transforms f into a corresponding boolean circuit c_f. Each gate in c_f has a truth table that details the gate's output. P_1 then turns c_f into its garbled version c_g by garbling each gate's truth table. Now P_1 also garbles his input so that it fits c_g and sends this garbled input over to P_2 along with the complete garbled circuit c_g. Now, P_2 holds c_g and P_1's garbled input, but not the garbling procedure, so P_2 does not know how to transform and where to use her own input. P_2 receives the garbled version of x_2 by using *1-out-of-2 Oblivious Transfer*. P_2 then computes the garbled circuit c_g gate by gate and outputs the ungarbled result to P_1 to complete the protocol.

4.2 Detailed description

After this brief explanation, we will give a detailed step-by-step explanation of Yao's protocol as described in [4] and [6].

4.2.1 Boolean Circuit Representation

Let $f(x_1, x_2)$ be the function to be evaluated securely. This function is transformed into a boolean circuit c_f that satisfies $\forall x, y : f(x_1, x_2) = c_f(x_1, x_2)$, which is assumed to be possible for all functions, although only proven correct for polynomial time functions with fixed size input[17, 4].

4.2.2 Garbling the circuit

P_1 has now completed the transformation of the function f into a boolean circuit c_f. This circuit consists of binary gates, and each of these gates has a truth table to compute the gate's output. The goal now is to garble these truth tables and turn c_f into the garbled version c_g.

To show how the garbling of a gate's truth table works, we will look at a logical AND gate, let's call this gate $g^{\&}$. The initial truth table for $g^{\&}$ is shown in figure 1a[3]. Then, P_1 generates an encryption key for each possible boolean value at each wire, so in total he generates six keys, two for each input and output.

Afterwards, P_1 encrypts all entries in the output column (w_2) with the help of the corresponding input keys. The table of garbled values is given in figure 1b[4], which also includes the final garbled value. In the figure, $g^{\&}$ is the gate identifier, which merely serves as a nonce to guarantee that no duplicate encrypted values[17] appear in the whole circuit. Also, P_1 reorders the entries of the garbled truth table to further abstract from the original truth table.

The encryption serves two purposes. On the one hand, each encryption produces a random output since it uses a cryptographic hash function that is assumed to mimic a random oracle. By doing so, it removes any correlation between input and output values. Our example gate $g^{\&}$ produces three identical outputs, but all the resulting garbled values are uniformly distributed and therefore they do not offer any possibility to gather information about the actual values.

On the other hand, the encryption makes it impossible for P_2 to tamper with the circuit during the evaluation process, as she will not be able to obtain any additional values than those she is intended to receive, because she will not get the keys for the respective inputs.

w_0	w_1	w_2
0	0	0
0	1	0
1	0	0
1	1	1

(a) Original table

w_0	w_1	w_2	garbled value
k_0^0	k_1^0	k_2^0	$H(k_0^0 \| \| k_1^0 \| \| g^{\&}) \oplus k_2^0$
k_0^1	k_1^0	k_2^0	$H(k_0^1 \| \| k_1^0 \| \| g^{\&}) \oplus k_2^0$
k_0^0	k_1^1	k_2^0	$H(k_0^0 \| \| k_1^1 \| \| g^{\&}) \oplus k_2^0$
k_0^1	k_1^1	k_2^1	$H(k_1^0 \| \| k_1^1 \| \| g^{\&}) \oplus k_2^1$

(b) After garbling

Figure 1: Truth table for $g^{\&}$

P_1 will garble the complete circuit, gate by gate and pass on the output values to the next gate, except for the final output gates, which do not need to be garbled as they display the actual function output which is to be learned by both participants, so P_2 can learn this value.

4.2.3 Transmitting the garbled values

After generating the garbled circuit c_g, P_1 needs to also create a garbled version of his input x_1. First, he will transform x_1 into the boolean value which corresponds to the inputs for the original circuit c_f. Then, he will replace each bit in this boolean value with the key for the corresponding input to c_g. Assume the following example, considering the gate example $g^{\&}$ from figure 1: the first bit of P_1's input is destined for w_0 and the value of this bit is 0. Then P_1 would select the corresponding key k_0^0. This replacement is done for all of P_1's input and finally results in a garbled version of P_1's input. This value is then sent to P_2, along with the garbled circuit c_g.

4.2.4 Receiving P_2's garbled input

P_2 now has c_g and a garbled version of P_1's input, but still needs a garbled version of her own input to be able to actually evaluate the garbled circuit. In the garbling process described in 4.2.2, P_1 has constructed the garbled values for P_2's possible inputs but does not have knowledge about P_2's actual input. P_2, on the other hand, knows her own input, but can not determine the keys generated for these input values.

This is where *1-out-of-2 Oblivious Transfer* as described in 3.1 comes into play. For each of P_2's input bits, she engages in an Oblivious Transfer to receive the garbled value corresponding to her input. In this case, P_1 is the sender and P_2 is the receiver, P_1 inputs (k_i^0, k_i^1), where i is the current wire's identifier, and P_2 inputs either 0 or 1, depending on the actual input. That way, P_2 learns a full garbled version of her input without either P_2 learning more than needed about the circuit or P_1 learning P_2's input, and is then able to compute the circuit.

4.2.5 Circuit evaluation

Now that P_2 has received her garbled input, she can look up the value for each output wire. Since she has no idea which table entry they belong to, she has to try the decryption for all four possible outputs. Provided that the protocol has been followed correctly by both parties, only one decryption will work, all others will result in $\perp$. The result is then used as input for the next gate in the circuit, or the output.

[3]Here, w_0 and w_1 are the gate's inputs whereas w_2 is the output

[4]The terminology for the keys is as follows: k_a^b means that this is the key for w_a having the value b

doi: 10.2313/NET-2016-09-1_10

A short example: again, we look at our example gate $g^{\&}$. Assume P_2 has looked up the input keys k_0^0 for w_0 and k_1^1 for w_1. She then computes $H(k_0^0|k_1^1|g^{\&})$, and by $\oplus$-ing that value with the garbled output value from the truth table, she receives the value for k_2^{05}, which is then carried over to the next gate and used as input key.
This procedure continues through the complete circuit, until finally, the circuit outputs the resulting bits to P_2 who then assembles them into the correct output for f. To complete the protocol, P_2 has to output the value to P_1.

4.3 Evaluation

In this section, we evaluate Yao's original protocol with respect to mainly two aspects, *security* and *performance*. In a later section, we will provide improvement proposals that have been found in recent research for these two aspects. We will then also evaluate these improvements with respect to their feasibility.

4.3.1 Security

As mentioned above, Yao's protocol is secure against *semi-honest* adversaries, but not against *malicious* ones. That finding is quite trivial to observe. On the one hand, the OT protocol we introduced is only secure against *semi-honest* adversaries, and on the other hand, there exist other points during execution where one of the parties could gather additional information by just not following the protocol. There are various possible attack points for *malicious* adversaries in Yao's protocol, a malicious P_1 could generate a corrupt circuit c_g computes a value different from the original function f, and also, one party could send over corrupt values to allow for some reverse engineering to get the other party's input, for example. The model of the *semi-honest* adversaries is not realistic in a real-world context, as we can not expect a communicating party that we do not trust with our input to not take the chance of stealing it by deviating from the protocol. At least, we need to develop a mechanism to discover *malicious* behavior.

4.3.2 Performance

The GCP presented above solves the SFE problem in polynomial time, at least against *semi-honest* adversaries[6]. But there exist SFE problems that demand for the generation of circuits with possibly over 1 billion gates[7]. Each gate in this circuit is represented and stored as four output keys (one for each wire-value combination), each key being a multi-byte string. Assuming a key length of 160 bit (e.g. with SHA-1 as hash function), this yields a total circuit size of about 800 megabytes for the billion-gate circuit. Each P_1 and P_2 need to keep this circuit stored during execution, and the complete circuit needs to be transferred from P_1 to P_2. This leads to different possible enhancement points for performance. The first one is optimizing the communication between the two parties to lower the amount of data that needs to be exchanged, then we could improve how P_2 evaluates the circuit, and we could develop a better way of constructing the circuit. We will show an example method for each of these fields in the following section.

[5]This is possible since $\oplus$ is self-inverse

[6]After applying the enhancements proposed in 5.1, this also holds true against *malicious* adversaries

5. POSSIBLE ENHANCEMENTS

As we have shown in the previous section, the GCP still has some weaknesses considering security and performance, which this section is seeking to abolish or at least improve.

5.1 Security

Here we introduce some solutions from recent research that attempt to solve these problems. For each of these areas, there are many different solutions that sometimes take completely different approaches, but we will limit the explanation and only introduce one protocol version at most.

5.1.1 Securing Oblivious Transfer

In this section, we introduce a protocol to secure OT against *malicious* adversaries as described in [1]. This protocol is based on the Diffie-Hellman[2] assumption: with a given g^x and g^y without knowledge of x or y, computing g^{xy} is hard. The protocol steps are listed below.
Let p be some prime number, g a generator for the group $\mathbb{Z}_p^*$ [7] and C some element in $\mathbb{Z}_p^*$. Assume that the tuple (p, g, C) is known to everyone, but the discrete logarithm of C is kept secret. Also, we have the sender S holding two strings $\{s_0, s_1\}$ and the receiver R wishing to receive s_i with $i \in \{0, 1\}$ as in our original OT protocol.
Before the actual OT starts, R needs to generate his public key. He does so by randomly picking $i \in 0, 1$ and $x_i \in \{0, \ldots, p-2\}$ and then calculating $\beta_i = g^{x_i}$ and $\beta_{1-i} = C \cdot (g^{x_i})^{-1}$. Then, β_0 is R's public key with x_i being the corresponding private key. We observe that the equation $\beta_0\beta_1 = C$ holds, therefore it is possible for S to check that R's public key is correct. Also, since the discrete logarithm of C is not known, R can not know the discrete logarithms of both β_0 and β_1, but only for the β_i that has been generated to be the working public key with x_i being the corresponding logarithm. In addition to that, S can not find out which β_i R knows because they are randomly distributed[8].
S now chooses random $y_0, y_1 \in \{0, \ldots, p-2\}$, calculates $\alpha_0 = g^{y_0}$ and $\alpha_1 = g^{y_1}$ and sends them to R. In the same course, S also initializes $z_0 = \beta_0^{y_0}$ and $z_1 = \beta_1^{y_1}$ and sends $r_0 = s_0 \oplus z_0$ and $r_1 = s_1 \oplus z_1$ to R. Now, $P2$ can compute $z_i = a_i^{x_i}$ and is then able to retrieve s_i by computing $s_i = z_i \oplus r_i$.
Those who are familiar with the matter will recognize a great similarity to a Diffie-Hellman key exchange which relies on the possibility for two parties to generate a mutual key pad without receiving the other party's key part. To decrypt r_i, R needs to compute z_i, which S initialized as $z_i = \beta_i^{y_i}$, but R does not have y_i, only x_i, and also $\alpha_i = g^{y_i}$. This makes $\alpha_i^{x_i} = g^{y_i^{x_i}} = g^{y_i \cdot x_i}$. In addition, recall that $\beta_i = g^{y_i}$ and z_i was constructed as $\beta_i^{x_i}$. This in return makes $\beta_i^{y_i} = g^{x_i^{y_i}} = g^{x_i \cdot y_i}$, which is notably the equal to $\alpha_i^{x_i}$, as multiplication is commutative over $\mathbb{Z}_p^*$. Therefore, R can construct the same pad as S and decrypt r_i.

5.1.2 Better security in garbling process

We also need to secure the actual circuit construction, because the circuit constructing party could easily generate a

[7]As our working group is $\mathbb{Z}_p^*$, all arithmetic in this section will be done mod p, so g^x is actually short for $g^x \bmod p$

[8]They are randomly distributed over the set of all tuples (x, y) with $x \cdot y = C$ and $x, y \in \mathbb{Z}_p^*$

doi: 10.2313/NET-2016-09-1_10

corrupt circuit that, using a simple example, just outputs the other party's input (a simple example would be $f(x, y) = y$). Therefore, we need to come up with a solution that forces P_1 to construct a correct and non-malicious circuit. An idea of such a solution is given here, based on [12].
The mechanism we introduce here is called *cut-and-choose*. Instead of just garbling the circuit, P_1 now generates m garbled versions c_i of the circuit, but each of them is garbled differently, i.e. with different keys for output encryption every time. In addition, P_1 also computes the corresponding garbled inputs x_i, with $0 \leq i < m$. Now, P_1 also generates a commitment $C_i = H(x_i)$ for each $0 \leq i < m$, where H is some *cryptographic hash function*. Then, P_1 sends all m circuits to P_2, along with the respective commitments C_i, and also structural information, i.e. which keys have been used on the respective gates.
P_2 then randomly chooses a $0 \leq j < m$ and has P_1 de-garble all circuits except for the jth one to inspect the other $m-1$ circuits. If any of them is malformed, i.e. does not match the garbled version, P_2 assumes that P_1 is malicious and aborts the execution. If all circuits have been checked and proven correctly formed, P_2 continues to receive P_1's garbled input like in a normal execution of the protocol. But first, P_2 verifies that P_1 did not send a corrupt input by checking if $C_j = H(x_j)$. If so, P_2 aborts, else P_2 assumes P_1 is not sending corrupt data and continues the protocol.
This example protocol is supposed to give an intuition of the *cut-and-choose* concept, the concept can be further improved by choosing a probabilistic approach as described by Lindell and Pinkas in [11]. Instead of taking $m-1$ circuits for proof of correctness, P_2 will only check circuit correctness on $m/2$ circuits, then compute the other half and take the majority result of this computation as correct output. Instead of immediately aborting execution when a corrupt circuit has been detected, P_2 will continue evaluating all $m/2$ circuits and then take the majority result. To understand the reasoning behind this, consider the following example: P_1 constructs all circuits correctly, except a single one, where the output is the actual function value $\oplus$'ed with P_2's first input bit. Then, P_1 could just observe if P_2 aborts the computation prematurely, thus has encountered the corrupt circuit, or returns a value, then P_1 can also draw conclusions on P_2's input. A more detailed description of this attack can be found in [17].

5.1.3 Corrupt inputs

By now, we have ensured that a *malicious* adversary can not exploit the *Oblivious Transfer* phase, and also that the circuit has been constructed correctly. But an adversary could still inject corrupt inputs to gain information from the other party, as shown in this short example:
During the actual OT process, S is not able to find out anything about R's input, but this does not imply that S does not have any other way to learn about R's input. S can send over the correct garbled value for 0, but a corrupt value for 1. If the requested value was 0, R will continue the protocol execution, if not, R will notice the corrupt value and abort due to our secured protocol. But either way, by watching R's behavior, S is able to find out which value S requested. Again, Lindell and Pinkas have come up with a solution to this security issue in [11]. Assume that S's input is the boolean form of x_2 with $|x_2|$ being the number of bits. To secure the protocol against leaking information to corrupt inputs, we have P_2 replace each bit of his input x_2 by a combination of XOR-gates that take s new input bits introduced by P_2. P_1 can now no longer corrupt P_2's input, but only the new inputs to the XOR-gates. The more new bits P_2 introduces, the more ways she has of constructing her actual input through the XOR-gates.

5.1.4 Open Problems

Although these measures have abolished a lot of the security issues of the original protocol, there are still a few remaining that are yet to be solved for the *malicious* setting. One of them is to make sure that P_2 returns the correct result of the circuit computation. By now, P_2 could just send P_1 a wrong result for the circuit computation and keep the actual result to herself, or even send no result at all, or the other way around, P_1 could leave the output encrypted and by that force P_2 to send the output, but then P_1 could keep the result to himself. A unilateral solution for this problem is yet to be found, some work has been put into at least making sure that a returned solution is correct.

5.2 Performance

In the last section, we have aimed to provide a more secure version of Yao's protocol, now we will try to improve the protocol's performance. But first, we need to give a few thoughts on the terms *performance* and *efficiency*. Snyder states that Yao's protocol can be executed in polynomial time, even with the security enhancements made in 5.1[17], which is comparably performant and efficient by means of execution time. But nevertheless, as mentioned before, the protocol is very costly, even quite simple operations on comparably small operands demand for the generation of circuits with possibly billions of gates, which for many cases is too expensive and therefore makes the use of GCP in this situation impractical and nearly impossible.
We have in general three possible points in the protocol where we could try to make the protocol execute more efficiently. These are optimizing the communication, i.e. reducing the amount of data that needs to be exchanged, optimizing the actual execution time of each gate and finally cutting unnecessary gates from the circuit by finding more efficient circuit constructions

5.2.1 Optimizing communication traffic

The major cost generator in the GCP is in the most cases the transmission of the garbled circuit from P_1 to P_2. Kreuter et al. [9] have found that calculating the *Levenshtein Distance*[9] of two strings of approximately 500 bytes requires around 5.9 billion gates. These gates are connected by wires, of which each is represented by four keys. A key can be assumed to be a multi-byte string, and even if you make the unrealistic assumption that each key is one byte long, this still results in the c_g's size exceeding 20 gigabytes (for our *Levenshtein distance* example. So, we will first have a look at some ideas on how to lower the transmission needs. Note that this section will solely focus on communication optimizations, which sometimes might come with extra computation

[9]The *Levenshtein Distance*, also known as the edit distance, is the minimum number of basic bit operations (insertion, deletion, bit flipping) needed to transform one string into another, and is not to be confused with the *Hamming distance*. It is commonly used as a benchmark for two-party-computation.

doi: 10.2313/NET-2016-09-1_10

time, but this is acceptable since communication traffic is making for the vast part of the GCP, because transmitting data via a network is slower than processing the same data on a CPU.
The procedure we will discuss here is called *Random Seed Checking* and has originally been presented by Goyal et al.[5]. In detail, this procedure consists of two parts that each base on the same idea. We recall that the construction of the c_g involves P_1 assigning each wire in the circuit a random value (see 4.2.2). Instead of doing so, we have P_1 choose a random seed and then construct the wire keys deterministically based on that seed, like a pseudo-random generator. For the second part of this version, we recall the *cut-and-choose* procedure described in 5.1.2 to secure the protocol in the presence of *malicious* adversaries, where P_1 would generate m semantically equivalent circuits and corresponding commitments and then P_2 would check the correctness of $m-1$ circuits to prove that P_1 is not generating corrupt circuits. We also discussed the majority result optimization to that concept, which will be further enhanced performance-wise here. In this optimization, P_1 will not send P_2 m full circuits, but only the *Random Seeds* that have been generated to deterministically build each circuit, and also a "commitment", that can, for simplification, again be seen as some sort of *Cryptographic Hash Function*. P_2 then generates the $m/2$ circuits to check herself from the seeds he got from P_1[10]. After the construction, P_2 can check a circuit's correctness by checking the corresponding commitment and will then, if the proof of correctness succeeded, request the remaining $m/2$ circuits to evaluate and continue as described before.

5.2.2 Optimizing Circuit Evaluation

After reducing the amount of communication traffic needed for the exchange of the garbled circuits, we will reduce the amount of resources needed for the actual circuit execution with respect to execution time and computational power. To delimit this section from the following one, in this section we discuss improvements to the circuit execution that can be applied without any changes to the circuit's internal structure.
The first approach towards achieving such optimization we will discuss here is the *Fast Table Lookups* technique as described in [7][11]. The idea of this procedure is to facilitate the gate evaluation for P_2 by hinting her on which table entry for the next gate to use. In the original protocol, P_2 needs to decrypt all four garbled output values to find out which is the real one (the one value where the decryption works). In this improved version, P_1 appends an extra bit to each garbled output value. The two extra bits from the input wires form an index in the range $\{0,\dots,3\}$ that points to the next encrypted value that can be decrypted with the received input keys. Due to the fact that the table rows are mixed, this does not constitute a security flaw, since there is still no way to deduce the mapping of keys to wire values.
Another possible approach for execution optimization is aiming to parallelize the execution. Normally, the protocol execution is done in a linear manner, P_1 generates the circuit (or possibly all m circuits when utilizing the *cut-and-choose* procedure), while P_2 is idle, because she needs the circuits first to start her procedure. The idea of *Pipelined Circuit Execution* has been originally presented by Huang et al. in [7] where they came to the conclusion that it is possible for P_1 to parallelize the execution by sending P_2 parts of the circuit as soon as possible, ideally right after generating them, and P_2 will start evaluating them, as long as there are gates to which the input is available. First, this reduces the amount of memory needed, since no party is obliged to store the entire circuit in their memory, but keeps on evaluating the gates. Additionally, it also reduces the computation time, roughly according to the following formula. Normally, the execution consists of the garbling time (t_{garble}) plus the time needed for transmitting P_2's garbled input via OT (t_{OT}) and the time P_2 needs to evaluate the garbled circuit (t_{eval}), so the total circuit execution time sums up to $t_{total} = t_{garble} + t_{OT} + t_{eval}$, whereas when using circuit pipelining, the total execution time will only be $t_{total} = max(t_{garble}, t_{eval}) + t_{OT}$[12]. A major downside of this proposed procedure is that it is only secure against *semi-honest* adversaries per default[17], because it doesn't comply with the *cut-and-choose* strategy proposed in 5.1.2, as this has P_1 generate m circuits that need to be generated and stored together for P_2 to be able to choose a set of circuits to evaluate, the same also holds true for *random seed checking*. Kreuter et al. have developed a solution to secure *pipelined execution* against *malicious* adversaries where P_1 has to generate each circuit twice, possibly using *random seed checking*, once like he normally would, starting to send the circuits to P_2 right away, and again after P_2 has chosen which circuits he wants to evaluate [9].

5.2.3 Optimizing Circuit Construction

While in the last section, we presented concepts to execute the circuit faster without actually altering it, we will try to achieve improvements to the GCP's resource needs by making the circuit construction more efficient. The general idea of this class of approaches is to significantly reduce the number of garbled values that need to be generated, stored and exchanged.
The first approach to this is the trivial strategy of simplifying the circuit and therefore having less gates to be garbled. The main part of this approach is to remove inefficiencies from the circuit, i.e. to eliminate partial circuits that always evaluate to a constant or that can be represented by fewer gates. While this might sound like a simple and minuscule improvement, Pinkas et al. [15] have shown that it is possible to improve *Fairplay's*[13] performance by 60 %.
Kolesnikov and Schneider[8] have developed another approach that goes beyond simply reducing the number of gates that are generated by the function-to-circuit transformation. They introduce the possibility of replacing a garbled XOR gate by a standard XOR operation, which they call the *free XOR* technique. In the default protocol, P_1 generates the garbled values representing 0 and 1 for each wire at each gate at random. When using the *free XOR* technique, we instead have the values be generated in relation to XOR gates. This allows P_2 to simply execute a single XOR operation for each such gate instead of the lookup and decryption of values from the garbled truth tables. The following description of

[10] P_1 also delivers any needed information about each circuit's structure
[11] The original concept has been postulated in [12].
[12] At least in the ideal case.
[13] *Fairplay* will be introduced in 6.1

doi: 10.2313/NET-2016-09-1_10

the *free XOR* algorithm is taken from Snyder[17].
Let k be the length of the garbled values generated for the circuit. We have P_1 generate a secret k-bit integer $K \in \{0,1\}^k$. Let G be the set of all XOR gates contained in the generated circuit, and g such a gate with two input wires, of which the input values are delivered by the two gate g_{in_0} and g_{in_1}. Also, let $k^b_{in_i}$ be the key that corresponds to the input gate g_{in_i} having the value $b \in \{0,1\}$. Then, set $k^1_{in_0} = K \oplus k^0_{in_0}$ and $k^1_{in_1} = K \oplus k^0_{in_1}$ for all gates $g \in G$ and replace the gate g with some function that returns $k_{in_0} \oplus k_{in_1}$. This eliminates the need to hold garbled truth tables for all XOR gates and thus reduces the size of the garbled circuit by $|G| \cdot |k|$, with k again being the size of a garbled value in the circuit. A more detailed example of how to derive the correct output values using the XOR operations is given by Kolesnikov and Schneider in their paper on *free XOR*[8].

5.3 Combining different strategies

As we mentioned before, some of the proposed improvements conflict with each other because they are doing different approaches to different fields of problems, and the respective measures contradict each other. In some cases, the different strategies don't interfere with each other, but other concepts might not be feasible at the same time by default. In these cases, additional effort needs to be taken to make the different concepts work together. We already discussed the possibility of running *pipelined circuit execution* together with a *cut-and-choose* approach, and the necessary additional computation load in this case about doubles the effort, as P_1 needs to generate each circuit twice. So, before blindly applying different improvements to an implementation of the GCP, one needs to consider potential side effects the enhancements have on each other. For some improvements that initially contradict each other, there exist solutions to make a parallel application of two conflicting approaches possible, but this requires the development of additional strategies dedicated to this single purpose. Another example where such extra work is needed is combining Huang et al.'s *Fast Table Lookups* and the *Free XOR* technique[17]. The *free XOR* technique is not trivially capable of handling the extra bits appended to the output to directly determine the next table entry used, and the solution to this problem presented by Kreuter et al.[9] requires some more effort and will thus not be discussed here, it is just used as an example to illustrate the difficulties that come with the desire to implement multiple enhancement strategies.

6. IMPLEMENTATIONS

Originally, Yao developed his GCP as a theoretical concept, and for a long time, an actual implementation of the protocol did not seem possible as the needs in computation power and memory consumption were apparently too expensive to afford a working implementation. But, as predicted by Moore's Law[14], the number of transistors on integrated circuits and thus the computational power grew exponentially over the years, and so did the possibility of realizing a computation-intensive protocol like the GCP. So, in this section we will present some actual implementations. The principle is similar throughout the different approaches, they offer the possibility to enter the function to be computed in a high-level language and compile those to a language dedicated to describing boolean circuits, which are then garbled and sent.

6.1 Fairplay

When new versions of the GCP are developed, the developers almost always compare them to the *Fairplay*[12] implementation. Malkhi et al. developed the approach in 2004, and it is considered one of the first ever implementations for the GCP. Recent implementations use *Fairplay* as a benchmark for comparison, and build upon it. *Fairplay* implements the simple *cut-and-choose* strategy proposed in 5.1.2 to provide basic security against *malicious* adversaries, and also uses the *Fast Table Lookups* technique. It uses its own *Secure Function Definition Language* (SFDL) and is compiled to VHDL-like *Secure Hardware Definition Language* (SHDL) that describes the circuit, which is then output as a Java object. The sample problems that were used to test and evaluate the *Fairplay* implementation can be considered rather simple: the "Billionaires' Problem", a version of the Millionaires' Problem with larger numbers, a bitwise AND operation, a database search for keyed elements and finding the median of two arrays of size 10. The latter problem required the construction of 4383 gates and a computation time of 7.09 seconds[12]. The problems might have been simple, but *Fairplay* was certainly a proof of concept for the possibility of implementing Yao's GCP.

6.2 Huang et al.

In [7], Huang et al. present an implementation that aims to achieve a high performance above all. They implemented various performance enhancements that we have presented in this paper, such as the *Fast Table Lookups*, *Free XORs* or *Pipelined Circuit Execution*, which led to astonishing computation speedups, but they only focused on performance, meaning they did not secure their protocol against *malicious* adversaries. In contrary to *Fairplay*, Huang et al. tested their protocol against more difficult problems and compared them to the previously best known implementation solving the respective problem.They used problems such as the *Hamming Distance* of 900 bit strings where they achieved an overall speedup of 4100 times (213 seconds to 0.051 seconds)[7], the *Levenshtein Distance* of 200 bit strings which produced the largest computed circuit with over 1 billion gates, or AES-128. AES had already been implemented using garbled circuits by Pinkas et al. [15] using *Fairplay*'s SFDL, but Huang et al. follow a different approach by orienting towards the traditional AES code. A complete description of their AES approach would be beyond the scope of this paper, though.

6.3 Kreuter et al.

Kreuter et al.[9] again focused on developing an implementation for security against *malicious* adversaries. They created an own language for entering functions, such as *Fairplay* did with the SFDL, and also a new compiler that they compared against the *Fairplay* compiler and which brought a vast speedup in generating larger circuits. Snyder labels Kreuter et al.'s implementation the "state of the art"[17] when it comes to providing an implementation of the GCP that is secure against *malicious* adversaries, they implemented all of the aforementioned security enhancements (5.1) and also all performance improvements (5.2), they also developed additional techniques to employ strategies that were originally

doi: 10.2313/NET-2016-09-1_10

contradicting each other (see 5.3). They evaluated their system against typical "benchmark problems" such as AES or the *Levenshtein distance*, as mentioned before, where they used two 4095-bit strings as input and computed the resulting circuit consisting of over 5.9 billion gates in 8.2 hours, which they compared to Huang et al.'s implementation that computed the result faster, but is only secure against *semi-honest* adversaries.

6.4 Mood et al.

The implementation we will present in this section is slightly different from the others, as it is written to make using the GCP on mobile devices possible. Mood et al.[13] wanted to adapt to the trend of smartphones becoming increasingly popular in 2012 and took the challenge of creating a GCP port for Android. They used *Fairplay*'s SFDL to define functions and a specialized *Pseudo Assembly Language Compiler* (PALC) to compile to *Pseudo Assembly Language* (PAL). The PALC was designed specifically for Android devices and provides a significant improvement over a direct *Fairplay* port. On Android devices, the problem size is not only limited by execution time and memory, but in addition also by Android itself restricting the file size to 4 GB[13]. Mood et al. evaluated their implementation on 2011 made HTC Thunderbolts, which have 768 MB of RAM, which is significantly lower than todays Android phones have available, so it is possible that larger circuits are feasible on today's Android phones than have been in 2012. In addition to the usual benchmarking process of evaluating their implementation against various problems, they also developed an actual Android application where a user can on the one hand solve the same benchmarking problems as Mood et al. did and on the other hand engage a password vault where a password can be encrypted and only unlocked if the two parties that created the vault enter their password.

7. CONCLUSION

The goal of this paper was to give the reader a firm understanding of Yao's GCP and all its basics, along with an overview of enhancements that have been introduced during the 30 years since this protocol was first introduced. We also presented some state of the art implementations that might be further developed into an actual GCP-based security framework in the following years, with computation power still growing, although Moore's Law might no longer be applicable[18]. It is to be noted that SFE is an open research field in cryptography, and there is a lot of effort put into developing new approaches to SFE as well as to further enhancing the GCP.

8. REFERENCES

[1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in Cryptology – CRYPTO '89 Proceedings*, pages 547–557. Springer, 1989.

[2] W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, pages 644–654. IEEE, 1976.

[3] O. Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, pages 86–97, 1998.

[4] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[5] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Advances in Cryptology - EUROCRYPT 2008*, pages 289–306. Springer, 2008.

[6] C. Hazay and Y. Lindell. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.

[7] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[8] V. Kolesnikov and T. Schneider. Free xor gates and applications. In *Automata, Languages and Programming*, pages 486–498. 2008.

[9] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 14–14. USENIX Association, 2012.

[10] Y. Lindell. Secure two party computation in practice. lecture given at technion-israel institute of technology tce summer school 2013, 2013. `https://www.youtube.com/watch?v=YvDmGiNzV5E`.

[11] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology - EUROCRYPT 2007*, pages 52–78. Springer, 2007.

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

[13] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography and Data Security*, pages 254–268. Springer, 2012.

[14] G. E. Moore. Cramming more components onto integrated circuits. In *Proceedings of the IEEE*, volume 86, pages 82–85, 1986.

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

[16] M. O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[17] P. Snyder. Yao's garbled circuits: Recent directions and implementations, 2014.

[18] M. M. Waldrop. `http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338`. Accessed: 2016-06-20, 10:21.

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

[20] A. C. Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.