Yao's Garbled Circuits: Recent Directions and Implementations

Peter Snyder
University of Illinois at Chicago
psnyde2@uic.edu

ABSTRACT

Secure function evaluation, or how two parties can jointly compute a function while keeping their inputs private, is 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 initially treated as theoretically interesting but too computationally expensive for practical use. However, in the decades since Yao published his solution, a great deal of work has gone into both optimizing the protocol for practical use, and further securing the protocol to make it useful in untrusted scenarios.

This paper provides a thorough explanation of Yao's original protocol and its security characteristics. The paper then details additions to the protocol to both secure it against untrusted parties and to make it practical for computation. 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 two parties can collaborate to correctly compute the output of a function without either 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 want to determine who is richer, without either party revealing how much money they have[25].

Many solutions have been proposed for SFE. One category of solution is function specific, and depends on specific properties of the function being executed to provide security[11]. These solutions, while interesting, are of less general interest, since they apply to only a limited set of problems.

Another category of solution is general in approach, 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 computation on encrypted data. Yao's *garbled circuits* protocol also fits in this second, general 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 masking the inputs and outputs of each gate so that the party executing the function cannot discern any information about the inputs or intermediate values to the function. The protocol is secure as long as both parties do not deviate. 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[25] and 1986[26]. These papers are broad and theoretical, and do not directly provide 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[7], though still crediting Yao for the approach.

That Yao developed this foundational protocol, but never published it, presents writers with the tricky question of what to cite when crediting the GCP approach. The common convention 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.

1.2 Aims of the Paper

This paper aims to provide a full description of Yao's GCP and its security characteristics. This paper also provides detailed explanations of related work by other researchers to improve the performance and security of the protocol.

This paper presumes no previous familiarity with cryptography (generally) or Yao's protocol (specifically) in the sections explaining the protocol, though the general concepts of symmetric and public key cryptography are referenced. Some background in cryptography is assumed in the sections on security and performance improvements to the protocol. Formal proofs of the underlying concepts are not discussed and are left to their original 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 specific details, like implementing 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 section 6 provides a similar discussion of the performance of Yao's protocol. Section 7 briefly describes some implementations of the protocol, and section 8 concludes.

2. SECURITY DEFINITIONS

This section defines several security-related terms that are used throughout the paper. The terminology is not identical throughout the literature, but in most cases has simple mappings to equivalent terms.

2.1 Required Properties for SFE Protocols

Attempting to abstractly but precisely define the characteristics of a SFE protocol is difficult and can quickly devolve into a long enumeration of characteristics a SFE system should *not* have. Yao instead suggests[26] 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 desired function. Note that this does not guarantee a correct result, since the function being securely computed could itself contain a logic error, nor does it guarantee the production of any answer, since one party could submit an input outside the domain of the function. This *validity* requirement merely requires that the secure version of 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} given an ideal-oracle too.

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 do as well as a ideal-oracle in preventing one party from learning the function's result without sharing it with the other party. In order words, *P1* should not be able to learn u while denying it to *P2*, nor vise-versa.

2.2 Adversary Models

In addition to defining the properties a SFE protocol should have, it is necessary to define under which conditions those properties must hold. While the relevant literature contains a variety of terms for an adversary's willingness to deviate from the protocol, for different gradations between 100% honest and 100% dishonest, 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 under a given adversary model if the 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 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 in any way that 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 necessarily secure against *semi-honest* adversaries.

2.3 Hash Function Assumptions

This paper makes the assumption that hash functions generally, or at least some existing efficient hash function, model a random oracle. More formally, the paper assumes that a hash operation can be treated as a uniformly distributed mapping from $f(\{0,1\}^*) \rightarrow \{0,1\}^h$, where h is the length of the produced message digest. This assumption implies that there is no correlation between the output of a hash function and its input. Put differently, the assumption implies that nothing can be learned about the input to a hash function by examining its output.

This assumption is a common one throughout the field and discussed research[22]. While some research mentions alternate constructions or other caveats if this random oracle assumption does not hold, these are are rarely the main assumption in the work, and thus are not discussed further in this paper.

3. OBLIVIOUS TRANSFER

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

While OT and SFE are approaches to distinct (though related) problems, understanding Yao's GCP and its security

properties requires some understanding of OT. Its a cryptographic primitive and a building block that Yao's GCP builds on. This section provides a brief overview of the OT problem and a simple OT protocol that is secure against *semi-honest* adversaries. The role of OT in Yao's protocol is discussed in section 4, and a more secure OT protocol is included and explained in section 5.

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 an index i from *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 N_j for all values of $j \neq i$.

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[23] in 1981 using an interactive 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[15] is a simple *1-out-of-2 OT* protocol that is secure against *semi-honest* adversaries. It is included here to assist 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 is indistinguishable from a public key, but for which *P2* has no corresponding private key to decrypt with.
4: *P2* then advertises these values as public keys (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 no party deviates from the protocol, *P2* is able to recover the desired string s_i but is not able to recover the other value, s_{i-1}. Similarly, *P1* never learns i.

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 Goldreich, Micali, and Wigderson[7], the terminology used in this section follows more recent publications[10]. In all cases though the concepts are similar and there is a direct mapping between the two.

The protocol is presented here twice, first in a less formal format that includes some reasoning for each step in the protocol, and a second time, fully describing each step taken by both parties. The former description is intended to make the latter one easier 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} in c_g.
5: *P2* calculates c_g with the garbled 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, as well as 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 begin doing so by modeling f as a boolean circuit. *P1* then "garbles" the circuit by replacing all boolean values in the circuit with pseudo-random looking strings, and then keeping this mapping secret. This is done for the input and output wires of every gate in the circuit, with the exception of the circuits output gates; the values of these gates' output wires are left un-garbled.

P1 then replaces each bit of his input with the pseudo-random string that maps to that bit's input on the corresponding input wire into circuit. *P1* then sends the garbled circuit and his garbled input to *P2*.

P2 receives both the garbled circuit and *P1*'s garbled input. However, since all input wires into the circuit have been garbled and only *P1* has the mapping between the garbled values and the 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. Note that the properties of OT prevent *P1* from learning about *P2*'s input in this process.

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: 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 the corresponding input wire and using

the resulting garbled output string as an 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 size of the value returned by the hash 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 values returned by the output gates in the circuit are not obscured. This results in *P2* learning the value of $f(i_{P1}, i_{P2})$ once the computation has finished. *P2* then completes the protocol by sharing this computed value with *P1*.

4.2 Detailed Description of the Protocol

This section provides a more precise explanation of each step of Yao's protocol, specifying how each step of the is carried out by both parties. The numbering of subsections here followings the numbering used in protocol 2.

4.2.1 *Generating A Boolean Circuit Representation of the Function*

Before it can be securely evaluated, the function f must be converted into an equivalent boolean circuit c so that $\forall x, y \rightarrow f(x,y) = c(x,y)$. The strategies for optimally doing so may be function specific, and are 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[7].

4.2.2 *Garbling Truth Tables*

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

To see how *P1* does this, first consider a single logical OR gate, g_1^{OR}, represented in figure 1. 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, one 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 using the keys used for the corresponding inputs. The gate identifier serves as a nonce and is only included in this construction to ensure that the same values are never encrypted twice in the circuit. *P1* then randomly orders the rows the table, further obscuring the underlying boolean values[1]).

This encryption plays two important roles in the protocol. First, since the output of each encryption operation is assumed be random (i.e. the hash 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 three identical boolean values, the garbled values all uniformly distributed, revealing nothing about the underlying value being encrypted.

[1]To simplify the presentation, this step is not shown in figure 1.

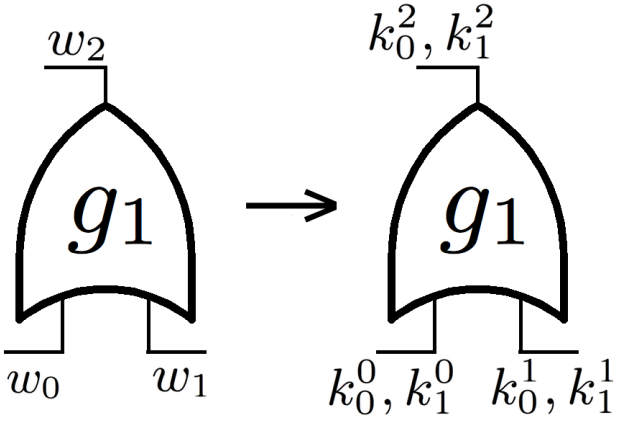

Figure 1: Garbling a single gate

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	garbled value
k_0^0	k_1^0	k_2^0	$H(k_0^0 \|\| k_1^0 \|\| g_1) \oplus k_2^0$
k_0^0	k_1^1	k_2^1	$H(k_0^0 \|\| k_1^1 \|\| g_1) \oplus k_2^1$
k_0^1	k_1^0	k_2^1	$H(k_0^1 \|\| k_1^0 \|\| g_1) \oplus k_2^1$
k_0^1	k_1^1	k_2^1	$H(k_0^1 \|\| k_1^1 \|\| g_1) \oplus k_2^1$

(b) Garbled Values

Figure 2: Computation table for g_1^{OR}

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.

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 unobscured since they are outputting the final result of the circuit, a value which *P2* is allowed to learn.

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 its 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, *P1*'s first bit was input into w_0, and the value of

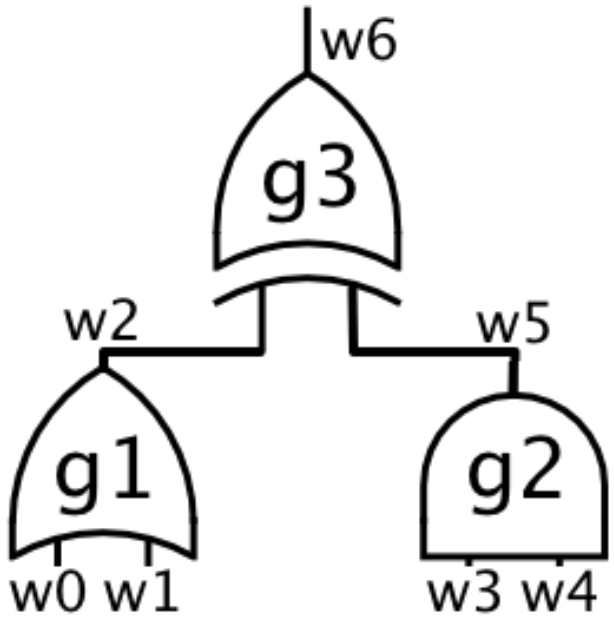

Figure 3: Composing several gates into a Simple Circuit

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	garbled value
k_3^0	k_4^0	k_5^0	$H(k_3^0 \| \| k_4^0 \| \| g_2) \oplus k_5^0$
k_3^0	k_4^1	k_5^0	$H(k_3^0 \| \| k_4^1 \| \| g_2) \oplus k_5^0$
k_3^1	k_4^0	k_5^0	$H(k_3^1 \| \| k_4^0 \| \| g_2) \oplus k_5^0$
k_3^1	k_4^1	k_5^1	$H(k_3^1 \| \| k_4^1 \| \| g_2) \oplus k_5^1$

(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	garbled value
k_2^0	k_5^0	k_6^0	$H(k_2^0 \| \| k_5^0 \| \| g_3) \oplus k_6^0$
k_2^0	k_5^1	k_6^1	$H(k_2^0 \| \| k_5^1 \| \| g_3) \oplus k_6^1$
k_2^1	k_5^0	k_6^1	$H(k_2^1 \| \| k_5^0 \| \| g_3) \oplus k_6^1$
k_2^1	k_5^1	k_6^0	$H(k_2^1 \| \| k_5^1 \| \| g_3) \oplus k_6^0$

(b) Garbled Values

Figure 5: Computation table for g_3^{XOR}

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 c_g and *P1*'s garbled input, but still needs the garbled representation of her own input to compute 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 her input wires in c_g.

P2 maps the first bit of her input to its corresponding garbled value by engaging in *1-out-of-2 OT*s with *P1*, where *P1*'s inputs are (k_1^0, k_1^1), and *P2*'s input is 0 or 1, depending on the first bit of *P2*'s input. *P2* performs additional OTs with *P1* 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 garbled inputs and the garbled circuit, she can straight forwardly compute the circuit. 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 these 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. This is a serious limitation for making 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) preventing *P1* from gaining an advantage by sending *P2* corrupt values for her input.

Finally, some discussion is given to the open problem of how to guarantee *fairness* in the *malicious* case, by ensuring *P2* returns output to *P1* at the end of the protocol.

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, 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[12].

As previously discussed, OT is a distinct, though related, field to both SFE and Yao's protocol. As such, this section does not attempt to assess the state-of-the-art in OT. A variety of other approaches to *malicious*-case secure *1-out-of-2 OT* protocols exist[19, 12, 7, 20], each with their own requirements, computation costs and underlying security

assumptions. The below protocol[1][2] is included to show that efficient *1-out-of-2 OT* with *malicious* adversaries is possible, and that researchers have used it and equivalent OT protocols to make Yao's protocol secure in the *malicious* case.

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 value C from $\mathbb{Z}_q^*$ 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 $\beta_{i-1} = C \bullet (g^{x_i})^{-1}$. (β_0, β_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* $a_0 = g^{y_0}$ and $a_1 = g^{y_1}$.
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 $z_i = a_i^{x_i}$ and then receives s_i by computing $s_i = z_i \oplus r_i$.

The purpose of many of the steps in the protocol are not explicit in the original work[1], so some explanation is provided below. Specifically, in step 5 *P1* checks that $\beta_0 \bullet \beta_1 = C$ to prevent *P2* from being able to decrypt under both β_0 and β_1, and 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.

Steps 7, 8 and 9 function similarly to a Diffie-Hellman key exchange. However, in step 9 it may not be immediately obvious why *P2* is able to reconstruct z_i to decrypt r_i and receive s_i. To understand why, recall that a_i is equal to g^{y_i}, making $a_i^{y_i} = g^{y_i^{x_i}} = g^{y_i \bullet x_i}$.

Similarly , recall that z_i was generated from $B_i^{y_i}$ and that $B_i = g^{x_i}$. This makes $B_i^{y_i} = g^{x_i^{y_i}} = g^{x_i \bullet y_i}$. Since *P2* is able to construct the same pad *P1* used to mask s_i, *P2* can undo the mask and receive s_i.

5.2 Securing Circuit Construction

A second way a malicious adversary could exploit Yao's protocol to learn information about the other party's input is by *P1* creating and garbling a circuit for a function other than the function expected by *P2*. Trivially, *P1* could send *P2* a garbled circuit that echos back *P2*'s input. More reasonably, *P1* could construct the circuit to output a value that leaks information about i_{P2} in some less obvious manner not known to *P2*. It is therefor necessary for *P2* to ensure that the garbled circuit she evaluates is actually modeling the expected function.

5.2.1 Zero-Knowledge Proofs

Two different general strategies for achieving this assurance have been promoted. The first approach has *P1* generate a zero knowledge proof of the garbled circuit's correctness, and then sends this proof to *P2* along with the garbled circuit and *P1*'s garbled inputs[7, 6].

This zero-knowledge strategy dates back to earlier in the history of Yao's protocol, when the protocol served more as proof that SFE was possible and less as a practical tool for actually achieving SFE. More recent, implementation-focused work on Yao's protocol has treated the zero-knowledge proof approaches as too expensive for practical use[16, 18, 17], and thus this strategy is not discussed further in this paper.

5.2.2 Cut-and-Choose

Instead, recent work on Yao's protocol has focused on a *cut-and-choose* strategy for securing circuit construction[17]. Work on this approach has developed in an arms-race fashion, with proposals being made, other researchers revealing shortcomings in the given strategy, and a revised strategy being developed to address the given weakness. Several rounds of this propose-attack-revise cycle are discussed below, to better explain the role of each proposed improvement.

Standard Cut-and-Choose

Under this approach, *P1* constructs m versions of the circuit, each structured identically but garbled differently so that the keys for each gate in each circuit are unique. *P1* does the same for his inputs to each of the m garbled circuits. Additionally, *P1* generates a "commitment" for each of his garbled inputs, which for simplicity can be understood to be a simple hash of the inputs[3].

P1 then sends each of these pairs of garbled circuits and associated input commitments to *P2*, who selects $m-1$ versions of the circuit to verify. *P1* de-garbles each of the $m-1$ selected circuits, so that *P2* can see the underlying circuit with the now unobscured boolean values in each gates' computation table. *P2* can then verify that each of the revealed circuits are constructed correctly and as expected.

If everything looks correct to *P2* she will continue with the computation by receiving *P1*'s garbled inputs, checking that they match their corresponding, previously sent commitment (again, most simply thought of as checking that the hash of the received garbled inputs matches the previously sent hash), and then proceed with Yao's protocol as normal. This reduces the chances of *P1* tricking *P2* into computing a corrupted circuit to $1/m$. This protocol is described more formally in protocol 4.

[2]This protocol is a slightly modified version of the protocol presented in [1]. It incorporates a change suggested by [19] to remove the reliance on a external zero knowledge proof or other out-side-the-protocol source for C.

[3]Though several more secure methods of committing are mentioned in [16], a simple hash function is used here to simplify the description of the cut-and-choose approach here, and commitment schemes in general throughout this paper. A more secure approach is described in [9].

Protocol 4 Securing Circuit Construction With Cut-and-Choose

1: *P1* generates m garbled versions of the circuit c, along with a corresponding garbled version of his input, called X_i for $0 \leq i < m$.
2: *P1* uses hash function H to generate commitments to each garbled input, $COM_i = H(X_i)$ for $0 \leq i < m$.
3: *P1* sends *P2* m garbled circuits and COM such that $|COM| = m$.
4: *P2* selects $0 \leq j < m$ and *P1* un-garbles all circuits except the jth.
5: *P2* inspects all $m-1$ circuits to check that they are correctly formed. If not, *P2* aborts.
6: *P2* receives *P1*'s garbled inputs to circuit j and confirms that *P1* did not change his inputs by verifying $COM_j = H(X_j)$. If not, *P2* aborts.
7: Otherwise, *P2* receives the continues with Yao's protocol as normal.

Further Securing Cut-and-Choose

However, for many applications one may wish for a stronger guarantee against executing a malicious circuit from *P1*. Lindell and Pinkas[16] discovered that *P1*'s odds of success can be dramatically reduced without the overhead of needing to generate additional circuits by altering the cut-and-choose strategy slightly.

Instead of *P1* revealing $m-1$ circuits, Lindell and Pinkas have *P2* select only $m/2$ circuits to be revealed. *P2* computes the remaining $m/2$ circuits and takes the majority result. Under this construction, a malicious *P1* would only succeed in having *P2* output a corrupt result if

1. *P1* constructs more than $m/4$ of the circuits corruptly, and
2. None of the corrupt $m/4$ circuits are among the $m/2$ circuits *P2* selected to be revealed.

Lindell and Pinkas measure *P1*'s chance of success in such an attack at $2^{-0.311m}$, where m is the number of circuits generated[16].

Majority Result as a Defense Against a Malicious Circuit

An immediate question that comes out of the above approach is why *P2* should take the majority result of the computed $m/2$ circuits, instead of immediately aborting when encountering the first corrupt circuit, especially given that computing a garbled circuit is an expensive operation. The reason is that, were *P2* to abort if all circuit outputs were not identical, she would become vulnerable to a different attack from *P1*.

Consider the case where *P1* constructs all circuits correctly, with a single exception. This corrupt circuit outputs the correct value of the function $\oplus$'ed with the first bit of *P2*'s input. By observing whether *P2* evaluates all $m/2$ evaluation-circuits, *P1* learns the first bit of *P2*'s input.

Figure 6 provides a full explanation of how *P1* performs this attack. The first column depicts the first bit of the correct value returned by f, and the second column shows the first bit of *P2*'s input. The third column shows the first bit of the value returned by *P1*'s malicious circuit, and column four describes the value *P2* returns from evaluating the entire $m/2$ set of circuits (or $\perp$ if *P2* aborts). Finally, column five shows what *P1* is able to learn about *P2*'s input, without having access to the first three columns of the table.

f_0	i_{P2_0}	$f_0 \oplus i_{P2_0}$	*P2* returns	*P1* learns
0	0	0	0	$i_{P2_0} = 0$
0	1	1	$\perp$	$i_{P2_0} = 1$
1	0	1	1	$i_{P2_0} = 0$
1	1	0	$\perp$	$i_{P2_0} = 1$

Figure 6: Using a corrupted circuit to learn first bit of *P2*'s input

Further Work Ensuring Consistent Inputs From *P1*

Lindell and Pinkas's[16] solution of having *P1* provide m versions of the garbled circuit succeeds in providing *P2* with a high degree of confidence that *P1* has not corrupted any circuits. However, it leads to another problem, of ensuring that *P1* provides the same input to each of the circuits *P2* evaluates. Solutions to this problem involve additional, involved protocols[16] or rely on other areas of cryptography[24] that are beyond the scope of this paper. More discussion of this problem, as well as possible solutions to it, are provided by other work[14].

5.3 Securing Against Corrupt Inputs

A third area where a malicious party can exploit the original construction of Yao's protocol is in the values *P1* returns to *P2* in the *1-out-of-2 OT* step. Recall that this OT step is taken to prevent *P1* from learning whether *P2* is requesting the garbled value for a 0 or a 1 bit in her input. The protections given in the previous two subsections provide no security against this attack. Subsection 5.1 only addresses ensuring that *P1* cannot learn *P2*'s inputs *during* the OT step, not that these inputs cannot be leaked elsewhere in the protocol. Subsection 5.2 provides *P2* guarantees that the circuits being evaluated are not corrupt, but provides no guarantees that the inputs to those circuits are not corrupt.

P1 can exploit this weakness in the protocol to gain information about *P2*'s input in the following manner. Instead of returning *P2* the correct garbled values for each bit of her input, *P1* returns the correct garbled value for 0, and a corrupt value for 1. *P1* can then learn whether *P2* received the 0 or 1 value by observing if *P2* aborts while computing the circuit. If *P2* received the 0 value, she will be able to compute the circuit as normal; if *P2* received the 1 value, she will not be able to compute the circuit and will be forced to abort. Either way, *P1* is able to learn the value of a bit of *P2*'s input.

Lindell and Pinkas[16] provide a method of securing the protocol against this attack. Their technique is depicted in figure 7. The defense works by adding $s|i_{P2}|$ input bits to the circuit, where s is a chosen security parameter. Each of *P2*'s input bits is replaced by the result of XOR of s new input bits, each chosen by *P2* from *P1* through the same OT protocol. The circuit is also augmented to reflect these new input wires and XOR gates. This step of indirection gives *P2* 2^{s-1} ways to receive her true input bits from *P1*, and prevents *P1* from gaining any knowledge about *P2*'s underlying input bit by corrupting the augmented, XORed inputs.

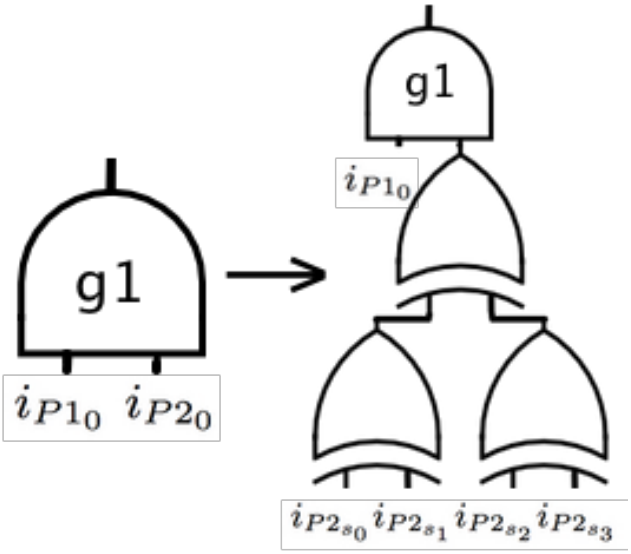

Figure 7: Securing against *P1* providing malicious inputs through s new $\oplus$ gates

Note that this construction does not prevent *P1* from forcing *P2* to abort when executing the circuit, it only prevents *P1* from learning anything about *P2*'s input.

5.4 Ensuring *P2* Returns At All

One remaining problem with Yao's protocol in the *malicious* setting is how to ensure *P2* returns to *P1* any output value from the function. There is no clear method to prevent *P2* from maliciously aborting the protocol early, right after computing the output of the garbled circuit, but right before sending the output to *P1*. That this problem is still open means that the *fairness* principle Yao described for SFE cannot be guaranteed in the *malicious* setting.

As a second best option, much work has been done to ensure that if *P2* does output a value, it is the correct computation returned by the function[24]. Another possibility though is that, until a solution to this problem is found, Yao's protocol is only appropriate in the *malicious* setting where the inputs to the function need to be kept private, but the output does not. An example of such a scenario is a secure voting systems. However, this restriction clearly limits the number of problems for which Yao's protocol is appropriate.

6. PROTOCOL PERFORMANCE

Yao's protocol gives a polynomial time solution for the SFE problem, both in the *semi-honest* and *malicious* cases (once the adjustments discussed in section 5 are made). However, while Yao's protocol is by this definition "efficient", it is also costly, and for many problems prohibitively so. For example, Kreuter, shelat and Shen[14] found that computing the edit distance of two 4095-bit strings required a circuit of over 5.9 billion gates and several hours of time, even with a highly optimized circuit.

A great deal of work has been done to make Yao's protocol less expensive to execute. This work broadly falls into three categories: 1) communication optimizations that reduce the amount of information that must be shared between the two parties, 2) execution optimizations that that allow for the same number of gates to be executed in a shorter amount of time, and 3) circuit optimizations that reducing the number of gates needed to compute a function.

Optimizations do not always cleanly fall into only one of these categories, and improvements in one area often have spill over benefits in another. For example, reducing the number of gates needed to compute a circuit also reduces the number of gates that need to be communicated between parties. The following categorization is more meant to provide an intuition about the main role of each optimization, and less a strict taxonomy of each contribution.

6.1 Communication Optimizations

The communication costs of transmitting a garbled circuit from *P1* to *P2* dwarfs all other communication related costs in Yao's protocol[4]. To see why, recall that circuits can grow to contain billions of gates, and that each wire connecting each of these gates is represented by four multi-byte strings, meaning each garbled circuit can be gigabytes in size. This problem is made worse when considering the protocol in the *malicious* setting, where the *cut-and-check* strategy requires *P1* to send many copies of the garbled circuit to *P2*. Minimizing the amount of information that must be communicated between the parties in the protocol is therefor a significant issue in making Yao's protocol practical.

6.1.1 Random Seed Checking

One solution to this problem was presented by Goyal, Mohassel and Smith[8]. Their technique consists of two modifications that together significantly reduce the communication costs of Yao's protocol.

First, instead of having *P1* assign values for each wire in the circuit randomly, *P1* selects a random seed for each garbled version of the circuit, then uses that random seed to deterministically generate each of the the pseudo-random values used in the circuit.

Second, instead of sending *P2* m copies of the garbled circuit during the *cut-and-check* phase, *P1* instead sends *P2* "commitments" for each version of the circuit. *P2* chooses the $m/2$ circuits for *P1* to reveal. Instead of sending complete versions of each garbled circuit to *P2*, *P1* sends the random seeds used for each selected circuit, along with any structural information *P2* needs to generate the garbled circuit from the random seed. *P2* can then generating the garbled circuit herself, using the random seed to duplicate the pseudo-random values *P1* generated.

Once *P2* has reconstructed a version of the garbled circuit that she knows to be correct using the random seed, she then checks that *P1*'s commitments for each circuit are correct (loosely, by hashing each random-seed generated circuit and seeing if it matches the corresponding commitment). Finally, *P1* sends *P2* the remaining $m/2$ circuits for *P2*'s evaluation.

This technique reduces the communication overhead of the protocol by approximately 1/2, since the commitments *P1* sends are constant in size and much much smaller than the size of a circuit.

[4]For all but the most trivial functions.

6.2 Execution Optimizations

Another area of optimization consists of techniques for reducing the number of resources, both in terms of time and computation power, needed to evaluate one or more garbled circuits. Optimizations in this section deal with how two parties can securely evaluate a garbled circuit more efficiently *without* needing to alter the structure of the circuit.

6.2.1 Fast Table Lookups

The *fast table lookups*[5] technique speeds up *P2*'s evaluation of a circuit by removing the need for *P2* to attempt to decrypt each row of each gate's garbled truth table until she finds a value that decrypts correctly. Instead, the circuit constructor adds an additional bit to the end of each garbled output value. This additional bit serves as half of an index into the next gate's garbled truth table. Since each garbled truth table contains four rows, and each gate has two input wires (each with one index bit), combining the index bits from both input values can uniquely identify which of the four rows in the next gate's garbled truth table the input values decrypt.

Note that since the order of the rows in each garbled truth table is randomized during construction, these index values do not reveal any information about the underlying values, and thus do not affect the security of the the system.

Further note that the approach described above is functionally equivalent to the method described in[17], but just described from opposite direction. Milkhi et al. decide the order of the entries in the garbled truth table based on the assigned index bits on each input value. The above description achieves the same outcome in reverse by randomly ordering the garbled truth table and then assigning the correct index bits to the values of the input strings[6].

6.2.2 Pipelined Circuit Execution

Garbled circuits for even simple functions can grow extremely large, making them difficult to store in memory for both the generating and computing party, as well as time consuming to securely evaluate (since *P2* waits idle in the protocol while *P1* garbles the circuit). Huang, Evans, et al.[11] realized that the garbling and executing processes could be partially parallelized, with *P1* sending *P2* the garbled gates as quickly as he is able to prepare them, and *P2* continuing to compute as long as she has holds at least one gate for which she has inputs for.

This technique has two benefits. First, it prevents either party from needing to keep an entire circuit in memory (though the optimal strategy for minimizing what subset of the circuit must be kept in memory is an open problem[14]), and second, it roughly reduces the time needed to compute a garbled circuit from $t_{garble} + t_{OT} + t_{evaluate}$ to $max(t_{garble}, t_{evaluate}) + t_{OT}$.

The above construction works in the *semi-honest* case, but seems unworkable in the *malicious* case. Recall that securing the protocol in the *malicious* case is done with the *cut-and-choose* technique, where *P1* creates many garbled versions of the circuit and *P2* selects a subset to be un-garbled. This would seem to require that *P1* hold all m circuits simultaneously. The previously discussed *random seed checking* approach similarly seems to make this pipelining and parallel execution strategy impossible, since it seems to require *P1* to hold all m copies of the circuit until *P2* has made her selection of circuits to verify.

A method for achieving the memory and execution time improvements of this *pipelined circuit execution* technique in the *malicious* case was developed by Kreuter, shelat and Shen[14]. Their solution is to have *P1* generate each garbled circuit twice, once before *P2* selects which circuits to verify, and then again after *P2* has made her choices.

In the first phase, *P1* generates each garbled circuit, generates a commitment for it, saves the generating random seed, and the discards the circuit before generating the next circuit and commitment.

In the second phase, once *P2* has selected which $m/2$ circuits to evaluate, *P1* can reconstruct each garbled circuit, one-at-a-time, and send them to *P2*. *P2* can, in turn, evaluate each circuit as she is receiving it, as she would in the *semi-honest* case. This achieves the intended optimization of neither party needing to store more than one circuit at a time, or hold an entire circuit in memory, without giving up the communication improvements of the *random seed checking* technique.

6.3 Circuit Optimizations

A straight forward way of reducing the cost of Yao's protocol is to reduce the size of the garbled gates that must be evaluated. This section discusses several strategies that have been used to reduce the number of garbled values needed in a garbled circuit.

6.3.1 Circuit Simplification

Reducing the number of gates in the pre-garbled circuit trivially reduces the number of garbled gates that need to be evaluated later on. This can be though of as a preprocessing stage that optimizes the circuit before garbling it. Put another way, this step attempts to remove inefficiencies introduced when the underlying function was being encoded as a circuit.

Circuit optimization strategies include looking for unused gates, gates that have no effect on the circuit's output, finding sub-circuits that can be more efficiently represented by a smaller number of gates, and removing identity gates and sub-circuits that are guaranteed to evaluate to 0 or 1 and replacing them with simpler constructions[14, 22]. The benefit of from this type of optimization will be inversely related to the quality of circuits generated by the function-to-circuit translating process. One study[22] found a 60% reduction in circuit size when optimizing circuits generated from a common circuit generator[17].

6.3.2 Free XORs

A second strategy for reducing the number of gates needed in a garbled circuit is the *free XOR* technique, discovered by Kolesnikov and Schneider[13]. This optimization allows for the circuit constructor to replace all garbled XOR gates in the circuit with a simple XOR operations. This results in the significant improvement of removing four garbled values from the circuit for every XOR gate.

The *free XOR* technique works by changing how some of the garbled values for wires in the circuit are selected.

[5]This name for the technique comes from[11], though versions of it are in work at least as early as[17].

[6]The latter approach was used because it lends itself better as an addition to the standard protocol, while the [17] description requires working into the initial construction of the protocol.

Recall that by default each garbled value of 0 and 1 for each wire in the circuit is selected randomly. The *free XOR* technique instead relates the values of the input wires to XOR gates so that the gate's correct output values can be computed with a single XOR operation, instead of needing to lookup and decrypt the output value in a garble truth table. Since garbled truth tables are no longer needed for all XOR gates, the size of the garbled circuit is reduced by $|XORgates| \bullet |k|$, where k is the size of the garbled values used in the circuit. The *free XOR* technique is described more formally in algorithm 5.

Algorithm 5 Free XOR Technique

1: *P1*, the circuit constructor, generates secret $R \in \{0,1\}^k$, where k is the length of each garbled value in the circuit.
2: Let G be the set of all XOR gates in the circuit, and let g_{in_0} and g_{in_1} refer to the gates in the circuit with output leading into gate g. Finally, let $k^b_{in_i}$ refer to the $b \in \{0,1\}$ value of wire leaving g_{in_i} and entering g.
3: **for** $g \in G$ **do**
4: Set $k^1_{in_0} = R \oplus k^0_{in_0}$ and $k^1_{in_1} = R \oplus k^0_{in_1}$.
5: Replace g with a function returning $k_{in_0} \oplus k_{in_1}$.
6: **end for**

Note that the description of the *free XOR* technique in figure 5 is not immediately compatible with the previously discussed *fast table lookups* technique. Such a construction is possible, though slightly more involved. It is provided in [14], but omitted here to avoid complicating the description of the *free XOR* technique.

6.3.3 Garbled Row Reduction

Pinkas et al.[22] developed a technique to further reduce the number of garbled values that need to be stored in the garbled circuit. Their technique, called *garbled row reduction*, removes the need to store one garbled value from each AND and OR gate in the circuit, and does so by building on the *fast table lookups* technique.

Garbled row reduction works by special casing one of the four possible indexes into a gate's garbled truth table. Pinkas et al. select $(0,0)$ for this special case, but that decision is arbitrary. For this special case, the garbled output value for the wire is defined to be a function of the input wire values, instead of a new, pseudo-random value. The circuit evaluator can then receive the output wire's value in this special case by performing some computation equivalent to $k^b_{out} \leftarrow H(k_{in_0}, k_{in_1}, g)$ (where g is a unique identifier for the gate) and assigning an index bit of 0. The value of b would depend on the type of gate (AND or OR) and the boolean values represented behind the input wire values that carried the $(0,0)$ index.

The circuit constructor would then need to correctly populating the rest gates garbled truth table, using the above value for any other rows where the output wire should carry value b, and using $k^b_{out} \oplus R$ in the other case (to maintain compatibility with the *free XOR* approach from the previous section).

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

w_2	w_5	w_6	garbled value
k^0_2	k^0_5	k^0_6	$H(k^0_2 \|\| k^0_5 \|\| g_3) \oplus k^0_6$
k^0_2	k^1_5	k^0_6	$H(k^0_2 \|\| k^1_5 \|\| g_3) \oplus k^0_6$
k^1_2	k^0_5	k^0_6	$H(k^1_2 \|\| k^0_5 \|\| g_3) \oplus k^0_6$
k^1_2	k^1_5	k^1_6	$H(k^1_2 \|\| k^1_5 \|\| g_3) \oplus k^1_6$

7. IMPLEMENTATIONS

This section briefly describes three significant implementations of Yao's protocol and some of their relevant security and performance characteristics. Other significant and important implementations of the protocol are noted[21, 24], but because of time and space constraints are not discussed further.

7.1 Fairplay

Fairplay[17] was developed in 2004 and was one of the first attempts to create a practical system to execute Yao's protocol. The system included a high level language for describing functions, which the system would transform into circuit representations suitable for garbling.

The system provided some security against *malicious* adversaries. For example, a simple cut-and-choose system, similar to the one discussed in section 5.2.2 was included to provide *P2* $1 - 1/m$ protection against corrupt circuits from *P1*. However, the system is not secure against other attacks from *P1*, such as the corrupt input attack discussed in section 5.3.

Fairplay was the first to implement one of the performance techniques discussed in section 6, *fast table lookups*. The system also implemented several different OT protocols and found that there were significant performance differences between them.

Fairplay was evaluated against four functions, an AND bitwise operation, the "billionaires problem", a simple key database search, and finding the median value in the combined array of the inputs of both parties. The largest computed circuit consisted of 4383 gates and the system computed these functions on the order of seconds. Interestingly, the researchers found that communication costs dominated computation costs in all functions. Given the simple nature of most of the problems, *Fairplay* could be said to have shown that Yao's protocol was closer to practical usage than most researchers probably expected, even if it was not there yet. *Fairplay* became the baseline that other implementations were judged against.

7.2 Huang, Evans, Katz, Malka

In 2011 Huang, et al.[11] presented a new practical system for carrying out Yao's protocol. Like *Fairplay*, the system used a high level language that users could write functions in, in this case Java. The system would then automatically translate these high level functions into circuit representations.

This approach focused on the *semi-honest* scenario, and thus made the trade off of less security for faster performance (the paper explicitly discusses the reasoning behind this decision). The system included the performance optimizations in *Fairplay*, but added many others, including free XORs (discussed in section 6.3.2), garbled gate reduction (discussed in section 6.3.3) and pipelined execution (discussed in section 6.2.2).

The system was compared against previously best known privacy preserving methods for computing Hamming distance,

Levenshtein distance, Smith-Waterman genome alignment and AES, and found an order of magnitude improvement in their system (though it is not discussed if the compared to approaches are secure in the *malicious* case or similarly just in the *semi-honest* case). The largest circuit computed consisted of over 1 billion gates, for the Levenshtein distance problem.

7.3 Kreuter, shelat, Shen

Most recently, Kreuter et al.[14] presented a system for carrying out Yao's protocol in the *malicious* case. Similar to the previously discussed methods, this system includes a way of converting from a high level representation of a function into a boolean circuit. Here the high level language was created by the authors. The compiler is also specifically compared to the *Fairplay* complier. The authors find that their boolean circuit compiler is able to generate dramatically larger circuits using fewer computational resources than previous efforts.

This work represents the state of the art in securing Yao's protocol in the *malicious* case. While the work is not quite able to claim full security against *malicious* adversaries (because of the open problem of *P2* being able to abort early and not reveal the circuit's output), it appears to be as close as possible given known methods. The work includes all security techniques discussed in section 5, as well as some other strategies that are beyond the scope of this paper.

Similarly, this work also includes all discussed performance optimizations discussed in section 6. It also incorporates strategies that are not discussed in this paper because they are either hardware dependent (the system uses hardware optimizations in the *Intel Advanced Encryption Standard Standard Instructions* provided on some Intel and AMD processors) or highly heuristic based and possibly not generally applicable (see the system's method for limiting the working set of a garbled circuit).

The system was evaluated on AES, RSA signing, dot product computation, and edit distance of large strings. The evaluations show that the random seed checking (section 6.1.1), garbled row reduction (section 6.3.3) and free XOR (section 6.3.2) optimizations provide the greatest improvement. The largest circuit computed in the system was in finding the edit distance between two 4095 bit strings, which required over 5.9 billion gates and 8.2 hours. This is compared to the Huang et al. approach to the same problem, which computed the same edit distance computation more quickly, but in the *semi-honest* setting.

8. CONCLUSION

This paper attempts to provide an overview of the field of SFE using Yao's garble circuits protocol. In addition to the techniques and approaches discussed in this paper, there is a great deal of related work in other fields that might be of interest for those interested in practical SFE, such as zero-knowledge proof systems, performance optimizing OT constructions, malleable claw-free collections, and verifiable secret sharing. Similarly, other practical-oriented implementations of Yao's protocol, such as the LEGO system[21], may also be of interest.

References

[1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in Cryptology-CRYPTO'89 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. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2009.

[7] 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.

[8] 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.

[9] S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *Advances in CryptologyâĂŤCRYPTOâĂŹ96*, pages 201–215. Springer, 1996.

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

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

[12] 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.

[13] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming*, pages 486–498. Springer, 2008.

[14] 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.

[15] 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.

[16] 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.

[17] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. San Diego, CA, USA, 2004.

[18] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography-PKC 2006*, pages 458–473. Springer, 2006.

[19] 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.

[20] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, 2005.

[21] J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *Theory of Cryptography*, pages 368–386. Springer, 2009.

[22] 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.

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

[24] C.-h. Shen et al. Two-output secure computation with malicious adversaries. In *Advances in Cryptology–EUROCRYPT 2011*, pages 386–405. Springer, 2011.

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

[26] 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.