CompGC: Efficient Offline/Online Semi-honest Two-party Computation

Adam Groce
Reed College
agroce@reed.edu

Alex Ledger*
MIT Lincoln Laboratory
alex.ledger@ll.mit.edu

Alex J. Malozemoff†
Galois
amaloz@galois.com

Arkady Yerukhimovich
MIT Lincoln Laboratory
arkady@ll.mit.edu

Abstract

We introduce a new technique, *component-based garbled circuits*, for increasing the efficiency of secure two-party computation in the offline/online semi-honest setting. We observe that real-world functions are generally constructed in a modular way, comprising many standard components for common tasks like arithmetic or cryptographic operations. Our technique allows circuits for these common tasks to be garbled and shared during an offline phase; once the function to compute is specified, these pre-shared components can be chained together to create a larger garbled circuit. We stress that we do *not* assume that the function is known during the offline phase — only that it uses some common, predictable components.

To measure the efficiency gains of this technique, we give two implementations: an optimized library, libgarble, for standard garbled circuits, and an implementation, CompGC, of component-based garbled circuits. Using these, we compare the online costs of garbled circuit-based secure two-party computation using the standard and component-based approaches for various examples, including several classes of machine learning classification. We find that our technique results in up to an order of magnitude online performance improvement for standard computations over a realistic network setup compared to prior work.

1 Introduction

Secure two-party computation allows a pair of parties, each with private input, to compute a function of those inputs without sharing them with each other. This is an extremely powerful tool, and it was shown by Yao to be feasible using an approach termed *garbled circuits* [Yao86]. Since then, a long line of work has aimed to increase the efficiency of garbled circuit-based secure computation. This paper continues that effort.

In particular, our goal is to allow the use of offline pre-processing to significantly reduce online computation time for garbled circuit-based computation. This is not a new goal. Beaver, for example, showed how precomputation can significantly increase the online speed of the required oblivious transfers (OTs) [Bea95]. Others have found similar ways to increase the online efficiency of the cut-and-choose technique needed for malicious security [HKK+14, LR14, LR15]. There is also a long history of precomputation in the setting of non-garbled circuit-based two-party computation [DPSZ12, NNOB12].

In the semi-honest setting in which all of our constructions work, it has long been known that precomputation can greatly increase efficiency *if the function is known ahead of time*, with only the inputs specified at the time of online computation. The protocol is simple: the garbler computes the entire garbled circuit ahead of time, with only OT computations (which can also be preprocessed, but still require some online

*Portion of work done while at Reed College.

†Portion of work done while at University of Maryland.

communication), communication of the inputs, and evaluation done online. However, requiring that the function be known ahead of time is a substantial limitation.

In this work, we show a way to achieve a similar benefit *without* prior knowledge of what circuit will be computed. Towards this goal, we note that most functions of interest are built in a modular way. Just as a programmer writes code for a complex function by using existing simpler functions, the circuits for these functions use components that perform common tasks. There might be a portion of the circuit that takes the maximum of two numbers, for example, or that computes a hash function. We show that one can precompute garbled circuits for these smaller components and then chain them together in the online phase when the function to be computed is specified. We call this *component-based* garbled circuit construction, and we provide an open-source implementation, CompGC, that achieves up to an order of magnitude improvement in online computation time for standard benchmarks over a real network, versus standard garbled circuit-based secure two-party computation.

We can imagine this system being used in several different ways. In the most narrow case, parties may know roughly what sort of function will be computed. For example, they might be unsure *only* of the input length. In this case, they can compute a narrow set of components specifically tailored to that function. This incurs slightly increased total computation cost in exchange for greatly improved online speed.

In a more general setting, parties might engage frequently in computation of a given general type. A library of common operations might be developed for that particular type of computation. Cryptographic functions, for example, commonly rely on a small set of operations, including large components like those for computing standard hash functions and blockciphers and smaller components for simple tasks like bitwise XOR of two strings. Geometric computations, on the other hand, might require a large number of matrix operations. Other libraries could be developed for computations in machine learning, finance, or other general areas, or specifically tuned to the needs of a larger application of which the secure computation was part.

We note that in the use cases discussed above, substantial storage would be required. There would also be significant setup cost. However, components in our scheme that are not used for one computation can be saved for the next. That means that the component library that the parties have precomputed can be maintained simply by replacing used components. As a result, the *amortized* total cost of each computation is not greatly increased, and latency is drastically reduced.

1.1 Our Contributions

Our contributions go well beyond pointing out the ability to divide a circuit into pieces. In particular, we provide a formal specification for how to create and connect components, give a practical, open-source implementation, and show experimentally that our method allows for drastically reduced online computation. Specifically, we make the following contributions.

Component-based garbled circuits. We give a protocol for precomputing garbled circuits for given components, and for combining these components as needed at runtime. We show that security is maintained by this protocol. This construction allows arbitrary linkage between component wires while requiring online communication of only *one* label per component input wire. This drastically reduces the need for online communication.

Implementations. We develop our own standalone library libgarble[1] for garbling circuits. Our library is based on the JustGarble implementation of Bellare et al. [BHKR13], but makes many internal improvements to the codebase. None of these constitute theoretical improvements to the underlying algorithm, but rather optimizations of the code. For example, we revise the data structure by which circuits are stored in order to speed access to certain data. We believe this is a valuable contribution on its own, and it is relevant even when not using our component-based precomputation strategy. Our library improves the performance of garbling and evaluating an AES circuit by 10% and 22%, respectively, as compared to JustGarble, along with many other improvements, including support for half-gates [ZRE15] and privacy-free garbled circuits [FNO15]

[1] `https://github.com/amaloz/libgarble`

alongside a consistent API. In subsequent work, libgarble was used to implement very efficient maliciously secure two-party computation protocol [WMK16].

We then use libgarble as a building block to create a complete secure computation system, CompGC[2]. This tool allows parties to precompute any specified library of components during the offline phase, using libgarble to garble each component. During the online phase, it creates a series of instructions for the evaluator that allows the chaining of the relevant components, and it handles the extra computation (outside of garbling and evaluation) that is required to distribute the input wire labels and decipher the output wire labels.

Experimental results. We use this implementation to conduct several experiments. We first compare to prior work on garbled circuits. Specifically, we consider three computations: (1) computing AES using a single-round AES component as a building block; (2) using this single-round AES component to allow for encryption of arbitrary length messages using CBC mode; and (3) computing Levenshtein distance, which can be used for any number of applications, including text processing and genetic analysis. Computations (2) and (3) above model a setting where only the input length is unknown during precomputation. Computations (1) and (2) also model a setting where a library of standard cryptographic components is used.

We measure total online time required to perform the secure computation over a simulated but realistic network configuration. In all of these measurements, we see substantial efficiency improvements due to precomputation. For example, when computing Levenshtein distance between two 60 symbol strings, where each symbol comes from an 8-bit alphabet, we see almost an order of magnitude improvement (from 3.9 seconds to 408 milliseconds) when using our approach over the naive approach of sending the entire circuit online.

Next, we consider a broad class of machine learning classification computations first considered by Bost et al. [BPTG15]. This is an ideal setting for a library that is limited and tailored to a specific application domain, but which can nevertheless carry out a substantial variety of computations. As Bost et al. observe, these computations can all be broken down into a small number of simple components. With this observation we measure the total and online time for garbled circuit evaluation for these computations. Surprisingly, we find that the naive garbled circuit approach using libgarble is already competitive with the performance of the special-purpose protocols of Bost et al. and is dramatically faster in the case of decision tree classifiers. Furthermore, using CompGC, we obtain significant further improvements in the online time for all of these computations, beating the runtime of Bost et al. in all cases.

All of our work is done in the *semi-honest* model. We believe there are many use cases of secure computation for which semi-honest security is sufficient. For example, when two mutually trusting companies or agencies are prevented from sharing data by policy or legal restrictions, but otherwise trust each other to behave honestly. We also view semi-honest security as a natural stepping stone, and we expect these techniques can, with additional work, be extended to the malicious setting as well.

1.2 Paper Organization

The remainder of this paper is organized as follows. Section 2 summarizes the related prior work. Section 3 provides background information on garbled circuits and secure two-party computation, introducing the necessary notation that we use in the remainder of the paper. Section 4 describes our component-based garbled circuit technique. Section 5 provides the details on our prototype implementation of the described primitives and Section 6 gives the experimental results evaluating the performance of our schemes for several common classes of functions.

2 Related Work

Garbled circuits were first introduced by Yao [Yao86] as a tool for general secure two-party computation. While they were originally viewed mainly as a theoretical tool, this view has changed significantly over the past decade or so. Starting with the Fairplay system of Malkhi et al. [MNPS04], garbled circuits have been

[2] `https://github.com/aled1027/CompGC`

built into prototypes of secure computation. This has led to a long line of work (e.g., [BHKR13, HKS+10, HEKM11, KsS12, LR15, Mal11, MGBF14, PSSW09, SHS+15, WMK16, NST16]) that aims to improve the efficiency of garbled circuits and to build usable and practical systems for various real-world applications. Out of this work, the most efficient known implementations of general garbled circuit-based computation are TinyGarble [SHS+15] for security against semi-honest adversaries, which is based on the efficient garbling procedure introduced by JustGarble [BHKR13], and the approach of Rindal and Rosulek [RR16] (in the offline/online model) and Wang et al. [WMK16] (in the standalone setting) for malicious adversaries.

In attempting to increase the online efficiency of secure computation, we are guided by many prior works that identified as a major bottleneck the time and bandwidth necessary to transmit the garbled circuit to the evaluator. Several works [KMR14, KS08, NPS99, PSSW09, ZRE15] aim to reduce the size of the circuit that must be communicated between the generator and evaluator. We see this paper as continuing this effort, aiming to reduce the amount of communication necessary in the online phase of garbled circuit evaluation. While we do not further reduce the overall size of the garbled circuit to be transmitted, we significantly reduce the amount of communication necessary in the online phase, after the function to compute and the inputs are chosen. As communication is the main bottleneck, Gueron et al. [GLNP15] argue that the speed improvements made by JustGarble disappear due to the need to transmit the circuit. Because we send the circuit components in the *offline* phase, communication is no longer the bottleneck and we can thus reap all the performance benefits of using a JustGarble-based garbling library.

The idea of breaking circuits into smaller pieces appeared previously in the work of Mood et al. [MGBF14], where it was called "partial garbled circuits," though with a very different purpose. Rather than use it to reduce online computation and communication time as we do here, Mood et al. used it as a way to reuse values in internal gates of a garbled circuit across multiple computations. Their technique also requires sending *two* correction labels per wire, whereas we can do it with just *one*. Additionally, several prior works using the "LEGO" approach build garbled circuits [NO09, FJN+13, FJNT15, NST16] out of pre-garbled NAND gates but do not consider larger components.

Secure computation of machine learning classifiers. A classifier (or *model*) is the output of a machine learning technique when applied to a training data set. The resulting classifier is capable of labeling new data points. For example, a hospital might have a classifier that can indicate whether a particular patient's medical record is indicative of a given disease. Because that classifier was constructed from private patient data (or in other settings, because it is proprietary) it cannot be released publicly. Similarly, a patient might not want to freely transmit their medical records. Securely computing classifier functions allows the patient to learn the result of the classifier without either the model or the patient's medical records being shared.

There are several works (e.g., [BFK+09, EFG+09]) that show how to securely compute specific classifiers, but the only work that shows how to compute a broad set of classifiers is that of Bost et al. [BPTG15]. Bost et al. show that three component operations (comparison, argmax, and dot product) are sufficient for computing hyperplane decision, naive Bayes, and decision tree classifiers. Because many different machine learning algorithms use classifiers of one of these three types, this means that three component operations are sufficient for carrying out secure classification using a wide range of underlying machine learning algorithms. Bost et al. show tailored protocols for secure computation of these components and therefore of the classifiers, which they claim to be more efficient than "general" garbled circuit-based computation.

3 Preliminaries

In this section we briefly introduce the notation and key primitives that we use, as well as some background.

3.1 Garbled Circuits

Garbled circuits are the main tool used for all of our constructions. Our presentation here follows [GLNP15, LR14] which is adapted from [BHR12], and we refer the reader to those works for a more detailed presentation.

Garbled circuits, proposed originally by Yao [Yao86], are a way of encoding a Boolean circuit that allow for secure evaluation of the function computed by that circuit. This encoding has the property that given

w_0 label	w_1 label	w_{out} label	garbled table entry
W_0^0	W_1^0	W_{out}^0	$\mathsf{Enc}_{W_0^0}(\mathsf{Enc}_{W_1^0}(W_{out}^0))$
W_0^0	W_1^1	W_{out}^0	$\mathsf{Enc}_{W_0^0}(\mathsf{Enc}_{W_1^1}(W_{out}^0))$
W_0^1	W_1^0	W_{out}^0	$\mathsf{Enc}_{W_0^1}(\mathsf{Enc}_{W_1^0}(W_{out}^0))$
W_0^1	W_1^1	W_{out}^1	$\mathsf{Enc}_{W_0^1}(\mathsf{Enc}_{W_1^1}(W_{out}^1))$

Table 1: Garbled AND Gate. Only the values in the last column are sent to the evaluator. If the input wires have values a and b, then the evaluator knows W_0^a and W_1^b and can therefore decrypt $W_{out}^{a \wedge b}$.

encodings of values for each input wire, it is possible to evaluate the function computed by this circuit (i.e., learn the values of the output wires) without learning the values of the input wires or any of the internal circuit wires. This enables two-party secure computation where one party produces the garbled circuit and the input labels, and the other party evaluates the circuit to produce the output. This is described in more detail in Section 3.3.

More formally, a *garbling scheme* consists of two algorithms (Garble, Eval). On input a security parameter 1^κ and a circuit C, $\textsc{Garble}(1^\kappa, C)$ returns the triple (GC, e, d) where GC is the garbled circuit, e is the ordered set of input wire labels $\{(W_i^0, W_i^1)\}_{i \in \mathsf{Inputs}(C)}$, and d is the ordered set of output labels $\{(W_i^0, W_i^1)\}_{i \in \mathsf{Outputs}(C)}$.

Given a garbled circuit GC and a set of input labels $X = \{W_i^{x_i}\}_{i \in \mathsf{Inputs}(C)}$, $\textsc{Eval}(GC, X)$ computes the garbled output Z such that using the set d, it is possible to recover the actual output z (i.e., by finding Z in the ordered set of output labels).

Example. The most straightforward example of a garbled circuit is Yao's original scheme. Each wire w_i has two associated labels, W_i^0 and W_i^1, corresponding to values 0 and 1 respectively. For each gate there is a table like Table 1. This table contains encryptions of the labels for the gate's output wire, using the labels of the input wires as keys. The encryptions are chosen so that the evaluator, knowing the labels of the two input wires, can decrypt the proper label of the output wire (and nothing else). Repeated evaluation of gates then propagates knowledge of the correct wire labels (for whatever initial input labels were given) through the entire circuit.

Privacy. In order to be useful for secure two-party computation, it is necessary that garbled circuits satisfy the following *privacy* notion. The values seen by the evaluator, GC, d, and X, should not reveal any information about x that is not revealed by the output $C(x)$. Formally, we require that there exist a polynomial time simulator S that on input $(1^\kappa, C, C(x))$ outputs a simulated garbled circuit that is indistinguishable from (GC, e, d) generated by Garble. Since S knows $C(x)$ but not x, this captures the fact that the output of Garble does not reveal anything (else) about x.

Free-XOR. Our constructions make use of one critical improvement to the original garbled circuits called free-XOR [KS08], which allows for XOR gates to be evaluated "for free" without requiring any garbled tables to be included in the garbled circuit. Specifically, this technique works by choosing a global random value R and then ensuring that the labels for all circuit wires have a difference of R. That is, for any wire w_i, $W_i^0 \oplus W_i^1 = R$. This enables the secure evaluation of an XOR gate by simply computing the XOR of the two incoming labels, as R cancels out appropriately.

3.2 Oblivious Transfer

Another key component for secure two-party computation is *oblivious transfer* (OT) [EGL82, Rab05]. OT is a two-party primitive where one party (the sender) has as input two κ-bit strings (m_0, m_1) and the other party (the receiver) has a bit b. OT enables the receiver to receive m_b from the sender, while preventing the sender from learning which string was received (the value of b) and preventing the receiver from learning anything about m_{1-b}. In this paper we use the semi-honest OT construction by Naor and Pinkas [NP99].

One technique for optimizing OT that we make critical use of is OT preprocessing [Bea95]. OT preprocessing allows splitting any OT protocol into an expensive offline phase and a much cheaper online phase. Specifically, in the offline phase, before the inputs are known, OT is performed on random inputs for both the sender and receiver. This requires a number of expensive cryptographic operations. However, in the online phase the pre-OT'd values are used to perform the OT on the parties' actual inputs without needing any additional expensive operations.

3.3 Secure Two-party Computation

We now briefly describe how garbled circuits and oblivious transfer can be used to realize secure two-party computation. That is, to enable two parties to compute a joint function on their inputs without either party learning more than what is implied by its input and output. In this work we focus on two-party computation that is secure against a *semi-honest* adversary corrupting either of the two parties. That is, such an adversary follows the protocol as specified, but attempts to learn extra information from its interactions. For a formal treatment of the security of two-party computation we refer readers to the book by Goldreich [Gol09].

In garbled circuit-based two-party computation of circuit C, we identify the two parties as the *garbler* who has input x and the *evaluator* who has input y. The garbler first runs $\textsc{Garble}(C)$ to produce (GC, e, d). He then sends GC and an encrypted form of d to the evaluator together with the wire labels corresponding to the bits of the garbler's input x. The encrypted form D of d corresponds to a random permutation of $\{\mathsf{Enc}_{W_i^0}(0), \mathsf{Enc}_{W_i^1}(1)\}_{i \in \mathsf{Outputs}(C)}$.

Now, for each bit of the evaluator's input y, the garbler and evaluator run an OT protocol by which the evaluator learns the appropriate wire label (without revealing that bit of y to the garbler). Now, the evaluator has all the inputs to run $\textsc{Eval}(GC, X)$ to recover the output wire labels. It then uses these wire labels to decrypt the entries in D to learn the appropriate output. If output by both parties is desired, the evaluator can send this output to the garbler.

4 Component-Based Garbled Circuits

We introduce the concept of component-based garbled circuits to allow for much of the work involved in building and transmitting a garbled circuit to be done in an offline phase before the inputs or even the function to compute are known. This allows us to significantly improve the online performance of secure two-party computation schemes using garbled circuits. Our improvements stem from the observation that a common way to build circuits (and programs) is to compose them out of common building blocks or components. For example, common components such as circuits for arithmetic operations, cryptographic functions, and text processing can form the base for large classes of general computation.

We show how to take advantage of such common components for designing efficient garbled circuits. Specifically, our approach is to pre-garble a large number of common component circuits in an offline phase. Note that we do not need to know the computation to be performed (besides the generic components used to create said computation) or the inputs during this offline phase. Then, in an efficient online phase, we show how to *link* these components to form the actual circuit we wish to compute. We only need to send a single wire label for each of the input wires in each component. Even if components are all single gates, this corresponds to sending only *one* label per wire, which is half the size of the best known garbled circuit construction [ZRE15]. However, components will rarely be a single gate. We believe that in many applications (including those used in our experiments) circuits will use many large components, and all wires internal to a given component require no communication at all. Since the time to communicate the garbled circuit is the major bottleneck, this leads to significant savings in the overall garbled circuit computation; see Section 6 for details.

More technically, a component-based garbling scheme is a triple of algorithms $(\textsc{Garble}, \textsc{Link}, \textsc{Eval})$. $\textsc{Garble}$ and $\textsc{Eval}$ are variants on the corresponding methods for standard garbled circuits, while $\textsc{Link}$ is new.

Garble. The GARBLE procedure is unchanged, but now is given a component c as input (in place of a complete circuit C). GARBLE(c) outputs the garbled component GC_c, input wire set e_c, and output wire set d_c, for this component.

Link. On input two garbled components $c_0 = (GC_0, e_0, d_0)$ and $c_1 = (GC_1, e_1, d_1)$ as well as a mapping of output wires of c_0 to input wires of c_1, LINK produces the *link* labels needed to convert from c_0 output wires to c_1 input wires. Specifically, suppose that output wire w_i of c_0 has labels (W_i^0, W_i^1) and input wire w_j of c_1 has labels $(\overline{W}_j^0, \overline{W}_j^1)$. Then, to link these two wires, LINK outputs $W_{ij} = W_i^0 \oplus \overline{W}_j^0$. Note that since we use the free-XOR optimization, we know that both $W_i^0 = W_i^1 \oplus R$ and $\overline{W}_j^0 = \overline{W}_j^1 \oplus R$ for some random value R. Therefore, we have that $W_i^0 \oplus \overline{W}_j^0 = W_i^1 \oplus \overline{W}_j^1$, so a single label W_{ij} is sufficient to connect both the zero and the one wire labels. This allows us to reduce the communication necessary to one label per component wire.

Eval. On input a list of garbled components $\{c_i\}$ and linking labels $\{W_{ij}\}$, EVAL computes the garbled outputs $\{Y_i\}$ as follows. Starting from the inputs, EVAL proceeds component by component, evaluating each component to get the component output wire labels. When appropriate, it uses these component output wire labels together with the appropriate link labels to recover the input labels for later components. Finally, once all the components are evaluated, EVAL recovers the garbled outputs $\{Y_i\}$ from the output components and uses d for that component to recover the (real) output y.

For details on the exact garbling scheme used to garble the components and several further optimization improvements, we refer the reader to the implementation details in Section 5.

Privacy. We now show how to adapt the standard privacy definition for garbled circuits [BHR12] to the component-based setting. Specifically, for a set of components $\{c_i\}_{i \in \mathsf{Components}}$, we want that the pre-garbled components $\{GC_i\}$, together with the input labels $\{W_j^{x_j}\}_{j \in \mathsf{Inputs}(C)}$, and the output map $d_{C_{out}}$ as well as all the link labels $\{W_{ij}\}_{i,j \in \mathsf{Components}}$ do not reveal any information about x. Formally, as in the case of garbled circuits, we require that there exist a polynomial time simulator $\mathcal{S}$ that on input $(1^\kappa, C, C(x))$, where $C(\cdot)$ is some polynomial size circuit, outputs simulated component garbled circuits for all components in C, input and output labels, as well as all the linking labels W_{ij} for linking all necessary wires that are indistinguishable from $(\{GC\}_i, e_{\mathsf{Input}(C)}, d_{\mathsf{Output}(C)})$ and W_{ij} generated by the real GARBLE and LINK procedures. Formally, security is captured by the following game:

The privacy experiment $\mathsf{Expt}^{\mathsf{priv}}_{\mathcal{A},\mathcal{S}}(\kappa)$:

1. Invoke adversary $\mathcal{A}$: compute $(C, x) \leftarrow \mathcal{A}(1^\kappa)$.
2. Choose a random $b \in_R \{0, 1\}$.
3. If $b = 0$: For each component c_i in C, compute $(GC_i, e_i, d_i) \leftarrow$ GARBLE$(1^\kappa, c)$. Additionally, for each pair of components (c_i, c_j) that need to be linked, compute all the link labels $\{W_{ij}\} \leftarrow$ LINK(c_i, c_j). Finally, compute input labels $X = \{W_i^{x_i}\}_{i \in \mathsf{Inputs}(C)}$ and output map $d_{\mathsf{Output}(C)}$. Then output challenge $\tau = (\{GC_i\}, \{W_{ij}\}, X, d_{\mathsf{Output}(C)})$.
   If $b = 1$: Compute $\tau \leftarrow \mathcal{S}(1^\kappa, C, C(x))$.
4. Give $\mathcal{A}$ the challenge τ and obtain a guess
   $b' \leftarrow \mathcal{A}(\tau)$.
5. Output 1 if and only if $b' = b$.

Definition 1. *A component-based garbled circuit scheme achieves* privacy *if for every probabilistic polynomial time A there exists a probabilistic polynomial time simulator S and a negligible function $\mu(\cdot)$ such that for every $\kappa \in$ N:*

$$\Pr\left[\mathsf{Expt}^{\mathsf{priv}}_{\mathcal{A},\mathcal{S}}(\kappa) = 1\right] \leq \frac{1}{2} + \mu(\kappa)$$

4.1 Component-Based Secure Two-Party Computation

We now briefly describe how to use component-based garbled circuits for secure two-party computation. In an offline stage, before inputs or even the computation to be performed are known, the garbler runs GARBLE on a number of components to pre-garble these components; it then sends $\{GC_i\}_{i \in \mathsf{Components}}$ and an encrypted form D of $d_{\mathsf{Output}(C)}$ (as specified in Section 3.3) to the evaluator. These components are circuit building blocks that comprise the eventual computation; however, their exact linking is not determined at this time. In parallel, the garbler and evaluator preprocess a number of instances of OT. Both the garbler and the evaluator store the received garbled components and preprocessed OTs.

When the function f to compute and the inputs (x, y) are known[3], the garbler assembles the circuit C out of the garbled components $\{c_i\}$. For each component pair that needs to be linked, the garbler runs LINK(c_i, c_j) and sends the link labels W_{ij} to the evaluator. Additionally, the garbler sends the input labels $\{W_i^{x_i}\}$ for the garbler's inputs. Finally, the garbler and evaluator complete the online phase of the OTs to retrieve the labels $\{W_i^{y_i}\}$ for the evaluator's input. Given this information, the evaluator runs EVAL to compute the circuit.

4.2 Analysis

To analyze the performance of component-based 2PC, we look separately at the online and offline phases. In the offline phase the garbling and transmission of garbled components is similar to the total communication normally done to garble and send a circuit. However, this communication can be done offline thus not affecting the online running time. The online phase, on the other hand, only sends one link label per pair of wires connecting any components. So, in total, the online communication necessary is just one label for each component input wire. We note that, even in the case when components are just single gates, this still enables us to achieve communication of one label per gate (and XOR gates remain free). This is 50% savings over the best known construction [ZRE15]. In the more realistic case, where components are substantially larger, the savings can be much greater.

This analysis assumes that the same circuit is used in both cases. In reality, requiring circuits be built of pre-made components will change their structure. Component-based construction limits global circuit optimization because components must be treated as impermeable black boxes. However, doing careful circuit optimization for each function is hard to begin with. It is an expensive computation, and the time to optimize the circuit counteracts (or even eliminates) the efficiency gains that optimization would provide. When building a library of components, one can very carefully optimize each component, since the optimization is an offline computation and is amortized over many uses of each component. As a result, the *feasible* amount of circuit optimization might be substantially higher with the component-based approach.

4.3 Security

We now sketch a proof of security for our offline/online construction. Roughly, what we need to prove is that the added linking labels do not break the security of the original garbled circuit construction. More formally, we need to show a simulator that, given the output y, is able to generate simulated garbled components and linking labels that would look indistinguishable from the true garbled circuit.

We must consider the view of each party, where the "view" includes any messages received during the protocol. (Values computed and sent by a party themselves cannot give them additional information.) First we note that the view of the garbler in this construction only consists of its side of the OT protocol executions. This is the same as its view in the standard garbled circuit protocol, so no additional security argument is needed.

Next we consider security against a semi-honest evaluator. Roughly, we can use a slightly modified version of the standard garbled circuit simulator. This simulator produces a garbled circuit GC for the overall circuit C. The simulator then divides this circuit into components matching the components that were pre-garbled

[3] We assume that the function to compute is agreed upon before the online phase starts, so the circuit description does not need to be sent.

by the protocol. These garbled components are then modified as follows. For each output wire w_i of each linked component, a random label $\widehat{W_i}$ is chosen and is XORed with the output wire label. The result is a new label for each output wire. (The tables in the final gate before each output wire are modified to match the new values.) The output wires still have truly random labels, so these simulated values are still indistinguishable from the evaluator's true view. We now simply note that the random values $\widehat{W_i}$ for each component output wire serve as the simulated linking value that would connect each component's output to the relevant input wires of the next component. They have the same mathematical relationship to the wire labels as the true linking values do. Therefore the simulator has produced a complete simulation of the evaluator's view, and security is achieved.

5 Implementation

We have implemented all the theoretical ideas discussed above in CompGC, a new system for secure computation with preprocessing. Here we describe the implementation in detail, and in the next section we present performance numbers from our experimental results.

To create CompGC we first develop the libgarble library, which is based on the JustGarble implementation of Bellare et al. [BHKR13]. We prefer libgarble to existing approaches, such as TinyGarble [SHS+15], due to its efficiency[4], the fact that it can be compiled as a shared library, and its consistent API. The libgarble library does just what its name implies — it creates a garbled version of a specified circuit and evaluates that circuit given inputs. It is a tool, rather than a complete implementation of secure computation. It does not carry out the oblivious transfers (OTs) necessary to share input, or the networked interactions necessary to send the garbled circuit (or the information for the OT protocols, or the output) between parties.

The libgarble library is based on JustGarble, but several improvements have been made to the code, including cleaning up the API, improving the structures for storing the garbled circuit, etc. With these modifications, we can now evaluate an AES circuit in around 17 cycles/gate, a computation that takes around 22 cycles/gate on the same hardware with the original JustGarble implementation, an improvement of around 22%.

We then use libgarble to build CompGC. CompGC has both an offline and an online phase. In the offline phase, CompGC is given a library of components and computes a specified number of each component. This library could be small and special-built for a certain class of functions, or it could be a huge library of many common computational steps, meant to allow faster online computation of most realistic functions.

In the offline phase, the garbler side of CompGC uses libgarble to generate and garble the component circuits. The garbler saves the garbled component circuits, each tagged with a unique ID, and input and output labels to disk. The garbler side also sends the garbled component circuits and their unique IDs to the evaluator side, which saves the received data to disk. The offline phase finishes by performing the offline portion of OT preprocessing as described by Beaver [Bea95].

We specify the function that the garbler and evaluator compute in the online phase with a JSON file. The file specifies what types of components are needed for the computation, and how the components' input and output wires should be connected. (In an actual deployment, a more efficient representation of the function would be given. We use JSON for our prototype because it's human-readable.)

The garbler and evaluator agree on the function to compute, including the types of components to use and how they are connected. Then, in the online phase, the garbler sends the necessary masks to connect wires between components, the necessary information to interpret the output wire labels, and the wire labels for the garbler's inputs to the evaluator.

Next, the garbler and evaluator perform the online phase of preprocessed oblivious transfer, resulting in the evaluator having input labels corresponding to its input. The evaluator now has all of the information necessary to perform the computation. It evaluates each component using libgarble (in an order specified by the instructions), and computes the input labels for each component from either input labels or processing

[4]Using libgarble as a building block, securely computing AES with a latency of 20 ms and bandwidth of 50 Mbit/s using precomputed OTs takes around 40 ms (cf. Table 2), whereas TinyGarble using their `--disable-OT` flag takes around 208 ms.

the output of a previous component. Finally, the evaluator computes the final output (and can then send it back to the garbler).

6 Experimental Results

In this section we describe two classes of experiments that we performed to demonstrate the efficiency gains provided by CompGC. The first of these demonstrates the performance improvements over traditional garbled circuit computation for several useful classes of computation. The second of these shows the efficiency of using CompGC to compute several machine learning classification algorithms.

Experimental setup. All experiments in this section were run on an Intel® Core™ i5-4210H CPU @ 2.90GHz with 8 GB of RAM. For network configuration, we used the built in Linux emulator `netem` to configure localhost to have a latency of 20 ms (below the average latency in the United States [lat]) and a bandwidth of 50 Mbits/sec. We chose to use a simulated network due to the ease of controlling the latency and bandwidth as well as the ease of reproducibility.

6.1 Improvements over Naive Garbled Circuits

We compared CompGC[5] with the traditional setting where the entire circuit is transferred online. We implemented a semi-honest protocol using libgarble in which the parties preprocess OTs in an offline stage, but the circuit garbling and transfer is done online. This is the closest setting to our work, as we assume that the parties do not know which circuit they would like to compute until the online stage.

We ran four experiments: AES, CBC mode, and Levenshtein distance using both 30 and 60 symbols. We discuss each experiment in turn.

AES: We treat each *round* of AES as a separate component. Thus, computing AES involves linking together 10 components (for each of the 10 rounds of AES when considering 128-bit inputs).

CBC mode: This algorithm provides a way of encrypting variable length messages using a blockcipher (in our case, AES) as an underlying building block. We use the same single-AES-round components as the above experiment, along with an XOR component. Our experiment involves running CBC mode over a 10 block message, and thus we use 110 components (100 for the AES rounds and 10 for the XOR components).

Levenshtein distance: This algorithm provides a measure of distance between two strings. We use as the core component the Levenshtein core circuit as explained by Huang et al. [HEKM11]; see also Figure 1. We use an 8-bit alphabet and run Levenshtein distance over strings containing both 30 and 60 symbols, which corresponds to 900 and 3600 components, respectively.

We note that these experiments are just a sample of what can be done using our tool. While the components we use are particular to our experiments, we note that, for example, an AES circuit could be used in other systems besides just CBC mode (e.g., any function that uses a blockcipher). Likewise, we could break the Levenshtein core circuit into its components (such as 2-MIN and AddOneBit; see Figure 1) which can likely be used in other circuit constructions.

Experimental results. Table 2 presents the results of the above experiments over our simulated network. We compare the running times of both standard semi-honest secure two-party computation with the OTs preprocessed, which we denote as Naive, and our component-based garbled circuit protocol, which we denote as CompGC. We execute 100 runs of each experiment, reporting the average and the 95% confidence interval. Looking at the running times we see drastic improvements of almost an order of magnitude for Levenshtein using 60 symbols, as well as very significant improvements for CBC mode and Levenstein over 30 symbols, and a sizeable improvement for computation of AES. We can see why this is the case by looking at the

[5]All experiments use commit `6af87990be49202fd2d957d8e36128e0ca294623`.

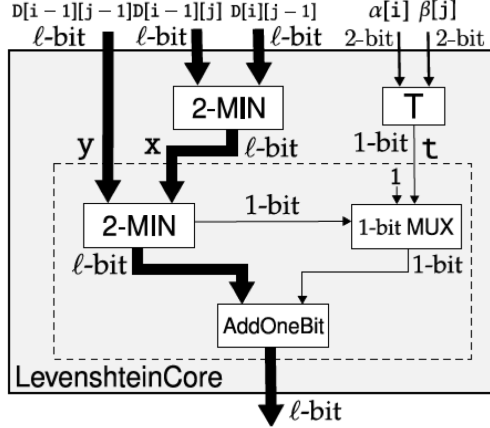

Figure 1: Levenshtein core circuit (taken from Figure 5(c) from the work of Huang et al. [HEKM11]).

	Time (simulated)		**Comm.**	
	Naive	CompGC	Naive	CompGC
AES	62.7 ± 0.1	40.8 ± 0.1	1.9	0.4
CBC mode	414.7 ± 0.6	127.2 ± 0.6	18.9	4.3
Leven. (30)	827.7 ± 0.8	114.8 ± 0.7	39.1	3.6
Leven. (60)	3903.0 ± 2.2	408.4 ± 4.1	191.2	15.7

Table 2: Experimental results; see Section 6 for the experimental setup. Leven. (XX) denotes Levenshtein distance over strings containing XX symbols. All times are in *milliseconds* and all communication is in *megabits*. Naive denotes our implementation of standard semi-honest 2PC using garbled circuits and preprocessed OTs using libgarble, whereas CompGC denotes our component-based implementation. Time is (online) computation time, not including the time to preprocess OTs, but including the time to load data from disk. All timings are of the *evaluator's* running time, and are the average of 100 runs, with the value after the ± denoting the 95% confidence interval. The communication reported is the number of bits received by the evaluator.

total communication of each approach; CompGC demonstrates the greatest *time* improvement for those experiments with the greatest *communication* improvement.

As the main use of CompGC is for more efficient *online* running time, we did not optimize the offline time (we do not use OT extension and do not use a highly optimized OT implementation). However, we note that our offline phase is still relatively efficient: around 30 ms for AES and around 450 ms for CBC mode and Levenshtein with 60 symbols, all over localhost[6]. Thus, we are not achieving efficient online secure computation at the cost of an expensive offline phase: the offline phase involves only preprocessing OTs and garbling and sending garbled components.

From these experiments, we validate the belief that communication *is* the bottleneck for semi-honest secure two-party computation using garbled circuits on realistic networks, and demonstrate that component-based garbling provides a powerful technique for reducing this bottleneck.

6.2 Machine Learning Classification

Our next set of experiments aims to examine a setting where a small library of simple components can enable a variety of computations in a particular domain. We choose private classification because it gives a good illustration of this sort of use, and because it's an area that others have claimed is difficult for garbled circuit-based protocols in the past.

[6]As a comparison point, TinyGarble takes around 120ms for AES.

The key prior work on private classification is that of Bost et al. [BPTG15]. They claim that garbled circuit protocols cannot feasibly compute classification functions (due to memory constraints) and then give specialized protocols that can. We first show that standard "naive" garbled circuits can indeed compute all three of the classification functions considered by Bost et al.[7] Furthermore, efficiency is comparable for two of the three functions, while the third is much faster using garbled circuits (cf. Table 3).

We then use our component-based approach and find, as before, that we can substantially improve online efficiency compared to prior approaches. The resulting speeds are all faster than those achieved by Bost et al. This shows that with precomputation, even without knowing the classification function(s) that will be needed ahead of time, garbled circuits can produce faster online times than even special-purpose protocols. Furthermore, the components we use all carry out simple operations that we expect would be useful for a large variety of functions, not only private classification.

As in Bost et al., we ran experiments to evaluate performance of three different machine learning classifiers: hyperplane decision, naive Bayes, and decision tree. We briefly describe these classifiers below, following the presentation of Bost et al. These classifiers cover a wide range of machine learning algorithms, because many different machine learning algorithms output classifiers of the same type. (For example, perceptron, least squares, and support vector algorithms all output hyperplane decision classifiers.)

For all the classifiers, the user's input $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$ is called a feature vector. Classifying an input x according to a model w amounts to evaluating a function $C_w : \mathbb{R}^d \to \{c_1, \ldots, c_k\}$ on x to classify x into class c_{k^*} for $k^* \in \{1 \ldots k\}$. For ease of notation, we often write k^* instead of c_{k^*}.

Hyperplane decision-based classifier: For this classifier the model consists of k vectors $\{w_1, \ldots, w_k\}$ each in $\mathbb{R}^d$ and the classifier function is

$$k^* = argmax_{i \in [k]} \langle w_i, x \rangle$$

where $\langle \cdot, \cdot \rangle$ represents an inner product. Commonly, as will be the case in all of our experiments, the model consists of only a single vector w and instead of argmax, the classifier determines whether $\langle w, x \rangle \geq 0$.

Naive Bayes: For this classifier, the model consists of the probabilities that each class c_i occurs $\left(\{\Pr[C = c_i]\}_{i=1}^{k}\right)$ and the conditional probabilities that the jth element of x is equal to some value v when x is in class c_i (i.e., $\{\{\{\Pr[x_j = v | C = c_i]\}_{v \in D_j}\}_{j=1}^{d}\}_{i=1}^{k}$). The classifier aims to classify an input x into the class resulting in the highest posterior probability:

$$k^* = argmax_{i \in [k]} \Pr[C = c_i | X = x].$$

The naive Bayes model additionally assumes that the features of x are independent so that

$$\Pr[C = c_1, X_1 = x_1, \ldots, X_d = x_d] = \Pr[C = c_i] \prod_{j=1}^{d} \Pr[X_j = x_j | C = c_i].$$

Decision Tree: For this classifier, the model is given as a binary tree T where each internal node specifies a partition rule on one feature of x and each leaf node corresponds to a class. Specifically, each internal node is labelled with an index $i \in [d]$ and a value w_i and the rule checks whether $x_i \leq w_i$. Classification is done by traversing this tree according to the partition rules and outputting the class corresponding to the leaf node reached.

Classifier components. As already pointed out by Bost et al., all the classifiers above can be assembled using only three basic components: comparison, inner product, and argmax. This makes these classifier

[7] The reason for this discrepancy is that the failure Bost et al. claim occurs when they attempt to use an automated tool to generate the circuits for the classification functions. We simply specify the circuits by hand. We believe this is the more fair comparison, since specifying the circuits is certainly easier than writing special-purpose protocols.

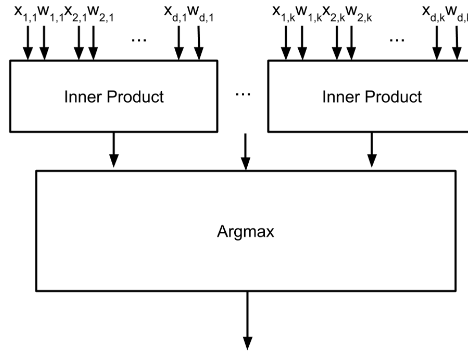

Figure 2: Hyperplane decision classifier circuit.

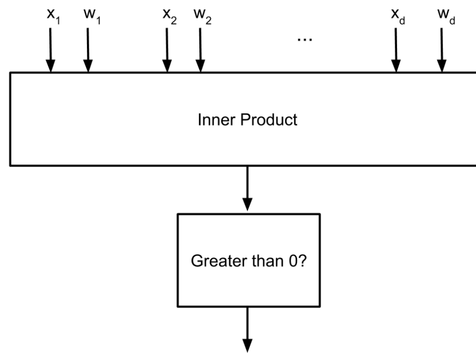

Figure 3: One feature hyperplane decision classifier circuit.

computations ideal for the component-based approach as a small collection of pre-garbled components is sufficient to evaluate all of these classifiers without knowing which one will be desired at preprocessing time. Below we show how circuits for these classifiers can be constructed from a slightly larger set of components.

Figure 2 shows how to build a hyperplane decision classifier out of inner product and argmax components. If, as is the case in the data sets used by Bost et al., the model only consists of a single vector w, an even simpler variant of this classifier can be built using a single inner product component and a component that compares to zero. This is shown in Figure 3.

Figure 4 shows how to build a naive Bayes classifier circuit using addition, select, and argmax components. The select component is one that takes an index and selects an element at that index from an array. We note that just as in Bost et al., we work with the logarithms of probabilities, so that products of probabilities can be computed as sums.

We implement a decision tree classifier by having the circuit mimic the structure of the decision tree. Figure 5 shows the circuit portion corresponding to a single internal node of the decision tree. This just consists of a single comparison component together with a few basic boolean logic gates. It takes as input a bit indicating whether or not this node is on the path of the decision tree that would be traversed for the given feature vector. It outputs two bits that indicate the same thing for each of its children. As the tree is evaluated, the traversed path is computed, and the leaf node that is reached receives a 1 while the others receive 0s. Leaf nodes then simply output their corresponding label if they receive 1 and a null value otherwise.

We stress that all the components used in these constructions are simple, standard operations and could serve as building blocks for many additional functionalities besides the ones discussed here. Thus, while here we consider a more limited case where a component library is specially-tailored for computing private

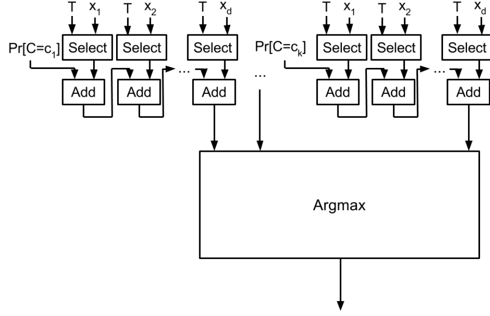

Figure 4: Naive Bayes classifier circuit.

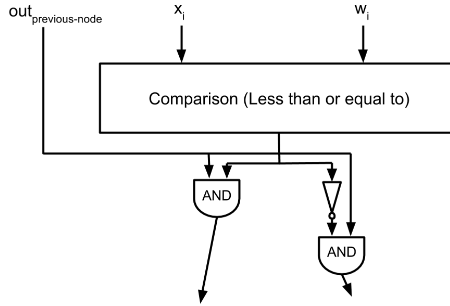

Figure 5: Decision tree classifier node evaluation circuit.

classification, a larger (but still quite modest) library would enable both private classification and a variety of other functions.

Experimental Results. Table 3 presents the results of the above experiments for machine learning classification. We evaluate our secure classification algorithms on the same data sets used by Bost et al.[8], which come from the UCI machine learning repository [BL13] and from [BFK+09]. We ran our experiments in two network settings, our default network (50Mbit/s bandwidth and 20ms latency) and another with no bandwidth limit. We believe that the former represents a more realistic network and also is the worst case for us, since our performance is bandwidth-limited, but the latter matches that used by Bost et al. and allows more direct comparison. We use simple, straightforward circuits including simple, optimized variants of MUX and comparison due to Kolesnikov et al. [KSS09]; our results could be improved by further optimizing the component circuits.

We note first that standard 2PC using no precomputation is already competitive with the protocols of Bost et al. The times in Table 3 do not include precomputation, so for honest comparison to Bost et al., the OT precomputation needs to be added to the listed times for Naive, but this time is consistently only 65ms. Even including this time the times for Naive in the unbounded bandwidth setting are similar to those of Bost et al. for hyperplane decision classifiers, somewhat faster for naive Bayes classifiers, and more than an order of magnitude faster for decision tree classifiers. (The former two classifiers make heavy use of arithmetic gates such as inner product and addition which work well with the homomorphic encryption-based tools

[8]We do not include the audiology naive Bayes data set due to a design decision in our implementation that stores the circuits in memory rather than pipelining them from disk; thus, we run out of memory when trying to load the necessary garbled circuits. We stress that this is an artifact of our implementation and not an inherent issue with our approach.

		Naive		CompGC		Bost et al. [BPTG15]		
Data Set	Size	Time	Comm.	Time	Comm.	Time	Comm.	Rnds
Cancer	30	172 (987)	47	56 (56)	0.7	204	0.3	3.5
Credit	47	256 (1,679)	82	65 (72)	1.1	217	0.3	3.5

(a) Hyperplane decision classifier. "Size" is the length of the model vector w.

	Specs.		Naive		CompGC		Bost et al. [BPTG15]		
Data Set	C	F	Time	Comm.	Time	Comm.	Time	Comm.	Rnds
Cancer	2	9	215 (961)	46	97 (485)	22	479	0.6	7
Nursery	5	9	391 (2,832)	138	169 (1,444)	68	1415	1.2	21

(b) Naive Bayes classifier. "C" is the number of classes and "F" is the number of features.

	Specs.		Naive		CompGC		Bost et al. [BPTG15]		
Data Set	N	D	Time	Comm.	Time	Comm.	Time	Comm.	Rnds
Nursery	4	4	40 (40)	0.2	40 (40)	0.0	2085	21.6	15
ECG	6	4	40 (40)	0.4	40 (41)	0.1	8816	29.1	22

(c) Decision tree classifier. "N" is the number of internal nodes in the tree and "D" is its depth.

Table 3: Experimental results; see Section 6.2 for the experimental setup. Times are in *milliseconds* and communication is in *megabits*. Naive denotes our implementation of standard semi-honest 2PC using garbled circuits and preprocessed OTs using libgarble, whereas CompGC denotes our component-based implementation. "Time" is (online) computation time, not including the time to preprocess OTs, but including the time to load data from disk. Times in parentheses are over a network with bounded bandwidth. All timings are of the *evaluator's* running time, and are the average of 100 runs. The communication reported is the number of bits received by the evaluator. The time and communication of Bost et al. are as reported in [BPTG15]. "Rnds" denotes the number of round trips between parties. For Naive and CompGC, the number of rounds is one throughout.

used by Bost et al. while the decision tree classifier only uses comparison and boolean operations making it ideal for a garbled circuit-based approach.) When bounding the bandwidth, the times for naive garbled circuit computation increase substantially. We do not have experimental measurements of the protocols of Bost et al. in that setting, but since they are less bandwidth-limited, it is likely that those protocols are still faster for hyperplane and naive Bayes classifiers in low-bandwidth settings.

We can then leave the results of Bost et al. aside and look at the bounded bandwidth setting to see whether CompGC provides increased online performance. The decision tree classifier is so efficient even in the traditional garbled circuit protocol that CompGC can do little to improve performance. However, we see a factor of two improvement for naive Bayes classifiers and an order of magnitude improvement for hyperplane decision classifiers. We note that with these improvements, we essentially match the performance of Bost et al. for the case of naive Bayes classifiers, and significantly beat their (online) performance for the other classifiers. These results show yet more evidence that CompGC can use precomputation to drastically reduce online time and can do it using only very simple components that would certainly be included in any substantial library.

Acknowledgments

Work of Alex J. Malozemoff conducted in part with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research, and in part through NSF award #1111599. Work of Arkady Yerukhimovich supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

References

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, Heidelberg, August 1995.

[BFK+09] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. In Michael Backes and Peng Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 424–439. Springer, Heidelberg, September 2009.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[BL13] K. Bache and M. Lichman. UCI machine learning repository. Available at `http://archive.ics.uci.edu/ml/`, 2013.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS 2015*. The Internet Society, February 2015.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[EFG+09] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 235–253. Springer, 2009.

[EGL82] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 205–210. Plenum Press, New York, USA, 1982.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 537–556. Springer, Heidelberg, May 2013.

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015. `http://eprint.iacr.org/2015/309`.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[GLNP15] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 567–578. ACM Press, October 2015.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In David Wagner, editor, *20th USENIX Security Symposium*, San Francisco, California, USA, August 8–12, 2011. USENIX Association.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, Heidelberg, August 2014.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10*, pages 451–462. ACM Press, October 2010.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg, August 2014.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09*, volume 5888 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2009.

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *21st USENIX Security Symposium*, Bellevue, Washington, USA, August 8–10, 2012. USENIX Association.

[lat] Global IP network latency. `http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html`. Accessed 2015-02-16.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, Heidelberg, August 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 579–590. ACM Press, October 2015.

[Mal11] Lior Malka. VMCrypt: modular software architecture for scalable secure computation. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 715–724. ACM Press, October 2011.

[MGBF14] Benjamin Mood, Debayan Gupta, Kevin R. B. Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 582–596. ACM Press, November 2014.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, San Diego, California, USA, August 9–13, 2004. USENIX Association.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Heidelberg, March 2009.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer with adaptive queries. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 573–590. Springer, Heidelberg, August 1999.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, 1999.

[NST16] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. Cryptology ePrint Archive, Report 2016/1069, 2016. `http://eprint.iacr.org/2016/1069`.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 297–314, 2016.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster two-party computation secure against malicious adversaries in the single-execution setting. Cryptology ePrint Archive, Report 2016/762, 2016. `http://eprint.iacr.org/2016/762`.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

Changelog

- Version 1.2 (June 15, 2017):
  - More extensive experimental results section.
- Version 1.1 (May 29, 2016):
  - Removed an optimization due to a security flaw.

  - Update acknowledgments.
  - Add additional references for related work.

- Version 1.0 (May 11, 2016): First release.