Faster Secure Two-Party Computation Using Garbled Circuits

Yan Huang David Evans
University of Virginia

Jonathan Katz
University of Maryland

Lior Malka
Intel[*]

Abstract

Secure two-party computation enables two parties to evaluate a function cooperatively without revealing to either party anything beyond the function's output. The *garbled-circuit technique*, a generic approach to secure two-party computation for semi-honest participants, was developed by Yao in the 1980s, but has been viewed as being of limited practical significance due to its inefficiency. We demonstrate several techniques for improving the running time and memory requirements of the garbled-circuit technique, resulting in an implementation of generic secure two-party computation that is significantly faster than any previously reported while also scaling to arbitrarily large circuits. We validate our approach by demonstrating secure computation of circuits with over 10^9 gates at a rate of roughly 10 μs per garbled gate, and showing order-of-magnitude improvements over the best previous privacy-preserving protocols for computing Hamming distance, Levenshtein distance, Smith-Waterman genome alignment, and AES.

1 Introduction

Secure two-party computation enables two parties to evaluate an arbitrary function of both of their inputs without revealing anything to either party beyond the output of the function. We focus here on the *semi-honest setting*, where parties are assumed to follow the protocol but may then attempt to learn information from the protocol transcript (see further discussion in Section 1.2).

There are two main approaches to constructing protocols for secure computation. The first approach exploits specific properties of f to design special-purpose protocols that are, presumably, more efficient than those that would result from generic techniques. A disadvantage of this approach is that each function-specific protocol must be designed, implemented, and proved secure.

The second approach relies on completeness theorems for secure computation [7, 8, 34] which give protocols for computing *any* function f starting from a Boolean-circuit representation of f. This generic approach to secure computation has traditionally been viewed as being of theoretical interest only since the protocols that result require several symmetric-key operations per gate of the circuit being executed and the circuit corresponding to even a very simple function can be quite large.

Beginning with Fairplay [22], several implementations of generic secure two-party computation have been developed in the past few years [11, 21, 27] and used to build privacy-preserving protocols for various functions (e.g., [4, 13, 16, 26, 29]). Fairplay and its successors demonstrated that Yao's technique could be implemented to run in a reasonable amount of time for small circuits, but left the impression that generic protocols for secure computation could not scale to handle large circuits or input sizes or compete with special-purpose protocols for functions of practical interest. Indeed, some previous works have explicitly rejected garbled-circuit solutions due to memory exhaustion [16, 26].

The thesis of our work is that design decisions made by Fairplay, and followed in subsequent work, led researchers to severely underestimate the applicability of generic secure computation. We show that protocols constructed using Yao's garbled-circuit technique can *outperform* special-purpose protocols for several functions.

1.1 Contributions

We show a general method for implementing privacy-preserving applications using garbled circuits that is both faster and more scalable than previous approaches. Our improvements are of two types: we improve the efficiency and scalability of garbled circuit execution itself, and we provide a flexible framework that allows programmers to optimize various aspects of the circuit for computing a given function.

[*]Work done while at the University of Maryland.

	Hamming Distance (900 bits)		**Levenshtein Distance**		**AES**	
	Online Time	Overall Time	Overall Time†	Overall Time‡	Online Time	Overall Time
Best Previous	0.310 s [26]	213 s [26]	92.4 s	534 s	0.4 s [11]	3.3 s [11]
Our Results	0.019 s	0.051 s	4.1 s	18.4 s	0.008 s	0.2 s
Speedup	16.3	4176	22.5	29	50	16.5

Table 1: Performance comparisons for several privacy-preserving applications.

† Inputs are 100-character strings over an 8-bit alphabet. The best previous protocol is the circuit-based protocol of [16].

‡ Inputs are 200-character strings over an 8-bit alphabet. The best previous protocol is the main protocol of [16].

Garbled-circuit execution. In previous garbled-circuit implementations including Fairplay, the garbled circuit (whose length is several hundreds bits per binary gate) is fully generated and loaded in memory before circuit evaluation starts. This impacts both the efficiency of the resulting implementation and severely limits its scalability. We observe that it is unnecessary to generate and store the entire garbled circuit at once. By topologically sorting the gates of the circuit and pipelining the process of circuit generation and evaluation we can significantly improve overall efficiency and scalability. Our implementation never stores the entire garbled circuit, thereby allowing it to scale to effectively an unlimited number of gates using a nearly constant amount of memory.

We also employ all known optimizations, including the "free XOR" technique [18], garbled-row reduction [27], and oblivious-transfer extension [14]. Section 2 provides cryptographic background and explains the protocol and optimizations we use.

Programming framework. Developing and debugging privacy-preserving applications using existing compilers is tedious, cumbersome, and slow. For example, it takes several hours for Fairplay to compile an AES program written in SFDL, even on a computer with 40 GB of memory. Moreover, the high-level programming abstraction provided by Fairplay and other tools for secure computation obscures important opportunities for generating more compact circuits. Although this design decision stems from the worthy goal of providing a high-level programming interface for secure computation, it is severely detrimental to performance. In particular, existing compilers (1) automatically garble the entire circuit, even when portions of the circuit can be computed locally without compromising privacy; (2) use more gates than necessary, since they always use the maximum number of bits needed for a particular variable, even when the number of bits needed at some intermediate stage might be significantly lower; (3) miss important opportunities to replace general gates with XOR gates (which can be garbled "for free" [18]); and (4) miss opportunities to use special-purpose (e.g., multiple input/output) gates that may be more efficient than binary gates. TASTY [11] provides a bit more control, by allowing the programmer to decide when to use depth-2 arithmetic circuits (which can be computed using homomorphic encryption) rather than Boolean circuits. However, this is not enough to support many important circuit optimizations and there are limited places where using homomorphic encryption improves performance over an efficient garbled-circuit implementation.

We present a new method and supporting framework for generating efficient protocols for secure two-party computation. Our method enables programmers to generate a secure protocol computing some function f from an existing (insecure) implementation of f, while providing enough control over the circuit design to enable key optimizations to be employed. Our approach allows users to write their programs using a combination of high-level and circuit-level Java code. Programmers need to be able to design Boolean circuits, but do not need to be cryptographic experts. Our framework enables circuits to be built and evaluated modularly. Hence, even very complex circuits can be generated, evaluated, and debugged. This also provides the programmer with opportunities to introduce important circuit-level optimizations. Although we hope that such optimizations can eventually be done automatically by sophisticated compilers, our emphasis here is on providing a framework that makes it easy to implement privacy-preserving applications. Section 3 provides details about our implementation and efficiency improvements.

Results. We explore applications of our framework to several problems considered in prior work including secure computation of Hamming distance (Section 4) and Levenshtein (edit) distance (Section 5), privacy-preserving genome alignment using the Smith-Waterman algorithm (Section 6), and secure evaluation of the AES block cipher (Section 7). As summarized in Table 1, our implementation yields privacy-preserving protocols that are an order of magnitude more efficient than prior work, in some cases beating even special-purpose protocols designed (and claimed) to be more efficient than what could be obtained using a generic approach.[1]

[1] Results for the Smith-Waterman algorithm are not included in the table since there is no prior work for meaningful comparison, as we discuss in Section 6.

1.2 Threat Model

In this work we adopt the *semi-honest* (also known as *honest-but-curious*) threat model, where parties are assumed to follow the protocol but may attempt to learn additional information about the other party's input from the protocol transcript. Although this is a very weak security model, it is a standard security model for secure computation, and we refer the reader to Goldreich's text [7] for details.

Studying protocols in the semi-honest setting is relevant for two reasons:

- There may be instances where a semi-honest threat model is appropriate: (1) when parties are legitimately trusted but are prevented from divulging information for legal reasons, or want to protect against future break-ins; or (2) where it would be difficult for parties to change the software without being detected, either because software attestation is used or due to internal controls in place (for example, when parties represent corporations or government agencies).
- Protocols for the semi-honest setting are an important first step toward constructing protocols with stronger security guarantees. There exist generic ways of modifying the garbled-circuit approach to give covert security [1] or full security against malicious adversaries [19, 20, 25, 30].

Further, our implementation could be modified easily so as to give meaningful privacy guarantees even against malicious adversaries. Specifically, consider a setting in which only one party P_2 (the circuit evaluator; see Section 2.1) receives output, and the protocol is implemented not to reveal to the other party P_1 anything about the output (including whether or not the protocol completed successfully). If an oblivious-transfer protocol with security against *malicious* adversaries is used (see Section 2.2), our implementation achieves full security against a malicious P_2 and privacy against a malicious P_1. In particular, neither party learns anything about the other party's inputs beyond what P_2 can infer about P_1's input from the revealed output. Understanding how much private information the output itself leaks is an important and challenging problem, but outside the scope of this paper.

Note that this usage of our protocols provides privacy, but does not provide any correctness guarantees. A malicious generator could construct a circuit that produces an incorrect result without detection. Hence, this approach is insufficient for scenarios where the circuit generator may be motivated to trick the evaluator by producing an incorrect result. Such scenarios would require further defenses, including mechanisms to prevent parties from lying about their inputs. Many interesting privacy-preserving applications do have the properties needed for our approach to be effective. Namely, (1) both parties have a motivation to produce the correct result, and (2) only one party needs to receive the output. Examples include financial fraud detection (banks cooperate to detect fraudulent accounts), personalized medicine (a patient and drug company cooperate to determine the best treatment), and privacy-preserving face recognition.

2 Cryptographic Background

This section briefly introduces the cryptographic tools we use: garbled circuits and oblivious transfer. We adapt and implement protocols from the literature, and therefore do not include proofs of security in this work. The protocol we implement can be proven secure based on the decisional Diffie-Hellman assumption in the random oracle model [2].

2.1 Garbled Circuits

Garbled circuits allow two parties holding inputs x and y, respectively, to evaluate an arbitrary function $f(x,y)$ without leaking any information about their inputs beyond what is implied by the function output. The basic idea is that one party (the garbled-circuit *generator*) prepares an "encrypted" version of a circuit computing f; the second party (the garbled-circuit *evaluator*) then obliviously computes the output of the circuit without learning any intermediate values.

Starting with a Boolean circuit for f (which both parties fix in advance), the circuit generator associates two random cryptographic keys w_i^0, w_i^1 with each wire i of the circuit (w_i^0 encodes a 0-bit and w_i^1 encodes a 1-bit). Then, for each binary gate g of the circuit with input wires i, j and output wire k, the generator computes ciphertexts

$$\mathrm{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right)$$

for all inputs $b_i, b_j \in \{0,1\}$. (See Section 3.4 for details about the encryption used.) The resulting four ciphertexts, in random order, constitute a *garbled gate*. The collection of all garbled gates forms the garbled circuit that is sent to the evaluator. In addition, the generator reveals the mappings from output-wire keys to bits.

The evaluator must also obtain the appropriate keys (that is, the keys corresponding to each party's actual input) for the input wires. The generator can simply send $w_1^{x_1}, \ldots, w_n^{x_n}$, the keys that correspond to its own input where each $w_i^{x_i}$ corresponds to the generator's i^{th} input bit. The parties use *oblivious transfer* (see Section 2.2) to enable the evaluator to obliviously obtain the input-wire keys corresponding to its own inputs.

Given keys w_i, w_j associated with both input wires i, j of some garbled gate, the evaluator can compute a key for the output wire of that gate by decrypting the appropriate ciphertext. As described, this requires up to four decryptions per garbled gate, only one of which will succeed. Using standard techniques [22], the construction can be modified so a single decryption suffices. Thus, given one key for each input wire of the circuit, the evaluator can compute a key for each output wire of the circuit. Given the mappings from output-wire keys to bits (provided by the generator), this allows the evaluator to compute the actual output of f. If desired, the evaluator can then send this output back to the circuit generator (as noted in Section 1.2, sending the output back to the generator is a privacy risk unless the semi-honest model can be imposed through some other mechanism).

Optimizations. Several optimizations can be applied to the standard garbled circuits protocol, all of which we use in our implementation. Kolensikov and Schneider [18] introduce a technique that eliminates the need to garble XOR gates (so XOR gates become "free", incurring no communication or cryptographic operations). Pinkas et al. [27] proposed a technique to reduce the size of a garbled table from four to three ciphertexts, thus saving 25% of network bandwidth.[2]

2.2 Oblivious Transfer

One-out-of-two oblivious transfer (OT_1^2) [5, 28] is a crucial component of the garbled-circuit approach. An OT_1^2 protocol allows a *sender*, holding strings w^0, w^1, to transfer to a receiver, holding a selection bit b, exactly one of the inputs w^b; the receiver learns nothing about w^{1-b}, and the sender does not learn b. Oblivious transfer has been studied extensively, and several protocols are known. In our implementation we use the Naor-Pinkas protocol [24], secure in the semi-honest setting. We also use *oblivious-transfer extension* [14] which can achieve a virtually unlimited number of oblivious transfers at the cost of (essentially) k executions of OT_1^2 (where k is a statistical security parameter) plus a marginal cost of a few symmetric-key operations per additional OT. In our implementation, the time for computing the "base" $k = 80$ oblivious transfers is about 0.6 seconds, while the on-line time for each additional OT_1^2 is roughly 15 μs.

For completeness, we note that there are known oblivious-transfer protocols with stronger security properties [10], as well as techniques for oblivious-transfer extension that are secure against malicious adversaries [9]. These could easily be integrated with our implementation to provide the stronger privacy properties for situations where the result does not go back to the circuit generator as discussed in Section 1.2.

[2] A second proposed optimization reduces the size by approximately 50%, but cannot be combined with the free-XOR technique.

3 Implementation Overview

Our implementation allows programmers to construct protocols in a high-level language while providing enough control over the circuit design to enable efficient implementations. The source code for the system and all the applications described in this paper are available under an open-source license from http://mightBeEvil.org. Our code base is very small: the main framework is about 1500 lines of Java code, and a circuit library (see Section 3.3) contains an additional 700 lines of code. The main features of our framework that enable efficient protocols are its support for pipelined circuit execution (Section 3.1) and the optimizations enabled by its circuit-level representation that allow developers to minimize the number of garbled gates needed (Section 3.2). Section 3 describes our circuit library and how a programmer defines a new circuit component. Section 3.4 describes implementation parameters used in our experiments.

3.1 Pipelined Circuit Execution

The primary limitation of previous garbled-circuit implementations is the memory required to store the entire circuit in memory. There is no need, however, for either the circuit generator or evaluator to ever hold the entire circuit in memory. The circuit generation and evaluation processes can be overlapped in time (pipelined), eliminating the need to ever store the entire garbled circuit in memory as well as the need for the circuit generator to delay transmission until the entire garbled circuit is ready. In our framework, the processing of the garbled gates is pipelined to avoid the need to store the entire circuit and to improve the running time. This is automated by our framework, so a user only needs to construct the desired circuit.

At the beginning of the evaluation both the circuit generator and the circuit evaluator instantiate the circuit structure, which is known to both of them and is fairly small since it can reuse components just like a non-garbled circuit. When the protocol is executed, the generator transmits garbled gates over the network as they are produced, in an order defined by the circuit structure. As the client receives the garbled gates, it associates them with the corresponding gate of the circuit. Note that the order of generating and evaluating the circuit does not depend on the parties' inputs (indeed, it cannot since that would leak information about those inputs), so there is no overhead required to keep the two parties synchronized.

The evaluator then determines which gate to evaluate next based on the available output values and tables. Gate

evaluation is triggered automatically when all the necessary inputs are ready. Once a gate has been evaluated it is immediately discarded, so the number of truth tables stored in memory is minimal. Evaluating larger circuits does not significantly increase the memory load on the generator or evaluator, but only affects the network bandwidth needed to transmit the garbled tables.

3.2 Generating Compact Circuits

To build an efficient two-party secure computation protocol, a programmer first analyzes the target application to identify the components that need to be computed privately. Then, those components are translated to digital circuit designs, which are realized as Java classes. Finally, with support from our framework's core libraries, the circuits are compiled and packaged into server-side and client-side programs that jointly instantiate the garbled-circuit protocol.

The cost of evaluating a garbled circuit protocol scales linearly in the number of garbled gates. The efficiency of our approach is due to the pipelined circuit execution technique described above, as well as several methods we use to minimize the number of non-XOR gates that need to be evaluated. One way to reduce the number of gates is to identify parts of the computation that only require private inputs from one party. These components can be computed locally by that party so do not require any garbled circuits. By designing circuits at the circuit level rather than using a high-level language like SFDL [22], we are able to take advantage of these opportunities (for example, by computing the key schedule for AES locally; see Section 7). For the parts of the computation that need to be done cooperatively, we exploit several opportunities enabled by our approach to reduce the number of non-XOR gates needed.

Minimizing bit width. To improve performance, our circuits are constructed with the minimal width required for the correctness of the programs. Our framework supports this by allowing most library circuits to be instantiated with a parameter that specifies the sizes of the inputs, a flexibility that was not present in prior implementations of secure computation. For example, SFDL's simplicity encourages programmers to count the number of 1s in a 900-bit number by writing code that leads to a circuit using 10-bit accumulators throughout the computation even though narrower accumulators are sufficient for early stages. The Hamming distance, Levenshtein distance, and Smith-Waterman applications described in this paper all reduce width whenever possible. This has a significant impact on the overall efficiency: for example, it reduces the number of garbled gates needed for our Levenshtein-distance protocol by 20% (see Section 5.2).

Fast table lookups. Constant-size lookup tables are frequently used in real-world applications (e.g., the score matrix for Smith-Waterman and the SBox for AES). Such lookup tables can be efficiently implemented as a single generalized m-to-n garbled gate, where m is number of bits needed to represent the index and n is the number of bits needed to represent each table entry. This, in turn, can be implemented within as a garbled circuit using a generalization of the standard "permute-and-encrypt" technique [22]. The advantage of this technique is that the circuit evaluator only needs to perform a single decryption operation to look up an entry in an arbitrarily large table. On the other hand, the circuit generator still needs to produce and transmit the entire table, so the cost for the circuit generator and the bandwidth are high. If the table entries have any structure there may be more efficient alternatives (see Section 7 for an example).

3.3 Circuit Library

Our framework includes a library of circuits defined for efficient garbled execution. Applications can be built by composing these circuits, but more efficient implementations are usually possible when programmers define their own custom-designed circuits.

The hierarchy of circuits is organized following the *Composite* design pattern [6] with respect to the build() method. Circuits are constructed in a modular fashion, using Wire objects to connect them together. Appendix A provides a UML class diagram of the core classes of our framework. The Wire and Circuit classes follow a variation of the *Observer* pattern, which offers a kind of publish/subscribe functionality [6]. The main difference is that when a wire w is *connected* to a circuit on port p (represented as a position index to the inputWires array of the circuit), all the observers of the port p automatically become observers of w.

The SimpleCircuit abstract class provides a library of commonly used functions starting with 2-to-1 AND, OR, and XOR gates, where the AND and OR gates are implemented using Yao's garbled-circuit technique and the XOR gate is implemented using the free-XOR optimization. Implementing a NOT gate is also free since it can be implemented as an XOR with constant 1.

The circuit library also provides more complex circuits for, e.g., adders, muxers, comparators, min, max, etc., where these circuits were designed to minimize the number of non-XOR gates using the techniques described in Section 3.2. Optimized circuits for additional functions can be added, as needed. A circuit for some desired function f can be constructed from the components provided in our circuit library, without needing to build the circuit

entirely from AND/OR/NOT gates.

Composite circuits are constructed using the build() method, with the general structure shown below:

```
public void build() throws Exception {
    createInputWires();
    createSubCircuits();
    connectWires();
    defineOutputWires();
    fixInternalWires();
}
```

To define a new circuit, a user creates a new subclass of CompositeCircuit. Typically it is only necessary to override the createSubCircuits(), connectWires(), and defineOutputWires() methods. If internal wires are fixed to known values, these can be set by overriding fixInternalWires(). Our framework automatically propagates known signals which improves the run-time whenever any internal wires are fixed in this way. For example, given a circuit designed to compute the Hamming distance of two 1024-bit vectors, we can immediately obtain a circuit computing the Hamming distance of two 512-bit vectors by fixing 512 of each party's input wires to 0. Because of the way we do value propagation, this does not incur any evaluation cost. As another example, when running the Smith-Waterman algorithm (see Section 6) certain values are fixed to public constants and these can be fixed in our circuit implementing the algorithm in the same way.

3.4 Implementation Details

Throughout this paper, we use 80-bit wire labels for garbled circuits and statistical security parameter $k = 80$ for oblivious-transfer extension. For the Naor-Pinkas oblivious-transfer protocol, we use an order-q subgroup of $\mathbb{Z}_p^*$ with $|q| = 128$ and $|p| = 1024$. These settings correspond roughly to the *ultra-short* security level as used in TASTY [11]. We used SHA-1 to generate the garbled truth-table entries. Each entry is computed as:

$$\mathrm{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right) = \text{SHA-1}\left(w_i^{b_i} \| w_j^{b_j} \| k\right) \oplus w_k^{g(b_i,b_j)}.$$

All cryptographic primitives were used as provided by the Java Cryptography Extension (JCE). Our experiments were performed on two Dell boxes (Intel Core Duo E8400 3GHz) connected on a local-area network.

4 Hamming Distance

The Hamming distance $Hamming(\boldsymbol{a}, \boldsymbol{b})$ between two ℓ-bit strings $\boldsymbol{a} = a_{\ell-1} \cdots a_1 a_0$ and $\boldsymbol{b} = b_{\ell-1} \cdots b_1 b_0$ is simply the number of positions i where $b_i \neq a_i$. Here we consider secure computation of $Hamming(\boldsymbol{a}, \boldsymbol{b})$ where one party holds $\boldsymbol{a}$ and the other has input $\boldsymbol{b}$. Secure Hamming-distance computation has been used as a subroutine in several privacy-preserving protocols [15, 26]. As part of their SciFI work, Osadchy et al. [26] show a protocol based on homomorphic encryption for secure computation of Hamming distance. To reduce the online cost of the computation, SCiFI uses pre-computation techniques aggressively. They report that for $\ell = 900$ their protocol has an "off-line" running time of 213s and an "on-line" running time of 0.31s. (Note that their measure of "off-line running time" includes the time for any processing done locally by one party before sending a message to the other party, even when the local processing depends on that party's input.)

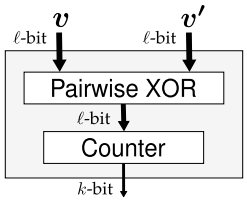


Figure 1: Circuit computing Hamming distance.

4.1 Circuit-Based Approach

We explore a garbled-circuit approach to secure Hamming-distance computation. The high level design of a circuit Hamming for computing the Hamming distance is given in Figure 1. The circuit first computes the XOR of the two ℓ-bit input strings $\boldsymbol{v}, \boldsymbol{v}'$, and then uses a subcircuit Counter to count the number of 1s in the result. The output is a k-bit value, where $k = \lceil \log \ell \rceil$.

A naïve design of the Counter submodule is to use ℓ copies of a k-bit AddOneBit circuit, so that in each of the ℓ iterations the Counter circuit accumulates one bit of $\boldsymbol{v} \oplus \boldsymbol{v}'$ in the k-bit counter.

Since XOR gates are free and an k-bit Adder needs only k non-XOR gates [17], the Hamming circuit with the naïve Counter needs $\ell \cdot \lceil \log \ell \rceil$ non-free gates. We improve upon this by changing the Counter design so as to reduce the number of gates while enabling the gates to be evaluated in parallel.

First, we observe that the widths of the early one-bit adders can be far smaller than k bits. At the first level, the inputs are single bits, so a 1-bit adder with carry is sufficient; at the next level, the inputs are 2-bits, so a 2-bit adder is sufficient. This follows throughout the circuit, halving the total number of gates to $\frac{(\ell \lceil \log \ell \rceil)}{2}$.

Second, the serialized execution order is unnecessary. We improved the naïve design to yield a parallel version of Counter given in Figure 2. Our current execution framework does not support parallel execution, but is designed so that this can be readily supported in a future version.

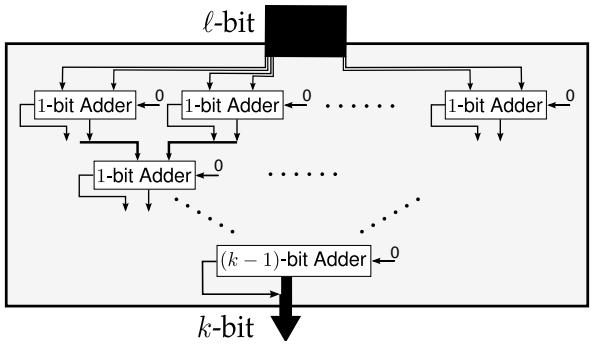


Figure 2: Parallelized Counter circuit.

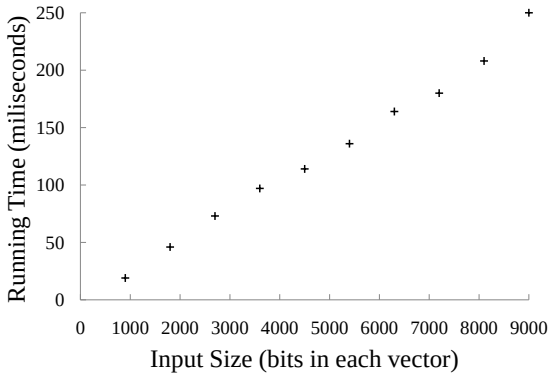


Figure 3: On-line running time of our Hamming-distance protocol for different input lengths.

4.2 Results

We implemented a secure protocol for Hamming-distance computation using the circuit from the previous section and the Java framework described in Section 3. Computing the Hamming distance between two 900-bit vectors took 0.019 seconds and used 56 KB bandwidth in the online phase (including garbled circuit generation and evaluation), with 0.051 seconds (of which the OT takes 0.018 seconds) spent on off-line preprocessing (including garbled circuit setup and the OT extension protocol setup and execution). For the same problem, the protocol used in SCiFI took 0.31 seconds for on-line computation, even at the cost of 213 seconds spent on preprocessing.[3] The SCiFI paper did not report bandwidth consumption, but we conservatively estimate that their protocol would require at least 110 KB. In addition to the dramatic improvement in performance, our approach is quite scalable. Figure 3 shows how the running time of our protocol scales with increasing input lengths.

The garbled-circuit implementation has another advantage as compared to the homomorphic-encryption approach taken by SCiFI: if the obliviously calculated Hamming distances are not the final result, but are only intermediate results that are used as inputs to another computation, then a garbled-circuit protocol is much better in that by its nature it can be readily composed with any subsequent secure computation. In contrast, this is very inconvenient for homomorphic-encryption-based protocols because arbitrary operations over the encryptions are not possible. As an example, in the SCiFI applications the parties do not want to reveal the computed Hamming distance h directly but instead only want to determine if $h > h_{\max}$ for some public value h_{max}. Osadchy et al. had to design a special protocol involving adding random noise to the h values and using an oblivious transfer protocol to handle this. In our case, however, we would only need to add a comparator circuit after the Hamming-distance computation. In fact, with our approach further optimizations would be possible when h_{max} is known since at most the $\lceil \log h_{max} \rceil$ low-order bits of the Hamming distance need to be computed.

[3] Osadchy et al. [26] used a 2.8 GHz dual core Pentium D with 2 GB RAM for their experiments, so the comparison here is reasonably close. Also note that for their experiments, Osadchy et al. configured their host to turn off the Nagle ACK delay algorithm, which substantially improved network performance. This is not realistic for most network settings and was not done in our experiments.

5 Levenshtein Distance

The *Levenshtein distance* (also known as *edit distance*) between two strings has applications in DNA and protein-sequence alignment, as well as text comparison. Given two strings α and β, the Levenshtein distance between them (denoted $Levenshtein(\alpha,\beta)$) is defined as the minimum number of basic operations (insertion, deletion, or replacement of a single character) that are needed to transform α into β. In the setting we are concerned with here, one party holds α and the other holds β and the parties wish to compute $Levenshtein(\alpha,\beta)$.

Algorithm 1 is a standard dynamic-programming algorithm for computing the Levenshtein distance between two strings. The invariant is that $\mathtt{D[i][j]}$ always represents the Levenshtein distance between $\alpha[1\ldots\mathtt{i}]$ and $\beta[1\ldots\mathtt{j}]$. Lines 2–4 initialize each entry in the first row of the matrix D, while lines 5–8 initialize the first column. Within the two for-loops (lines 8–13), $\mathtt{D[i][j]}$ is assigned at line 11 to be the smallest of $\mathtt{D[i-1][j]}+1$, $\mathtt{D[i][j-1]}+1$, or $\mathtt{D[i-1][j-1]}+\mathtt{t}$ (where t is 0 if $\alpha[\mathtt{i}]=\beta[\mathtt{j}]$ and 1 if they are different). These correspond to the three basic operations *insert* $\alpha[\mathtt{i}]$, *delete* $\beta[\mathtt{j}]$, and *replace* $\alpha[\mathtt{i}]$ with $\beta[\mathtt{j}]$.

5.1 State of the Art

Jha et al. give the best previous implementation of a secure two-party protocol for computing the Levenshtein distance [16]. Instead of using Fairplay, they developed their own compiler based on Fairplay, while borrow-

Algorithm 1 *Levenshtein*(α, β)

```
1:  Initialize D[α.length][β.length];
2:  for i ← 0 to α.length do
3:      D[i][0] ← i;
4:  end for
5:  for j ← 0 to β.length do
6:      D[0][j] ← j;
7:  end for
8:  for i ← 1 to α.length do
9:      for j ← 1 to β.length do
10:         t ← (α[i] = β[j]) ? 0 : 1;
11:         D[i][j] ← min(D[i − 1][j]+1, D[i][j − 1]+1,
                          D[i − 1][j − 1]+t);
12:     end for
13: end for
```

ing the function-description language (SFDL) and the circuit-description language (SHDL) directly from Fairplay. Jha et al. investigated three different strategies for securely computing the Levenshtein distance. Their first protocol (Protocol 1) directly instantiated Algorithm 1 as an SFDL program, which was then compiled into a garbled-circuit implementation. Because their garbled-circuit execution approach required keeping the entire circuit in memory, they concluded that garbled circuits could not scale to large inputs. The largest problem size their compiler and execution environment could handle before crashing was where the parties' inputs were 200-character strings over an 8-bit (256-character) alphabet.

Their second protocol combined garbled circuits with an approach based on *secure computation with shares*. The resulting protocol was scalable, but extremely slow. Finally, they proposed a hybrid protocol (Protocol 3) by combining the first two approaches to achieve better performance with scalability.

According to their results, it took 92 seconds for Protocol 1 to complete a problem of size 100×100 (i.e., two strings of length 100) over an 8-bit alphabet. This protocol required nearly 2 GB of memory to handle the 200×200 case [16]. Their flagship protocol (Protocol 3), which is faster for larger problem sizes, took 658 seconds and used 364.3 MB bandwidth on a problem of size 200×200 over an 8-bit alphabet.

5.2 Circuit-Based Approach

We observed that the circuit used for secure computation of Levenshtein distance can be much smaller than the circuit produced from a high-level SFDL description. The main reason is that the SFDL description does not distinguish parts of the computation that can be performed locally by one of the parties, nor does it take advantage of the actual number of bits required for values at intermediate stages of the computation.

The portion of the computation responsible for initializing the matrix (lines 2–7) does not require any collaboration, and thus can be completed by each party independently. Moreover, since the length of each party's private string is not meant to be kept secret, the two for-loops (lines 8–9) can be managed by each party independently as long as they keep the inner executions synchronized, leaving only two lines of code (lines 10–11) in the innermost loop that need to be computed securely.

Let ℓ denote the length of the parties' input strings, assumed to be over a σ-bit alphabet. Figure 5a presents a circuit, LevenshteinCore, that is computationally equivalent to lines 10–11. The T (stands for "test") circuit in that figure outputs 1 if the input strings provided are different. Figure 4 shows the structure of the T circuit. (For the purposes of the figures in this section, we assume $\sigma = 2$ since this is the alphabet size that would be used for genomic comparisons. Nevertheless, everything generalizes easily to larger σ.) For a σ-bit alphabet, the T circuit uses $\sigma - 1$ non-free gates.

The rest of the circuit computes the minimum of the three possible edits (line 11 in Algorithm 1). We begin with the straightforward implementation shown in Figure 5a. The values of D[i − 1][j], D[i][j − 1], and D[i − 1][j − 1] are each represented as ℓ-bit inputs to the circuit. For now, this is fixed as the maximum value of any D[i][j] value. Later, we reduce this to the maximum value possible for a particular core component. Because of the way we define ℓ there is no need to worry about the carry output from the adders since ℓ is defined as the number of bits needed to represent the maximum output value. The circuit shown calculates exactly the same function as line 11 of Algorithm 1, producing the output value of D[i][j]. The full Levenshtein circuit has one LevenshteinCore component for each *i* and *j* value, connected to the appropriate inputs and producing the output value D[i][j]. The output value of the last LevenshteinCore component is the Levenshtein distance.

Recall that each ℓ-bit AddOneBit circuit uses ℓ non-free gates, and each ℓ-bit 2-MIN uses 2ℓ non-free gates. So, for problems on a σ-bit alphabet, each ℓ-bit NaiveLevenshteinCore circuit uses $7\ell + \sigma - 1$ non-free gates. Next, we present two optimizations that reduce the number of non-free gates involved in computing the

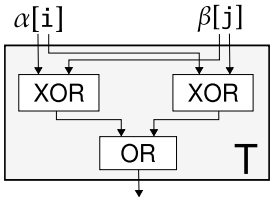
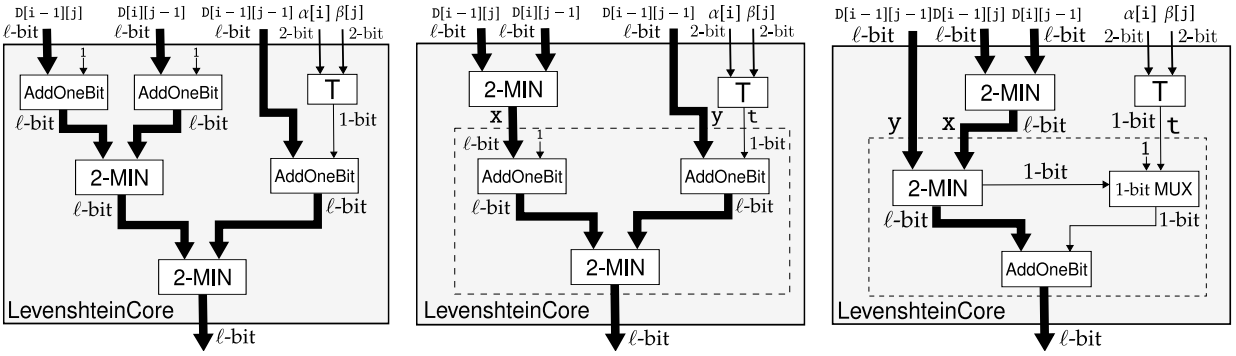


Figure 4: T circuit.



(a) Naïve (b) Better (c) Final

Figure 5: Implementations of the Levenshtein core circuit.

Levenshtein core to $5\ell+\sigma$.

Since $\texttt{min}(\texttt{D[i}-\texttt{1][j}]+1,\texttt{D[i][j}-\texttt{1]}+1)$ is equivalent to $\texttt{min}(\texttt{D[i}-\texttt{1][j}],\texttt{D[i][j}-\texttt{1]})+1$, we can combine the two AddOneBit circuits (at the top left of Figure 5a) into a single one, and interchange it with the subsequent 2-MIN as shown in Figure 5b. The circuits in the dashed box in Figure 5b compute $\texttt{min}(\texttt{x}+1,\texttt{y}+\texttt{t})$, where $\texttt{t}\in\{0,1\}$. This is functionally equivalent to:

$$\textbf{if } (\texttt{y}>\texttt{x}) \textbf{ then } \texttt{x}+1 \textbf{ else } \texttt{y}+\texttt{t}.$$

Hence, we can reuse one of the AddOneBit circuits by putting it after the GT logic embedded in the MIN circuit. This leads to the optimized circuit design shown in Figure 5c. Note that the 1-bit output wire connecting the 2-MIN and 1-bit MUX circuits is essentially the 1-bit output of the GT sub-circuit inside 2-MIN. This change reduces the number of gates in the core circuit to $2\times 2\ell+\ell+\sigma-1+1=5\ell+\sigma$.

The final optimization takes advantage of the observation that the minimal number of bits needed to represent $\texttt{D[i][j]}$ varies throughout the computation. For example, one bit suffices to represent $\texttt{D[1][1]}$ while more bits are required to represent $\texttt{D[i][j]}$ for larger $\texttt{i}$'s and $\texttt{j}$'s. The value of $\texttt{D[i][j]}$ can always be represented using $\lceil\log\min(\texttt{i},\texttt{j})\rceil$ bits. The number of gates decreases by:

$$1-\frac{\sum_{i=1}^{\ell}\sum_{j=1}^{\ell}\left\lceil\log\left[\min(i,j)\right]\right\rceil}{\ell^2\lceil\log\ell\rceil}.$$

For $\ell=200$ this results in a 25% savings, but the effect decreases as ℓ grows.

Although it would be possible to describe such a circuit using a high-level language like SFDL, it would be very tedious and awkward to do so and would require a customized program for each input size. Hence, SFDL programs tend to allocate more than the number of bits needed to ensure correctness of the protocol output.

5.3 Results

We implemented a protocol for secure computation of Levenshtein distance using the circuit described above and our framework from Section 3. The protocol handles arbitrary input lengths ℓ (it also handles the case where the input strings have different lengths) and arbitrary alphabet sizes 2^σ. It completes a problem of size 200×200 over a 4-character alphabet in 16.38 seconds (of which less than 1% is due to OT) using 49 MB bandwidth. The dependence of the running time on σ is small: for $\sigma=8$ our protocol takes 18.4 seconds in the 200×200 case, which is 29 times faster than the results of Jha et al. [16].

Our protocol is highly scalable, as shown in Figure 6. The largest problem instance we ran is 2000×10000 (not shown in the figure), which used a total of 1.29 billion non-free binary gates and completed in under 223 minutes (at a rate of over 96,000 gates per second). In addition, our approach enables further optimizations for many practical scenarios. For example, if the parties are only interested in determining whether the Levenshtein distance is below some threshold d, then only the $\lceil\log d\rceil$ low-order bits of the result need to be computed and the number of bits for an entry can be reduced.

6 Smith-Waterman

The Smith-Waterman algorithm (Algorithm 2) is a popular method for genome and protein alignment [23,31]. In contrast to Levenshtein distance which measures *dissimilarity*, the Smith-Waterman score measures *similarity* between two sequences (higher scores mean the sequences are more similar). The algorithm has a basic structure similar to the algorithm for computing Levenshtein distance. The differences are: (1) the preset entries (the first row and the first column) are initialized to 0; (2) the algorithm has a more sophisticated core (lines 10–12) that involves an affine gap function $\texttt{gap}$ and computes the maximum score across all previous entries in the row and

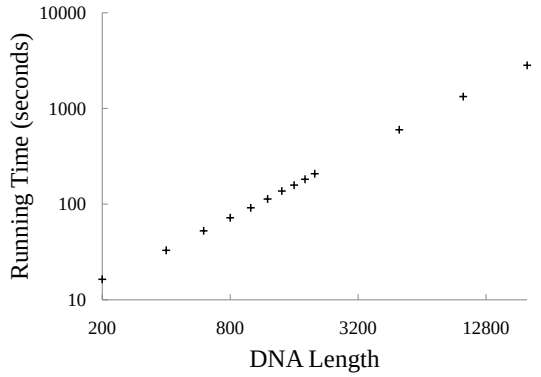


Figure 6: Overall running time of our Levenshtein-distance protocol. (Plotted on a log-log scale; the problem size is 200×DNA Length and $\sigma = 2$.)

column; and (3) the algorithm uses a fixed 2-dimensional score matrix `score`.

In practice, the gap function is typically of the form $\texttt{gap}(x) = a + b \cdot x$ where a, b are publicly known, negative integer constants. By choosing a and b appropriately, one can account for the fact that the evolutionary likelihood of inserting a single large DNA segment is much greater than the likelihood of multiple insertions of smaller segments (of the same total length). A typical gap function is $\texttt{gap}(x) = -12 - 7x$, which is what we use in our evaluation experiments.

The 2-dimensional score matrix `score` quantifies how well two symbols from an alphabet match each other. In comparing proteins, the symbols represent amino acids (one of twenty possible characters including stop symbols). The entries on the diagonal of the `score` matrix are larger and positive (since each symbol aligns well with itself), while all others are smaller and mostly negative numbers. The actual numbers vary, and are computed based on statistical analysis of a genome database. We use the BLOSUM62 [12] score matrix for computation over randomly generated protein sequences.

To obtain the optimal alignment, one first computes matrix `D` using Algorithm 2, then finds the entry in `D` with the maximum value and traces the path backwards to find how this value was derived. In a privacy-preserving setting, the full trace may reveal too much information. Instead, it may be used as an intermediate value for a continued secure computation, or just aspects of the result (e.g., the score or starting position) could be revealed.

6.1 State of the Art

The only previous attempt to implement a secure Smith-Waterman computation is by Jha et al. [16]. (An alternate approach, suggested by Szajda et al. [32], is to perform the computation normally but operating on transformed data instead of the parties' private data. It is unclear, however, what privacy or correctness properties can be achieved by this approach.) Jha et al.'s protocol follows a similar approach to their Levenshtein-distance protocols described in Section 5, and led them to conclude that garbled-circuit implementations could not handle even small inputs (their garbled-circuit implementation for Smith-Waterman could not handle a 25×25 size input). Hence, they invented a hybrid protocol (Protocol 3) to implement the Smith-Waterman algorithm.

Their prototype had two limitations that prevent direct performance comparisons:

1. They use only 8 bits to represent each entry of the dynamic-programming matrix, but for most protein-alignment problems the *similarity* scores between even two short sequences of length 25 can overflow an 8-bit integer, and for larger sequences it is bound to overflow. In the BLOSUM62 scoring table, the typical score for two matching proteins is 6 (and as high as 11).

2. They used a constant gap function ($\texttt{gap}(x) = -4$) that is inappropriate for practical scenarios.

Despite these simplifications in their work, our complete Smith-Waterman implementation (that does not make any of these simplifications) still runs more than twice as fast as their implementation.

Algorithm 2 `Smith-Waterman`(α, β, `gap`, `score`)

```
1:  Initialize D[α.length][β.length];
2:  for i ← 0 to α.length do
3:    D[i][0] ← 0;
4:  end for
5:  for j ← 0 to β.length do
6:    D[0][j] ← 0;
7:  end for
8:  for i ← 1 to α.length do
9:    for j ← 1 to β.length do
10:     rMax ← max_{1≤o≤i}(D[i − o][j] + gap(o));
11:     cMax ← max_{1≤o≤j}(D[i][j − o] + gap(o));
12:     D[i][j] ← max(0, rMax, cMax,
                      D[i − 1][j − 1] + score[α[i]][β[j]]);
13:   end for
14: end for
```

6.2 Circuit-Based Approach

The core of the Smith-Waterman algorithm (lines 10–12 of Algorithm 2) involves ADD and MAX circuits. To reduce the number of non-free gates, we replace lines 10–11 with the code in Algorithm 3. This allows us to

Algorithm 3 Restructured Smith-Waterman core

```
rMax ← 0;
for o ← 1 to i do
    rMax ← max(rMax, D[i − o][j] + gap(o));
end for
cMax ← 0;
for o ← 1 to j do
    cMax ← max(cMax, D[i][j − o] + gap(o));
end for
```

use much narrower ADD and MAX circuits for some entries since we know the value of D[i][j] is bounded by $\lceil \log(\min(i, j) \cdot maxscore) \rceil$, where *maxscore* is the greatest number in the score matrix. We only need to make sure that values are appropriately sign-extended (a free operation) when they are carried between circuits of different width.

We also note that gap(o), which serves as the second operand to every ADD circuit, can always be safely computed without collaboration since it does not depend on any private input. Thus, instead of computing gap(o) using a complex garbled circuit, it can be computed directly with the output value fed directly into the ADD circuit. Being able to tightly bound the part of the computation that really needs to be done privately is another advantage of our approach.

The matrix-indexing operation on score does need to be done in a privacy-preserving way since its inputs reveal symbols in the private inputs of the parties. Since the row index and column index each can be denoted as a 5-bit number, we could view the score table as a 10-to-1 garbled circuit (whereas each entry in truth table is an encryption of 5 wire keys representing the output value). Using an extension of the *permute-and-encrypt* technique, it leads to a garbled table containing $2^{10} = 1024$ ciphertexts (of which 624 are null entries since the actual table is 20×20, but which must be transmitted as random entires to avoid leaking information). However, observe that one of the two indexes is known to the circuit generator since it corresponds to the generator's input value at a known location. Hence, we use the index known to the circuit generator to specialize the two-dimensional score table lookup to a one-dimensional table lookup. This reduces the cost of oblivious table lookup to computing and transmitting 20 ciphertexts and 12 random entries (to fill the 2^5-entry table) for the circuit generator, while the work for the circuit evaluator is still performing one decryption.

6.3 Results

Our secure Smith-Waterman protocol takes 415 seconds and generates 1.17 GB of network traffic running on two

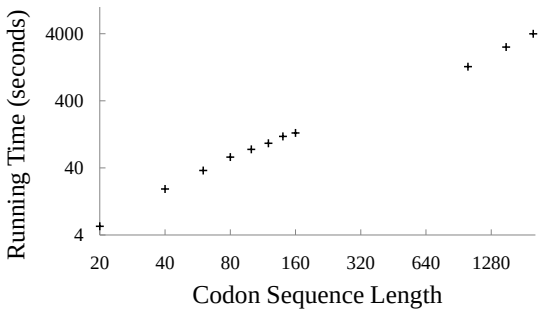


Figure 7: Overall running time of the Smith-Waterman protocol. (Plotted on a log-log scale; problem size 20×Codon Sequence Length.)

protein sequences of length 60. The garbled-circuit implementation by Jha et al. did not scale to a 60×60 input size, but their Protocol 3 was able to complete on this input length in nearly 1000 seconds (but recall that due to simplifications they used, their implementation would not usually produce the correct result). Figure 7 shows the running time of our implementation as a function of the problem size.

7 AES

AES is a standardized block cipher. We focus on AES-128 which uses a 128-bit key as well as a 128-bit block length. The high-level operation of AES is shown in Listing 1 (based on Daemen and Rijmen's report [3]). It takes a 16-byte array msg and a large byte array key, which is the output of the AES key schedule. The variable Nr denotes the number of rounds (for AES-128, Nr=10).

In privacy-preserving AES, one party holds the key k and the other holds an input block x. At the end of the protocol, the second party learns $AES_k(x)$. This functionality has a number of interesting applications including encrypted keyword search (see Pinkas et al. [27]).

7.1 Prior Work

Pinkas et al. [27] implement AES as an SFDL program, which is in turn compiled to a huge SHDL circuit consisting of more than 30,000 gates. Henecka et al. [11] used the same circuit, but obtained better online performance by moving more of the computation to the precomputation phase. The best performance results they reported are 3.3 seconds in total and 0.4 seconds online per block-cipher evaluation.

7.2 Our Approach

We also use garbled circuits to implement privacy-preserving AES. However, our technique is distinguished

```
public static byte[] Cipher(byte[] key, byte[] msg) {
    byte[] state = AddRoundKey(key, msg, 0);
    for (int round = 1; round < Nr; round++) {
        state = SubBytes(state);
        state = ShiftRows(state);
        state = MixColumns(state);
        state = AddRoundKey(key, state, round);
    }

    state = SubBytes(state);
    state = ShiftRows(state);
    state = AddRoundKey(key, state, Nr);
    return state;
}
```

Listing 1: The AES block cipher.

from previous ones in that instead of constructing a huge circuit, we derive our privacy-preserving implementation around the structure of a traditional program, following the code in Listing 1. Our guiding principle is to identify the minimal subset of the computation that needs to be performed in a privacy-preserving manner, and only use garbled circuits for that portion of the computation. Specifically, we observe that the entire key schedule can be computed locally by the party holding the key. There is no need to use garbled circuits to compute the key schedule since it only depends on one party's data.

Overview. To make the implementation simpler, we explicitly group the wire labels of every 8-bit byte into a *State* object, representing the intermediate results of garbled circuits. Compared to the original code (Listing 1), we only need to replace the built-in data type `byte` with our custom type `State` in building the code for implementing the garbled circuit. Since the state is represented by garbled wire labels, we can compose circuits implementing each execution phase to perform the secure computation.

As noted earlier, the value of the `key` which is the output of the key schedule can be executed by Alice alone, and then used as effective input to a circuit. This enables us to replace the expensive privacy-preserving key schedule computation with less expensive oblivious transfers (which, due to the oblivious-transfer extension, are cheaper than using garbled circuits).

Second, as in many other real-world AES cipher implementations, the `SubBytes` subroutine dominates the resource (e.g., time and hardware area) consumption. We consider two possible designs for implementing the `SubBytes` subroutine. The first design minimizes online time for situations where preprocessing is possible; the second minimizes total time in the absence of idle periods for preprocessing.

Third, the `ShiftRows` subroutine imposes no cost for our circuit implementation since this subroutine merely impacts the wiring but requires no additional gates.

The `MixColumns` subroutine requires secure computation, but we design a circuit for this that uses only XORs. The `AddRoundKey` subroutine is realized by a BitWiseXOR circuit that simply juxtaposes 128 XOR gates.

SubBytes. The `SubBytes` component dominates the time for AES, so we consider two alternate designs.

Minimizing online time. Our first design seeks to minimize the online execution time by moving as much of the work as possible to the preprocessing phase. The `SubBytes` subroutine can be implemented with sixteen 8-bit-to-8-bit garbled tables, similar to the score matrix used in the Smith-Waterman application. From the perspective of the circuit generator, this results in a garbled "gate" with $2^8 \times 8 = 2048$ ciphertexts. The circuit evaluator need only decrypt 8 of these (i.e., one table entry) at a cost of 4 hash evaluations (since we use 80-bit wire labels and SHA-1, with 160-bit output length, for the encryption). This design is distinguished by its very low online cost, so is well suited to situations where the primary goal is to minimize the online execution time.

Minimizing total time. Our second design aims to minimize the total execution time by implementing `SubBytes` with an efficient circuit derived from the work of Wolkerstorfer et al. [33]. The two logical components of `SubBytes` are computing an inverse over $\mathrm{GF}(2^8)$ and an affine transformation over $\mathrm{GF}(2)$. The circuit we use to compute the inverse over $\mathrm{GF}(2^8)$ is given in Figure 8. In essence, $\mathrm{GF}(2^8)$ is viewed as an extension of $\mathrm{GF}(2^4)$, so that an element of $\mathrm{GF}(2^8)$ is mapped to a vector of length two over $\mathrm{GF}(2^4)$. A series of operations over $\mathrm{GF}(2^4)$ are applied to these values, which are then mapped back to an element in $\mathrm{GF}(2^8)$. In this circuit diagram, Map and Inverse Map circuits realize the bijection between $\mathrm{GF}(2^8)$ and $(\mathrm{GF}(2^4))^2$; $\oplus$ and $\otimes$ represent *addition* and *multiplication* over $\mathrm{GF}(2^4)$, respectively. The affine transform over finite field $\mathrm{GF}(2)$ and all of the component circuits except for the $\otimes$ and $\mathrm{GF}(2^4)$Inverse circuits can be implemented using XOR gates alone. Since each $\otimes$ circuit has 16 non-free gates and each $\mathrm{GF}(2^4)$Inverse has 10 non-free gates, the total number of non-free gates per $\mathrm{GF}(2^8)$Inverse circuit is $16 \times 3 + 10 = 58$.

MixColumns. The core functionality of `MixColumns` is to compute $s'_c(x) = a(x) \otimes s_c(x)$, where $0 \le c < 4$ specifies the column, $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, and $\otimes$ denotes multiplication over finite field $\mathrm{GF}(2^8)$. Let $s_c(x) = s_{3,c}x^3 + s_{2,c}x^2 + s_{1,c}x + s_{0,c}$ and $s'_c(x) =$

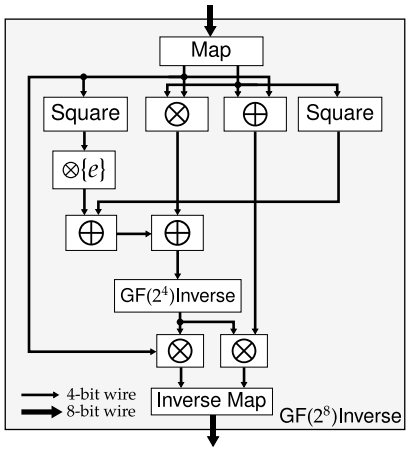


Figure 8: Inverse Circuit over GF(2^8).

$s'_{3,c}x^3 + s'_{2,c}x^2 + s'_{1,c}x + s'_{0,c}$. This is equivalent to

$$\begin{aligned}
s'_{0,c} &= (\{02\}\cdot s_{0,c}) \oplus (\{03\}\cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\}\cdot s_{1,c}) \oplus (\{03\}\cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\}\cdot s_{2,c}) \oplus (\{03\}\cdot s_{3,c}) \\
s'_{3,c} &= (\{03\}\cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\}\cdot s_{3,c}).
\end{aligned}$$

It follows that:

$$\begin{aligned}
s'_{0,c} &= (\{02\}\cdot s_{0,c}) \oplus (\{02\}\cdot s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\}\cdot s_{1,c}) \oplus (\{02\}\cdot s_{2,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\}\cdot s_{2,c}) \oplus (\{02\}\cdot s_{3,c}) \oplus s_{3,c} \\
s'_{3,c} &= (\{02\}\cdot s_{0,c}) \oplus s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\}\cdot s_{3,c}).
\end{aligned}$$

The operation $\{02\}\cdot b$ is defined as multiplying b by $\{02\}$ modulo $\{1b\}$ in GF(2^8). If $b = b_7 \cdots b_1 b_0$, and $z = z_7 \cdots z_1 z_0 = \{02\}\cdot b$, the output bits can be computed using only XOR gates:

$$\begin{aligned}
&z_7 = b_6, && z_6 = b_5, && z_5 = b_4, && z_4 = b_3 \oplus b_7, \\
&z_3 = b_2 \oplus b_7, && z_2 = b_1, && z_1 = b_0 \oplus z_7, && z_0 = b_7
\end{aligned}$$

For every column of 4-byte numbers, the equations above are implemented by the MixOneColumn circuit (Figure 9). Each invocation of `MixColumns` involves processing four columns, so we can build the MixColumns circuit by juxtaposing four MixOneColumn circuits. Thus, the MixColumns circuit can be implemented using only XOR gates.

7.3 Results

Using the first (online-minimizing) SubBytes design, there are no non-free gates and 160 oblivious table lookups. The total time for the computation is 1.6 seconds without preprocessing. With preprocessing, the online time to evaluate the circuit is 0.008 seconds (since the evaluator can always identify the right entry in the table to decrypt), more than 50 times faster than the best previous results [11].

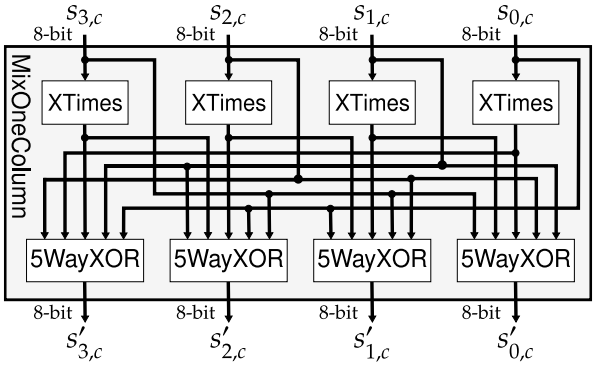


Figure 9: MixOneColumn Circuit.

With our second design, the total number of non-free gates for the entire AES computation is $58 \times 16 \times 10 = 9280$. The overall time is 0.2 seconds (of which 0.08 seconds is spent on oblivious transfer) without preprocessing, more than 16 times faster than the best previous results [11]. The online time is 0.06 seconds with preprocessing enabled.

8 Conclusion

Misconceptions about the performance and scalability of garbled circuits are pervasive. This perception has led to the development of several complex, special-purpose protocols for problems that are better addressed by garbled circuits. We demonstrate that a simple pipelining approach, along with techniques to minimize circuit size, is enough to make garbled circuits scale to many large problems, and practical enough to be competitive with special-purpose protocols.

We hope improvements in the efficiency of privacy-preserving computing will enable many sensitive applications to be deployed. Ours is just a first step towards that goal, and more work needs to be done before secure computation can be used routinely in practice. Although our approach enables circuits to scale arbitrarily and make evaluation substantially faster than previous work, it is still far slower than normal computation. Further performance improvements are needed before large problems can be computed securely in interactive systems. In addition, our work assumes the semi-honest threat model which is only suitable for certain scenarios where only one party obtains the output or both parties can rely on verified implementations. Efficient protocols secure against a malicious adversary model appear to be much more challenging to design.

Acknowledgments

The authors thank Ian Goldberg for his extensive and very helpful comments and suggestions on this paper. Peter Chapman, Jiamin Chen, Yikan Chen, Austin DeVinney, Brittany Harris, Sang Koo, abhi shelat, Chi-Hao Shen, Dawn Song, David Wagner, and Samee Zahur also provided valuable comments on this work. The authors thank Somesh Jha and Louis Kruger for providing their Smith-Waterman secure computation implementation and answering our questions about it.

This work was partly supported by grants from the National Science Foundation, DARPA, and a MURI award from the Air Force Office of Scientific Research. The contents of this paper do not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

References

[1] Y. Aumann and Y. Lindell. Security against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In *4th Theory of Cryptography Conference*, 2007.

[2] M. Bellare and P. Rogaway. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security* (CCS), 1993.

[3] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer Verlag, 2002.

[4] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving Face Recognition. In *9th International Symposium on Privacy Enhancing Technologies*, 2009.

[5] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 28(6), 1985.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995.

[7] O. Goldreich. *Foundations of Cryptography, Volume 2: Basic Applications*. Cambridge University Press, Cambridge, UK, 2004.

[8] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority. In *19th ACM Symposium on Theory of Computing* (STOC), 1987.

[9] D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-combiners via Secure Computation. In *5th Theory of Cryptography Conference*, 2008.

[10] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Computation: Techniques and Constructions*. Springer, 2010.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-party Computations. In *ACM Conference on Computer and Communications Security* (CCS), 2010.

[12] S. Henikoff and J. G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. In *Proceedings of the National Academy of Sciences of the United States of America*, 1992.

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

[14] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *Advances in Cryptology — Crypto*, 2003.

[15] A. Jarrous and B. Pinkas. Secure Hamming Distance Based Computation and its Applications. In *Applied Cryptography and Network Security* (ACNS), 2009.

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

[17] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Cryptology and Network Security* (CANS), 2009.

[18] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages, and Programming* (ICALP), 2008.

[19] 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.

[20] Y. Lindell and B. Pinkas. Secure Two-party Computation via Cut-and-Choose Oblivious Transfer. In *7th Theory of Cryptography Conference*, 2011.

[21] Y. Lindell, B. Pinkas, and N. Smart. Implementing Two-party Computation Efficiently with Security against Malicious Adversaries. In *International*

Conference on Security and Cryptography for Networks (SCN), 2008.

[22] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a Secure Two-party Computation System. In *13th USENIX Security Symposium*, 2004.

[23] R. Mott. Smith-Waterman Algorithm. In *Encyclopedia of Life Sciences*. John Wiley & Sons, 2005.

[24] M. Naor and B. Pinkas. Computationally Secure Oblivious Transfer. *Journal of Cryptology*, 18(1), 2005.

[25] J. B. Nielsen and C. Orlandi. LEGO for Two-party Secure Computation. In *6th Theory of Cryptography Conference*, 2009.

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

[27] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-party Computation is Practical. In *Advances in Cryptology — Asiacrypt*, 2009.

[28] M. O. Rabin. How to Exchange Secrets with Oblivious Transfer. Technical Report 81, Harvard University, 1981.

[29] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient Privacy-preserving Face Recognition. In *ICISC 09: 12th International Conference on Information Security and Cryptology*, 2009.

[30] A. Shelat and C.-H. Shen. Two-output Secure Computation with Malicious Adversaries. In *Advances in Cryptology — Eurocrypt*, 2011.

[31] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 1981.

[32] D. Szajda, M. Pohl, J. Owen, and B. G. Lawson. Toward a Practical Data Privacy Scheme for a Distributed Implementation of the Smith-Waterman Genome Sequence Comparison Algorithm. In *Network and Distributed System Security Symposium* (NDSS), 2006.

[33] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC Implementation of the AES S-boxes. In *Cryptographers' Track — RSA*, 2002.

[34] A. C.-C. Yao. How to Generate and Exchange Secrets. In *27th Symposium on Foundations of Computer Science* (FOCS), 1986.

A Core Classes

The core classes in our framework are shown in the UML diagram below.