Efficient Privacy-Preserving Computing Using Garbled Circuits

Anonymized for Submission

***Abstract*—Secure two-party computation enables two parties to cooperatively evaluate a function that involves private inputs from each party without revealing anything beyond the function's output. Yao developed a general approach to secure two-party computation using garbled circuits in the 1980s, but because of its apparent inefficiency and high memory requirements it has not been used widely in real systems. This paper demonstrates that the garbled circuit technique can be made much more efficient than previously thought. We avoid the memory problem by pipelining the process of generating and evaluating the circuit so the entire circuit never needs to be stored in memory. This allows us to run garbled circuits at an arbitrary scale (over a billion non-free gates for computing Levenshtein distance between two DNA sequences of lengths 2000 and 10000). Further, we develop several techniques for making the circuits more efficient by designing them to minimize the number of non-free gates. We demonstrate our approach by achieving order of magnitude improvements over the best previous secure protocols for Hamming distance, Levenshtein Distance, Smith-Waterman alignment, and privacy-preserving AES.**

***Keywords*-secure computation, secure two-party computation, privacy-preserving protocols.**

I. Introduction

It has been a long-term pursuit of the cryptographic community to enable two mutually untrusted parties to cooperatively compute an arbitrary function, $f(x,y)$, where x and y are the each party's respective private input to f, without leaking any information about x or y beyond what the output inherently reveals [33]. Andrew Yao gave a general solution, known as the *garbled circuit* technique, to this problem for any two-party computation scenario [34]. It has run-time and communication complexity that is linear in the size of the circuit needed to compute the original function, but with a large constant factor because of the encryption operations needed to generate and evaluate each binary gate.

A well-known implementation of garbled circuits is Fairplay [23], which produces a garbled circuit protocol from a program written in a high-level procedural language. Many subsequent projects either used Fairplay directly [4, 13, 20] or indirectly [18, 8, 28, 17, 27] to build privacy-preserving protocols. The success of Fairplay shaped the security community's general view of the poor performance of the garbled circuit technique. Due to certain design decisions taken by Fairplay, and followed by its successors, the efficiency of the garbled circuit technique was largely underestimated. Driven by this false impression, some subsequent research on secure computation developed alternative solutions to problems that can be better solved by straightforward garbled circuits [26, 18].

The first design flaw of previous garbled circuit implementations is that the circuit, including all the garbled tables, must be fully generated and loaded in memory before the evaluation starts. This is problematic since security of the technique requires that each gate can only be used once, so a huge garbled circuit is needed to solve any large problem. Previous authors (e.g., [18]) have explicitly rejected garbled circuits solutions because of memory exhaustion. However, as the circuit evaluation only needs to follow the topological sort order, parts of the circuit can be executed regardless whether the rest of it is available. Requiring the circuit representation to be fully loaded before execution is both prohibitive and unnecessary. Our framework pipelines the circuit generation and evaluation processes, evaluating the circuit as it is generated without ever needing to store the entire circuit.

One main contribution of this work is designing, implementing, and evaluating a tool for generating efficient protocols for two-party secure computation. Section III describes our method and Section VIII explains how we implement it as a Java framework. Our approach uses garbled circuits, but uses pipelining aggressively to produce implementations that scale far better than previous approaches. In addition, our framework enables circuits to be built and evaluated modularly, so even very complex circuits can be easily generated and evaluated.

The second reason for the poor performance of garbled circuits generated by Fairplay and similar tools is that the high-level programming abstraction they provide obscures important opportunities to generate more efficient circuits. Although this design decision stemmed from the worthy goal of providing a high-level programming interface to secure computation, it is severely detrimental to the resulting protocols' performance. In particular, (1) it automatically garbles everything in the input program, even though many parts of the program could be run without garbling since they only involve private data from one party; (2) it allocates more bits than necessary for many operations, since the programmer tends to allocate maximum number of bits needed to store all values that may be stored in a variable over its lifetime, where a circuit can be designed to use the minimum number of bits needed to represent that maximum possible value at each use; (3) it misses important opportunities to replace expensive gates with free (XOR) gates; and

	Hamming Distance		Levenshtein Distance		AES	
	Online Time	Overall Time	Overall Time	Overall Time	Online Time	Overall Online
Best Previous Result	0.310s [26]	213s [26]	534s†	92.4s‡	0.4s [13]	3.3s [13]
Our Results	0.019s	0.051s	18.4s	4.1s	0.008s*	0.2s**
Speedup Multiple	16.3	4176	29.0	22.5	50	16.5

Table I. Performance Comparison for Several Privacy-Preserving Applications.

†. Flagship protocol of [18] on 200×200 input. ‡. Purely circuit-based protocol in [18] on 100×100 input. $*$. Online-time minimizing `SubBytes` circuit design. $**$. Overall-time minimizing `SubBytes` circuit design.

(4) it misses opportunities to use special-purpose multiple input and output gates that may be more efficient for some operations than standard binary gates.

Our other main contribution is demonstrating that taking advantage of these optimization opportunities can produce order of magnitude improvements in circuit evaluation costs. Although we hope that such optimizations can eventually be done automatically by sophisticated compilers, our emphasis here is on manually producing efficient garbled circuit designs and providing a framework that makes it easy to develop and deploy such designs. We show this by producing pure garbled circuit implementations of several applications that dramatically outperform previous results, including custom-designed protocols that incorporate other techniques such as additive homomorphic encryption.

Table I summarizes our results for several privacy-preserving computational applications. Section IV presents our privacy-preserving Hamming distance protocol that is 16 times faster than the one used by SCiFI [26] (which advocated using a complex homomorphic encryption protocol to avoid the costs of garbled circuit evaluation). Sections V and VI present protocols for Levenshtein distance and Smith-Waterman genome alignment that can scale to arbitrarily large problems. Section VII presents two versions of a privacy-preserving AES protocol that are 50 times faster online and 16.5 times faster in total time, respectively, than the best previous results [13].

II. Background

In this section, we summarize the cryptographic tools used in this paper and closely related work.

Threat model. We assume a semi-honest threat model, where both parties are assumed to follow the protocol as specified but attempt to learn additional information about the other party's private inputs from the protocol transcript. Although this is a weak threat model, it is used by many previous works on secure multi-party computation (e.g, [21, 5, 32, 18, 1, 26, 13]) including the ones we use in our comparisons. This model is appropriate for some realistic situations in which both parties have a vested interest in the computation finishing correctly. Several techniques have been proposed for converting a protocol that is secure under the semi-honest model into a protocol that is secure against a malicious adversary [12, 7, 22], but it remains to be seen if this can be done efficiently. The oblivious transfer, free-XOR, and garbled row reduction techniques (described later in this section) we use are secure only assuming the existence of a random oracle [2].

Garbled circuits. Garbled circuits allow two semi-honest parties, a circuit *generator* and a circuit *evaluator*, to compute an arbitrary function $f(x_0, x_1)$, where x_0 and x_1 are private inputs from each party, without leaking any information about their respective secret inputs beyond what is revealed by the function output itself.

Any binary gate, f, which has two input wires W_0, W_1 and one output wire W_2, can be realized as a garbled gate. First, generate random nonces w_i^0 and w_i^1 to represent signal 0 and signal 1 on each wire W_i. Then, generate a truth table of four entries $\mathrm{Enc}_{w_0^{s_0}, w_1^{s_1}}(w_2^{f(s_0,s_1)})$ (where s_0, s_1 denote the 1-bit plain signal on wire W_0, W_1, respectively) and permute the table. We call this encrypted and permuted truth table a *garbled table*. The encrypted garbled tables have been blamed for the inefficiency and limited scalability of garbled circuit technique since a new table is needed for each gate evaluation.

Next, the garbled table and $w_0^{s_0}$, which represents the generator's secret input, are sent to the evaluator. To obtain the appropriate wire label for her own input (without revealing the input), $w_1^{s_1}$, the evaluator and generator execute an *oblivious transfer* protocol. Thus, the evaluator can decrypt one and only one entry that corresponds exactly to their inputs. Following this construction strategy, an arbitrary number of binary gates can be assembled to accomplish general purpose computation using the output wire labels of one gate as the input labels of the next gate.

In summary, a garbled circuit protocol involves three main steps: (1) the circuit generator garbles the circuit's truth tables; (2) the circuit generator directly transfers the circuit and garbled truth tables, and obliviously transfers the appropriate input labels to the evaluator; and (3) the circuit evaluator evaluates the circuit by successively decrypting the entry of each garbled table corresponding to the available input wires to learn the output wire labels necessary to fully evaluate a single path through the circuit.

Oblivious Transfer. An oblivious transfer protocol allows a *sender* to send one of a possible set of values to a *receiver*. The receiver selects and learns only one of the values, and

the sender cannot learn which value the receiver selected. For example, a 1-out-of-2 oblivious transfer protocol (denoted OT_1^2) allows the sender, who has two bits b_0 and b_1, to transfer b_σ to the receiver, where $\sigma \in \{0,1\}$ is kept secret to the receiver throughout the protocol. OT_1^2 was first proposed by Even, Goldreich, and Lempel [9]. Naor and Pinkas developed an efficient OT_1^2 protocol based on Decisional Diffie-Hellman (DDH) hardness assumption [25]. We use this technique in our implementation. Based on the random oracle assumption, Ishai et al. devised a novel technique to reduce the cost of doing m OT_1^2 transfers to k OT_1^2, where k, ($k \ll m$), serves as a configurable security parameter [16]. We take advantage of this technique to efficiently transfer the input wire labels in our protocols.

Related work. Fairplay [23] is a high-level language (SFDL), compiler, and interpretation (SHDL) framework working designed for constructing privacy-preserving protocols for general purpose computation. Fairplay introduced SFDL as a procedural programming language interface for programmers and SHDL as the intermediate circuit description language. In comparison, our framework offers a library of efficient circuit components to enable users to construct secure computation protocols at the circuit-level.

Fairplay used the *permute-and-encrypt* technique, which saves the evaluator from needing to use trial-and-error on every entry of a garbled truth table to identify the entry that is encrypted with the evaluator's wire labels. Our framework implementation uses this technique, and extends it to support any number of inputs for arbitrary m-to-n gates. We use this extension to perform table lookup operations with a single encryption operation for the evaluator.

The free-XOR technique was introduced by Kolensikov and Schneider [20]. This technique allows all XOR gates be executed by just XOR-ing the input wire labels, without needing any garbled table or encryption operations. Security relies on assuming the random oracle model [2]. This technique was used for privacy-preserving auctions and computing global minima [19], and we make extensive use of it in our circuit designs.

Huang et al. [15] proposed a better privacy-preserving minimum protocol that avoids the need to propagate indices by using the garbled labels obtained in circuit evaluation to execute a backtracking tree protocol and used this in a biometric identification protocol. For our applications, we focus on the core operations and do not include the oblivious retrieval that would be necessary for a complete biometric application. We could integrate this technique with our solutions to efficiently provide profile information.

Pinkas et al. [27] proposed the *garbled row reduction* technique, which reduces the size of a garbled table to three entries (saving 25% of network bandwidth) for all non-free gates and is composable the with free-XOR technique. We use this technique in our implementation. Pinkas et al. [27] also presented an optimization based on secure secret sharing over finite fields that saves 50% of network bandwidth for all kinds of binary garbled circuits. Although this optimization works without the correlation robust assumption on the key derivation function, it cannot be combined with the free-XOR technique. They tested the effectiveness of different combinations of optimization techniques by implementing a privacy-preserving AES protocol based on a SHDL file output by Fairplay. In comparison, our AES protocol is more than an order of magnitude faster than their implementation.

TASTY [13] extended Fairplay's SFDL to allow the programmer to specify where in the digital circuit to integrate some arithmetic circuits (limited to addition and constant multiplication) that are realized by homomorphic encryption schemes. They also incorporated the free XOR technique [20]. However, their approach still started from compiling enhanced SFDL programs, so the programmer does not have enough control over the circuit construction to make optimal use of free XORs as is possible with our approach.

III. Method

To build an efficient two-party secure computation, the 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 cooperatively instantiate the garbled circuit protocol. The efficiency of our approach is due to the pipelined circuit evaluation technique and several methods we use to minimize the number of non-free gates that need to be evaluated.

A. Pipelined Circuit Evaluation

The primary limitation of previous garbled circuit implementations is the memory required to store the entire circuit in memory. For example, Pinkas et al.'s privacy-preserving AES implementation involved 11,500 non-free gates [27], each of which requires a garbled table of encrypted wire values. For problems like Levenshtein distance (Section V) the size of the circuit scales with the size of the input, so only relatively small inputs can be handled.

There is no need, however, for either the circuit generator or circuit evaluator to ever hold the entire circuit in memory. The circuit generating and evaluating processes of different gates can actually be overlapped in time. In our framework, the processing of the garbled truth tables is *pipelined* to avoid the need to store the entire circuit and to save processing time. At the beginning of the evaluation, both the generator and evaluator instantiate the circuit structure, which is known to both and fairly small since it can reuse components just like a non-garbled circuit. Note that the process of generating and evaluating the circuit does not

(indeed, it cannot, because of privacy) depend on the inputs, so there is no overhead required to keep the two parties synchronized.

Our framework automates the pipelined execution, so a user only needs to construct the desired circuit. When the protocol is executed, the generator transmits garbled truth tables over the network as they are produced, in the order defined by the circuit structure. As the client receives the garbled truth tables, it associates them with the corresponding gate. The client 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 (see Section VIII for details). 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 increase the memory load on the generator or evaluator, only the network bandwidth needed to transmit the garbled tables.

B. Generating Efficient Circuits

Since pipelined execution eliminates the memory bottleneck, the cost of evaluating a garbled circuit protocol scales linearly in the number of garbled gates. 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 directly 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, we are able to take advantage of these opportunities (for example, by computing the key schedule for AES locally and transmitting it obliviously).

For the parts of the computation that need to be done cooperatively, we exploit several opportunities enabled by our framework for minimizing the number of non-free gates in our circuits.

Circuit Library. Programmer can create circuits using a library of basic circuits (comparator, adder, muxer, min, etc.) designed to make the best use of free XOR techniques. This serves as one solution to the more general goal of trading off more XOR gates, which are free, for expensive AND and OR gates. Our goals are distinguished from conventional hardware circuit design in that the latter aims to optimize circuits under a completely different set of criteria, such as total number of gates, area, and power consumption. Also, since each garbled gate can only be evaluated once, the reuse goals of hardware circuit design do not apply to garbled circuits.

Minimizing Width. Circuits are constructed with the minimal width required for the correctness of the programs. For example, if one wants to count the number of 1's in a 900-bit number, as is encountered in a face recognition application, SFDL's simplicity encourages programmers to write code that leads to a circuit that uses 10-bit accumulators throughout the computation. However, narrower accumulators are sufficient for early stages. The Hamming distance, Levenshtein distance, and Smith-Waterman applications all take advantage of this technique. For example, this reduces the cost for our Levenshtein distance protocol by about 20% (see Section V-B).

Propagating Known Values. Our framework automatically propagates known wire signals when the circuit is built. For example, given a circuit designed for Hamming distance of 1024×1024 vectors, we can immediately obtain a 900×900 Hamming distance circuit by fixing 248 of the 2048 input wires to 0. Because of the value propagation, this does not incur any evaluation cost. As another example, all the initial states (i. e., entries in the first row and the first column of the state matrix) in both the Levenshtein and Smith-Waterman protocols are filled with known values agreed upon by both parties. Hence, our circuit computes on known signals without any needing any encryption.

Fast Table Lookups. Constant lookup tables are frequently seen in real world applications (e.g., the score matrix for Smith-Waterman and the SBox for AES). In this work, we show that 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 to represent each table entry. We extend the 4-to-1 gate *permute-and-encrypt* technique [23] to work with arbitrary m-to-n gates. The advantage of this technique is the short online cost since the circuit evaluator only needs to perform a single encryption operation to lookup an entry in an arbitrarily large table. On the other hand, the circuit generator still needs to produce the entire table, so if the table entries have structure there may be more efficient alternatives if total execution time is the driving concern (see Section VII for an example).

The next four sections illustrate our approach on important privacy-preserving applications. Section VIII provides details on how our framework is implemented.

IV. Hamming Distance

Hamming distance is an essential metric with applications in myriad fields. It is a core operation in biometric identification systems [26] and m-point-SPIR (Symmetric Private Information Retrieval) [17]. Given two ℓ-bit binary strings $\boldsymbol{a}$ and $\boldsymbol{b}$, where $\boldsymbol{a} = a_{\ell-1} \cdots a_1 a_0$ and $\boldsymbol{b} = b_{\ell-1} \cdots b_1 b_0$, the Hamming distance between $\boldsymbol{a}$ and $\boldsymbol{b}$, $Hamming(\boldsymbol{a}, \boldsymbol{b})$, is simply the total number of correspondingly different bits between $\boldsymbol{a}$ and $\boldsymbol{b}$. In a privacy-preserving scenario, $\boldsymbol{a}$ is the private input of Alice and $\boldsymbol{b}$ is the private input of Bob. Alice and Bob wish to collaboratively compute $Hamming(\boldsymbol{a}, \boldsymbol{b})$, or use its value as an intermediate result in a subsequent computation, without revealing their respective private inputs to the other.

A. State of the Art

Due to the perceived inefficiency and limited scalability of garbled circuits, researchers have proposed alternate approaches to computing Hamming distances [17, 26]. Osadchy et al. used secure Hamming distance as the core computation for the SCiFI privacy-preserving face recognition system [26]. They computed the Hamming distance by a novel use of an additive homomorphic encryption system. Additive homomorphic encryption allows anyone to compute $[\![a+b]\!]$ from $[\![a]\!]$ and $[\![b]\!]$ without learning a or b ($[\![x]\!]$ denotes the encryption of a number x using a specified public key). Next, we summarize their technique.

Let $\boldsymbol{c} = c_{\ell-1}\cdots c_1 c_0$, where $c_i = a_i \oplus b_i$. Let $[\![c_i]\!]$ denote the encryption of bit c_i with Bob's public key pk_{bob}. First, Alice computes $[\![c_i]\!]$ by computing $[\![a_i]\!]\cdot[\![b_i]\!]^{1-2a_i}$, where $[\![a_i]\!]$ is computed by Alice on her own and $[\![b_i]\!]$ is received from Bob. This works because,

$$\begin{aligned}[\![c_i]\!] &= [\![a_i \oplus b_i]\!] = [\![a_i\overline{b_i} + \overline{a_i}b_i]\!] = [\![a_i(1-b_i)+(1-a_i)b_i]\!] \\ &= [\![a_i]\!]\cdot[\![b_i]\!]^{-a_i}\cdot[\![b_i]\!]^{1-a_i} = [\![a_i]\!]\cdot[\![b_i]\!]^{1-2a_i}.\end{aligned}$$

By homomorphically summing all c_i's, Bob obtains the encryption of $h = Hamming(\boldsymbol{a},\boldsymbol{b})$:

$$[\![h]\!] = \left[\!\!\left[\sum_{0\le i<\ell} c_i\right]\!\!\right] = \prod_{0\le i<\ell} [\![c_i]\!].$$

To reduce the on-line cost of the computation, SCiFI uses pre-computation techniques aggressively.

In practice, the value of the Hamming distance is rarely revealed directly. Instead, it is used obliviously in some subsequent computation. For example, in SCiFI, h is compared to a threshold value to see if it signifies a close enough match. They accomplish this using a two step protocol. First, Alice computes $[\![h+r]\!]$ and sends it to Bob, where r is a random noise added to h to prevent Bob from learning h. Bob, who is able to decrypt $[\![h+r]\!]$ using his private key, only learns $(h+r)$. Second, Bob calculates whether h is below a threshold value t using a 1-out-of-$(h_{max}+1)$ oblivious transfer protocol $\mathrm{OT}_1^{h_{max}+1}$, where h_{max} is the maximal possible value of h. For the $\mathrm{OT}_1^{h_{max}+1}$ oblivious transfer, Alice, who defines the threshold t, is the *sender*, with her $(h_{max}+1)$-bit private input $\boldsymbol{x} = x_{h_{max}}\cdots x_1 x_0$, where

$$x_i = \begin{cases} 1, & \text{if } 0 \le (i-r) \bmod (h_{max}+1) \le t; \\ 0, & \text{otherwise,}\end{cases}$$

while Bob is the *receiver*, using $(h+r) \bmod (h_{max}+1)$ as his private input choice. At the end of the oblivious transfer, Alice learns nothing and Bob learns 1 if $0 \le h \le t$, or 0 otherwise. If the party (Bob in current settings) who doesn't define t is designated as the only receiver of the final outcome, a more expensive $\mathrm{OT}_1^{2d_{max}+1}$ protocol is required.

Note that if there are a total of n records stored in the database, everything in the above two paragraphs has to be repeated n times. In SCiFI, the length of each bit string representing a face was 900 and the experimental results were derived with a database of 100 entries. The on-line time required for one Hamming distance computation was reported as 310 milliseconds, in addition to an off-line time of 213 seconds. Note that the off-line time needs to be done by the client once for each candidate face.

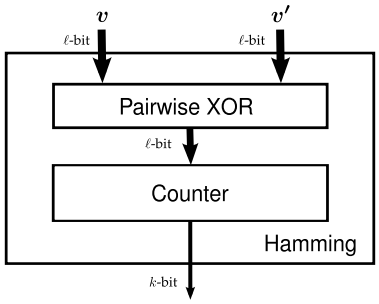

Figure 1. Hamming Distance Circuit.

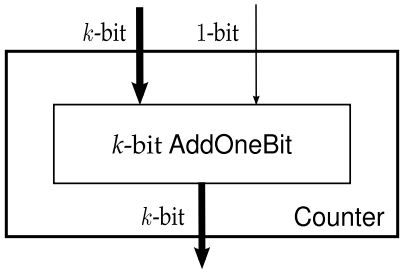

Figure 2. Core Serial Counter Circuit.

B. Circuit-Based Approach

Supporting our main argument that straightforward garbled circuit designs can solve many problems efficiently, we give a pure garbled circuit design that is much faster than the SCiFI implementation.

The high level design of a Hamming distance circuit is given in Figure 1. It is basically an ℓ-way pair-wise XOR followed by a Counter circuit that counts the number of 1 signals among its input wires. The output of the Hamming circuit is a k-bit value, where $k = \lceil \log \ell \rceil$.

A naïve design of the Counter submodule, shown in Figure 2, is to iteratively use a k-bit AddOneBit circuit that is evaluated ℓ times, where ℓ is the width of Counter. (Recall that in garbled circuits we can only evaluate each gate once, so this means ℓ copies of the circuit are needed.) In each iteration, the Counter circuit accumulates one bit of $\boldsymbol{v} \oplus \boldsymbol{v'}$ to the k-bit counter. The output signals of the AddOneBit in the i^{th} iteration are the input signals in the $i+1^{th}$ iteration.

Since XOR is free [20] and an ℓ-bit Adder needs only ℓ non-free gates [19], an ℓ-bit Hamming circuit with the naïve Counter needs $\ell\lceil \log \ell \rceil$ non-free gates. We improve the Counter design by minimizing the number of gates and enabling them to execute in parallel.

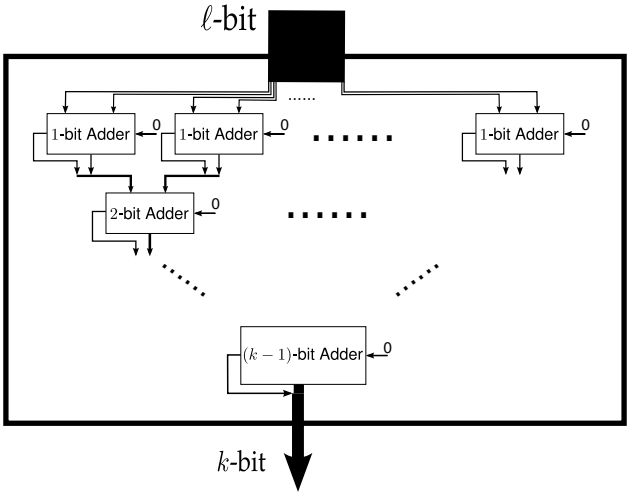

Figure 3. Parallel Counter Circuit.

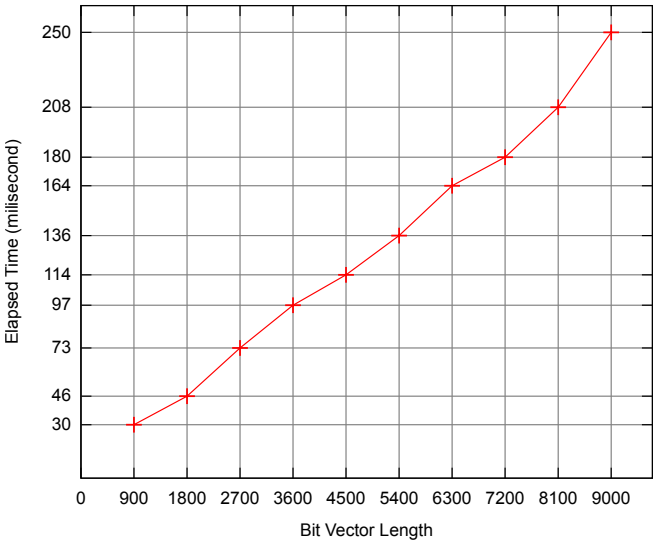

Figure 4. Scalability of our Hamming Distance Protocol.

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 $\frac{(\ell\lceil\log\ell\rceil)}{2}$ non-free gates.

Second, the serialized execution order is unnecessary. Therefore, we improved the naïve design to yield a parallel version of Counter given in Figure 3. (Our current execution framework, see Section VIII, does not support parallel execution, but is designed in a way that enables this to be readily supported in a future version.)

With this Hamming circuit component, it is fairly easy to implement either a local threshold comparison functionality by composing Hamming with a greater-than (GT) circuit, or a global minimum functionality for finding the best match by composing Hamming with a MIN circuit using the GT and MIN circuits from [15].

C. Evaluation

We implemented the Hamming distance protocol using the Java framework as described in Section VIII. Throughout this paper, we used an 80-bit nonce to represent each wire label. The security parameters used for the extended OT protocol [16] are $(80, 80)$. These settings correspond to the *ultra-short* security level defined in TASTY [13], and the same parameters are used by the comparison systems. On two Dell boxes (Intel Core Duo E8400 3GHz) connected on a LAN, computing the Hamming distance between two 1024-bit vectors takes 0.026 seconds and 56 KB bandwidth in the online phase, with 0.06 seconds spent on offline pre-processing (for the 900-bit oblivious transfer). As a comparison, SCiFI used 0.31 seconds for on-line computation, even at the cost of 213 seconds spent on pre-processing.[1] The SCiFI paper did not report bandwidth consumption, but the 900 homomorphic encryptions for each 900-bit vector will require at least $2048 \times 900/8 = 225$KB. Note that the very expensive pre-processing phase used in SCiFI's approach must be done for every candidate face. For example, it takes 38 seconds and 225KB bandwidth just to compute the 900 encryptions for a candidate face, and 71 seconds to send them to the server. The optimization techniques only help to shift this cost to pre-processing phase, but it still must be done once for every candidate face. Our circuit-based protocol does not have this problem. It takes only 0.06 seconds and 42KB on-line bandwidth cost if the circuit preparation and transmission time is counted as part of the offline preprocessing.

In addition to the dramatic improvement in performance, our approach is much more scalable than SCiFi's additive homomorphic encryption based approach. Figure 4 shows how the running time scales linearly with increasing input lengths. The circuit-based implementation has several other advantages over the homomorphic encryption approach:

- Without considering oblivious transfer, the homomorphic encryption-based protocol needs two rounds of communication; our protocol needs only a single round.
- SCiFi's Hamming distance protocol requires both parties to predict and agree on the value of h_{max}. In many applications where Hamming distance is used it may be

[1] Their experiments used a 2.8 GHz dual core Pentium D with 2GB RAM for the client machine, so although it is always difficult to make direct comparisons between different execution environments, the comparison here is reasonably close. Also note that for the SCiFI experiments they configured their host to turn off the Nagle ACK delay algorithm, which substantially improved network performance, a dominant factor in the execution time. This is not realistic for most network settings and was not done in our experiments.

difficult to predict an appropriate value, and the actual value may reveal properties of the selecting party's training data. The performance of the SCiFI protocol is severely influenced by the value of h_{max} since for every record in the database there is either an $\mathrm{OT}_1^{h_{max}+1}$ or an $\mathrm{OT}_1^{2h_{max}+1}$ phase, which for large h_{max} values is very expensive. In contrast, the running time of our protocol is independent of the value of h_{max}, although our circuit could be further optimized in cases where a small h_{max} is known. This would lend a opportunity for further reducing the cost of the garbled circuit approach since we can compute the result using much narrower adders down the tree and OR-ing the carry-out bits of all necessary gates at the end of the circuit.

- If the obliviously calculated Hamming distances are intended to be intermediate results that are used as inputs to an arbitrary computation, the garbled circuit protocol is much better in that by its nature it can be readily composed with any subsequent secure computation. However, this is very inconvenient for additive homomorphic encryption based protocols, because arbitrary operations over the encryptions are not possible.

V. Levenshtein Distance

Levenshtein distance (also known as *edit distance*) is a classic example for dynamic programming techniques [3]. It has applications in aligning DNA or protein sequences and comparing text files. Given two strings α and β, the Levenshtein distance between them (denoted $Levenshtein(\alpha,\beta)$) is defined as the minimum number of basic operations (add, delete, or replace a single character) needed to transform string α into β. The Levenshtein algorithm is given in Algorithm 1.

The invariant is that $\mathtt{D}[\mathtt{i}][\mathtt{j}]$ 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), $\mathtt{D}[\mathtt{i}][\mathtt{j}]$ is assigned at line 11 to be the smallest of three possible values, $\mathtt{D}[\mathtt{i}-1][\mathtt{j}]+1$, $\mathtt{D}[\mathtt{i}][\mathtt{j}-1]+1$, or $\mathtt{D}[\mathtt{i}-1][\mathtt{j}-1]+\mathtt{t}$ (where t is 0 if $\alpha[\mathtt{i}]=\beta[\mathtt{j}]$ and 1 if they are different). This corresponds to the three basic operations: *insert* $\alpha[\mathtt{i}]$, *delete* $\beta[\mathtt{j}]$, and *replace* $\alpha[\mathtt{i}]$ with $\beta[\mathtt{j}]$, respectively.

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
```

A. State of the Art

Jha et al. gave the best known previous implementation of a secure two-party Levenshtein distance protocol [18]. Instead of using Fairplay, they developed their own circuit compiler based on Fairplay. They borrowed the function description language SFDL and the circuit description language SHDL directly from Fairplay.

They investigated three different strategies. Their first protocol (Protocol 1) directly instantiated the Levenshtein algorithm in Algorithm 1 into an SFDL program, which was then compiled into a garbled circuit implementation. Because this approach requires 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 was 200×200 (both inputs are 200-character string from a 256-character alphabet). Their second protocol combined garbled circuits with *secure computation with shares* (SCWC) [11]. This circuit was scalable, but extremely slow. Finally, they proposed a hybrid protocol (Protocol 3) by combining first two approaches to achieve better performance with scalability.

According to their performance results, it took 92.4 seconds time and 86.7MB bandwidth for Protocol 1 to complete a problem of size 100×100, and this protocol required nearly 2GB of memory to handle the 200×200 input [18]. Their flagship protocol (Protocol 3), which is faster for larger problem sizes such as 200×200, took 658 seconds and used 364.3MB bandwidth on a problem of size 200×200.

B. Circuit-Based Approach

First, we observe that the circuit used to compute Levenshtein distance can be much smaller than the implementation produced from a high-level SFDL description. That implementation does not distinguish parts of the computation that can be done without cooperation or take advantage of knowledge about the actual input sizes at different stages.

The part of the computation responsible for initializing the matrix (lines 2–7) does not require any collaboration, hence can be completed by each party independantly. 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 so long as they keep the inner executions synchronized, leaving only the two lines (lines 10–11) of code nested in the innermost loop that need to be computed securely.

Figure 6 presents a circuit, LevenshteinCore, that is computationally equivalent to lines 10–11. For genomics

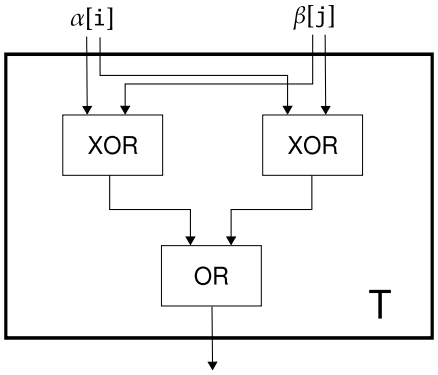

Figure 5. T circuit.

comparisons, two bits suffice to represent a nucleotide, and the t bit is set to 0 if and only if the two bits coming from α and β both match. We capture this functionality with the T circuit shown in Figure 5. For a σ-bit alphabet (e. g., an 8-bit alphabet necessary to cover the 256-character alphabet used by Jha et al. [18]), the T circuit needs $\sigma-1$ binary non-free gates in the σ-to-1 OR circuit.

The rest of the circuit corresponds to computing the minimum of the three possible edits (line 11 in Algorithm 1). We begin with the straightforward implementation shown in Figure 6. The values of $\mathtt{D[i-1][j]}$, $\mathtt{D[i][j-1]}$, and $\mathtt{D[i-1][j-1]}$ are each represented as ℓ-bit inputs to the circuit (for now, ℓ is fixed as the maximum value of any $\mathtt{D[i][j]}$ value; later we reduce this to the maximum value possible for a particular core component). Note that 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 $\mathtt{D[i][j]}$. The full Levenshtein circuit has one LevenshteinCore component for each i and j value, connected to the appropriate inputs and producing the output value $\mathtt{D[i][j]}$. 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 Levenshtein core to $5\ell+\sigma$.

First, we observe that

$$\begin{aligned} & \mathtt{min}(\mathtt{D[i-1][j]}+1,\mathtt{D[i][j-1]}+1) \\ = \;& \mathtt{min}(\mathtt{D[i-1][j]},\mathtt{D[i][j-1]})+1 \end{aligned}$$

so we can combine the two AddOneBit circuits (at the top-left of Figure 6) into a single one and interchange its relative position with the subsequent 2-MIN as shown in Figure 7.

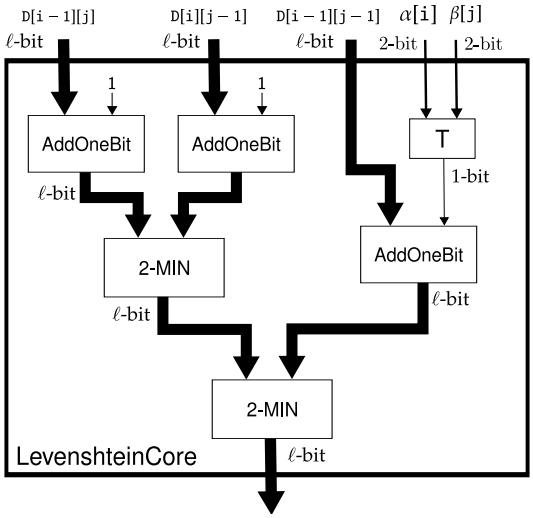

Figure 6. Naïve LevenshteinCore Circuit.

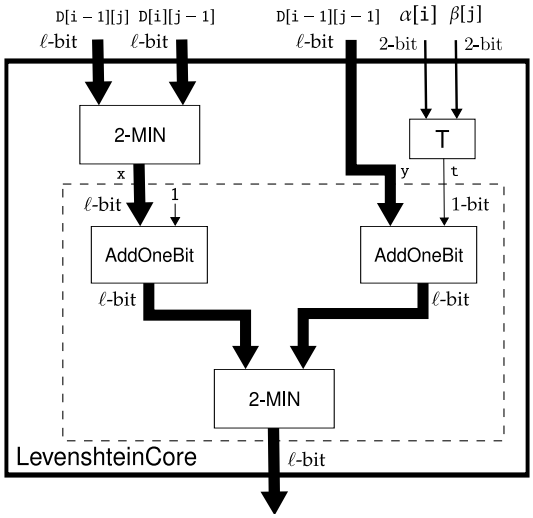

Figure 7. Better LevenshteinCore Circuit.

The circuits in the dashed box in Figure 7 compute

$$\mathtt{min}(\mathtt{x}+1,\mathtt{y}+\mathtt{t}),$$

where $\mathtt{t}\in\{0,1\}$. This is functionally equivalent to:

$$(\mathtt{y}>\mathtt{x}) \quad ? \quad \mathtt{x}+1:\mathtt{y}+\mathtt{t}$$

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 8. Note that the 1-bit output wire connecting the 2-MIN and 1-bit MUX circuits is basically 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 $\mathtt{D[i][j]}$ is not a constant. For example, one bit suffices to represent

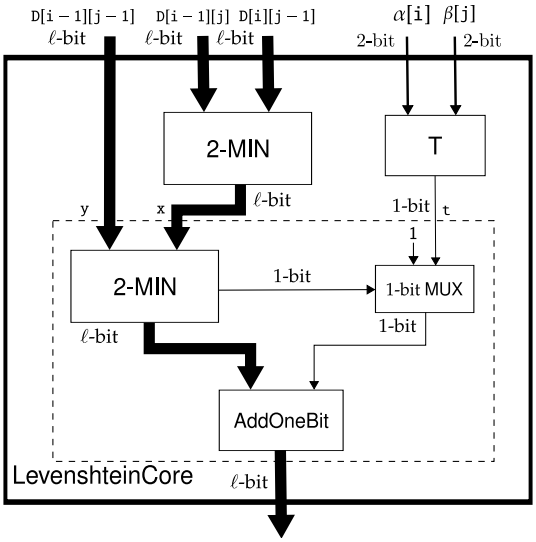

Figure 8. Final LevenshteinCore Circuit.

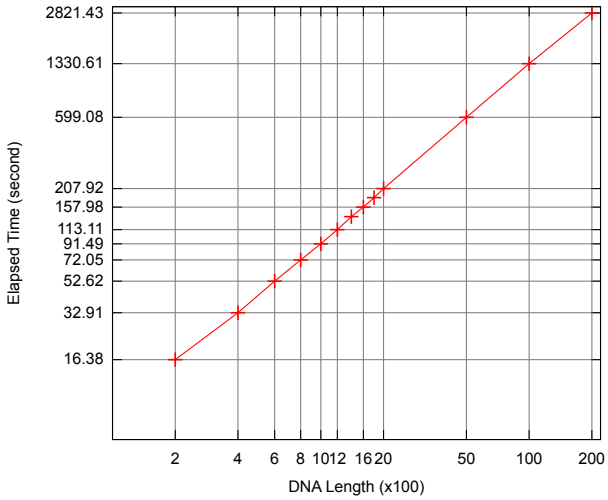

Figure 9. Scalability of Levenshtein Distance Protocol (problem size 200×DNA Length)

D[1][1] while more bits are required to represent D[i][j] for larger i's and j's. To be precise, $\lceil \log \min(\mathtt{i},\mathtt{j}) \rceil$ bits are sufficient to represent the value of D[i][j]. Over a problem instance of (M,N), the rate of savings is

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

For $M = 200, N = 200$, this savings rate is 25%, but the rate slowly decreases as M and N grow.

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 require a customized program for each input size. Hence, SFDL programs tend to allocate a constant but much more than necessary number of bits to ensure the global correctness of the protocol output.

C. Evaluation

We implemented a prototype Levenshtein protocol with our framework (Section VIII). It completes a problem of size 200×200 (4-character alphabet) in 16.38 seconds using 49MB bandwidth. Our Levenshtein distance protocol works with alphabet of any size σ. For $\sigma = 8$ as in Jha et al.'s results, our protocol takes 18.4 seconds, which is 29 times faster than their results [18].

Our approach is highly scalable, as shown in Figure 9 (of problem size $200 \times x$, where x ranges from 200 to 20 000), which invalidates the conclusion of [18]. The largest problem instance we have run is 2000×10000 (not shown in the figure), which required 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, the parties are likely to be only interested in whether a sufficiently good match (represented by a very small threshold d) could happen. In this case, the number of bits for an entry can be upper bounded by $\lceil \log d \rceil$.

VI. Smith-Waterman Genome Alignment

The Smith-Waterman algorithm is a popular method for genome and protein alignment [29, 24]. In contrast to the Levenshtein distance measuring *dissimilarity*, Smith-Waterman measures the *similarity* between two sequences (higher scores mean the sequences are more similar). The algorithm, shown in Algorithm 2, has a basic structure that is similar to the Levenshtein distance. The only differences are that the preset entries (the first row and the first column) are initialized to 0; that it computes a more sophisticated core (lines 10–12) which involves an affine gap function (gap), finding the maximum score across all previous entries in the row and column; and using a fixed 2-dimensional score matrix (score).

In practice, the gap function is typically of the form $\mathtt{gap}(x) = a + b \cdot x$ where a, b are publicly known, normally negative integer constants. By choosing a and b appropriately, biologists can account for the evolutionary likelihood of inserting large DNA segments being much greater than the likelihood of the same number of single insertions. A typical gap function is $\mathtt{gap}(x) = -12 - 7x$, 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 letters including stop symbols). The entries on the diagonal of a score matrix are larger positive numbers (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. In this paper, we use the BLOSUM62 [14] score matrix for computation over randomly generated protein sequences.

To obtain the optimal alignment using the Smith-Waterman algorithm, one first computes the 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 private information, so it may be used as an intermediate value for a continued secure computation, or just aspects of the result (e.g., the score and starting and finishing positions) could be revealed. Determining how much information leaks in the result is an important problem, but outside the scope of this paper.

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
```

A. State of the Art

The only previous attempt of which we are aware to implement Smith-Waterman securely is by Jha et al. [18]. An alternate approach, suggested by Szajda et al., is to perform the computation normally but by transforming the private sequence before disclosing it [30]. This approach can work for some scenarios, but cannot provide the privacy or correctness properties achieved by garbled circuits.

Jha et al.'s design follows a similar approach to their Levenshtein distance protocols described in Section V, and led to the conclusion that garbled circuit implementations could not handle even small inputs (their garbled-circuit implementation for Smith-Waterman could not handle 25×25 input). Hence, they adapted the hybrid protocols for Levenshtein distance to implement the Smith-Waterman protocol using secure computation with shares.

Their prototype had two limitations that make direct performance comparisons awkward: (1) Only 8 bits were used to represent each entry of the dynamic programming matrix, however, for the majority of the 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 ($\mathtt{gap}(x) = -4$) that is inappropriate for practical scenarios. Implementing a normal affine transformation in SFDL would involve another secure addition and a very expensive secure multiplication operation. Despite these simplifications, our complete Smith-Waterman implementation still runs more than twice as fast as their implementation.

B. Circuit-Based Approach

From lines 10–12 of Algorithm 2, the core functions needed to implement the Smith-Waterman algorithm are ADD and MAX. To reduce the number of non-free gates, we replace lines 10–11 with

```
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.
```

This allows us to use much narrower ADD and MAX circuits for some entries since we know the value of D[i][j] is bounded by $\lceil \log(\min(i, j) \cdot \mathit{maxscore}) \rceil$, where *maxscore* is the greatest number in the score matrix. We only need to make sure that values are appropriately sign-extended, which is 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 used in the code above, 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 is computed directly, with the output value fed directly into the ADD circuit. Being able to tightly bound the part of computation that really needs to be privacy-preserving is one of several big advantages of our approach to that of SFDL.

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. We implement the 20×20 score table as a 10-to-5 circuit, since 10 input bits are used to represent the row and column indices and 5 bits are used to represent the output score. With the extended *permute-and-encrypt* technique, the circuit generator generates symmetric encryptions of size 400×5 wire label lengths, of which the garbled truth table consists, while the circuit evaluator needs only to process an encryption of size 5 wire label lengths to

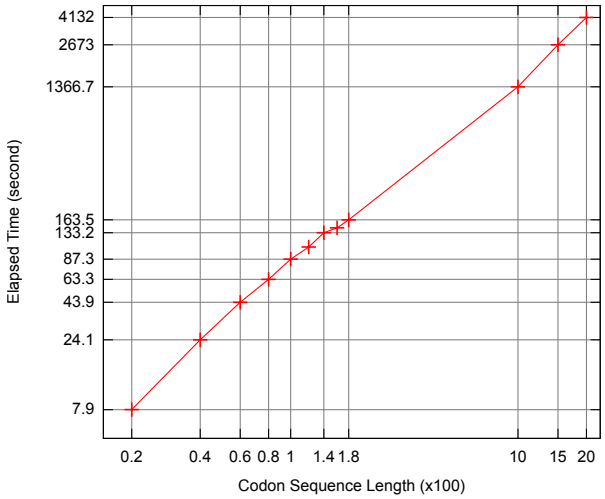

Figure 10. Scalability of Smith-Waterman Protocol (problem size 20×Codon Sequence Length)

obtain the output wire labels.[1]

C. Evaluation

Our Smith-Waterman protocol takes 447 seconds and generates 1331MB of network traffic running on two protein sequences of length 60. Jha et al.'s simplified Smith-Waterman implementation did not scale to 60×60 for the garbled circuit implementation, but for their Protocol 3 was able to complete a 60×60 in nearly 1000 seconds (but recall that the 8-bit value they used to accumulate the score almost certainly overflowed for this example, so it is not a meaningful comparison). Figure 10 shows the scalability of evaluation times in terms of the problem size ($20 \times x$, where x ranges from 20 to 180). We use the BLOSUM62 score matrix with the gap function $-12-7x$. Since the performance does not depend on the actual data (indeed is must not since this would be a privacy leak), we use random sequences for our testing.

VII. Privacy-Preserving AES

AES is a standard symmetric encryption scheme. The high level operation of the cipher is shown in Listing 1 (based on [6]). 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). A privacy-preserving AES cipher allows Alice, who has a private key, to encrypt a private message from Bob without Alice knowing the message, nor Bob learning the private key. A privacy-preserving AES primitive has a number of interesting applications such as keyword searching and blind signatures [27]. In addition, it follows immediately from the definition of privacy-preserving AES protocol that it is inherently invulnerable to side-channel attacks.

[1]The size of an entry in the garbled truth table depends on the number of bits, n, needed to represent an element of the constant lookup table. However, the number of symmetric encryptions needed to produce an entry in the garbled truth table depends both on n, and the block width of the symmetric cipher. For example, in our case, SHA-1 has a block width of 160 whereas each wire label has 80 bits. Therefore, it costs $\lceil 80n/160 \rceil = \lceil n/2 \rceil$ cipher operations for generate each entry of a garbled truth table.

Listing 1. AES Cipher.

```
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;
}
```

A. Prior Work

The best previous results on private AES were developed by Pinkas et al. [27]. Their approach implements AES encryption as an SFDL program, which is in turn compiled to a huge SHDL circuit consisting of more than 30,000 gates. Henecka et al. used the same circuit, but obtained better online performance results 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 encryption cipher block [13].

B. Our Approach

We also use garbled circuits to implement privacy-preserving AES. However, our technique is distinguished from previous ones in that instead of constructing a huge circuit, we derive our privacy-preserving protocol implementation around the structure of a traditional program, following the code Listing 1. Our guiding principle is to identify the minimal subset of the computation that needs to be privacy-preserving, and only use expensive cooperative computation for those computations.

Overview. To make the implementation simpler, we explicitly group the wire labels of every 8-bit byte into *State*, representing the intermediate results of garbled circuits, so that they can be easily manipulated and passed around in the high level program. 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.

Our first observation is that the state of the `key`, the output of the key schedule, can be obliviously transferred from Alice to Bob so that the key scheduler can be executed by Alice alone. This enables us to replace the expensive privacy-preserving key schedule computation with less expensive oblivious transfers.

Second, as in many other real world AES cipher implementations, the `SubBytes` subroutine dominates the resource (e. g., time and hardware) 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 idling periods for preprocessing.

Third, the `ShiftRows` subroutine can be safely executed independently by Alice and Bob on their own data. So nothing special is needed to make it privacy-preserving.

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

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

Minimizing Online Time. Our first design targets minimizing the online execution time by moving as much of the work as possible to the preprocessing phase. The `SubBytes` subroutine can be implemented with 16 8-to-8 garbled circuits with 256 truth table entries, similar to the score matrix used in the Smith-Waterman application (Section VI-B). From the perspective of the circuit generator (Alice), this results symmetric encryptions of size $256 \times 8 = 2048$ wire label lengths (and $256 \times 4 = 1024$ cipher operations) for every entry of the garbled truth table. However, in the circuit evaluator's (Bob) view, the cost is only 4 symmetric decryption operations over an encryption of size 8 wire label lengths.[1] 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. In most scenarios, however, the total time required to perform encrypt a block is most important. Our second design aims to minimize the total execution time by implementing `SubBytes` with an efficient circuit derived from the Wolkerstorfer et al.'s hardware circuit design [31] (see that paper for details of the mathematical derivations behind this SBox implementation strategy). The two logical components of `SubBytes` are inverse over $\mathrm{GF}(2^8)$ and affine transformation over $\mathrm{GF}(2)$. The circuit we use to compute the inverse over $\mathrm{GF}(2^8)$ is given in Figure 11. 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 its two $\mathrm{GF}(2^4)$ term representation, on which a series of operators including *inverse* over $\mathrm{GF}(2^4)$ are applied, and then mapped back to element in $\mathrm{GF}(2^8)$. In this circuit diagram, Map and Inverse Map circuits realize the bijections between $\mathrm{GF}(2^8)$ and $(\mathrm{GF}(2^4))^2$; $\oplus$ and $\otimes$ mean *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 $\mathsf{GF}(2^4)\mathsf{Inverse}$ circuits can be implemented using free XOR gates alone. Since each $\otimes$ circuit has 16 non-free gates and each $\mathsf{GF}(2^4)\mathsf{Inverse}$ 10, the total number of binary non-free gates per $\mathsf{GF}(2^8)\mathsf{Inverse}$ circuit is $16 \times 3 + 10 = 58$.

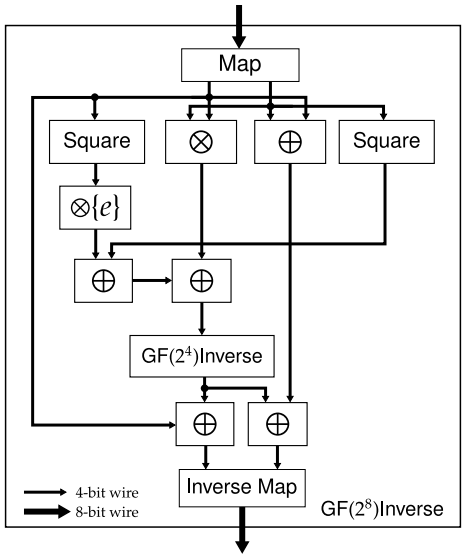

Figure 11. Inverse Circuit over $\mathrm{GF}(2^8)$.

MixColumns. The core functionality of `MixColumns` is $s'_c(x) = a(x) \otimes s_c(x)$, 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 $s'_c(x) = s'_{3,c}x^3 + s'_{2,c}x^2 + s'_{1,c}x + s'_{0,c}$. 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 $\{02\} \cdot b$ where b is an arbitrary 8-bit number, known as *xtimes*, is defined as multiplying $\{02\}$ modulo $\{1b\}$ in $\mathrm{GF}(2^8)$. 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 free XOR gates:

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

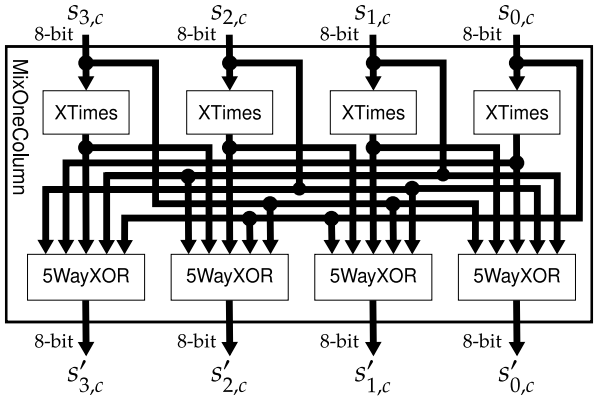

Figure 12. MixOneColumn Circuit.

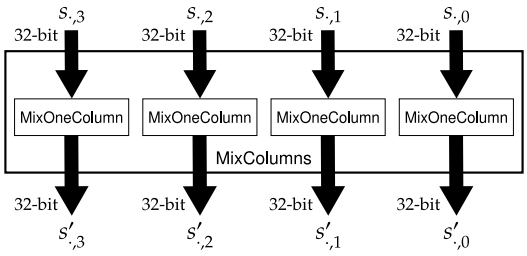

Figure 13. MixColumns Circuit.

This can be computed using one three free XOR gates.

For every column of 4-byte numbers, the equations above are implemented by the MixOneColumn circuit (Figure 12). Each invocation of `MixColumns` involves processing four columns, so we can build the MixColumns circuit by juxtaposing four MixOneColumn circuits as shown in Figure 13. Thus, the MixColumns circuit can be implemented using only free XOR gates.

C. Evaluation

Using the first (online-minimizing) SubBytes design, it is equivalent to $64 \times 16 \times 10 \times 8 = 81920$ binary non-free gates for every AES-128 cipher operation from Alice's bandwidth perspective, and $16 \times 10 \times 8 = 1280$ from Bob's bandwidth perspective.[1] The total time is 1.6 seconds without preprocessing phase. With preprocessing, the online time to evaluate the circuit is 0.008 seconds.

With our second design, the total number of binary non-free gates is $58 \times 16 \times 10 = 9280$ for both Alice and Bob. The overall time is 0.2 seconds (of which 0.08 seconds is spent on OT) without preprocessing. The online time is 0.06 seconds with preprocessing enabled.

VIII. Implementation

Our implementation uses a Java framework designed to enable programmers to define secure computations using a high-level language while providing enough control over the circuit design to enable efficient implementation. Users of our framework write a combination of high-level code and code for constructing circuits. Users are not expected to be cryptographic experts, but do need some familiarity with digital circuit design. Our framework and all the applications are available under an open source license.[2]

Figure 14 depicts a UML class diagram of the core classes of our framework. Concrete circuits are constructed using their build() method. The hierarchy of circuits is organized following the *Composite* design pattern [10] with respect to the build() method. Circuits are constructed in a highly modularized way, using Wire objects to connect them all together. The relation between Wire and Circuit follows a variation of the *Observer* pattern (a kind of publish-subscribe) [10]. The main difference is that, when a wire w is *connected* to a circuit on a *port* p (represented as a position index to the inputWires array of the circuit), all the observers of the input port wire p are automatically become observers of w.

Subclasses of the SimpleCircuit abstract class provide the functions commonly required by any binary gates such as 2-to-1 AND, OR, and XOR. The AND and OR gates are implemented using standard Yao's circuit technique, whereas the XOR gate is implemented with free XOR optimization technique [20]. Our framework automatically propagates known signals which saves the protocol run-time cost whenever any internal wires can be fixed to a known value. The binary circuits form a core part of our framework, in that users generally only need to worry about creating new concrete classes deriving from CompositeCircuit.

In order to make the users life simple when it comes to building new composite circuits, the build() method of CompositeCircuit abstract class is designed with the *Factory Method* pattern [10]. The code below shows the general structure of the build() method:

```
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.

In cases where internal wires can be fixed to known values (e. g., the carry-in port of an ADD is fixed to 0), the user might need to override fixInternalWires(). As an example, the code listing of AddOneBit circuit is given in Listing 2. Note that AddOneBit is written based solely on a (2ℓ-to-ℓ+1)-bit adder. Also, the way we write AddOneBit, it is easy to instantiate an AddOneBit circuit of arbitrary input width.

[2]URL removed following anonymity requirements; an anonymized version of our code will be made available through the PC chairs upon request.

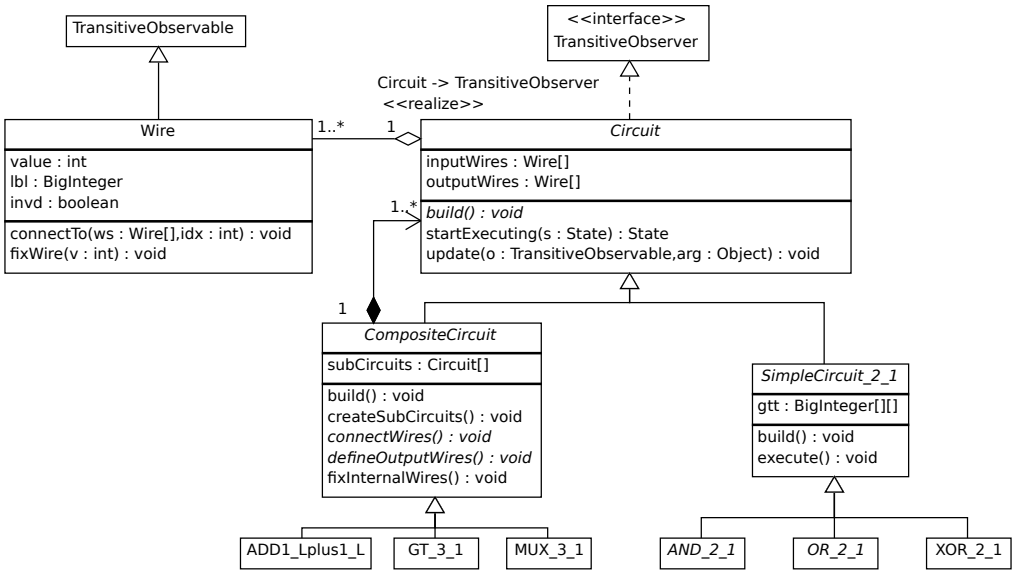

Figure 14. Core Classes in Framework.

Because of its design, our framework code base is very small. The core is implemented with about 1500 lines of Java code. The utility circuits comprise an additional 700 lines for the implementations of efficient circuits for adders, muxers, comparators, min's, max's, etc. The small size of the framework offers the possibility of providing the user with good confidence in the integrity of the protocol software implementation, since it is small enough to be reasonably audited. In view of the fact that any party involved in a privacy-preserving computation has to fully trust their end software for protecting their secrets, this small-code base property is essential in practice.

IX. Conclusion

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

Our framework enables users to take advantage of low-level circuit design to produce efficient and scalable privacy-preserving protocols. We identified several techniques for producing efficient circuits, and a framework for evaluating them efficiently, and demonstrated its impact by implementing several privacy-preserving applications and executing secure computations orders of magnitude larger than has been done before.

We hope improvements in the efficiency of privacy-preserving computing will enable many sensitive applications to be deployed with privacy. This is just a first step towards that goal, and much 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 could be computed securely in interactive systems. In addition, our work assumes the semi-honest threat model. Efficient execution under a malicious adversary model appears to be much more challenging. Finally, the secure two-party computation approach raises many deployment questions since it moves the trust burden from the other part to the software used to generate and execute the secure protocol. The small size of our framework provides some hope that necessary security properties could be validated independently, but even for small programs this remains a challenge.

References

[1] M. Barni, T. Bianchi, D. Catalano, M. D. Raimondo, R. D. Labati, P. Faillia, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, and A. Piva. Privacy-preserving Fingercode Authentication. In *ACM Multimedia and Security Workshop*, 2010.

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

[3] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS*, 2008.

[5] J. Brickell and V. Shmatikov. Privacy-preserving Graph Algorithms in the Semi-honest Model. In *ASIACRYPT*, 2005.

[6] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[7] I. Damgård and C. Orlandi. Multiparty Computation for Dishonest Majority: from Passive to Active Security at Low Cost. *eprint.iacr.org*, 2010.

Listing 2. Source Code for AddOneBit Circuit.

```
class AddOneBit_Lplus1_L extends CompositeCircuit {
  private final int L;

  public AddOneBit_Lplus1_L(int m) {
   super(m + 1, m, 1, "AddOneBit_Lplus1_L");
   L = m;
  }

  protected void createSubCircuits() {
   subCircuits[0] = new ADD_2L_Lplus1(L);
   super.createSubCircuits();
  }

  protected void connectWires() {
   inputWires[0].connectTo(subCircuits[0].inputWires, 0);
   for (int i = 0; i < L; i++)
    inputWires[i+1].connectTo(subCircuits[0].inputWires,
                              2*i + 1);
  }

  protected void defineOutputWires() {
   System.arraycopy(subCircuits[0].outputWires, 0,
                    outputWires, 0, L);
  }

  protected void fixInternalWires() {
   for (int i = 1; i < L; i++)
    subCircuits[0].inputWires[2*i].fixWire(0);
  }
}
```


[8] 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.

[9] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 1985.

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

[11] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. In *ACM Symposium on Theory of Computing*, 1987.

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

[14] 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.

[15] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient Privacy-Preserving Biometric Identification. In *NDSS*, 2011.

[16] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, 2003.

[17] A. Jarrous and B. Pinkas. Secure Hamming Distance Based Computation and Its Applications. In *Applied Cryptography and Network Security*, 2009.

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

[19] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *International Conference on Cryptology and Network Security*, 2009.

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

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

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

[23] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — A Secure Two-Party Computation System. In *USENIX Security Symposium*, 2004.

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

[25] M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *ACM-SIAM Symposium on Discrete Algorithms*, 2001.

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

[27] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, 2009.

[28] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient Privacy-Preserving Face Recognition. In *International Conference on Information Security and Cryptology*, 2009.

[29] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147, 1981.

[30] D. Szajda, M. Pohl, J. Owen, and B. Lawson. Toward A Practical Data Privacy Scheme for A Distributed Implementation of the Smith-Waterman Genome Sequence Comparison Algorithm. In *NDSS*, 2006.

[31] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC Implementation of the AES SBoxes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, 2002.

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

[33] A. C. Yao. Protocols for Secure Computations. In *Symposium on Foundations of Computer Science*, 1982.

[34] A. C. Yao. How to Generate and Exchange Secrets. In *Symposium on Foundations of Computer Science*, 1986.