Towards Billion-Gate Secure Computation with Malicious Adversaries

Benjamin Kreuter *abhi shelat* *Chih-hao Shen*
University of Virginia
$\{$*brk7bx, abhi, shench*$\}$*@virginia.edu*

Abstract

The goal of this paper is to assess the feasibility of two-party secure computation in the presence of a malicious adversary. Prior work has shown the feasibility of billion-gate circuits in the semi-honest model, but only the 35k-gate AES circuit in the malicious model, in part because security in the malicious model is much harder to achieve. We show that by incorporating the best known techniques and parallelizing almost all steps of the resulting protocol, evaluating billion-gate circuits is feasible in the malicious model. Our results are in the standard model (i.e., no common reference strings or PKIs) and, in contrast to prior work, we do not use the random oracle model which has well-established theoretical shortcomings.

1 Introduction

Protocols for secure computation allow two or more mutually distrustful parties to collaborate and compute some function on each other's inputs, with privacy and correctness guarantees. Andrew Yao showed that two-party secure protocols can be constructed for any computable function, and laid out a framework for general purpose secure computation systems [32]. Yao's protocol involves representing the function as a boolean circuit, and having one party (called the *generator*) encrypt the circuit in such a way that it can be selectively decrypted by the other party (called the *evaluator*) to compute the output, a process called "garbling." Oblivious transfer is used by the evaluator to obtain a subset of the decryption keys that can be used to compute the output of the function for the evaluator's input. The first implementation of Yao's protocol, the Fairplay framework [25], involves describing functions using a high-level language, which is compiled into a circuit description that can be used by the generator for the garbling process.

Yao's protocol is of great practical significance. In many real-world situations, the inputs to a function may be too valuable or sensitive to share. Huang et al. explored the use of secure computation for biometric identification [14] in national security applications, in which it is desirable for individual genetic data to be kept private but still checked against a classified list. In a similar security application, Osadchy et al. described how facial recognition could be performed in a privacy-preserving manner [28]. The more general case of multiparty computation has already seen real-world use in computing market clearing prices in Denmark [2].

Yao's original protocol ensures that the privacy of each party's input and the correctness of the output will be preserved if both parties follow the protocol honestly. This assumption is often made in other protocols, such oblivious transfer or protocols based on homomorphic encryption, and has been the basis for a number of scalable secure computation systems [4, 10, 12, 17, 22, 28, 31]. It is conceivable, however, that one of the parties may deviate from the protocol in an attempt to violate security or correctness. Participants in an auction may attempt to manipulate the output in their favor; spies may attempt to obtain sensitive information; a computer being used for secure computation may be infected with malware. Securing against *malicious* participants in an efficient, practical manner is more technically challenging.

Implementations of practical systems with security against active, malicious adversaries have been presented by others. Nielsen et. al. presented the LEGO+ system [27], which uses an approach based on oblivious transfer to achieve efficient 2-party computation in the malicious model. Lindell and Pinkas presented an approach based on garbled circuits that uses the cut-and-choose technique [23], with an implementation of this system having been given by Pinkas et. al. [29]. The protocol compiler presented by Ishai, Prabhakaran, and Sahai [16] also uses an approach based on oblivious transfer, and was implemented by Lindell, Oxman, and Pinkas [21]. In all these cases, AES was used as a benchmark for performance tests. These protocols are proved secure in the *random oracle model*.

Protocols for general multiparty computation with security against a malicious *majority* have also been presented. Canetti et. al. gave a construction of a *universally composable* protocol in the *common reference string* model [5]. The protocol compiler of Ishai et. al., mentioned above, can be used to construct a multiparty protocol with security against a dishonest majority in the UC model [16]. Bendlin et. al. showed a construction based on homomorphic encryption [1], which was improved upon by Damgård et. al. [7]; these protocols were also proved secure in the UC model, and require additional setup assumptions. The protocol of Damgård et. al. is based on a preprocessing model, which improves the amortized performance of the protocol. Damgård et. al. presented an implementation of their protocol, which could evaluate the function $(x \boxtimes y) \boxplus z$ in about 3 seconds with a 128 bit security level, but with an amortized time of a few milliseconds.

This paper presents a scalable two-party secure computation system which guarantees privacy and correctness in the presence of a malicious party. The system we present can handle circuits with hundreds of millions or even billions of gates, while requiring relatively modest computing resources. Our system follows the Fairplay framework, allowing general purpose secure computation starting from a high level description of a function. We present a system with numerous technical advantages over the Fairplay system, both in our compiler and in the secure computation protocol. Unlike previous work, we do not rely solely on AES circuits as our benchmark; our goal is to evaluate circuits that are orders of magnitude larger than AES in the malicious model, and we use AES only as a comparison with other work. We prove the security of our protocol assuming *circular 2-correlation robust* hash functions and the hardness of the elliptic curve discrete logarithm problem, and require neither additional setup assumptions nor the random oracle model.

2 Contributions

Our principal contribution is to build a high performance secure two-party computation system that integrates state-of-the-art techniques for dealing with *malicious* adversaries efficiently. Although some of these techniques have been reported individually, we are not aware of any attempt to incorporate them all into one system, while ensuring that a security proof can still be written for that system. Even though some of the techniques are claimed to be compatible, it is not until everything is put together and someone has gone through all the details can a system as a whole be said to be provably secure.

We start by using Yao's garbled circuit [32] protocol for securely computing functions in the presence of semi-honest adversaries, and Shelat and Shen's cut-and-choose-based transformation [30] that converts Yao's garbled circuit protocol into one that is secure against malicious adversaries.

We then modify the above to use Ishai et al.'s oblivious transfer extension [15] that has efficient amortized computation time for oblivious transfers secure against malicious adversaries, and Lindell and Pinkas' random combination technique [23] that defends against selective failure attacks. We implement Kiraz's randomized circuit technique [18] that guarantees that the generator either gets no output or an authentic output (in other words, the evaluator cannot trick the generator into accepting a random output).

For evaluating garbled circuits, we incorporate Kolesnikov and Schneider's free-XOR technique [20] that minimizes the computation and communication cost for XOR gates in a circuit. To reduce communication costs, we adopt Pinkas et al.'s garbled-row-reduction technique [29] that reduces the communication cost for k-fan-in non-XOR gates by $1/2^k$, which means at least a 25% communication saving in our system since we only have gates of 1-fan-in or 2-fan-in. Finally, we implement Goyal et al.'s technique [11] for reducing communication as follows: during the cut-and-choose step, the check-circuits are given to the evaluator by revealing the random seeds used to produce them rather than the check-circuits themselves. Thus, along with the 60%-40% check-evaluation ratio proposed by Shelat and Shen [30] this technique provides a near 60% saving in communication. As far as we know, although these techniques exist individually, ours is the first system to incorporate all of these mutually-consistent state of the art techniques.

The most important new technique that we use is to exploit the embarrassingly parallel nature of the Shelat and Shen protocol for achieving security in the malicious model, in particular, the circuit-level parallelism. We parallelize all computation-intensive operations such as oblivious transfers or circuit construction by splitting the generator-evaluator pair into hundreds of slave pairs. Each of the pairs works on one copy of the circuit in a parallel but synchronized manner as synchronization is required by the Shelat and Shen's transformation [30].

For the running time of a secure computation, there are two main contributing factors: the input processing time I (due to oblivious transfers) and the circuit processing time C (due to garbled circuit construction and evaluation). In the semi-honest model, the system's running time is simply $I + C$. Security in the malicious model, however, requires several extra checks. In the first instantiation of our system, through heavy use of circuit-level parallelism, our system needs at most $I + 2C$ to compute hundreds of copies of circuit. In other words, when the circuit size is sufficiently larger than the input size, our system (secure in the malicious model) needs roughly twice as much computation time as that needed by the original Yao protocol (secure in the semi-honest model). This is a tremendous improvement over prior work [29, 30] which needed 100x more time than the semi-honest Yao. In a second instantiation of our scheme, we are able to achieve $I+C$ computation time, albeit at the cost of moderately more communication.

To get a sense of our improvements, we list the experimental results of the benchmark function—AES-128—from the most recent literature and our system. The latest reported system in the semi-honest model is built by Huang et al. [13] and needs 1.3 seconds (where $I = 1.1$ and $C = 0.2$) to complete a block of secure AES-128 computation. The fastest known system in the malicious model is proposed by Nielson et al. [27] and has an amortized performance 1.6 seconds per block (or more precisely, $I = 79$ and $C = 6$ for 54 blocks). Our system provides security in the malicious model and needs 1.1 ($= I+2C$, where $I = 0.7$ and $C = 0.2$) seconds per block. It is worth mentioning that both the prior systems require the full power of a random oracle, while ours requires a weaker cryptographic primitive called 2-circular correlation robust function, which was recently proven by Choi et al. [6] to be sufficient for the free-XOR technique's security.

Challenges of Parallelism Integrating existing techniques with circuit-level parallelism, however, requires careful engineering in order to achieve good performance while maintaining security. In Fairplay [25], a garbled circuit is fully constructed before being sent over a network for the other party to evaluate. Huang et al. [13] pointed out that keeping the whole garbled circuit in memory is unnecessary. The generation and evaluation of garbled gates could be conducted in a pipeline manner. Consequently, not only do both parties spend less time idling, but merely a constant amount of garbled gates need to reside in memory at one time, even when dealing with large circuits. However, this pipelining idea does not work trivially with other optimization techniques for the following two reasons:

- The cut-and-choose technique requires the generator to finish constructing circuits before the coin flipping (which is used to determine check-circuits and evaluation-circuits), but the evaluator cannot start checking or evaluating before the coin flipping. A naive approach would ask the evaluator to hold the circuits and wait for the results of the coin flipping before she proceeds to do her jobs. When the circuit is of large size, keeping hundreds of copies of such a circuit in memory is undesirable.

- Similarly, the random seed checking technique [11] requires the generator to send the hash for each garbled circuit, and later on send the random seeds for check-circuits so that the communication for check-circuits is vastly reduced. Note that the hash for an evaluation-circuit is given away before the garbled circuit itself. However, a hash is calculated only after the whole circuit is generated. So the generation-evaluation pipelining cannot be applied directly.

Our system, however, integrates the pipelining idea and both aforementioned techniques with a price of $I+2C$ computation time. We also propose an alternative that does not enjoy the benefits of the random seed checking technique (thus, no 60% saving in communication) but needs only as much computation time as that in the semi-honest setting, that is, $I+C$.

Besides the improvements by the algorithmic means, the latest hardware technologies are adopted in our system too. In particular, we incorporate the Intel Advanced Encryption Standard Instructions (AES-NI) in our system. While the encryption is previously suggested to be

$$\mathrm{Enc}_{X,Y}(Z) = H(X||Y) \oplus Z$$

in the literature [6, 20], where H is a 2-circular correlation resistant function instantiated either with SHA-1 [13] or SHA-256 [29], we propose an alternative that

$$\mathrm{Enc}^{k}_{X,Y}(Z) = \text{AES-256}_{X||Y}(k) \oplus Z,$$

where k is the index of the garbled gate. With the help of the latest instruction set, an AES-256 operation could take as little as 30% of the time for SHA-256. Since this operation is heavily used in circuit operations, with the help of the hardware optimized AES instructions, we are able to reduce the circuit computation time C by at least 20%. Unfortunately, we are unable to find a grid of hundreds of nodes that all support AES-NI, and thus, we are unable to provide experimental results in the malicious setting. However, recall that for certain circuits the running time in the semi-honest setting is roughly half of that in the malicious setting. Thus, the results in the semi-honest setting are reported as an estimation to that in the malicious setting.

Scalable Circuit Compiler One of the major bottlenecks that prevents large-scale secure computation is the need for a scalable compiler that generates a circuit description from a function written in a high-level programming language. Prior tools could barely handle circuits with 50,000 gates, requiring significant computational resources to compile such circuits. While this is just enough for an AES circuit, it is not enough for the large circuits that we evaluate in this paper.

We present a scalable boolean circuit compiler that can be used to generate circuits with hundreds of millions of gates, with moderate hardware requirements. This compiler performs some simple but highly effective optimizations, and tends to favor XOR gates. The toolchain is flexible, allowing for different levels of optimizations and can be parameterized to use more memory or more CPU time when building circuits.

As a first sign that our compiler advances the state of the art, we observe that it automatically generates a smaller boolean circuit for the AES cipher than the hand-optimized circuit reported by Pinkas et al. [29]. AES plays an important role in secure computation, and oblivious AES evaluation can be used as a building block in cryptographic protocols. Not only is it one of the most popular building blocks in cryptography and real life security, it is often used as a benchmark in secure computation. With the textbook algorithm, the well-known Fairplay compiler can generate an AES circuit that has 15,316 non-XOR gates. Pinkas et al. were able to develop an optimized AES circuit that has 11,286 non-XOR gates. By applying an efficient S-box circuit [3] and using our compiler, we were able to construct an AES circuit that has 9,100 non-XOR gates. As a result, our AES circuit only needs 59% and 81% of the communication needed by the other two, respectively.

Most importantly, with our system and the scalable compiler, we are able to run experiments on circuits with sizes in the range of hundreds of millions of gates. To the best of our knowledge, secure computation with such large circuits has never been run in the malicious model before. These circuits include 256-bit RSA (266,150,119 gates) and 1023-bit edit distance (242,594,574 gates). As the circuit size grows, resource management becomes crucial. A circuit of hundreds of millions of gates can easily result in several GB of data stored in memory or sent over network. Special care is required to handle these difficulties.

The organization of this paper is as follows. A variety of security decisions and optimization techniques will be covered in Section 3 and Section 4, respectively. Then, our system, including a compiler, will be introduced in Section 5 and Section 6. Finally, the experimental results are presented in Section 7 followed by the conclusion and future work in Section 8.

3 Techniques Regarding Security

The Yao protocol, albeit efficient, assumes honest behaviors. To achieve security in the malicious model, a method to enforce honest behaviors is necessary. The *cut-and-choose* technique is one of the most efficient methods in literature and is used in our system. Its main idea is for the generator to prepare multiple copies of the garbled circuit with independent randomness, and the evaluator picks a random fraction of the received circuits, whose randomness is then revealed. If any of the chosen circuits (called *check-circuits*) is not correctly generated with the revealed randomness, the evaluator aborts; otherwise, she evaluates the remaining circuits (called *evaluation-circuits*) and takes the majority of the outputs, one from each evaluation-circuit, as the final output.

The intuition is that to pass the check, a malicious generator can only sneak in a few faulty circuits, and the influence of these (supposedly minority) faulty circuits will be eliminated by the majority operation at the end. On the other hand, if a malicious generator wants to manipulate the final output, she needs to construct faulty majority among evaluation-circuits, but then the chance that none of the faulty circuits is checked will be negligible. So with the help of the cut-and-choose method, a malicious generator either constructs many faulty circuits and gets caught with high probability, or constructs merely a few and has no influence on the final output.

However, the cut-and-choose technique is not a cure-all. Several subtle attacks have been reported and would be a problem if not properly handled. These attacks include the *generator's input inconsistency attack*, the *selective failure attack*, and the *generator's output authenticity attack*, which are discussed in the following sections. Note that in this section, n denotes the input size and s denotes the number of copies of the circuit.

Generator's Input Consistency Recall that in the cut-and-choose step, multiple copies of a circuit are constructed and then evaluated. A malicious generator is therefore capable of providing altered inputs to different evaluation-circuits. It has been shown that for some functions, there are simple ways for the generator to extract information about the evaluator's input [23]. For example, suppose both parties agree to compute the inner-product of their input, that is, $f([a_2,a_1,a_0],[b_2,b_1,b_0]) \mapsto a_2b_2 + a_1b_1 + a_0b_0$ where a_i and b_i is the generator's and evaluator's i-th input bit, respectively. Instead of providing $[a_2,a_1,a_0]$ to all evaluation-circuits, the generator could send $[1,0,0]$, $[0,1,0]$, and $[0,0,1]$ to different copies of the evaluation-circuits. After the majority operation from the cut-and-choose technique, the generator learns $\text{major}(b_2,b_1,b_0)$, the majority bit in the evaluator's input, which is not what the evaluator agreed to reveal in the first place.

There exist several approaches to deter this attack. Mohassel and Franklin [26] proposed the equality-checker that needs $O(ns^2)$ commitments to be computed and exchanged. Lindell and Pinkas [23] developed an approach that also requires $O(ns^2)$ commitments. Later, Lindell and Pinkas [24] proposed a pseudorandom synthesizer that relies on efficient zero-knowledge proofs for specific hardness assumptions and requires $O(ns)$ group operations. Shelat and Shen [30] suggested the use of malleable claw-free collections, which also uses $O(ns)$ group operations, but they showed that witness-indistinguishability suffices, which is more efficient than zero-knowledge proofs by a constant factor.

In our system, we incorporate the malleable claw-free collection approach because of its efficiency. Although the commitment-based approaches can be implemented using lightweight primitives such as collision-resistant hash functions, they incur high communication overhead for the extra complexity factor s, that is, the number of copies of the circuit. On the other hand, the group-based approach could be more computationally intensive, but this discrepancy is compensated again due to the parameter s.[1] Hence, with similar computation cost, group-based approaches enjoy lower communication overhead.

Selective Failure A more subtle attack is *selective failure* [19, 26]. A malicious generator could use inconsistent keys to construct the garbled gate and OT so that the evaluator's input can be inferred from whether or not the protocol completes. In particular, a cheating generator could assign (K_0, K_1) to an input wire in the garbled circuit while using (K_0, K_1^*) instead in the corresponding OT, where $K_1 \neq K_1^*$. As a result, if the evaluator's input is 0, she learns K_0 from OT and completes the evaluation without complaints; otherwise, she learns K_1^* and gets stuck during the evaluation. If the protocol expects the evaluator to share the result with the generator at the end, the generator learns whether or not the evaluation failed, and therefore, the evaluator's input is leaked.

Lindell and Pinkas [23] proposed the random input replacement approach that involves replacing each of the evaluator's input bits with an XOR of s additional input bits, so that whether the evaluator aborts due to a selective failure attack is almost independent (up to a bias of 2^{1-s}) of her actual input value. Both Kiraz [18] and Shelat and Shen [30] suggested a solution that exploits committing OTs so that the generator commits to her input for the OT, and the correctness of the OTs can later be checked by opening the commitments during the cut-and-choose. Lindell and Pinkas [24] also proposed a solution to this problem using cut-and-choose OT, which combines the OT and the cut-and-choose steps into one protocol to avoid this attack.

Our system is based on the random input replacement approach due to its scalability. It is a fact that the committing OT or the cut-and-choose OT does not alter the circuit while the random input replacement approach inflates the circuit by $O(sn)$ additional gates. However, it has been shown that $\max(4n, 8s)$ additional gates suffice [29]. Moreover, both the committing OT and the cut-and-choose OT require $O(ns)$ group operations, while the random input replacement approach needs only $O(s)$ group operations. Furthermore, we observe that the random input replacement approach is in fact compatible with the OT extension technique. Therefore, we were able to build our system which has the group operation complexity independent of the evaluator's input size, and as a result, our system is particularly attractive when handling a circuit with a large evaluator input.

Generator's Output Authenticity It is not uncommon that *both* the generator and evaluator receive outputs from a secure computation, that is, the goal function is $f(x,y) = (f_1, f_2)$, where the generator with input x gets output f_1, and the evaluator with input y gets f_2.[2] In this case, the security requires that *both* the input and output are hidden from the other player. In the semi-honest setting, the straightforward solution is to let the generator choose a random number c as an extra input, convert $f(x,y) = (f_1, f_2)$ into a new function $f^*((x,c),y) = (\lambda, (f_1 \oplus c, f_2))$, run the original Yao protocol for f^*, and instruct the evaluator to pass the encrypted output $f_1 \oplus c$ back to the generator, who can then retrieve her real output f_1 with the secret input c chosen in the first place. However, the situation gets complicated when either of the participants could potentially be malicious. Note that the two-output protocols we consider are not *fair* since the evaluator always learns her own output and may refuse to send the generator's output. However, they can satisfy the notion that the evaluator cannot trick the generator into accepting arbitrary output.

Lindell and Pinkas [23] proposed a solution similar to the aforementioned solution in the semi-honest setting, where the goal function is modified while everything else remains the same, with the price of adding $O(n^2)$ gates to the circuit. Kiraz [18] presented a two-party computation protocol in which a zero knowledge proof of size $O(s)$ is conducted at the end to assure the authenticity of the generator's output. Shelat and Shen's [30] signature-based solution, similar to Kiraz's, adds n additional gates to the circuit, but however, requires only a witness-indistinguishable proof of size $O(s+n)$ at the end.

[1] To give concrete numbers, with an Intel Core i5 processor and 4GB DDR3 memory, a SHA-256 operation (from OpenSSL) requires 1,746 cycles, while a group operation (160-bit elliptic curve from PBC library with preprocessing) needs 322,332 cycles. It is worth-mentioning that s is at least 256 in order to achieve security level 2^{-80}.

[2] Here f_1 and f_2 are abbreviations of $f_1(x,y)$ and $f_2(x,y)$ for simplicity.

4 Techniques Regarding Performance

Yao's garbled circuit has been studied for decades. It has been drawing a lot of attention for its simplicity, constant round complexity, and computational efficiency (since circuit evaluation only requires fast symmetric operations). The fact that it incurs high communication overhead has provoked interest that has led to the development of fruitful results.

In this section, we will first briefly present the Yao garbled circuit, and then discuss the optimization techniques that greatly reduce the communication cost while maintaining the security. These techniques include free-XOR, garbled row reduction, random seed checking, and large circuit pre-processing. In addition to these original ideas, practical concerns involving large circuits and parallelization will be addressed.

4.1 Baseline Yao's Garbled Circuit

Given a circuit consists of AND and XOR gates, the generator constructs a corresponding garbled circuit as follows: For each wire w, the generator randomly picks $w_0, w_1 \in \{0,1\}^{k-1}$, respectively, and $\pi_w \in \{0,1\}$ as a permutation bit. Let $W_0 = w_0||\pi_w$ and $W_1 = w_1||(\pi_w \oplus 1)$, which are associated with value 0 and 1 for wire w. Next, level by level, for each gate $g \in \{\text{XOR}, \text{AND}\} \subset \{f | f : \{0,1\} \times \{0,1\} \mapsto \{0,1\}\}$ that has input wire x with (X_0, X_1, π_x), input wire y with (Y_0, Y_1, π_y), and output wire z with (Z_0, Z_1, π_z). The garbled truth table for this gate has four entries:

$$GTT_g \begin{cases} \text{Enc}(X_{0\oplus\pi_x}||Y_{0\oplus\pi_y},\ Z_{g(0\oplus\pi_x, 0\oplus\pi_y)}) \\ \text{Enc}(X_{0\oplus\pi_x}||Y_{1\oplus\pi_y},\ Z_{g(0\oplus\pi_x, 1\oplus\pi_y)}) \\ \text{Enc}(X_{1\oplus\pi_x}||Y_{0\oplus\pi_y},\ Z_{g(1\oplus\pi_x, 0\oplus\pi_y)}) \\ \text{Enc}(X_{1\oplus\pi_x}||Y_{1\oplus\pi_y},\ Z_{g(1\oplus\pi_x, 1\oplus\pi_y)}) \end{cases}$$

where $\text{Enc}(k,m)$ denotes message m encrypted with key k. Note that the key is a concatenation of two messages and the message refers to the concatenation of the random key and the corresponding permutation bit hereafter. Intuitively, π_x and π_y permute the entries in GTT_g so that for $i_x, i_y \in \{0,1\}$, the $(2i_x + i_y)$-th entry represents the input pair $(i_x \oplus \pi_x, i_y \oplus \pi_y)$ for gate g, in which case output value $g(i_x \oplus \pi_x, i_y \oplus \pi_y)$ along with the corresponding permutation bit for the output wire should be retrieved. In practice, to evaluate the garbled gate GTT_g, let $X||b_x$ and $Y||b_y$ be the retrieved messages for input wire x and wire y, respectively. The evaluator will use $X||b_x||Y||b_y$ to decrypt the $(2b_x + b_y)$-th entry in GTT_g and retrieve message $Z||b_z$, which is then used to evaluate next-level gates. As a result, the introduction of the permutation bit helps to identify the correct entry, and thus, only one, rather than all, of the four entries will be decrypted.

4.2 Free-XOR

Kolesnikov and Schneider [20] proposed the free-XOR technique that aims for removing the communication cost and decreasing the computation cost for XOR gates.

The idea is, instead of randomly picking W_0 and W_1 for wire w, the generator first randomly picks a global key R, where $R = r||1$ and $r \in \{0,1\}^{k-1}$. This global key has to be hidden from the evaluator. Then for each wire w, either W_0 or W_1 is randomly chosen from $\{0,1\}^k$, and the other key is determined by $W_b = W_{1\oplus b} \oplus R$. Note that π_w remains the rightmost bit of W_0. Moreover, for an XOR gate having input wire x with $(X_0, X_0 \oplus R, \pi_x)$, input wire y with $(Y_0, Y_0 \oplus R, \pi_y)$, and output wire z, the generator lets $Z_0 = X_0 \oplus Y_0$ and $Z_1 = Z_0 \oplus R$. Observe that

$$\begin{aligned} X_0 \oplus Y_1 = X_1 \oplus Y_0 = X_0 \oplus Y_0 \oplus R = Z_0 \oplus R = Z_1 \\ X_1 \oplus Y_1 = X_0 \oplus R \oplus Y_0 \oplus R = X_0 \oplus Y_0 = Z_0. \end{aligned}$$

This means that during the circuit evaluation, XORing the messages for the two input wires of an XOR gate will directly retrieve the message for the output wire. Therefore, no garbled truth table is needed, and the price of evaluating a garbled XOR gate is reduced from a decryption operation to a bitwise XOR operation.

This technique is secure when $\text{Enc}(K,m) = H(K) \oplus m$, where $H : \{0,1\}^{2k} \mapsto \{0,1\}^k$ is a random oracle. Furthermore, Choi et. al [6] have shown that a weaker cryptographic primitive, *2-circular correlation robust functions*, suffices. Our system instantiates this primitive either with $H(X||Y) = \text{SHA-256}(X||Y)$. However, when AESNI instructions are available, our system automatically instantiates the function as $H^k(X||Y) = \text{AES-256}(X||Y, k)$, where k is the gate index.

On a machine with 2.53 GHz Intel Core i5 processor and 4GB 1067 MHz DDR3 memory, it takes 784 clock cycles to run a single SHA-256 (with OpenSSL), while it needs only 225 cycles for AES-256 (with AES-NI). To measure the benefits of AES-NI, we use two instantiations to construct various circuits, listed in Table 1, and observe a consistant 20% saving in circuit construction.[3]

4.3 Garbled Row Reduction

The GRR (Garbled Row Reduction) technique suggested by Pinkas et. al [29] is used to reduce the communication overhead for non-XOR gates. In particular, it reduces the size of the garbled truth table for 2-fan-in gates by 25%.

[3]The reason that saving 500+ cycles does not lead to more improvements is that this encryption operation is merely one of the contributing factors to generating a garbled gate. Other factors, for example, include GNU `hash_map` table insertion (∼1,200 cycles) and erase (∼600 cycles).

	size (gates)	AES-NI	SHA-256	Ratio
AES-128	31,112	0.08	0.10	0.80
EDT-255	12,183,494	33.84	42.50	0.80
Dot_{64}^{256}	29,721,019	84.79	106.73	0.80
RSA-256	266,150,889	710.71	889.63	0.79

Table 1: Garbled circuit construction time (in seconds) with different implementations (AES-NI vs SHA-256). The circuits are for AES-128 encryption, 255-bit edit distance computation, 256-dimensional dot-product over 64-bit field, and 256-bit RSA encryption.

Recall that in the baseline Yao's garbled circuit, both the 0-key and 1-key for each wire are randomly chosen. After the free-XOR technique is integrated, the 0-key and 1-key for an XOR gate's output wire depend on input key and R, but the 0-key for a non-XOR gate's output wire is still free. The GRR technique is to make a smart choice for this degree of freedom, and thus, reduce one entry in the garbled truth table to be communicated over network.

In particular, the generator picks (Z_0, Z_1, π_z) by letting $Z_{g(0\oplus\pi_x, 0\oplus\pi_y)} = H(X_{0\oplus\pi_x}||Y_{0\oplus\pi_y})$, that is, either Z_0 or Z_1 is assigned to the encryption mask for the 0-th entry of the GTT_g, and the other one is computed by the equation $Z_b = Z_{1\oplus b} \oplus R$. Therefore, when the evaluator gets $(X_{0\oplus\pi_x}, Y_{0\oplus\pi_y})$, both $X_{0\oplus\pi_x}$ and $Y_{0\oplus\pi_y}$ have rightmost bit 0, indicating that the 0-th entry needs to be decrypted. However, with GRR technique, she is able to retrieve $Z_{g(0\oplus\pi_x, 0\oplus\pi_y)}$ by running $H(\cdot)$ without inquiring GTT_g.

Pinkas et al. claimed that this technique is compatible with the free-XOR technique [29]. For rigorousness purposes, we carefully went through the details and came up with a security proof for our protocol that confirms this compatibility. The proof will be included in the full version of this paper.

4.4 Random Seed Checking

Recall that the cut-and-choose approach requires the generator to construct multiple copies of the garbled circuit, and more than half of these garbled circuits will be fully revealed, including the randomness used to construct the circuit. Goyal, Mohassel, and Smith [11] therefore pointed out an insight that the evaluator could examine the correctness of those check-circuits by receiving a hash of the garbled circuit first, acquiring the random seed, and reconstructing the circuit and hash by herself.

This technique results in the communication overhead for check-circuits independent of the circuit size. This technique has two phases that straddle the coin-flipping protocol. Before the coin flipping, the generator constructs multiple copies of the circuit as instructed by the cut-and-choose procedure. Then the generator sends to the evaluator the hash of each garbled circuit, rather than the circuit itself. After the coin flipping, when the evaluation-circuits and the check-circuits are determined, the generator sends to the evaluator the full description of the evaluation-circuits and the random seed for the check-circuits. The evaluator then computes the evaluation-circuits and tests the check-circuits by reconstructing the circuit and comparing its hash with the one received earlier. As a result, even for large circuits, the communication cost for each check-circuit is simply a hash value plus the random seed. Our system provides that 60% of the garbled circuits are check-circuits. As a result, this optimization significantly reduces communication overhead.

4.5 Working with Large Circuits

A circuit for a reasonably complicated operation can easily consist of hundreds of millions of gates. For example, a 1023-bit edit distance has 242 million gates. When circuits grow to such a size, the task of achieving high performance secure computation becomes challenging.

An $(I + 2C)$-time solution Our solution for handling large circuits is based on Huang et. al's work [13], which is the only prior work capable of handling large circuits (of up to 1.2 billion non-XOR gates) in the semi-honest setting. Intuitively, the generator could work with the evaluator in a pipeline manner so that small chunks of gates are being processed at a time. The generator could start to work on the next chunk while the evaluator is still processing the current one. However, this technique does not work directly with the random seed checking technique described above in §4.4 because the generator has to finish circuit construction and hash calculation before the coin flipping, but the evaluator could start the evaluation only after the coin flipping. As a result, the generator needs a way to construct the circuit first, wait for the coin flipping, and send the evaluation-circuits to the evaluator without keeping them in memory the whole time. We therefore propose that the generator constructs the evaluation-circuits all over again after the coin flipping, with the same random seed used before and the same keys for input wires gotten from OT.

We stress that when fully parallelized, the second construction of an evaluation-circuit does not incur overhead to the overall execution time. Although we suggest to construct an evaluation-circuit twice, the fact is that according to the random seed checking, a check-circuit is already being constructed twice—once before the coin flipping by the generator for hash computation and once after by the evaluator for correctness verification. As a result, when each generator-evaluator pair is working on a single copy of the garbled circuit, the constructing time for a evaluation-circuit totally overlaps with that for a check-circuit. We therefore achieve the overall computation time $I+2C$ mentioned earlier, where the first C is for the generator to calculate the circuit hash, and the other C is either for the evaluator to reconstruct a check-circuit or for both parties to work on an evaluation-circuit in a pipeline manner as suggested by Huang et. al [13].

Achieving an $(I+C)$-time solution We observe that there is a way to achieve $I+C$ computation time, which exactly matches the running time of Yao in the semi-honest setting. This idea, however, is not compatible with the random-seed technique, and therefore incurs a tradeoff with communication size. Recall that the generator has to finish circuit construction and hash evaluation before the coin flipping, whereas the evaluator could start evaluating only after getting the result of the coin flipping. The idea is to run the coin flipping in the way that only the evaluator gets the result and does not reveal it to the generator until the circuit construction is completed. Since the generator is oblivious to the coin flipping result, she sends every garbled circuit to the evaluator, who could then either evaluate or check the received circuit. Moreover, in order for the evaluator to get the generator's input keys for evaluation-circuits and the random seed for the check-circuits, they run an OT, where the evaluator uses the coin flipping result as input and the generator provides either the random seed (for the check-circuit) or his input keys (for the evaluation-circuit). After the generator completes circuit construction and gives away the circuit hash, the evaluator compares the hash with her own calculation, if the hashes match, she proceeds with the rest of the original protocol. Note that this approach comes with a price that it does not enjoy the benefits provided by the random seed checking technique, that is, the 60% saving in communication.

Working Set Optimization Another problem encountered while dealing with large circuits is the *working set minimization problem*. Note that the *circuit value problem* is log-space complete for P. It is suspected that $L \neq P$, that is, there exist some circuits that can be evaluated in polynomial time but require more than logarithmic space. This open problem captures the difficulty of handling large circuits during both the construction and evaluation, where at any moment there is a set of wires, called *working set*, that are available and will be referenced in the future. For some circuits, the working set is inherently super-logarithmic. A naive approach is to keep the most recent D wires in the working set, where D is the upper bound of the input-output distance of all gates. However, there may be wires which are used as inputs to gates throughout the entire circuit, and so this technique could easily result in adding almost the whole circuit to the working set, which is especially problematic when there are hundreds of copies of circuit of billions of gates. While reordering the circuit or adding identity gates to minimize D would mitigate this problem, doing so while maintaining the topological order of the circuit is known to be an NP-complete problem, the *graph bandwidth problem* [9].

Our solution to this difficulty is to pre-process the circuit so that each gate comes with a usage count. Our system has a compiler that converts a program in high-level language into a boolean circuit. Since the compiler is already using global optimization in order to reduce the circuit size, it is easy for the global optimizer to analyze the circuit and calculate the usage count for each gate. With this information, it is easy for the generator and evaluator to decrement the counter for each gate whenever it is being referenced and to toss away the gate whenever its counter becomes zero. In Table 2, we show that with the help of the usage count, it could still be memory efficient to deal with even very large circuits.

	size (gates)	working set size
AES-128	31,112	323
EDT-255	12,183,494	2,829
Dot_{64}^{256}	29,721,019	32,968
RSA-256	266,150,889	1,794

Table 2: The size of the working set for various circuits

5 System Setup

In our system, both the generator and the evaluator consist of an equal amount of processes, including a root process and many slave processes. A root process has responsibility for coordinating its own slave processes and the other root process, while the slave processes work together on repeated and independent tasks. There are three pieces of code in our system: the generator, the evaluator, and the IP exchange server. Both the generator's and evaluator's program are implemented with Message Passing Interface (MPI) library. The reason for the IP exchanger server is that it is normally easy to run jobs on a cluster with dynamic working node assignment. However, when the nodes are dynamically assigned, the generator running on one cluster might have a hard finding the evaluator running on another. Therefore, a fixed location IP exchanger helps the matching up process as shown in Figure 1. The process starts with the root evaluator process collecting addresses of its slaves and forwarding them to a publicly known IP server. Then the root generator process will come to acquire the addresses and dispatch them to its slaves, who then proceed to pair up with one of the slave evaluator processes. However, we glue the whole system into one MPI program due to the ease of simulation purposes.

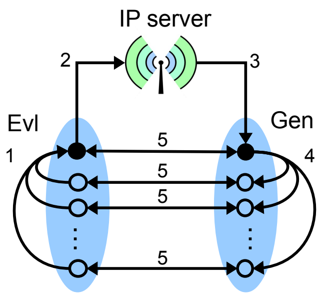

Figure 1: The process of building network connections between two parties, the generator and the evaluator, each of which consists of a root process (solid dot) and slave processes (hollow dots).

6 Boolean Circuit Compiler

Although the Fairplay circuit compiler can generate circuits, it requires a very large amount of computational resources to generate even relatively small circuits. Even on a machine with 48 gigabytes of RAM, Fairplay terminates with an out-of-memory error after spending 20 minutes attempting to compile an AES circuit. This makes Fairplay impractical for even relatively small circuits, and infeasible for some of the circuits tested in this project. One goal of this project was to have a general purpose system for secure computation, and so writing application specific programs to generate circuits, a technique used by others [13], was not an option.

To address this problem, we have implemented a new compiler that generates the same output format as Fairplay, but which requires far lower computational resources to do so. We were able to generate the AES circuit on a typical laptop computer with only 1GB of RAM, and were able to generate and test much larger non-trivial circuits. We used the well-known *flex* and *bison* tools to generate our compiler, and implemented an optimizer as a separate tool. We also use the results from [29] to reduce 3 arity gates to 2 arity gates.

As a design decision, we created an imperative, untyped language with static scoping. We allow code, variables, and input/output statements to exist in the global scope; this allows very simple programs to be written without too much extra syntax. Functions may be declared, but may not be recursive. Variables do not need to be declared before being used in an unconditional assignment; variables assigned within a function's body that are not declared in the global scope are considered to be local. Arrays are a language feature, but array indices must be constants or must be determined at compile time. Variables may be arbitrarily concatenated, and bits or groups of bits may be selected from any variable.

We use some techniques from the Fairplay compiler in our own compiler. In particular we use the single assignment algorithm from Fairplay, which is required to deal with assignments that occur inside of *if* statements. Otherwise, our compiler has several distinguishing characteristics that make it more resource efficient than Fairplay. The front end of our compiler attempts to generate circuits as quickly as possible, using as little memory as possible and performing only rudimentary optimizations before emitting its output. This can be done with very modest computational resources, and the intermediate output can easily be translated into a circuit for evaluation. The main optimizations are performed by the back end of the compiler, which identifies gates that can be removed.

Unlike the Fairplay compiler, we avoided the use of hash tables in our compiler, and used Red-Black trees instead. Although this multiplies the time complexity by a logarithmic factor, it allows far more data to be stored in-core, which greatly improves the scalability of our system. For circuits that are too large to fit in core, we use *Berkeley DB* as a key-value store.

In the following sections, we describe these contributions in more detail, and provide experimental results.

6.1 Circuit Optimizations

The front-end of our compiler tends to generate inefficient circuits, with large numbers of removable gates. As an example, for some operations the compiler generates large numbers of identity gates i.e. gates whose outputs follow one of their inputs. It is therefore essential to optimize the circuits emitted by the front end, particularly to meet our system's overall goal of practicality.

Our compiler uses several stages of optimization, most of which are global. As a first step, a local optimization removes redundant gates, i.e. gates that have the same truth table and input wires. This first step operates on a fixed-size chunk of the circuit, but we have found that there are diminishing improvements as the size of this window is increased, as shown in figure 3. We also remove constant gates, identity gates, and inverters, which are generated by the compiler and which may be inadvertently generated during the optimization process. Finally, we remove gates that do not influence the output, which can be thought of as dead code elimination. The effectiveness of each optimization on different circuits is shown in figure 2. The circuit that was least optimizable was the edit distance circuit, being reduced to only 82% of its size from the front end, whereas the RSA circuit was the most optimizable, being reduced to 16% of its original size.

RSA Size	Circuit Size	Time to Compile (s)	**Gates/sec**	Edit-Dist Size	Circuit Size	Time to Compile	**Gates/sec**
16	31,539	1.71	**18,444**	15	21,081	0.403	**52,310**
32	257,139	15.51	**16,579**	31	113,733	1.703	**66,784**
64	2,076,960	141.05	**14,725**	127	2,707,308	48.574	**55,736**
96	7,032,181	595.56	**11,808**	255	12,182,984	260.22	**46,818**
128	16,696,353	1416.25	**11,789**	511	54,743,160	1398.00	**39,158**

Table 3: Time required to compile and optimize RSA and edit distance circuits on a workstation with an Intel Xeon 5506 CPU and 8GB of RAM, using the textbook modular exponentiation algorithm. Note that the throughput for edit distance is much higher even for comparable sized circuits; this is because the front end generates a more efficient circuit without any optimization.

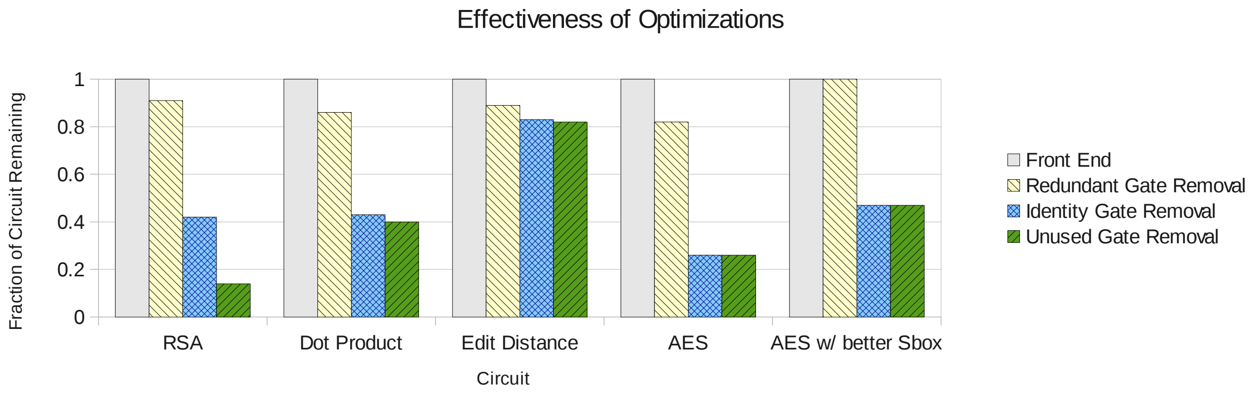

Figure 2: Average fraction of circuits remaining after each optimization is applied in sequence. We see that the *relative change* in circuit sizes after each optimization is dependent on the circuit itself, with some circuits being optimized more than others.

Gate Removal The front-end of the compiler emits gates in topological order, and similar to Fairplay, our compiler assigns explicit identifiers to each emitted gate. To remove gates efficiently, we store a table that maps the identifiers of gates that were removed to the previously emitted gates, and for each gate that is scanned the inputs are rewritten according to this table. The table itself is then emitted, so that the identifiers of non-removed gates can be corrected. This allows the optimization stages to be $\Omega(n \lg n)$, but requires $\Omega(n)$ space (with a relatively small constant factor).

Removing Redundant Gates Some of the gates generated by the front end of our compiler have the same truth table and input wires as previously generated gates; such gates are redundant and can be removed. This removal process has the highest memory requirement of any other optimization step, since a description of every non-redundant gate must be stored. However, we found during our experiments that this optimization can be performed on discrete chunks of the circuit with results that are very close to performing the optimization on the full circuit, and that there are diminishing improvements in effectiveness as the size of the chunks is increased. Therefore, we perform this optimization using chunks, and can use hash tables to improve the speed of this step.

Removing Identity Gates and Inverters The front end may generate identity gates or inverters, which are not necessary. This may happen inadvertently, such as when a variable is incremeneted by a constant, or as part of the generation of a particular logic expression. While removing identity gates is straightforward, the removal of inverters requires more work, as gates which have inverted input wires must have their truth tables rewritten. There is a cascading effect in this process; the removal of some identity gates or inverters may transform later gates into identity gates or inverters. This step also removes gates with constant outputs, such as an XOR gate with two identical inputs. Constant propagation and folding occur as a side effect of this optimization.

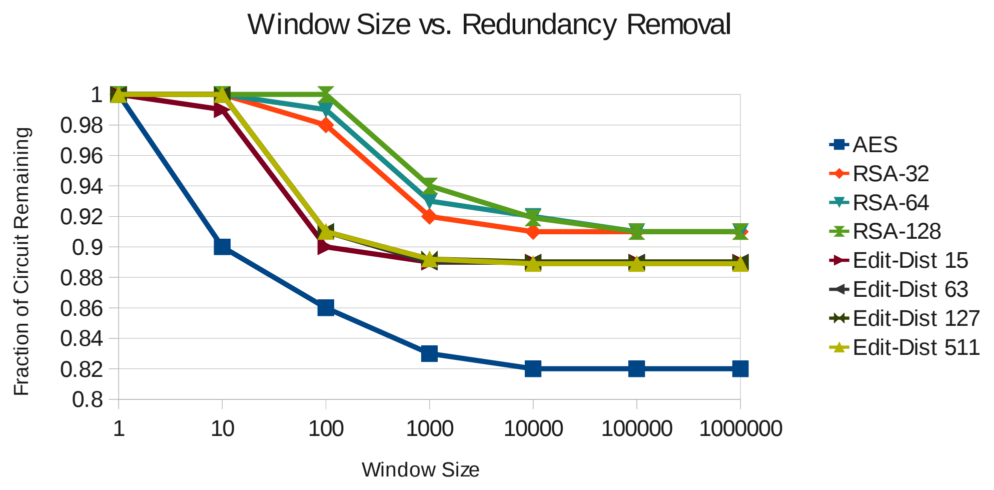

Figure 3: Size of the window used in redundant gate removal versus fraction of the circuit remaining after the optimization. This levels off, and we found that for the circuits we worked with a window of size 1,000,000 was sufficient.

Removing Unused Gates Finally, some gates in the circuit may not affect the output value at all. For this step, we scan the circuit backwards, and store a table of live gates; we then re-emit the live gates in the circuit and skip the dead gates. Immediately following this step, the circuit is prepared for the garbled circuit generator, which includes generating a usage count for each gate.

Using Berkeley DB as a Key-Value Store Unfortunately, even though our compiler is more resource efficient than Fairplay, it still requires space that is linear in the size of the circuit. For very large circuits, circuits with billions of gates or more, this may exceed the amount of RAM that is available. Our system can make use of *Berkeley DB* to store the tables, which can store tables both on disc and in RAM. Although it is possible to use Berkeley DB to manage an in-memory database, we found that this was still slower than using red-black trees (see table 4). However, by using Berkeley DB, we are able to compile very large circuits like RSA-256 (266,150,119 gates) or edit distance 1023 (242,594,574 gates). While this allows such large circuits to be compiled on inexpensive hardware, it should be noted that in both the RSA and edit distance cases more than 24 hours of computer time was required to complete the circuits, much of which was spent waiting on IO.

6.2 Compiler Testing Methodology

We tested the performance of our compiler using five circuits. The first was AES, to compare our compiler with the Fairplay system. We also used AES with an improved S-box, which results in a smaller AES circuit. We used RSA with various key sizes to test our compiler's handling of large circuits; RSA circuits have cubic size complexity, allowing us to generate very large circuits with small inputs. We also used an edit distance circuit, which was chosen because its inner loop is very different from our other tests, having no multiplication subroutine. Finally we used a dot product with error, a basic sampling function for the LWE problem, which is similar to RSA in creating large circuits, but also demonstrates our system's ability to handle large input sizes.

Circuit	Size	R-B Trees	Berkeley DB
AES	49,912	1.294s	5.025s
RSA-16	31,571	1.825s	6.594s
Dot_4^{64}	460,018	10.675s	36.464s

Table 4: Compile times for in-memory Berkeley DB versus Red-Black trees for small circuits (sizes include input gates).

6.3 Summary of Compiler Performance

Our compiler is able to emit and optimize large circuits in relatively short periods of time, less than an hour for circuits with tens of millions of gates on an inexpensive workstation. In figure 2 we summarize the effectiveness of the various optimization stages on different circuits; in circuits that involve multiplication in finite fields or modulo an integer, the identity gate removal step is the most important, removing more than half of the gates emitted by the front-end. The edit distance circuit is the best-case for our front end, as less than $1/5$ of the gates that are emitted can be removed by the optimizer. The throughput of our compiler is dependent on the circuit being compiled, with circuits which are more efficiently generated by the front-end being compiled faster; in table 3 we compare the generation of RSA circuits to edit distance circuits. The main bottleneck in the compiler is the key-value store, which leads to $O(n \lg n)$ performance.

7 Experimental Results

In this section, we implement various real world applications with secure computation. The experiment environment is the Ranger cluster in Texas Advanced Computing Center. Ranger is a blade-based system, where each node is a SunBlade x6240 blade running Linux kernel and has four AMD Opteron quad-core 64-bit processors, as an SMP unit. Each node in the Ranger system has 2.3 GHz core frequency and 32 GB of memory, and the point-to-point bandwidth is 1 GB/sec.

Timing methodology When there is more than one process on each side, care must be taken in measuring the timings of the system. The numbers reported in Table 7 is the time required by the root process at each stage of the system. This was chosen because the root process will always be the *longest* running process, as it must wait for each slave process to run to completion. Moreover, in addition to doing all the work that the slaves do, the root processes also perform the input consistency check and the coin tossing protocol.

AES We used AES as a benchmark to compare our compiler to the Fairplay compiler, and as a test circuit for our evaluator. We tested the full AES circuit, as specified in FIPS-97 [8].

In this experiment, two parties collaboratively compute the function $f : (x,y) \mapsto (\perp, \text{AES}_x(y))$, where the circuit generator owns the encryption key x, while the evaluator has the message y to be encrypted. After the computation, the generator will not get any output, whereas the evaluator will get the eiphertext $\text{AES}_x(y)$.

In order to achieve security level of 2^{-80}, that a malicious player cannot successfully cheat with probability better than 2^{-80}, it requires at least 250 copies of the garbled circuit [30]. For the balance between computing nodes, we use 256 copies in the experiment. The results reported in Table 6 and Table 7 come from an average of 30 experiment samples.

Gate Type	non-XOR	XOR
Fairplay	15,316	35,084
Ours-A	15,300	34,228
Pinkas et al.	11,286	22,594
Ours-B	9,100	21,628

Table 5: The components of the AES circuits from different sources. Ours-A comes from the textbook AES algorithm, and Ours-B uses optimized S-box circuit. (Sizes do not include input or output wires)

		Gen (sec)	Eval (sec)	Comm (KB)	Comm (%)
OT Time	Comp.	157.8	33.8	5,516	16.0
	Comm.	0.1	124.1		
Const.	Comp.	40.2	–	3	<0.1
	Comm.	–	40.2		
Inp. Check	Comp.	–	1.7	266	0.8
	Comm.	–	–		
Eval.	Comp.	14.6	34.2	28,781	83.3
	Comm.	21.1	3.2		
Total		237.3	237.3	44,805	100.0

Table 6: The result of $(x,y) \mapsto (\perp, \text{AES}_x(y))$, where $x,y \in \{0,1\}^{128}$ with parameters $k = 80$ and $s = 256$.

In Table 6, both the computational and communicational costs for each main stage are listed under the traditional setting, where there is only one process on each side. These main stages include oblivious transfer, garbled circuit construction, the generator's input consistency check, and the circuit evaluation. Each row includes both the computation and communication time used. Note that network conditions could vary from place to place. Our experiments run in a local area network, and the data can merely give a rough idea on how fast it could be in an ideal environment. However, we also report how much information must be exchanged during the process.

We see from Table 6 that the evaluator spends most of the time in communication in both the oblivious transfer and circuit construction stages, while there really is not much data needed to be transmitted. This is because the evaluator spends that time on waiting for the generator to finish computation-intensive tasks. The same reason explains that in the circuit evaluation stage the generator spends more time in communication than the evaluator. This waiting results from that the two parties need to run the protocol in a synchronized manner, which explains that the time both parties spend on each stage is about

core #	2		4		8		16		32		64		128		256	
	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl
OT	79.1	16.8	39.5	8.4	19.8	4.2	9.9	2.1	4.9	1.1	2.5	0.6	1.3	0.3	0.6	0.2
Gen.	19.6	–	9.3	–	4.5	–	2.2	–	1.1	–	0.5	–	0.3	–	0.1	–
Inp. chk	–	0.9	–	0.4	–	0.2	–	0.1	–	–	–	–	–	–	–	–
Eval.	7.1	16.2	3.2	7.6	1.7	3.5	0.8	1.7	0.4	0.8	0.3	0.4	0.2	0.2	0.1	0.1
Inter-com	11.8	83.6	5.7	41.3	2.5	20.5	1.4	10.2	0.6	5.1	0.3	2.6	0.1	1.4	0.1	0.7
Intra-com	0.5	0.5	0.2	0.3	0.2	0.2	0.1	0.2	0.1	0.1	–	0.1	–	0.1	–	0.1
Total time	118.0	118.1	58.0	58.0	28.6	28.6	14.4	14.4	7.2	7.2	3.7	3.7	1.9	1.9	1.0	1.0

Table 7: The time (in seconds) of running AES circuit with security parameter k=80 and s=256. The number of nodes represents the degree of parallelism on each side. "–" means that the time is smaller than 0.05 second and thus omitted.

the same. This synchronization is crucial since our protocol's security is guaranteed only under sequential execution. While the parallelization of the program introduces high performance execution, it does not and should not change this essential quality. Otherwise, a stronger notion of security such as universal security will be required. By using TCP sockets in "blocking" mode, we enforce this synchronization.

Note that the low communication during the circuit construction stage is due to the random seed checking technique. Also, the fact that the generator spends more time in the evaluation stage than she traditionally does comes from the second construction for evluation-circuits. Recall that only the evaluation-circuits need to be sent to the evaluator. Since only 40% of the garbled circuits (102 out of 256) are evaluation-circuits, the ratio of the generator's computation time in the generation and evaluation stage is 40.2:14.6 $\simeq$ 5:2.

Table 7 shows that the Yao protocol really benefits from the circuit-level parallelization. Starting from Table 6, where each side only has one process, all the way to when each side has 256 processes, as the degree of parallelism doubles, the total time reduces into a half. Note that the communication costs between the generator and evaluator remain the same, as shown in Table 6. It may seem odd that the communication costs also reduce as the number of processes increase. The real interpretation of this data is that as the number of processes increases, the "waiting time" decreases.

Notice that as the number of processes increases, the ratio of the time the generator spends in the construction and evaluation stage decreases from 5:2 to 1:1. The reason is that the number of garbled circuit each process handles is getting smaller and smaller. Eventually, we reach the limit of the benefits that the circuit-level parallelism could possibly bring. In this case, each process is dealing with merely a single copy of the garbled circuit, and the time spent in both the generation and evaluation stages is the time to construct a garbled circuit.

The intra-communication time in Table 7 is the communication within each side.

To the best of our knowledge, completing an execution of secure AES in malicious model within 1.1 seconds is the best result that has ever been reported. The next best result from Nielsen et al. [27] is 1.6 seconds, and it is an amortized result (85 seconds for 54 blocks of AES encryption in parallel).

RSA In this experiment, we run the 256-bit RSA encryption circuit, that is, $(x,y) \mapsto (\perp, x^y \mod m)$, where $x,y,m \in \{0,1\}^{256}$ and m is a public number. The circuit generated by our compiler has 266,150,119 gates, and 66,782,203 of which are non-XOR. Roughly, our system can handle 125,000 gates per second. Note that Huang et al.'s system is the only one, to the best of our knowledge, that is capable of handling large circuits [13], in particular, at a rate of over 96,000 (non-XOR) gates per second on an edit-distance circuit.

		Gen (sec)	Eval (sec)	Comm (MB)	Comm (%)
OT	Comp.	1.4	1.1	11	<0.1
	Comm.	–	0.2		
Gen.	Comp.	1210.7	–	<1	<0.1
	Comm.	–	1210.7		
Evl.	Comp.	718.3	542.6	204,355	99.9
	Comm.	199.2	375.5		
Total		1930.6	2130.1	204,368	100.0

Table 8: The result of $(x,y) \mapsto (\perp, x^y \mod m)$, where $x,y,m \in \{0,1\}^{256}$ and m is public parameter. The security parameters $k = 80$ and there are $s = 256$ copies of the circuit. Each party is comprised of 256 cores in a cluster.

8 Conclusion

We have presented a general purpose secure two party computation system which offers security against malicious adversaries and which can efficiently evaluate circuits with hundreds of millions of gates on affordable hardware. Our compiler can generate large circuits using fewer computational resources than similar compilers, and offers improved flexibility to users of the system. Our evaluator can take advantage of parallel computing resources, which are becoming increasingly common and affordable. As future work, we intend to extend our work to allow for the generation and evaluation of circuits with billions of gates, while maintaining security against malicious adversaries. The source code for this system can be downloaded from the authors' website.

References

[1] Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT* (2011), pp. 169–188.

[2] Bogetoft, P., Christensen, D. L., Damgård, I., Geisler, M., Jakobsen, T. P., Krøigaard, M., Nielsen, J. D., Nielsen, J. B., Nielsen, K., Pagter, J., Schwartzbach, M. I., and Toft, T. Secure multiparty computation goes live. In *Financial Cryptography* (2009), pp. 325–343.

[3] Boyar, J., and Peralta, R. A new combinational logic minimization technique with applications to cryptology. In *Experimental Algorithms*, P. Festa, Ed., vol. 6049 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 178–189.

[4] Brickell, J., and Shmatikov, V. Privacy-preserving Graph Algorithms in the Semi-honest Model. In *Asiacrypt* (2005).

[5] Canetti, R., Lindell, Y., Ostrovsky, R., and Sahai, A. Universally composable two-party and multi-party secure computation. Cryptology ePrint Archive, Report 2002/140, 2002. http://eprint.iacr.org/.

[6] Choi, S. G., Katz, J., Kumaresan, R., and Zhou, H.-S. On the Security of the "Free-XOR" Technique. Crypto ePrint Archive, 2011. http://eprint.iacr.org/2011/510.

[7] Damgard, I., Pastro, V., Smart, N., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. http://eprint.iacr.org/.

[8] FIPS. *Advanced Encryption Standard (AES)*, Nov. 2001.

[9] Garey, M. R., Graham, R. L., Johnson, D. S., and Knuth, D. E. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics 34*, 3 (1978), pp. 477–495.

[10] Gentry, C., Halevi, S., and Smart, N. P. Homomorphic evaluation of the aes circuit. Cryptology ePrint Archive, Report 2012/099, 2012. http://eprint.iacr.org/.

[11] Goyal, V., Mohassel, P., and Smith, A. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology* (Berlin, Heidelberg, 2008), EUROCRYPT'08, Springer-Verlag, pp. 289–306.

[12] Henecka, W., K ogl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security* (2010).

[13] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium* (2011).

[14] Huang, Y., Malka, L., Evans, D., and Katz, J. Efficient Privacy-Preserving Biometric Identification. In *Network and Distributed System Security Symposium* (2011).

[15] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., vol. 2729 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 145–161.

[16] Ishai, Y., Prabhakaran, M., and Sahai, A. Founding cryptography on oblivious transfer efficiently. In *Advances in Cryptology CRYPTO 2008*, D. Wagner, Ed., vol. 5157 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 572–591.

[17] Jha, S., Kruger, L., and Shmatikov, V. Towards Practical Privacy for Genomic Computation. In *IEEE Symposium on Security and Privacy* (2008).

[18] Kiraz, M. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[19] Kiraz, M., and Schoenmakers, B. A Protocol Issue for The Malicious Case of Yao's Garbled Circuit Construction. In *27th Symposium on Information Theory in the Benelux* (2006).

[20] Kolesnikov, V., and Schneider, T. Improved Garbled Circuit: Free XOR Gates and Applications. In *ALP 2008* (2008), L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126 of *LNCS*, Springer, pp. 486–498.

[21] Lindell, Y., Oxman, E., and Pinkas, B. The ips compiler: Optimizations, variants and concrete efficiency. In *CRYPTO* (2011), pp. 259–276.

[22] Lindell, Y., and Pinkas, B. Privacy Preserving Data Mining. *Journal of Cryptology 15*, 3 (2002).

[23] Lindell, Y., and Pinkas, B. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Eurocrypt* (2007).

[24] Lindell, Y., and Pinkas, B. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the 8th conference on Theory of cryptography* (Berlin, Heidelberg, 2011), TCC'11, Springer-Verlag, pp. 329–346.

[25] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay: A Secure Two-Party Computation System. In *13th Conference on USENIX Security Symposium* (2004), vol. 13, USENIX Association, pp. 287–302.

[26] Mohassel, P., and Franklin, M. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Public Key Cryptography* (2006).

[27] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Report 2011/091, 2011. http://eprint.iacr.org/.

[28] Osadchy, M., Pinkas, B., Jarrous, A., and Moskovich, B. SCiFI: A System for Secure Face Identification. In *IEEE Symposium on Security and Privacy* (2010).

[29] Pinkas, B., Schneider, T., Smart, N., and Williams, S. Secure Two-Party Computation Is Practical. In *Asiacrypt* (2009), M. Matsui, Ed., vol. 5912 of *LNCS*, Springer, pp. 250–267.

[30] Shelat, A., and Shen, C.-H. Two-output secure computation with malicious adversaries. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology* (Berlin, Heidelberg, 2011), EUROCRYPT'11, Springer-Verlag, pp. 386–405.

[31] Yang, Z., Zhong, S., and Wright, R. Privacy-preserving Classification of Customer Data without Loss of Accuracy. In *SIAM International Conference on Data Mining* (2005).

[32] Yao, A. Protocols for Secure Computations. In *23rd Symposium on Foundations of Computer Science* (1982), IEEE Computer Society, pp. 160–164.