Yao's Garbled Circuits: Recent Directions and Implementations

Peter Snyder
University of Illinois at Chicago
Chicago, Illinois, USA
psndye2@uic.edu

ABSTRACT

Secure function evaluation, or how two parties can jointly compute a function without any other party learning about any other party's inputs, has been an active field in cryptography. In 1986 Andrew Yao presented a solution to the problem called *garbled circuits*, based on modeling the problem as a series of binary gates and encrypting the result tables. This approach was initial treated as theoretically interesting but too computationally expensive for practical use. However, in the decades since Yao's solution was initially published, much work has gone into both optimizing the protocol for practical use, and further securing the protocol to make it further secure.

This paper provides a thorough explanation of both Yao's original protocol and its security characteristics. The paper then details additions to the protocol to make it both practical for computation and secure against untrusted parties. Implementations of Yao's protocol are also discussed, though the paper's emphasis is on the underlying enabling improvements to the protocol.

1. INTRODUCTION

Secure function evaluation (SFE) referrers to the problem of how can two parties collaborate to correctly compute the output of a function without any party needing to reveal their inputs to the function, either to each other or to a third party. A common example of this problem is the "millionaires problem", in which two millionaires wish to determine which of them has more money, without either party revealing how much money they have[13].

Many solutions have been developed for SFE. One category of solution is function specific, and depends on specific attributes of the function being executed to provide security[8]. These solutions, while interesting, are by definition of less general interest, since they apply to only a limited set of problems.

Another category of approach is more general, and seeks to provide a general solution for SFE by transforming arbitrary functions into secure functions. Approaches in this category include homomorphic encryption systems[4] which allow for arbitrary execution on encrypted data. Yao's *garbled circuits* protocol fits in this second category.

Yao's *garbled circuits protocol* (GCP) transforms any function into a function that can be evaluated securely by modeling the function as a boolean circuit, and then encrypting the inputs and outputs of each gate so that the party executing the function cannot discern any information about the inputs or intermediate values of the function. The protocol is secure as long as both parties follow the protocol. A full description of the protocol and the related security definitions are provided later in this paper.

1.1 History of Protocol

Interestingly, Yao never published his GCP. Several of his publications discuss approaches to the SFE problem generally, specifically papers from 1982[13] and 1986[14]. These papers are much broader in scope and are much more abstract than providing a protocol that could be implemented. Yao first discussed the *garbled circuits* approach in a public talk on the latter paper, as a concrete example of how his broader strategies could be applied[2]. Only later and by other researchers would the protocol be documented formally[6], though still crediting Yao for the approach.

Yao having developed this foundational protocol, but never having published it, presents authors with the tricky question of what to cite when crediting to the GCP approach. The common approach seems to be to cite Yao's two papers discussing his general approach the problem, even though those papers make no mention of garbled circuits or any similar concept.

1.2 Aims of the Paper

This paper aims to provide a full description of Yao's GCP and its security characteristics, namely what security the protocol does and does not provide. This paper also provides detailed explanations of related work done by other authors to improve the performance and security provided by the protocol.

This paper presumes no previous familiarity with Yao's protocol or cryptography in general in the explanation explanation of the protocol, beyond the general concepts of symmetric and asymmetric cryptography. Some background in cryptography is assumed in the sections on improvements and additions to the protocol. Formal proofs of the underlying concepts are not discussed and are left to their originating papers.

Some discussion is included of existing implementations of Yao's protocol. However, the focus here is on the promises, improvements and general techniques of the implementations, and not on implementation details like programming languages or hardware characteristics. Discussion of the implementations is mainly meant to inform how the protocol has developed and been improved, as opposed to a detailed comparison of how different implementations compare with each other.

1.3 Organization of the Paper

The remainder of the paper is structured as follows. Section 2 provides some security definitions used throughout the rest of the paper. Section 3 discusses oblivious transfer (OT), its role in the protocol, and a method for achieving OT in a manner that is compatible with the security guarantees of the standard version of Yao's protocol. Section 4 then provides a full explanation of the Yao's protocol and how to use *garbled circuits* to solve the *SFE* problem. Section 5 discusses the security of the protocol and proposed improvements, and **??** provides a similar discussion of performance issues in Yao's protocol. Section 8 provides a brief overview of some implementations of the protocol, and section 9 concludes.

2. SECURITY DEFINITIONS

This section defines several security related terms that are used through out this paper. The terminology is not identical throughout the literature, but have mappings onto similar, equivalent terms.

2.1 Properties of SFE System

Attempting to abstractly but precisely defining the characteristics of a SFE protocol is difficult and can quickly devolve into a long enumeration of characteristics a SFE system should *not* do. Instead, Yao suggests[14] that a correct system should be compared to an ideal-oracle that fulfills three properties, and that a SFE system is correct if it performs identically to this imagined ideal-oracle.

This imagined ideal-oracle takes a function to execute (f), the first party (*P1*)'s input (i_{P1}) and the second party (*P2*)'s input (i_{P2}), executes the given function with the values provided, and then returns the function's output to both parties ($u \leftarrow f(i_{P1}, i_{P2})$).

2.1.1 Validity

A SFE system must perform indistinguishably from an ideal-oracle in being able to correctly calculate the given function. Note that this does not guarantee a correct result, since the function being computed in a secure manner could itself have a logic error in it, nor does it guarantee to produce any answer, if one of the parties submits an invalid input to the computation. This *validity* requirement merely requires that the function produce the same result as the insecure (or "pre-secured") version of the function being evaluated, given the same inputs.

2.1.2 Privacy

A SFE system must also perform indistinguishably from an ideal-oracle in preventing preventing *P2* from learning about i_{P1} provided *P1* follows the protocol. The same must also hold for preventing *P1* from learning about i_{P2}.

Note that that this definition of *privacy* does not guarantee that *P1* is not able to learn *P2*'s input by examining the function's result (if the function being executed allows for such reverse engineering). If, for example, the function being evaluated securely is multiplication, the fact that *P1* can learn i_{P2} through u/i_{P1} does not violate this *privacy* property; *P1* could learn i_{P2} in this scenario given an ideal-oracle as well.

This does not imply that SFE cannot be used to protect the *privacy* of each parties' inputs, only that some functions (such as integer multiplication) do not make sense in the context of SFE.

2.1.3 Fairness

Finally, a SFE system must perform indistinguishably from an ideal-oracle in preventing one party from learning the output of f while learning it themselves. In order words, *P1* should not be able to learn the output of f while denying it to *P2*, and vise-versa.

2.2 Adversary Models

In addition to defining the properties a SFE should have, its necessary to define under which conditions those properties must hold. While the relevant literature contains many different terms for an adversary's willingness to deviate from the protocol, and many gradations between 100% honest and 100% malicious, this paper generalizes the types of attackers into two categories, at the extremes of the attacker spectrum.

A SFE protocol is said to be secure against under a given adversary model if the given SFE protocol can provide the three above mentioned security properties against any party following the assumptions of the adversary model.

2.2.1 Semi-Honest

A *semi-honest* adversary is assumed to follow all required steps in a protocol, but will also look for all advantageous information leaked from the execution of the protocol, such as intermediate values, control flow decisions, or values derivable from the same[5]. Additionally, *semi-honest* adversaries are assumed to be selfish, in that they will take any steps that will benefit themselves if the benefit is greater than the harm, within the constraints imposed by the protocol.

2.2.2 Malicious

A *malicious* adversary is assumed to arbitrarily deviate from the protocol at any point, and in whenever way it might benefit them[5]. This includes proving deceptive or incorrect values, aborting a protocol at anytime, or otherwise taking any steps that could reach a desirable outcome. This is the most difficult type of adversary to secure against; a system that is secure against *malicious* adversaries is also therefor secure against *semi-honest* adversaries.

3. OBLIVIOUS TRANSFER

OT refers to methods for two parties to exchange 1 of several values, with the sending party blinded to what value was selected, and the receiving party blinding to all other possible values that could have, but were not, selected.

While OT and SFE are approaches to distinct (though related) problems, understanding the Yao's GCP and the security properties requires some understanding of OT and how it is used in the protocol. Its a cryptographic primitive that serves as a building block that the security of Yao's GCP relies on.

This section provides a brief overview of the OT problem, a simple protocol that provides a solution to the OT problem against *semi-honest* adversaries. The role of OT in Yao's protocol is discussed in section 4.

3.1 Problem Definition

A general form of OT is *1-out-of-N oblivious transfer*, a two party protocol where *P1*, the sending party, has a collection of values. *P2* is able to select one of the values from this set to receive, but is not able to learn any of the other values.

More formally, a *1-out-of-N oblivious transfer* protocol takes as inputs a set of values N from *P1*, and i form *P2*, where $0 \leq i < |N|$. The protocol then outputs nothing to *P1*, and N_i to *P2* in a manner that prevents *P2* from learning another other values in *N*.

A special case of the above is the *1-out-of-2 oblivious transfer* problem, where *N* is fixed at 2. Here *P1* has just two values, and *P2* is accordingly limited to $i \in \{0, 1\}$. All versions of Yao's GCP discussed in this paper rely on *1-out-of-2 oblivious transfer* protocols.

3.2 Example 1-out-of-2 Protocol

The problem of *1-out-of-2 OT* was first addressed by Rabin[12] in 1981 using an online approach with multiple rounds of message passing, but was later adapted into an offline approaches using an techniques similar to the Diffie-Hellman key exchange protocol[3].

The following protocol[10] is a very simple *1-out-of-2 OT* protocol that is secure against *semi-honest*. It is included here to help in the next section's explanation of how the full GCP works, and to provide a easy-to-understand example of OT to build from later.

Protocol 1 Semi-Honest 1-out-of-2 Oblivious Transfer

```
1: P1 has a set of two strings, $S = \{s_0, s_1\}$.
2: P2 selects $i \in \{0, 1\}$ corresponding to whether she wishes
   to learn $s_0$ or $s_1$.
3: P2 generates a public / private key pair $(k^{pub}, k^{pri})$,
   along with a second value $k^{\perp}$ that externally appears to
   be public key, but for which P2 has no corresponding
   private key to decrypt with.
4: P2 then advertises both public keys as $k_0^{pub}, k_1^{pub}$, and
   sets $k_i^{pub} = k^{pub}$ and $k_{i-1}^{pub} = k^{\perp}$.
5: P1 generates $c_0 = E_{k_0^{pub}}(s_0)$ and $c_1 = E_{k_1^{pub}}(s_1)$ and
   sends $c_0$ and $c_1$ to P2.
6: P2 computes $s_i = D_{k^{pri}}(c_i)$.
```

Note that the protocol is secure by the *semi-honest* definition. As long as all parties do not deviate from the protocol, *P2* is able to recover the desired string from *S* but is not able to recover the other value from *S*. Similarly, *P1* does not know which value from *S P2* learned.

4. YAO'S PROTOCOL

This section provides a complete description of Yao's *garbled circuits protocol* and how the protocol incorporates OT. Though the protocol described here was first published by[6], the terminology used in this section follows more recent publications[7]. In all cases though the concepts are similar and there is a direct mapping between the two.

The protocol is presented twice, once in a less formal format that includes explanations of why the protocol requires each step, and a second time more formally, fully describing each step taken by both parties. The former section is intended to make the latter section either to follow.

Protocol 2 Yao's Garbled Circuits Protocol

```
1: P1 : generates a boolean circuit representation $c_c$ of $f$
   that takes input $i_{P1}$ from P1 and $i_{P2}$ from P2.
2: P1 transforms $c_c$ by garbling each gate's computation
   table, creating garbled circuit $c_g$.
3: P1 sends both $c_g$ and the values for the input wires in
   $c_g$ corresponding to $i_{P1}$ to P2.
4: P2 uses 1-out-of-2 OT to receive from P1 the garbled
   values for $i_{P2}$ to $c_g$.
5: P2 calculates $c_g$ with the encrypted versions of $i_{P1}$ and
   $i_{P2}$ and outputs the result.
```

4.1 Intuitive Description of the Protocol

This section attempts to provide a high level explanation of how Yao's protocol works and some of the reasoning behind its construction. It is included to make the following detailed description of the protocol easier to follow.

P1 and *P2* wish to compute function f securely, so that their inputs to the function remain secret. They will do so by modeling f as a boolean circuit. *P1* will then "garble" the circuit by replacing all boolean values in the circuit with pseudo-random looking strings, and then keeping the mapping secret. This is done for all gates in the circuit except for the output gates of the circuit. The values of the output wires for these gates are left un-garbled.

P1 will then similarly replace each bit of his input with the pseudo-random string that maps to that bit's input into the circuit. *P1* then sends the garbled circuit and the strings corresponding to his input bits to *P2*.

P2 receives both the garbled circuit and *P1*'s garbled input values. However, since all input wires into the circuit have been garbled and only *P1* has the mapping between the garbled values and underlying bits, *P2* does not know what values to input into the circuit to match her input bits. In other words, for each input wire into the circuit, *P2* can select one of two random strings to input (corresponding to 0 or 1), but does not know which of these correspond to her desired input bit.

In order to learn which pseudo-random string to select for each of *P2*'s input wires, *P2* engages in a *1-out-of-2 OT* with *P1* for each bit of *P2*'s input. For each round of the OT, *P2* submits the bit she wishes to learn, receives the corresponding string, and, *P1* learns nothing.

Once *P2* has received all of the strings corresponding to her input into the circuit, she holds everything needed to compute the output of the circuit, namely her garbled inputs, *P1*'s garbled inputs, and the garbled circuit itself. Further, she has obtained these values without *P1* learning her inputs, nor *P2* learning *P1*'s inputs.

P2 then begins to compute the circuit by entering the pseudo-random strings that correspond to each bit of her and *P1*'s input into each corresponding input gate and using the resulting string as the input to the next gate. *P2* may try to learn information about *P1*'s inputs by watching the execution of the circuit. The protocol prevents *P2* from doing so though the manner that each computation table for each gate was constructed.

Recall that the computation table for every gate in the circuit was constructed so that each pair of inputs produces a output string that represents the correct boolean result, but which appears pseudo-random to *P2*. In other words, instead of mapping from $\{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$, all gates in the circuit become a function mapping two random looking strings to another uniformly distributed pseudo-random string, or $f(\{0,1\}^{|k|}, \{0,1\}^{|k|}) \rightarrow \{0,1\}^{|k|}$, where $|k|$ is the key size of the of the encryption function. Since *P2* never learns the mapping between strings used in the table and their underlying boolean values, *P2* learns nothing by watching the outputs of each gate.

Recall that the output values of the output gates in the circuit are not masked. This results in *P2* learning the value of $f(i_{P1}, i_{P2})$ once the computation has finished. *P2* then shares this computed value with *P1*.

4.2 Detailed Description of the Protocol

This section provides a more detailed explanation of how each step of Yao's protocol, specifying how each step of the protocol can be implemented. The numbering of subsections here is intended to following the number of the protocol's steps in Protocol 2.

4.2.1 *Generating Boolean Circuit Representation of the Function*

The function f being securely evaluated must first be converted into and equivalent boolean circuit c so that $\forall x, y \in \{f(x,y) = c(x,y)\}$. The strategies for doing so are function specific, and thus is beyond the scope of the protocol. For the purposes of this paper though, it is sufficient to note that there exists a mapping from any polynomial time function with fixed sized inputs to a boolean circuit that calculates the same output[6].

4.2.2 *Garbling Truth Tables*

Once *P1* has constructed the boolean circuit representation c of f, the next step is to garble the truth table for each gate in c, or generating a garbled version of c from the clear version of c ($c_c \rightarrow c_g$).

To see how *P1* does, this first, consider a single logical OR gate, g_1^{OR}, represented in figure **??**. Initially *P1* generates the values for this gate as normal, resulting in the truth table in figure 2a. *P1* then generates a key for each possible value for each wire in the gate. This results in 6 keys being generated, for each of the two possible boolean values on each of the three wires in the gate.

P1 then encrypts each entry in the table for the output wire under the keys used for the corresponding inputs. The gate identifier serves as a nonce and is only included in this construction to to ensure that the same values are never encrypted twice in the circuit.

This encryption plays two important roles in the protocol. First, since the output of each encryption is assumed be random (i.e. the encryption function is assumed to perform like a random oracle), it removes any correlation between the underlying truth values in the table and the resulting garbled values. Even though this gate produces 3 identical boolean values, the garbled values all independently distributed, revealing nothing about the underlying value being masked.

Second, encrypting the output keys under the input keys prevents *P2*, the circuit evaluator, from playing with the circuit and considering other inputs other than those provided by *P1*. *P2* can only obtain one of the output keys from the table, since she will only have, at most, the necessary input keys to the gate to decrypt one value for the output wire.

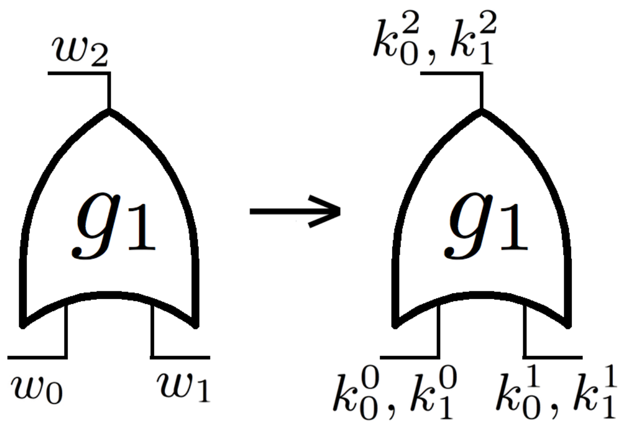

Figure 1: Garbling a single gate

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

(a) Original Values

w_0	w_1	w_2	output value
k_0^0	k_1^0	k_2^0	$E_{k_0^0}(E_{k_1^0}(k_2^0, g_1))$
k_0^0	k_1^1	k_2^1	$E_{k_0^0}(E_{k_1^1}(k_2^1, g_1))$
k_0^1	k_1^0	k_2^1	$E_{k_0^1}(E_{k_1^0}(k_2^1, g_1))$
k_0^1	k_1^1	k_2^1	$E_{k_0^1}(E_{k_1^1}(k_2^1, g_1))$

(b) Garbled Values

Figure 2: Computation table for g_1^{OR}

Once *P1* has garbled the values for one gate, he can continue the process to compose an arbitrarily large circuit. Figure 5 shows how multiple garbled gates can be composed together into a simple circuit, and the how the keys from each gate are carried forward into the next gate, blinding the computing party from the learning the intermediate values being calculated.

The only gates in the circuit that do not need to be garbled are the output gates, or gates who's wires do not serve as input wires to another gate. The values from these gates can remain obscured since they are outputting the final result of the circuit, a value which *P2* is allowed to learn.

Finally, once all the gates in the circuit are garbled, *P1* randomly permutes the order of each row in each table for each circuit, to further obscure the boolean values being input and output by each gate.

4.2.3 *Sending Garbled Values to* P2

Once *P1* has finished generating the garbled circuit, he then needs to garble his input to the function, creating a mapping of i_{P1} to the garbled equivalents. *P1* begins this process by replacing the first bit of his input with the corresponding key for that input wire in the circuit. For example, if in the circuit *P1*'s first bit was input into w_0, and the value of i_{P1}^0 was 1, *P1* would select k_0^1 to be the first value in his input to the garbled circuit. *P1* then repeats this procedure for the remaining bits in his input, creating *P1*'s garbled input. *P1* then sends the garbled circuit c_g and his garbled input to *P2*.

4.2.4 *Receiving* P2*'s Input Values through OT*

P2 receives the c_g and *P1* garbled inputs, but still needs the garbled representations of her own inputs to compute the

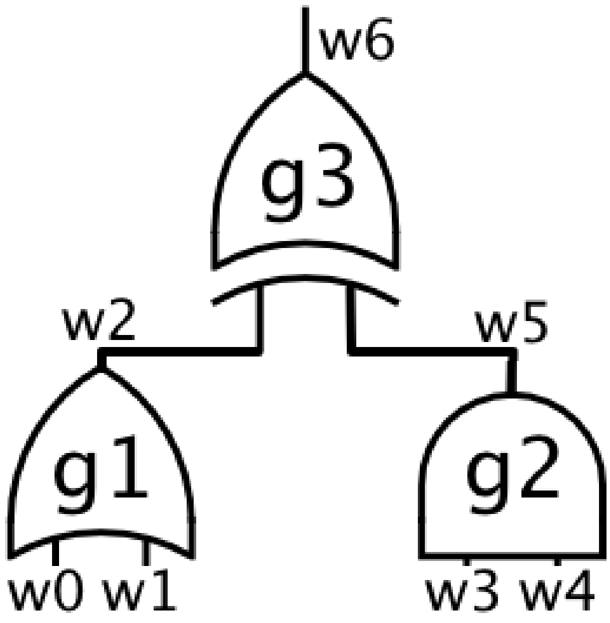

Figure 3: Composing several gates into a Simple Circuit

w_3	w_4	w_5
0	0	0
0	1	0
1	0	0
1	1	1

(a) Original Values

w_3	w_4	w_5	output value
k_3^0	k_4^0	k_5^0	$E_{k_3^0}(E_{k_4^0}(k_5^0, g_2))$
k_3^0	k_4^1	k_5^0	$E_{k_3^0}(E_{k_4^1}(k_5^0, g_2))$
k_3^1	k_4^0	k_5^0	$E_{k_3^1}(E_{k_4^0}(k_5^0, g_2))$
k_3^1	k_4^1	k_5^1	$E_{k_3^1}(E_{k_4^1}(k_5^1, g_2))$

(b) Garbled Values

Figure 4: Computation table for g_2^{AND}

w_2	w_5	w_6
0	0	0
0	1	1
1	0	1
1	1	0

(a) Original Values

w_2	w_5	w_6	output value
k_2^0	k_5^0	k_6^0	$E_{k_2^0}(E_{k_5^0}(k_6^0, g_3))$
k_2^0	k_5^1	k_6^1	$E_{k_2^0}(E_{k_5^1}(k_6^1, g_3))$
k_2^1	k_5^0	k_6^1	$E_{k_2^1}(E_{k_5^0}(k_6^1, g_3))$
k_2^1	k_5^1	k_6^0	$E_{k_2^1}(E_{k_5^1}(k_6^0, g_3))$

(b) Garbled Values

Figure 5: Computation table for g_3^{XOR}

output of the circuit.Recall that *P1* has the garbled values for all of *P2*'s input wires, but has no knowledge of what values correspond to *P2*'s true input. *P2*, inversely, knows the bits of her own input, but not the corresponding keys for the input wires into c_g.

P2 maps the bits of her input into their corresponding garbled values by engaging in a series of *1-out-of-2 OT*s with *P1*, where *P1*'s inputs are (k_0^0, k_0^1), and *P2*'s input is 0 or 1, depending on the first bit of *P2*'s input. *P2* the repeats the OT for all values $0 < i < |i_{P2}|$ to achieve her full garbled input into c_g.

4.2.5 Computing the Garbled Circuit

Once *P2* has both sets of garbled input values, and the garbled circuit, computing the final value is straight forward. For each input gate, *P2* looks up the corresponding value from *P1* and *P2*'s garbled input values and uses them as keys to decrypt the output value from the gate's garbled truth table. Since *P2* does not know which output key her two input keys correspond to, *P2* must try to decrypt each of the four output keys. If the protocol has been carried out correctly, only one of the four values will decrypt correctly. The other three decryption attempts will produce $\perp$. The newly decrypted key then becomes an input key to the next gate.

P2 continues this process until she reaches the output wires of the circuit. Each of these wires output a single, unencrypted bit. *P2* then reassembles the output bits and has the correct solution for the f encoded by c_g. *P2* completes the protocol by sending the output of the circuit to *P1*.

5. PROTOCOL SECURITY

Yao's protocol is designed to provide SFE against *semi-honest* adversaries. These security guarantees do not carry over against *malicious* adversaries though. This is a serious shortcoming for being able to make the protocol practical; there are relatively few real-world scenarios where you do not trust the other party to see your inputs to a function, but do trust them to forgo the opportunity to discover those same inputs by deviating from the protocol.

Much work has been conducted to extend Yao's protocol to be secure against *malicious* adversaries. This work can generally be classified into three areas, 1) creating *1-out-of-2 OT* protocols that are secure against *malicious* adversaries, 2) ensuring that the circuit constructing party correctly constructs the garbled circuit, and 3) that the circuit executing party returns provides the value output by the circuit to the other party.

5.1 Securing the OT Protocol

The *1-out-of-2 OT* protocol described in the section 3 is trivially vulnerable in the *malicious* case. Instead of generating $((k_b^{pub}, k_b^{pri}), (k_{b-1}^{\perp}, \perp))$, *P2* could easily generate two valid public / private key pairs, allowing her to recover both values sent by *P1*. Applied to Yao's protocol, this would allow *P2* to learn both the garbled versions of the 0 and 1 values for all of her input bits. *P2* having these additional keys would allow *P2* to decrypt additional values throughout the circuit garbled gate, violating the *privacy* requirement of SFE. Others have detailed several additional ways that using an insecure-in-the-*malicious*-case, OT protocol can be exploited by an attacker[9].

As previously discussed, OT is a distinct, though related, field to SFE in general and Yao's protocol in particular. As such, this section does not attempt to assess the state of the art of in the field of OT. A variety of other approaches to *malicious*-case secure *1-out-of-2 OT* protocols exist[11, 9, 6], each with their own tradeoffs, computation cost and underlying security assumptions. The below protocol[1][1] is included to show that efficient *1-out-of-2 OT* is possible, and that researchers have used it and equivalent OT protocols to make Yao's GCP secure in the *malicious* case.

[1]This protocol is a slightly modified version of the protocol presented in [1], to incorporate a change suggested by [11] to remove the reliance on a external zero knowledge proof or other out-side-the-protocol source for *C*.

Protocol 3 Malicious-Secure 1-out-of-2 Oblivious Transfer

```
1: P1 has a set of two strings, $S = \{s_0, s_1\}$.
2: P1 (sender) and P2 (receiver) agree on some $q$ and $g$ such that $g$ is a generator for $\mathbb{Z}_q^*$.
3: P1 selects a random $C$ from $\mathbb{Z}_q^*$, or more generally such that P2 does not know the discrete log of $C$ in $\mathbb{Z}_q^*$.
4: P2 selects $i \in \{0, 1\}$ corresponding to whether P2 wants $s_0$ or $s_1$. P2 also selects a random $0 \leq x_i \leq q - 2$.
5: P2 sets $\beta_i = g^{x_i}$ and $\beta_{i-1} = C \bullet (g^{x_i})^{-}1$. $(\beta_0, \beta_1)$ and $(i, x_i)$ form P1 public and private keys, respectively.
6: P1 checks the validity of P2's public keys by verifying that $\beta_0 \bullet \beta_1 = C$. If not, P1 aborts.
7: P1 selects $y_0, y_1$ such that $0 \leq y_0, y_1 \leq q - 2$, and sends P2 $a_0 = g^{y_0}, a_1 = g^{y_1}$.
8: P1 also generates $z_0 = \beta_0^{y_0}, z_1 = \beta_1^{y_1}$ and sends P2 $r_0 = s_0 \oplus z_0$ and $r_1 = s_1 \oplus z_1$.
9: P2 computes $z_i = a_i^{xi}$ and then receives $s_i$ by computing $s_i = z_i \oplus r_i$.
```

Step 5 is not in mentioned in the original[1] statement of the protocol, but was added by the author to better explain how the protocol works. Here, note that *P1* checks that $\beta_0 \bullet \beta_1 = C$ to prevent *P2* from being able to decrypt under both β_0 and β_1, to force *P2* to choose one or the other. As long as the assumption that *P2* does not know the discrete log of C holds, then it follows that *P2* cannot know the discrete log of both β_0 and β_1.

5.2 Securing Circuit Construction

References

[1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in CryptologyâĂŤCRYPTOâĂŹ89 Proceedings*, pages 547–557. Springer, 1990.

[2] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[4] C. Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[5] O. Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 1998.

[6] 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.

[7] C. Hazay and Y. Lindell. Efficient secure two-party protocols. *Information Security and Cryptography. Springer, Heidelberg*, 2010.

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

[9] M. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yaoâĂŹs garbled circuit construction. In *27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006.

[10] Y. Lindell. Secure two-party computation in practice. Lecture given at Technion-Israel Institute of Technology TCE Summer School 2013, `https://www.youtube.com/watch?v=YvDmGiNzV5E`, 2013.

[11] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

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

[13] A. C.-C. Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

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