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

Adam Groce
Reed College
agroce@reed.edu

Alex Ledger
Reed College
aledger@reed.edu

Alex J. Malozemoff
University of Maryland
amaloz@cs.umd.edu

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 such as arithmetic operations and other common tasks. 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.

We give an implementation, CompGC, of this technique and measure the efficiency gains for various examples. We find that our technique results in roughly an order of magnitude performance improvement over standard garbled circuit-based secure two-party computation.

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 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 one would use functions in a programming language, 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

*An earlier version of this paper contained an additional optimization not present here. We thank Jesper Buus Nielsen for pointing out an error in that section.

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. We show cryptographic protocols for carrying it out, and we provide an open-source implementation, CompGC, that achieves large efficiency gains, upwards of an order of magnitude improvement in online computation time, 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 greater 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.

Finally, in the most general setting, parties engaging in a great deal of computation over time could compute an enormous library with a huge number of possible component types. This would allow extraordinarily fast (online) computation of a wide array of functions.

We note that in the last two 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. We also allow load balancing, since parties can replace used components whenever computational resources are available.

1.1 Our Contributions

Our contributions go well beyond pointing out the ability to divide a circuit into pieces. We give formal specifications for how to create and connect components. We also give a practical, open-source implementation, CompGC, 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. We note that this technique is very similar to the "partial garbled circuits" of Mood et al. [MGBF14], although it was used for a different purpose in that work and, as described, required two labels per connection, whereas we only need a single label per connection. Additionally, a long line of work [NO09, FJN+13, FJNT15] building on the so called "LEGO" approach to maliciously secure garbled circuits uses essentially the same technique to *solder* garbled circuits out of pre-garbled NAND gates. However, none of these three papers give an implementation or experimental evaluation to demonstrate the practical benefit of this technique for real applications.

CompGC *implementation.* We develop our own standalone library `libgarble`[1] for garbling circuits. Our library is a based on the JustGarble implementation of Bellare et al. [BHKR13], but makes many internal improvements to the codebase. None of these improvements 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 is relevant even when not using our component-based precomputation strategy. Our library

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

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] alongside a consistent API.

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 consider three settings: (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. Here, again, we are eliminating the need to know the input length before computation. We measure total online time required to perform the secure computation over both localhost and a simulated 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 a greater than order of magnitude improvement (from 10.6 seconds to 752 milliseconds) when using our approach over the naive approach of sending the circuit online. See Section 6 for more details.

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. We conclude in Section 7.

2 Related Work

Garbled circuits were first introduced by Yao in the 1980s [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 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]) 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 (not using specialized massively-parallel hardware [KsS12]) 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 "Blazing Fast 2PC" system [LR15] for malicious adversaries (in the offline/online model).

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

One method for increasing the efficiency of garbled circuit-based secure computation is to work in the offline/online model and use preprocessing to reduce the online running time. A substantial line of work has focused on reducing the cost of the cut-and-choose technique [LP07] for malicious security using preprocessing [HKK+14, LR14, LR15]. However, all of these works require that the function to compute be defined *during* the pre-processing phase. Our goal is to allow the benefits of pre-processing *even when* one knows little about the function that might be computed.

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". 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 to building garbled circuits [NO09, FJN+13, FJNT15] use this idea to assemble circuits out of pre-garbled NAND gates.

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 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 $(\textsc{Garble}, \textsc{Eval})$. 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

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 $W_{out}^{a \wedge b}$.

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 $\text{Garble}(C)$ 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

As our first contribution, 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 is 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 $(\textsc{Garble}, \textsc{Link}, \textsc{Eval})$. Garble and Eval are variants on the corresponding methods for standard garbled circuits, while 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). $\textsc{Garble}(c)$ 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 (together with a specification of which wire to link to which 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, the format for indicating which wires to link, 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 $(GC_i, e_i, d_i) \leftarrow \textsc{Garble}(1^\kappa, c)$. Additionally, for each pair of components (c_i, c_j) that need to be linked, compute all the link labels $\{W_{ij}\} \leftarrow \textsc{Link}(c_i, c_j)$. 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 $\tau = (\{GC\}_i, \{W\}_{ij}, X, d_{\mathsf{Output}(C)}) \leftarrow \mathcal{S}(1^\kappa, C, C(x))$.
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 $\kappa \in \mathrm{N}$:*

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

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, 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 $\textsc{Link}(c_i, c_j)$ and sends the link labels W_{ij} along with the indices of the wires to be linked 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 (along with information on which input wires map to which output wires). 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] (again, discounting the metadata required to link these wires together). In the more realistic case, where components are substantially larger, the savings can be much greater.

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 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.

CompGC uses as its primary building block the libgarble library, which is based on the JustGarble implementation of Bellare et al. [BHKR13]. We chose to use libgarble over existing approaches, such as TinyGarble [SHS+15], due to its efficiency[3], the fact that it can be compiled as a shared library, and that it has a 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

[3]Using libgarble as a building block, securely computing AES over localhost using precomputed OTs takes 4.4ms (cf. Table 2), whereas TinyGarble using their `--disable-OT` flag takes 13ms.

of around 22%. Note that, while implemented in libgarble, we do *not* use the half-gates approach of Zahur et al. [ZRE15], which reduces the size of each garbled gate to two labels at the cost of two calls to the hash function H during evaluation. We instead use a scheme proposed by Bellare et al. [BHKR13] which requires three labels be transferred but only *one* call to H during evaluation. As we are only concerned with the online time, the benefits of a smaller circuit are outweighed by the extra cost in evaluation.

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. (Another format could be used to gain a small efficiency improvement, but we value the fact that the JSON file is human-readable.)

The garbler receives this function and the garbler's input to the function at the beginning of the online phase. It then generates a set of instructions for the evaluator. The instructions specify particular pre-shared garbled circuits (by ID, rather than just by type). The instructions also specify an order for their evaluation and specify how to feed the outputs of one component into the inputs of others. (This requires both specifying what wires connect where and specifying the relevant mask for each pair of wires that are being connected.) Finally, the instructions include the necessary information to convert the output wire labels to bits, as well as the wire labels for the garbler's input. The garbler sends these instructions 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 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

We compared CompGC[4]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.

Experimental setup. All experiments were run on an Intel® Core™ i5-4210H CPU, and were conducted over two network settings. The first involved running both parties on the default localhost configuration, which on our machine has a latency of 0.012 ms and bandwidth of 35.2 Gb/sec. For the second network setting, we used the built in Linux emulator `netem` to configure localhost to have a latency of 33 ms (the average latency in the United States [lat]) and a bandwidth of 50 Mbits/sec (more than the average bandwidth of 31 Mbits/sec in the United States as of September 2014 [ban]). We chose to use a simulated network due to the ease of controlling the latency and bandwidth as well as the ease of reproducibility. Our implementation also requires reading data from disk: on our experimental machine we measured the cached reads speed as 7.6 GB/sec and the buffered disk reads speed as 96 MB/sec.

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

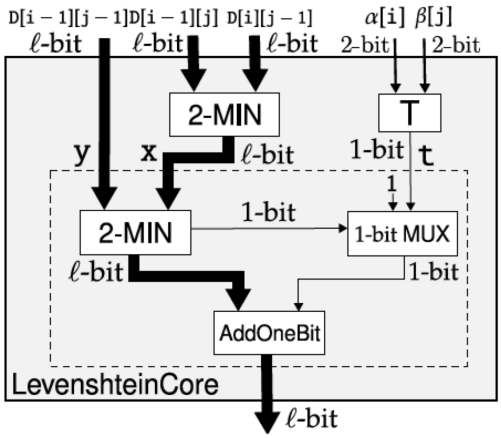

Figure 1: Levenshtein core circuit (taken from Figure 5(c) from the work of Huang et al. [HEKM11]).

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. {AY-0: I removed all mention of the localhost experiments since we dont have numbers for these. It would be really nice to add these back in.} Looking at the running times on the simulated network we see drastic improvements of upwards of an order of magnitude for CBC mode and Levenshtein using 60 symbols, as well as significant improvements for the other two cases. We can see why this is the case by looking at the 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 30ms for AES and around 450ms for CBC mode

	Time (simulated)		Comm.	
	Naive	CompGC	Naive	CompGC
AES	542.6 ± 0.7	134.4 ± 0.1	24	0.656
CBC mode	4800 ± 0.0	321.5 ± 0.9	235	7.4
Leven. (30)	2200 ± 0.0	371.0 ± 0.9	108	10.0
Leven. (60)	10600 ± 0.0	1119.6 ± 2.1	524	44

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.

and Levenshtein with 60 symbols, all over localhost[5]. {AY-1: Should we remove the above since we dont have numbers over the simulated network?} 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 circuits.

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

7 Conclusion

Our new technique, component-based garbled circuits, has greatly reduced online computation time for secure two-party computation. For functions we tested, the time needed for computation was reduced by almost an order of magnitude. This is done by decreasing the amount of data that must be communicated during the online phase. While in principle one could construct functions for which our technique is unlikely to produce more than 50% savings with any realistic set of precomputed components, the benefit for *realistic* functions is much, much greater.

We have shown this in several cases where the general type of function is known ahead of time, but the specifics (e.g., input length) are not. However, the principle itself has much wider application than this. To make full use of our technique, libraries of circuits must be designed. These could be application-specific libraries for certain domains of computation, or there could be large, general-purpose libraries meant to provide useful components for most functions. Designing these sorts of libraries would also allow careful optimization of circuit size for each component.

We work only in the two-party and semi-honest settings, but multi-party and malicious settings could be amenable to a similar technique. We leave the task of designing specific protocols for these settings as future work.

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

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

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

[ban] Measuring fixed broadband report – 2015. `https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-broadband-america-2015`. Accessed 2015-02-16.

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

[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.

[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.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[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.

[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.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

[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.

[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.

[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.

[SV11] N.P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Cryptology ePrint Archive, Report 2011/133, 2011. `http://eprint.iacr.org/2011/133`.

[SZ13] Thomas Schneider and Michael Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 275–292. Springer, Heidelberg, April 2013.

[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 1.1 (May 28, 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.