DRP: Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values

Benjamin Mood
University of Oregon

This majority of this paper was submitted to USENIX Security '14 and was co-authored with Kevin Butler, Debayab Gupta, and Joan Feigenbaum. For my part, I created the original versions of all sections other than the current abstract and the first half of the security section. I created the protocol, programmed, and tested the implementation. The background section has been expanded from the original submission.

Abstract

Two-party secure-function evaluation (SFE) has become significantly more feasible, even on resource-constrained devices, because of advances in server-aided computation systems. However, there are still bottlenecks, particularly in the input-validation stage of a computation. Moreover, SFE research has not yet devoted sufficient attention to the important problem of retaining state after a computation has been performed so that expensive processing does not have to be repeated if a similar computation is done again. This paper presents PartialGC, an SFE system that allows the reuse of encrypted values generated during a garbled-circuit computation. We show that using PartialGC can reduce computation time by as much as 96% and bandwidth by as much as 98% in comparison with previous outsourcing schemes for secure computation. We demonstrate the feasibility of our approach with two sets of experiments, one in which the garbled circuit is evaluated on a mobile device and one in which it is evaluated on a server. We also use PartialGC to build a privacy-preserving "friend-finder" application for Android. The reuse of previous inputs to allow stateful evaluation represents a new way of looking at SFE and further reduces computational barriers.

1 Introduction

Secure function evaluation, or *SFE*, allows multiple parties to jointly compute a function and maintain the privacy of their inputs and outputs. The two-party variant, known as 2P-SFE, was first introduced by Yao in the 1980s [36] and was largely a theoretical curiosity. Developments in recent years have made 2P-SFE vastly more efficient [17, 26, 35]. However, computing a function using SFE is still usually much slower than doing so in a non-privacy-preserving manner.

As mobile devices become more powerful and ubiquitous, users expect more services to be accessible through them. When SFE is performed on mobile devices (where resource constraints are tight), it is extremely slow – *if* the computation can be run at all without exhausting the memory. In fact, for non-trivial input sizes and algorithms, most programs run out of memory [8]. One way to allow mobile devices to perform SFE is to use a server-aided, or "outsourced," computational model [8, 21]. These systems allow the majority of an SFE computation to be outsourced to a more powerful device while still preserving privacy.

Although outsourcing makes resource-constrained SFE more practical, removing the prohibitive cost of on-device circuit evaluation, it is still resource heavy and slow. We have carefully examined CMTB, an outsourced SFE scheme presented by Carter et al. [8], and determined that the new bottleneck is in the cryptographic consistency checks performed on the inputs to the SFE program. We observed that it would be more efficient to adopt other mechanisms for input validation, specifically those proposed by shelat and Shen [35] (herein referred to as sS13). We also noted that even with more efficient input validation, there are many programs that, in order to function properly, require the ability to save state, a feature that current garbled-circuit implementations do not possess. Additionally, the ability to save state and reuse an intermediate value from one garbled circuit execution in another would be useful in many other ways – *e.g.*, we could split a large computation into a number of smaller pieces. Combined with efficient input validation, this becomes an extremely attractive proposition.

In this paper, we show that it is possible to reuse an

encrypted value in an outsourced SFE computation even if one is restricted to primitives that are part of standard garbled circuits. Our system, PartialGC, which is built off of CMTB [8], provides a way to take encrypted output wire values from one SFE computation, save them, and then reuse them as input wires in a new garbled circuit. Our method vastly reduces the number of cryptographic operations compared to the trivial mechanism of simply XOR'ing the results with a one-time pad. We provide a performance analysis and a sample application to illustrate our system.

Our system comprises three parties - a generator, an evaluator, and a third party ("the cloud"), to which the evaluator outsources its part of the computation. Our protocol is secure against a malicious adversary, assuming that there is no collusion with the cloud. We also provide a semi-honest version of the protocol.

In order to reuse intermediate values in a 2P-SFE, information from both parties - the generator and the evaluator - has to be saved. In our protocol, instead of requiring the evaluator to save information, the cloud saves it. This allows multiple distinct evaluators that outsource to the same cloud to participate in a large computation over time by saving state in the cloud between the different garbled circuit executions. For example, in a scenario where a mobile phone is outsourcing computation to a cloud, PartialGC can save the encrypted intermediate outputs to the cloud instead of the phone (Figure 1). This allows the phones to communicate with each other by storing encrypted intermediate values in the cloud, which is more efficient than requiring them to directly participate in the saving of values, as required by earlier 2P-SFE systems. For instance, the application we discuss later is a *friend finder*, where multiple friends save their locations inside the saved intermediate values in the cloud. Other friends can use these saved values to check whether or not someone is in the same map cell as themselves without having to copy and send data.

Figure 2 shows how PartialGC works at a high level: First, a standard SFE execution (blue) takes place, at the end of which we "save" some intermediate output values. All further executions use intermediate output values from previous executions.

Our Contributions:

1. *Reusable Encrypted Values* – We show how to reuse an encrypted value, using only garbled circuits, by mapping one garbled value into another.

2. *Reduced Runtime and Bandwidth* – We show how reusable encrypted values can be used in practice to reduce the execution time for a garbled-circuit computation; we get a 96% reduction in runtime and a 98% reduction in bandwidth over CMTB. Impressively, we can reduce the amount of bandwidth required by the mobile party *arbitrarily* when no input checks have to be performed on the partial inputs, *i.e.*, the intermediate inputs in our protocol.

3. *Outsourcing Stateful Applications* – We show how our system increases the scope of SFE applications by allowing multiple evaluating parties over a period of time to operate on the saved state of an SFE computation without the need for these parties to have any knowledge of each other.

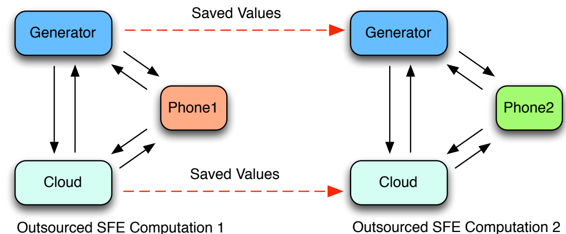


Figure 1: In our system there are three parties. Only the cloud party and generator have to save intermediate values, this means the phone in one computation could be different from a phone in another computation.

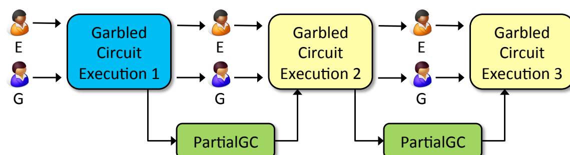


Figure 2: Overview of the PartialGC system. The blue box is a standard execution that produces partial outputs (garbled values). The yellow boxes represent executions that receive partial inputs and produce partial outputs. E is evaluator and G is generator.

The remainder of our paper is organized as follows: Section 2 provides some background on SFE. Section 3 introduces partial garbled circuits in detail. The PartialGC protocol and its implementation are described in Section 4. Section 6 lists the results from our experiments; our implementation of the aforementioned friend-finder application is described in Section 6.3. Section 7 discusses related work.

2 Background

Secure function evaluation (SFE) addresses scenarios where two or more mutually distrustful parties $P_1,...,P_n$, with private inputs $x_1,...,x_n$, want to compute a given function $y_i = f(x_1,...,x_n)$ (y_i is the output received by P_i), such that no P_i learns anything about any x_j or y_j, $i \neq j$ that is not logically implied by x_i and y_i. Moreover, there exists no trusted third party - in that case, the P_is

could simply send their inputs to the trusted party, which would evaluate the function and return the y_is.

SFE was first proposed in the 1980s in Yao's seminal paper [36]. The area has been studied extensively by the cryptography community, leading to the creation of the first general purpose implementation of SFE, Fairplay [29] in the early 2000s. Today, there exist many such implementations [6, 9, 15, 16, 25, 34, 37].

The classic implementations of 2P-SFE, including Fairplay, use garbled circuits. A garbled circuit is a Boolean circuit, which is encrypted in such a way that it can be evaluated when the proper input wires are entered. The party that evaluates this circuit does not learn anything about what any particular wire represents. In 2P-SFE, the two parties are: the *generator*, which creates the garbled circuit, and the *evaluator*, which evaluates the garbled circuit. Additional cryptographic techniques are used for input and output; we discuss these later.

A two-input Boolean gate has four truth table entries. A two-input garbled gate also has a truth table with four entries representing 1s and 0s, but these entries are encrypted and can only be retrieved when the proper keys are used. The values that represent the 1s and 0s are random strings of bits. The truth table entries are permuted such that the evaluator cannot determine which entry she is able to decrypt, only that she is able to decrypt an entry. The entirety of a garbled gate is the four encrypted output values.

Each garbled gate is then encrypted in the following way: Each entry in the truth table is encrypted under the two input wires, which leads to the result, $truth_i = Enc(input_x||input_y) \oplus output_i$, where $truth_i$ is a value in the truth table, $input_x$ is the value of input wire x, $input_y$ is the value of input wire y, and $output_i$ is the non-encrypted value, which represents either 0 or 1.We use AES as the *Enc* function. If the evaluator has $input_x$ and $input_y$, then she can also receive $output_i$, and the encrypted truth tables are sent to her for evaluation.

For the evaluator's input, 1-out-of-2 oblivious transfers (OTs) [1, 19, 31, 32] are used. In a 1-out-of-2 OT, one party offers up two possible values while the other party selects one of the two values without learning the other. The party that offers up the two values does not learn which value was selected. Using this technique, the evaluator gets the wire labels for her input without leaking information.

The only way for the evaluator to get a correct output value from a garbled gate is to know the correct decryption keys for a specific entry in the truth table, as well as the location of the value she has to decrypt.

During the permutation stage, rather than simply randomly permuting the values, the generator permutes values based on a specific bit in $input_x$ and $input_y$, such that, given $input_x$ and $input_y$ the evaluator knows that the location of the entry to decrypt is $bit_x * 2 + bit_y$. These bits are called the *permutation bits*, as they show the evaluator which entry to select based on the permutation; this optimization, which does not leak any information, is known as *point and permute* [29].

Example Execution

We give an example execution for the above garbled circuit protocol where the circuit is solely composed of a single AND gate. In this circuit both the generator and evaluator enter in input to the AND gate and both parties receive the output from the AND gate.

Creation of Wire Labels

In our protocol, the generator must first create the wire labels for the three garbled wires (two input and one output wires to the AND gate). Each wire needs two labels since each wire can have a value of 0 or 1. Wire e is the evaluator's input, g is the generator's input, and o is the output. We give the possible wire values for each wire below.

Wire	0 value	1 value
e	0x40FF	0x5788
g	0x9143	0x8634
o	0x78F2	0x6F85

Creation of Garbled Gate

The generator creates the garbled gate's truth table using the following equations:

$$\begin{aligned} table_{00} &= hash(e_0||g_0) \oplus o_0 \\ table_{01} &= hash(e_0||g_1) \oplus o_0 \\ table_{10} &= hash(e_1||g_0) \oplus o_0 \\ table_{11} &= hash(e_1||g_1) \oplus o_1 \end{aligned}$$

The equations above show how the generator encrypts the output values, o_i, using a 1-time pad generated by hashing together both input wire labels. Take note that the entries for the output of the garbled gate, o_0, o_0, o_0, o_1, correspond to the truth table for an AND gate.

Next, the truth table is permuted such that a bit from the input wire labels, the point and permute bit, correctly informs the evaluator which entry she can decrypt.

In this example we use the least significant bit as the point and permute bit. Both our example wire labels, g and e, have 0-value wire labels with point and permute bits of 1. This means we have to permute the truth table's values to swap the 0 and 1 values for both the g and e wire values. After permuting the truth table we receive the following table:

$$\begin{aligned} table_{11} &= hash(e_1||g_1) \oplus o_1 \\ table_{10} &= hash(e_1||g_0) \oplus o_0 \\ table_{01} &= hash(e_0||g_1) \oplus o_0 \\ table_{00} &= hash(e_0||g_0) \oplus o_0 \end{aligned}$$

After removing the identifying from the truth table tables, the complete truth table sent by the generator to the evaluator:

$$hash(e_1||g_1) \oplus o_1$$
$$hash(e_1||g_0) \oplus o_0$$
$$hash(e_0||g_1) \oplus o_0$$
$$hash(e_0||g_0) \oplus o_0$$

Inputs

After creation of the garbled gate, the evaluator still needs to know her input wire label. The generator can send over his encrypted input bit, as he knows the correct wire label for his input. For the evaluator's input, both parties participate in an OT. In this OT, the evaluator enters her input bit and the generator enters both possible wire label values. At the conclusion of the OT, the evaluator receives the input wire label corresponding to her input.

For this example, we assume the evaluator entered a 1 (0x5788) as input and the generator entered a 0 (0x9143) as input.

Evaluation

The evaluator receives the following truth table, as this is the entire garbled circuit:

$$hash(e_1||g_1) \oplus o_1$$
$$hash(e_1||g_0) \oplus o_0$$
$$hash(e_0||g_1) \oplus o_0$$
$$hash(e_0||g_0) \oplus o_0$$

The evaluator examines the point and permute bits, a 0 on the evaluator's input wire label and a 1 on the generator's input wire label, and knows she needs to decrypt the $0*2+1$ entry in the truth table.

The evaluator then hashes the wires together, XORs the truth table entry with the hash she just generated and receives o_0. The hash she created using the input wires and the hash in the truth table, which encrypted o_0, cancel out.

$$hash(e_1||g_0) \oplus o_0 \oplus hash(e_1||g_0) = o_0$$

Output

The evaluator has the output wire label but does not know how to interpret it. The generator sends the evaluator a table that informs the evaluator what the output wire label represents. This table contains hashes of both possible output wire values. Then evaluator sends the output wire label to the generator.

2.1 Threat Models

Traditionally, there are two threat models discussed in SFE work, semi-honest and malicious. The above description of garbled circuits is the same in both threat models. In the semi-honest model users stay true to the protocol but may attempt to learn extra information from the system by looking at any message that is sent or received. In the malicious model, users may attempt to change anything with the goal of learning extra information or giving incorrect results without being detected; extra techniques must be added to achieve security against a malicious adversary.

There are several well-known attacks a malicious adversary could use against a garbled circuit protocol. A protocol secure against malicious adversaries must have solutions to all potential pitfalls, described in turn:

Generation of incorrect circuits: If the generator does not create a correct garbled circuit, he could learn extra information by modifying truth table values to output the evaluator's input; he is limited only by the external structure of the garbled circuit the evaluator expects.

To solve this problem, the parties perform the computation many different times with different garbled circuits all computing the same function. Some of the garbled circuits are evaluated and many garbled circuits are checked for correctness. The output for the computation is then a majority vote from point and permute bit of the output values from the evaluation circuits. Using this concept, the evaluator knows the generator has a negligible chance of successfully gaining extra information with respect to then number of circuits, which is a security parameter. This type of protocol is known as a *cut and choose* protocol.

As an example we again look at the AND gate from the previous example. In the permuted gate the truth table should be:

$$hash(e_1||g_1) \oplus o_1$$
$$hash(e_1||g_0) \oplus o_0$$
$$hash(e_0||g_1) \oplus o_0$$
$$hash(e_0||g_0) \oplus o_0$$

Instead a malicious generator could create a different garbled gate where that the output is always the evaluator's input:

$$hash(e_1||g_1) \oplus o_1$$
$$hash(e_1||g_0) \oplus o_1$$
$$hash(e_0||g_1) \oplus o_0$$
$$hash(e_0||g_0) \oplus o_0$$

Using the cut and choose, the generator has to the commit to multiple garbled gates before any other operation takes place. Based on selection by a fair coin, the generator gives the evaluator keys to "open" a portion of the garbled circuits so the evaluator can verify they were correct.

If the generator tries to do the aforementioned attack, he will be caught with a s degree of certainty, depending upon the total number of garbled circuits evaluated.

Selective failure of input: Select failure attacks occur when the generator can cause the evaluator to abort the protocol based on her inputs to the function. This reveals information about the evaluator's input.

If the generator does not offer up correct input wires labels to the evaluator in the OT, the generator will learn a single bit of information based on whether the computation produced correct outputs. This attack is prevented by encoding each of the evaluator‘s input bits into multiple input bits. Using this encoding, if a single bit fails it does not leak any information about the evaluator's input to the generator, as the single bit of input held no information without knowing all of the encoding bits.

As an example of this attack, instead of using e_0 and e_1 as the two values the generator inputs into the OT, the generator uses e_0 and X where X causes the computation to fail. To prevent this attack the evaluator breaks these bits into many bits, which, by XORing them together inside of the garbled circuit, will generate the real input. The parties perform the OTs over two input bits, e_i and e_j, to represent to input bit e. The garbled circuit is then augmented to take in e_i and e_j and XOR them to form the true input bit e. Notice now, assuming we use a large enough amount of bits, if the computation fails then the input bit is not revealed.

Related to the attack of incorrect OT output wire labels is the attack of reversing the OT output wire labels. In this attack, the generator swaps the two possible OT inputs in the garbled circuit to invert the input wires. To prevent this, the protocol forces the generator to commit ahead of time to the possible input wire values. Those commitments are then checked during the evaluation. If the commitment fails then the evaluator knows the generator did not perform the correct OT operation. Since the input is already encoded the generator learns nothing from the abort.

To use a more concrete example, instead of using e_0 and e_1 as input into the OT, the generator swaps the order and enters e_1 and e_0. The commitment prevents the generator from performing this attack.

Input consistency: To prevent the generator from using an incorrect garbled circuit we use many different garbled circuits. However, if either party's input is not consistent across all circuits then it is possible for extra information to be retrieved during the evaluation. If the generator's input is not consistent then a single incorrect circuit could give extra information, bypassing most of the security gained from the parameter of the cut and choose. To prevent this attack the parties engage in a cryptographic technique that guarantees the consistency of the generator's input.

The generator has a reasonable chance (40%) of getting a single incorrect circuit into the evaluation portion of the garbled circuits. If we assume there are three evaluation circuits and the generator knows the output should be 0 for one possible input of his and 1 for a different input of his, and uses two almost evenly distributed inputs, the generator can use a single incorrect circuit to change the output by changing ouput vote from a $2-1$ vote to a $1-2$ vote. The generator learns a single bit of information based upon the output he saw in this example.

If the evaluator's input is not consistent then it will give out extra information, as she would know the output for many computations instead of just a single computation (as she can observe the output for each circuit with different inputs). The solution to this problem is to force the evaluator to generate all her inputs for circuits $2 \dots n$ from the inputs to circuit 1.

If we again assumed three evaluation circuits and the evaluator entered different inputs to all three circuits, it allows her to see the results from three the different inputs, instead of the single set of outputs she is suppose to receive.

Output consistency: In the two-party case, the output consistency check verifies that the evaluator did not modify the generator's output before sending it to the generator. Without the output check, it is impossible for the generator to know the evaluator did not modify the output before sending it to the generator. To prevent the evaluator from learning the actual output the generator adds 1-time pads into the output bits.

A simple solution to the problem of output consistency is to output X concatenated with a $MAC(X)$ and encrypt the total output value under a 1-time pad. For the evaluator to successfully modify the output she would have to guess how modifying X modifies $MAC(X)$ without knowing either X or $MAC(X)$. This is very unlikely given a large enough private key.

2.2 CMTB Protocol

As we are building off of the CMTB garbled circuit execution system, we give an abbreviated version of the protocol. In our description we refer to the generator, the cloud, and the evaluator. The cloud is the party the evaluator outsources her computation to.

Circuit generation and check: The template for the garbled circuit is augmented to add one-time XOR pads on the output bits and split the evaluator's input wires per the input encoding scheme. The generator generates the necessary garbled circuits and commits to them and sends the commitments to the evaluator. The generator then commits to input labels for the evaluator's inputs.

CMTB relies on the random seed check created by Goyal et al. [13], which was then implemented by Kreuter et al. [26] to combat generation of incorrect cir-

cuits. This technique uses a cut-and-choose style protocol to determine whether or not the generator created the correct circuits by creating and committing to many different circuits. Some of those circuits are used for evaluation, while the others are used as check circuits.

Evaluator's inputs: Rather than a two-party oblivious transfer, we perform a three-party *outsourced oblivious transfer*. An outsourced oblivious transfer is an OT that gets the select bits from one party, the wire labels from another, and returns the selected wire labels to a third party. The party that selects the wire labels does not learn what the wire labels are, and the party that inputs the wire labels does not learn which wire was selected; the third party only learns the selected wire labels. In CMTB, the generator offers up wire labels, the evaluator provides the select bits, and the cloud receives the selected labels. CMTB uses the Ishai OT extension [19] to reduce the number of OTs.

CMTB uses an encoding technique from Lindell and Pinkas [28], which prevents the generator from finding out any information about the evaluator's input if a selective failure attack transpires. CMTB also uses the commitment technique of Kreuter et al. [26] to prevent the generator from swapping the two possible outputs of the oblivious transfer. To ensure the evaluator's input is consistent across all circuits, CMTB uses a technique from Lindell and Pinkas [28], whereby the inputs are derived from a single oblivious transfer.

Generator's input and consistency check: The generator sends his input to the cloud for the evaluation circuits. Then the generator, evaluator, and cloud all work together to prove the input consistency of the generator's input. For the generator's input consistency check, CMTB uses the malleable-claw free construction from shelat and Shen [34].

Circuit evaluations: The cloud evaluates the garbled circuits marked for evaluation and checks the circuits marked for checking. The cloud enters in the generator and evaluator's input into each garbled circuit and evaluates each circuit. The output for any particular bit is then the majority output between all evaluator circuits. The cloud then recreates each check circuit. The cloud creates the hashes of each garbled circuit and sends those hashes to the evaluator. The evaluator then verifies the hashes are the same as the ones the generator previously committed to.

Output consistency check and output: The three parties prove together that the cloud did not modify the output before she sent it to the generator or evaluator. Both the evaluator and generator receive their respective outputs. All outputs are blinded by the respective party's one-time pad inside the garbled circuit to prevent the cloud from learning what any output bit represents.

CMTB uses the XOR one-time pad technique from Kiraz [23] to prevent the evaluator from learning the generator's real output. To prevent output modification, CMTB uses the witness-indistinguishable zero-knowledge proof from Kreuter et al. [26].

2.2.1 Non-collusion

In the CMTB protocol there is an assumption of non-collusion. Non-collusion is stated in CMTB as:

"The outsourced two-party SFE protocol securely computes a function f(a,b) in the following two corruption scenarios: (1)The cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, (2)All but one party is malicious, while the cloud is semi-honest."

This is the standard definition of non-collusion used in server-aided works such as Kamara et al. [21].

3 Partial Garbled Circuits

We introduce the concept of *partial garbled circuits* (PGCs), which allow the encrypted wire outputs from one SFE computation to be used as inputs to another. This can be accomplished by *mapping* the encrypted output wire values to valid input wire values in the next computation. In order to better demonstrate their structure and use, we first present PGCs in a semi-honest setting, before showing how they can aid us against malicious adversaries.

3.1 PGCs in the Semi-Honest Model

In the semi-honest model, for each wire value, the generator can simply send two values to the evaluator, which transform the wire label the evaluator owns to work in another garbled circuit. Depending on the point and permute bit of the wire label received by the evaluator, she can map the value from a previous garbled circuit computation to a valid wire label in the next computation. At the beginning of the protocol, for a given wire, the generator has both possible encrypted output input wires and the evaluator has the correct wire label values. At the end of the protocol, the evaluator has the correct input wire label for the output label she previously owned.

Specifically, for a given wire pair, the generator has wires w_0^{t-1} and w_1^{t-1}, and creates wires w_0^t and w_1^t. Here, t refers to a particular computation in a series, while 0 and 1 correspond to the values of the point and permute bits of the $t-1$ values. The generator sends the values $w_0^{t-1} \oplus w_0^t$ and $w_1^{t-1} \oplus w_1^t$ to the evaluator. Depending on the point and permute bit of the w_i^{t-1} value she possesses, the evaluator selects the correct value and then XORs her

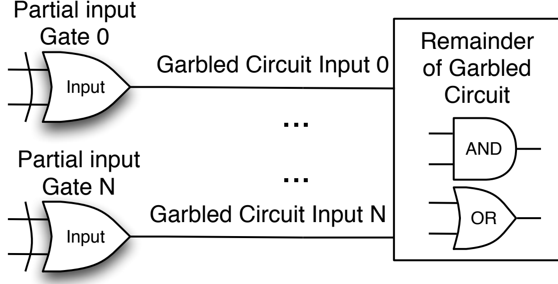


Figure 3: This figure shows how we create a single *partial input gate* for each input bit for each circuit and then link the *partial input gates* to the remainder of the circuit.

w_i^{t-1} with the $(w_i^{t-1} \oplus w_i^t)$ value, thereby giving her w_i^t, the valid partial input wire.

3.2 PGCs in the Malicious Model

In the malicious model, either party could choose to not perform their side of the protocol correctly. We must allow the evaluation of a circuit with partial inputs and verification of the mappings, while preventing a selective failure attack. The following features are necessary to accomplish these goals:

1. Verifiable Mapping

The generator G is able to create a secure mapping from a saved garbled wire value into a new computation that can be checked by the evaluator E, without E being able to reverse the mapping. For the evaluation circuits, G sends E the information required to transform the values and E maps the values. For check circuits, E is able to check the mapping by simulating the mappings created by G. During the evaluation and check phase, E must be able to verify whether the correct encrypted value was mapped and that G did not send an incorrect mapping. G must have either previously committed to the mappings before deciding the partition of evaluation and check circuits, or never learned which circuits are in the check versus the evaluation sets.

2. Partial Generation and Partial Evaluation

G creates the garbled gates necessary for E to enter the previously output intermediate encrypted values into the next garbled circuit. These garbled gates are called *partial input gates*. As shown in Figure 3 each garbled circuit is made up of two pieces: the partial input gates and the remainder of the garbled circuit.

3. Revealing Incorrect Transformations

Our last goal is to let E inform G that incorrect values have been detected. Without a way to limit leakage, G could gain information based on whether or not E informs G that she caught him cheating. This is a selective failure attack and is not present in our protocol.

4 PartialGC Protocol

We start with the CMTB protocol and add cut-and-choose operations from sS13 before introducing the mechanisms needed to save and reuse values. We defer to the original papers for details of the outsourced oblivious transfer [8] and the generator's input consistency check [35] sub-protocols that we use.

Our system operates in the same threat model as CMTB: we are secure against a malicious adversary with the assumption that no parties are colluding to defeat the protocol. We discuss this further in Section 5.

4.1 Preliminaries

There are three participants in the protocol:

Generator – The generator is the party that generates the garbled circuit for the 2P-SFE.

Evaluator – The evaluator is the other party in the 2P-SFE, which is outsourcing computation to a third party, the cloud.

Cloud – The cloud is the party that executes the garbled circuit outsourced by the evaluator.

Notation

C_{key} - Circuit key used for the free XOR optimization [24], which we employ. The key is randomly generated and then used as the difference between the 0 and 1 wire labels for a circuit C.

C_{seed} - This value is created by the generator's PRNG, and is used to generate a particular circuit C.

$POut_i$ - The *partial output* values are the values output from an SFE computation. These are encrypted garbled circuit values which can be reused in another garbled circuit computation. The i determines whether the value represents a 0 or a 1 value (as it does for all subsequent values in this list).

PIn_i - The *partial input* values are the $POut$ values after they have been hashed to remove the previous circuit key, C_{key}. These values are input to the *partial input gate*.

GIn_i - The *garbled circuit input* values are the results of the partial input gates and are input into the remaining garbled circuit.

Partial Input Gates - These are garbled gates that take in PIn values and output GIn values. Their purpose is to transform the PIn values, to values that work under the key C_{key} for the current circuit.

GIn_i - The *garbled circuit input* values are the results of the partial input gates and are input into the remainder of the garbled circuit, as shown in Figure 3.

4.2 Protocol

Common Inputs: The program circuit file, the bit level security, the circuit level security (number of circuits), and encryption and commitment functions.

Private Inputs: The evaluator and the generator both have inputs to the computation.

Outputs: The evaluator and the generator receive garbled circuit outputs.

Phase 1: Cut-and-choose

We use the cut-and-choose mechanism described in sS13. Here, the cloud selects which circuits are evaluation circuits and which are check circuits; the generator does not learn this information.

To decide how each circuit is to be used, the cloud and generator participate in an oblivious transfer where the generator offers up either the decryption key for its input to a circuit or the decryption key for the circuit seed. If the cloud selects the circuit seed, then that circuit is used as a check circuit. If the cloud selects the generator's input then that circuit is used as an evaluation circuit.

The generator then encrypts and sends his garbled input values and the check circuit information for all circuits to the cloud. The cloud is able to decrypt the check circuit information for the check circuits and decrypt the encrypted garbled input value for evaluation circuits.

We only perform the cut and choose oblivious transfers for the initial computation. For any subsequent computations, we require the parties hash the last decryption keys and use these hashes as the new decryption keys.

To allow the evaluator to know the circuit split, we perform the following:

The generator sends a hash of both possible encryption keys the cloud could have selected to the evaluator for each circuit as an ordered pair.

$$GeneratorSend(hash(check_key)) \quad (1)$$

$$GeneratorSend(hash(evaluation_key)) \quad (2)$$

The cloud then sends a hash of the value received to the evaluator for each circuit. The cloud also sends bits to indicate what circuits were selected as check or evaluation circuits to the evaluator.

$$CloudSend(hash(selected_key)) \quad (3)$$

$$CloudSend(circuit_split) \quad (4)$$

Using the hashes received from both parties and the circuit selection map from the cloud, the evaluator matches the hashes the cloud sent with the circuit selection to the hash the generator. If the selected hash from the generator for a given circuit does not match the hash the cloud sent over for the selected circuit, then the evaluator knows one of the two parties tried to cheat her.

Phase 2: Oblivious Transfer

We use the outsourced oblivious transfer (OOT) of CMTB. The generator inputs both possible input wires for the evaluator's input and the evaluator inputs its own private input values. After the OOT is performed, the cloud has the input wires, which represent the evaluator's input.

We make the following modification to the outsourced oblivious transfer: we perform the OOT on all circuits (this is required since the generator cannot know which circuits are for checking and which for evaluation), but the outputs of the OOT are composed of the original output of the OOT and an added blind that the generator creates and sends to the evaluator. There is one blind per output wire. The evaluator then sends the blinds to the cloud only for evaluation circuits.

Phase 3: Generator's Input Consistency Check

We use the input consistency check of sS13. A universal hash structure is used to prove the consistency of the generator's input across each evaluation circuit. If the proof is incorrect then the cloud knows the generator did not enter consistent input across all evaluation circuits.

Phase 4: Partial Input Gate Generation, Check, and Evaluation

Generation

For all circuits in the computation the generator creates the partial input gates, which transform previously output values into values that can be used in the current garbled circuit execution, and other necessary information to transform the previously output values, $POut$, into wires which are used in the complete circuit, GIn.

The generator creates a pseudorandom transformation value R and the *partial input gates*. One R is created for each circuit C, and one *partial input gate* is created for each saved wire for each C.

$$R = PRNG.random() \quad (5)$$

For each $POut$, the wires previously output, the generator XORs the wire with R. The value is then hashed.

$$temp_0 = hash(R \oplus POut_0) \quad (6)$$

$$temp_1 = hash(R \oplus POut_1) \quad (7)$$

A function, *setPPBitGen*, is then used to pseudorandomly find a bit which is different between the wire's two values to be used as the "point and permute" bit. We refer to the final value as the *partial input value*, PIn, which is used as the input to the *partial input gate*. Once per circuit, *setPPBitGen* is seeded using a seed derived from C_{seed}.

$$PIn_0, PIn_1 = setPPBitGen(temp_0, temp_1) \quad (8)$$

For each *PIn* wire, a wire *GIn* is created under the master key of that specific circuit – where C_{key} is the difference between 0 and 1 wire labels for the circuit. We reseed the PRNG with C_{seed} and add 1 to prevent overlap from the PRNGs.

$$GIn_0 = PRNG.random() \tag{9}$$

$$GIn_1 = GIn_0 \oplus C_{key} \tag{10}$$

The generator creates two truth table values, TT, for each *partial input gate* such that the truth table values change *PIn* to the corresponding *GIn*.

$$TT_0 = PIn_0 \oplus GIn_0 \tag{11}$$

$$TT_1 = PIn_1 \oplus GIn_1 \tag{12}$$

The generator sends *R* to the cloud for every circuit.

The generator sends each *partial input gate* to the cloud. This includes both TT values and the point and permute bit location. The truth table values are permuted such that the point and permute bits of the *PIn* values correctly map to the TT values using the point and permute optimization.

Check

For all check circuits, the cloud checks the partial input gates the generator created by recreating them using the check circuit information sent to the cloud during the cut-and-choose. The cloud uses the *R* value the generator sent for the circuit and verifies that the *partial input gates* sent by the generator matches the *partial input gates* created by the *R* value and circuit seed.

The cloud is able to recreate the partial input gates by recreating both *PIn* values, both *GIn* values, each point and permute bit location, and then creating the TT values. As previously noted, each *GIn* value is created directly from the circuit seeds. Each *PIn* value is a combination of *R* and the point and permute bit location. Using *PIn* and *GIn*, the cloud creates the TT values. After the creation of these values, the cloud is able to verify that the generator sent the correct *partial input gate* values.

Evaluation

For all evaluation circuits, the cloud evaluates the partial input gates to receive the garbled input values *GIn*. The cloud creates the *PIn* values by using *R* and *setPPBitEval* with the received point and permute bit location. *setPPBitEval* reads the location of the point and permute bit from the partial gate sent by the generator and applies it to the wire label.

$$temp = hash(R \oplus POut) \tag{13}$$

$$PIn = setPPBitEval(temp) \tag{14}$$

The cloud inputs each *PIn* value into the corresponding *partial input gate*. It then evaluates them by XORing the *PIn* value with the truth table value, which is selected by the point and permute bit.

$$GIn = (PIn \oplus TT_{PPbit}) \tag{15}$$

The cloud then uses the *GIn* values to evaluate the remainder of the garbled circuit.

Phase 5: Circuit Generation and Evaluation

The generator creates each garbled circuit for the cloud. The cloud then verifies that the check circuits are correct and uses the evaluation circuits to compute the result of the garbled circuit.

Phase 6: Output and Output Consistency Check

As the final step of the garbled circuit execution, a MAC of the output is generated inside the garbled circuit, based on a *k*-bit secret key entered into the function. Both the resulting garbled circuit output and the MAC are are encrypted under a one-time pad. The cloud receives the encrypted outputs of both the generator and evaluator from the garbled circuit execution, and sends the correct encrypted outputs to each party.

The generator and evaluator then decrypt the received ciphertext, perform a MAC over it, and verify the cloud did not modify the output by comparing the generated MAC with the one calculated within the garbled circuit.

Phase 7: Partial Output

The generator saves both possible wire values for each partial output wire. The cloud its value for each partial output wire associated with the evaluation circuits and both possible output values for all check circuits. The generator and cloud both save all decryption keys used during the cut-and-choose.

4.3 Implementation

As with most garbled circuit systems there are two stages to our implementation. The first stage is a compiler for creating garbled circuits, while the second stage is an execution system to evaluate the circuits.

For our compiler, we modified the KSS12 [26] compiler to allow the saving of input and output wire label values. By using the KSS12 compiler, we have an added benefit of being able to compare circuits of almost identical size and functionality between our system and CMTB, whereas other protocols generate circuits of sometimes vastly different sizes.

For our execution system, we started with the CMTB system and modified it according to our protocol requirements. PartialGC automatically performs the output consistency check, and we implemented this check at the circuit level. We became aware and corrected issues with CMTB relating to too many primitive OT operations performed in the outsourced oblivious transfer when using a high circuit parameter and too low a general security parameter in general. After the fixes were applied, this reduced the run-time of the OOT.

5 Security of PartialGC

In this section, we provide a basic proof sketch of the PartialGC protocol, showing that our protocol preserves the standard security guarantees provided by traditional garbled circuits - that is, none of the parties learns anything about the private inputs of the other parties that is not logically implied by the output it receives. Since we borrow heavily from [8] and [35], we focus on our additions, and defer to the original papers for detailed proofs of those protocols.

We know that the protocol described in [8] allows us to garble individual circuits and securely outsource their evaluation. In this paper, we modify certain portions of the protocol to allow us to transform the output wire values from a previous circuit execution into input wire values in a new circuit execution. These transformed values, which can be checked by the evaluator, are created by the generator using circuit "seeds."

We also use some aspects of [35], notably their novel cut-and-choose technique which ensures that the generator does not learn which circuits are used for evaluation and which are used for checking - this means that the generator must create the correct transformation values for all of the cut-and-choose circuits.

Because we assume that the CMTB garbled circuit scheme can securely garble any circuit, we can use it individually on the circuit used in the first execution and on the circuits used in subsequent executions. We focus on the changes made at the end of the first execution and the beginning of subsequent executions which are introduced by PartialGC.

The only difference between the initial garbled circuit execution and any other garbled circuit in CMTB is that the output wires in an initial PartialGC circuit are stored by the cloud, and are not delivered to the generator or the evaluator. This prevents them from learning the output wire labels of the initial circuit, but cannot be less secure than CMTB, since no additional steps are taken here.

Subsequent circuits we wish to garble differ from ordinary CMTB garbled circuits only by the addition, before the first row of gates, of a set of partial input gates. These gates don't change the output along a wire, but differ from normal garbled gates in that the two possible labels for each input wire are not chosen randomly by the generator, but are derived by using the two labels along each output wire of the initial garbled circuit.

This does not reduce security. In PartialGC, the input labels for partial input gates have the same property as the labels for ordinary garbled input gates: the generator knows both labels, but does not know which one corresponds to the evaluator's input, and the evaluator knows only the label corresponding to its input, but not the other label. This is because the evaluator's input is exactly the output of the initial garbled circuit, the output labels of which were saved by the evaluator. The evaluator does not learn the other output label for any of the output gates because the output of each garbled gate is encrypted. If the evaluator could learn any output labels other than those which result from an evaluation of the garbled circuit, the original garbled circuit scheme itself would not be secure.

The generator, which also generated the initial garbled circuit, knows both possible input labels for all partial evaluation gates, because it has saved both potential output labels of the initial circuit's output gates. Because of the outsourced oblivious transfer used in CMTB, the generator did not know which input labels to use for the initial garbled circuit, and therefore will not have been able to determine the output labels for that circuit. Therefore, the generator will likewise not know which input labels are being used for subsequent garbled circuits.

Generator's Input Consistency Check

We use the generator's input consistency check from sS13. We note there is no problem with allowing the cloud to perform this check; for the generator's inconsistent input to pass the check, the cloud would have to see the malicious input and ignore it, which would violate the non-collusion assumption.

Correctness of Saved Values

Scenarios where either party enters incorrect values in the next computation reduce to previously solved problems in garbled circuits. If the generator does not use the correct values, then it reduces to the problem of creating an incorrect garbled circuit. If the evaluator does not use the correct saved values then it reduces to the problem of the evaluator entering garbage values into the garbled circuit execution; this would be caught by the output consistency check.

Abort on Check Failure

If any of the check circuits fail, the cloud reports the incorrect check circuit to both the generator and evaluator. At this point, the remaining computation and any saved values must be abandoned. However, as is standard in SFE, the cloud cannot abort on an incorrect evaluation circuit, even when she knows that it is incorrect.

Concatenation of Incorrect Circuits

If the generator produces a single incorrect circuit and the cloud does not abort, the generator learns that the circuit was used for evaluation, and not as a check circuit. This leaks no information about the input or output of the computation; to do that, the generator must corrupt a majority of the evaluation circuits without modifying a check circuit. An incorrect circuit that goes undetected in one execution has no effect on subsequent executions as long the total amount of incorrect circuits is less than the majority of evaluation circuits.

Using Multiple Evaluators

One of the benefits of our outsourcing scheme is that the state is saved at the generator and cloud allowing the use of different evaluators in each computation. Previously, it was shown a group of users working with a single server using 2P-SFE was not secure against malicious adversaries, as a malicious server and last k parties, also malicious, could replay their portion of the computation with different inputs and gain more information than they can with a single computation [14]. However, this is not a problem in our system as at least one of our servers, either the generator or cloud, must be semi-honest due to non-collusion, which prevents the attack stated above.

Threat Model

As we have many computations involving the same generator and cloud, we have to extend the threat model for how the parties can act in different computations. There can be no collusion in each singular computation. However, the malicious party can change between computations as long as there is no chain of malicious users that link the generator and cloud – this would break the non-collusion assumption.

6 Performance Evaluation

We now demonstrate the efficacy of PartialGC through a comparison with the CMTB outsourcing system. Apart from the optimized cut-and-choose from sS13, PartialGC provides other benefits through generating partial input values after the first execution of a program. On subsequent executions, the partial inputs act to amortize overall costs of execution and bandwidth.

We demonstrate that the evaluator in the system can be a mobile device outsourcing computation to a more powerful system. We also demonstrate that other devices can act as an evaluator, such a server-class machine, to show the generality of this system. Our testing environment includes a 64-core server containing 1 TB of RAM, which we use to model both the Generator and Outsourcing Proxy parties. We run separate virtual machines for the Generator and Outsourcing Proxy roles, giving them each 32 threads. For the evaluator, we use a Samsung Galaxy Nexus phone with a 1.2 GHz dual-core ARM Cortex-A9 and 1 GB of RAM running Android 4.0, connected to the server through an 802.11 54 Mbps WiFi connection in an isolated environment. In our tests, which outsource the computation from a single server process we create that process on our 64-core server as well. We ran the CMTB implementation for comparison tests on the same experimental setup.

6.1 Execution Time

The PartialGC system is particularly well suited to complex computations that require multiple stages and the saving of intermediate state. To date, previous garbled circuit execution systems have focused on single-transaction evaluations, such as computing the "millionaires" problem (i.e., a joint evaluation of which party inputs a greater value without revealing the values of the inputs) or evaluating an AES circuit.

Our evaluation considers two comparisons: the improvement of our system when compared with CMTB without reusing saved values, and comparing our protocol for saving and reusing values against CMTB if such reuse was implemented in that protocol. We also benchmark the overhead for saving and loading values on a per-bit basis for 256 circuits, a necessary number to achieve a security parameter of 2^{-80} in the malicious model. In all cases we run 10 iterations of each test and give timing results with 95% confidence intervals.

The programs used for our evaluation are exemplars of differing input sizes and differing circuit complexities:

Keyed Database: In this program, one party enters a database and keys to it while the other party enters a key that indexes into the database, receiving a database entry for that key. This is an example of a program expressed as a small circuit that has a very large amount of input.

Matrix Multiplication: Here, both parties enter 32-bit numbers to fill a matrix. Matrix multiplication is performed before the resulting matrix is output to both parties. This is an example of a program with a large amount of inputs with a large circuit.

Edit Distance: This program (also known as Levenstein distance) finds the distance between two strings of the same length and returns the difference. This is an example of a program with a small number of inputs and a medium sized circuit.

Millionaires: In this classic SFE program, both parties enter a value, and the result is a one-bit output to each party to let them know whether their value is greater or smaller than that of the other party. This is an example of a small circuit with a large amount of input.

Gate counts for each of our programs can be found in Table 1. The only difference for the programs described above is the additional of a MAC function in PartialGC. We discuss the reason for this check in Section 6.4.

Table 2 shows the results from our experimental tests. In the best case, execution time was reduced by a factor of 32 over CMTB, from 1200 seconds to 38 seconds, a 96% speedup over CMTB. Ultimately, our results show that our system outperforms CMTB when the input checks are the bottleneck. This run-time improvement is due to improvements we added from sS13 and occurs in the keyed database, millionaires, and matrix multiplica-

	16 Circuits			64 Circuits			256 Circuits		
	CMTB	PartialGC		CMTB	PartialGC		CMTB	PartialGC	
KeyedDB 64	18 ± 2%	3.5 ± 3%	5.1x	72 ± 2%	8.3 ± 5%	8.7x	290 ± 2%	26 ± 2%	11x
KeyedDB 128	33 ± 2%	4.4 ± 8%	7.5x	140 ± 2%	9.5 ± 4%	15x	580 ± 2%	31 ± 3%	19x
KeyedDB 256	65 ± 2%	4.6 ± 2%	14x	270 ± 1%	12 ± 6%	23x	1200 ± 3%	38 ± 5%	32x
MatrixMult8x8	48 ± 4%	46 ± 4%	1.0x	110 ± 8%	100 ± 7%	1.1x	400 ± 10%	370 ± 5%	1.1x
Edit Distance 128	21 ± 6%	22 ± 3%	0.95x	47 ± 7%	50 ± 9%	0.94x	120 ± 9%	180 ± 6%	0.67x
Millionaires 8192	35 ± 3%	7.3 ± 6%	4.8x	140 ± 2%	20 ± 2%	7.0x	580 ± 1%	70 ± 2%	8.3x

Table 2: Timing results comparing PartialGC to CMTB without saving any values. All times in seconds.

	CMTB	PartialGC
KeyedDB 64	6,080	20,891
KeyedDB 128	12,160	26,971
KeyedDB 256	24,320	39,131
MatrixMult8x8	3,060,802	3,305,113
Edit Distance 128	1,434,888	1,464,490
Millionaires 8192	49,153	78,775
LCS Incremental 128	4,053,870	87,236
LCS Incremental 256	8,077,676	160,322
LCS Incremental 512	16,125,291	306,368
LCS Full 128	2,978,854	-
LCS Full 256	13,177,739	-

Table 1: Non-XOR gate counts for the various circuits. In the first 6 circucts, the differences between CMTB and PartialGC gate counts is in the consistency checks. The explanation for the difference in size between the incremental versions of longest common substring (LCS) is given in *Reusing Values*.

tions programs. In the other program, edit distance, the input checks are not the bottleneck and PartialGC does not outperform CMTB. The total run-time increase for the edit distance problem is due to overhead of using the new sS13 OT cut-and-choose technique which requires sending each gate to the evaluator for check circuits and evaluation circuits. This is discussed further in Section 6.4. The typical use case we imagine for our system, however, is more like the keyed database program, which has a large amount of inputs and a very small circuit. We expand upon this use case later in this section.

Reusing Values

Until now, we have not given a comparison of PartialGC to CMTB while demonstrating encrypted value reuse. For a true comparison of our system's capabilities, we created an incremental version of an LCS program for both PartialGC and CMTB. This program determines the length of the longest common substring between two strings. Rather than use a single dynamic program for the solution, our version incrementally adds a single bit to the computation each time the program is run and outputs the results each time to the evaluator. We believe this is a realistic comparison to a real-world application that incrementally adds data during each computation.

To properly compare the PartialGC incremental program to CMTB, we created a way to save and reuse values in that system. We saved every desired output bit under a one-time pad and re-entered that information into the next program, as well as the necessary information to decrypt the ciphertext. We add a MAC of the saved information to verify that the party saving the output bits did not modify their contents. The MAC over the output bits are compared to a subsequent MAC computed over the input bits to the next computation, to verify that the user re-entered the correct bits. For our proof of concept implementation we use the AES circuit of KSS12 as the in-garbled circuit MAC function and we use AES to generate a one-time pad inside the garbled circuit. We used the AES circuit, as this was the only cryptographically secure function used in CMTB. Both parties enter private keys to the MAC function. This program is labeled *CMTB-Inc*. We also create a circuit that computes the complete largest common substring up to a given value in one computation. This program is labeled *CMTB-Full*.

The resulting size of the PartialGC and CMTB circuits are shown in Table 1, and the results of the computation are shown in Figure 4. This result shows that saving and reusing values in PartialGC is more efficient than completely rerunning the computation. The additional costs of the one-time pad generation and the MAC performed within the garbled circuit add considerably to the costs of *CMTB-Inc* and in the case of input bit 512, the *CMTB-Inc* program will not run. In the case of the 512-bit *CMTB-Full*, the program would not complete compilation in over 42 hours. In our *CMTB-Inc* program, we assume the cloud saves the output bits (so that we can have multiple phones with a shared private key). We can have multiple evaluators if the result MAC is output to the server under a one-time pad generated by the phones' shared key.

Note that the growth of *CMTB-Inc* and *CMTB-Full* are different. *CMTB-Full* grows at a larger rate (4x for each 2x factor increase) than *CMTB-Inc* (2x for each 2x factor increase), implying that although at first it seems more efficient to rerun the program if small changes are desired in the input, eventually this will not be the case. The number of gates PartialGC requires to run a full ex-

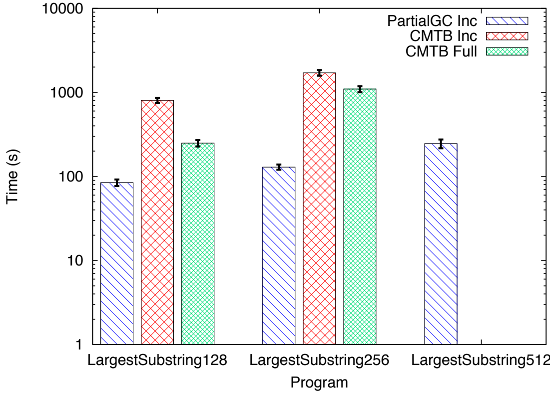


Figure 4: Results from testing our largest common substring (LCS) programs for PartialGC and CMTB. This shows when changing a single input value is more efficient to use PartialGC than either CMTB program. CMTB crashed on running LCS Incremental of size 512 due to memory requirements. We were unable to complete the compilation of CMTB Full of size 512.

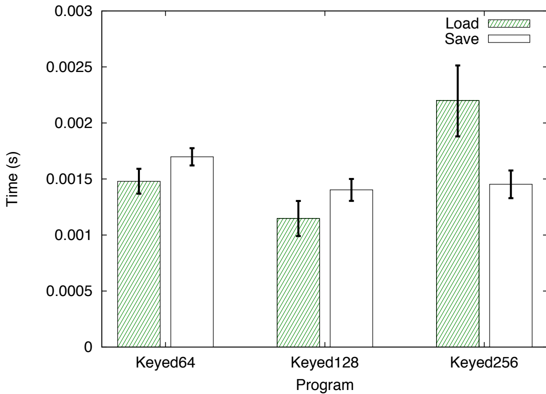


Figure 5: The amount of time it takes to save and load a bit in PartialGC when using 256 circuits.

ecution of the LCS program is the same as *CMTB-Full*.

Overhead of Reusing Values

We created several versions of the keyed database program to determine the runtime of saving and loading the database on a per bit basis using our system (See Figure 5). This figure shows it is possible to save and load a large amount of saved wire labels in a relatively short time. The time to load a wire label is larger than the time to save a value since saving only involves saving the wire label to a file and loading involves reading from a file and creating the partial input gates. Although not shown in the figure, the time to save or load a single bit also increases with the circuit parameter. This is because we need *s* copies of that bit - one for every circuit.

Outsourcing to a Server Process

	256 Circuits		
	CMTB	PartialGC	
KeyedDB 64	64992308	3590416	18x
KeyedDB 128	129744948	3590416	36x
KeyedDB 256	259250228	3590416	72x
MatrixMult8x8	71238860	35027980	2.0x
Edit Distance 128	2615651	4108045	0.64x
Millionaires 8192	155377267	67071757	2.3x

Table 3: Bandwidth comparison of CMTB and PartialGC. Bandwidth counted by instrumenting PartialGC to count the bytes it was sending and receiving and then adding them together. Results in bytes.

PartialGC can be used in other scenarios than just outsourcing to a mobile device. It can outsource garbled circuit evaluation from a single server process and retain performance benefits over a single server process of CMTB. For this experiment, the generator and cloud each have 32 threads and the outsourcing party has a single thread. Table 4 displays these results and shows that in the KeyedDB 256 program, PartialGC has a 92% speedup over CMTB. As with the outsourced mobile case, keyed database problems perform particularly well in PartialGC. Because the computationally-intensive input consistency check is a greater bottleneck on mobile devices than servers, these improvments for most programs are less dramatic. In particular, both edit distance and matrix multiplication programs benefit from higher computational power and their bottlenecks on a server are no longer input consistency; as a result, they execute faster in CMTB than in PartialGC.

6.2 Bandwidth

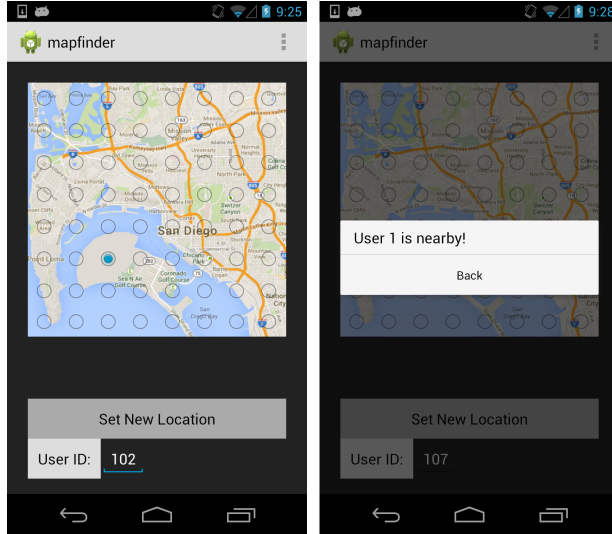
Since the main reason for outsourcing a computation is to save on resources, we give results showing a decrease in the evaluator's bandwidth. Bandwidth is counted by instrumenting the evaluator to count the number of bytes PartialGC sends and receives to either server. Our best result gives a 98% reduction in bandwidth (see Table 3). For the edit distance, the extra bandwidth used in the outsourced oblivious transfer for all circuits, instead of only the evaluation circuits, exceeds any benefit we would otherwise have received.

6.3 Secure Friend Finder

Many privacy-preserving applications can benefit from using PartialGC to cache values for state. As a case study, we developed a privacy-preserving friend finder application, where users can locate nearby friends without any user divulging their exact location. In this application, many different mobile phone clients can use the same

	16 Circuits			64 Circuits			256 Circuits		
	CMTB	PartialGC		CMTB	PartialGC		CMTB	PartialGC	
KeyedDB 64	$6.6 \pm 4\%$	$1.4 \pm 1\%$	4.7x	$27 \pm 4\%$	$5.1 \pm 2\%$	5.3x	$110 \pm 2\%$	$24.9 \pm 0.3\%$	4.4x
KeyedDB 128	$13 \pm 3\%$	$1.8 \pm 2\%$	7.2x	$54 \pm 4\%$	$5.8 \pm 2\%$	9.3x	$220 \pm 5\%$	$27.9 \pm 0.5\%$	7.9x
KeyedDB 256	$25 \pm 4\%$	$2.5 \pm 1\%$	10x	$110 \pm 7\%$	$7.3 \pm 2\%$	15x	$420 \pm 4\%$	$33.5 \pm 0.6\%$	13x
MatrixMult8x8	$42 \pm 3\%$	$41 \pm 4\%$	1.0x	$94 \pm 4\%$	$79 \pm 3\%$	1.2x	$300 \pm 10\%$	$310 \pm 1\%$	0.97x
Edit Distance 128	$18 \pm 3\%$	$18 \pm 3\%$	1.0x	$40 \pm 8\%$	$40 \pm 6\%$	1.0x	$120 \pm 9\%$	$150 \pm 3\%$	0.8x
Millionaires 8192	$13 \pm 4\%$	$3.2 \pm 1\%$	4.1x	$52 \pm 3\%$	$8.5 \pm 2\%$	6.1x	$220 \pm 5\%$	$38.4 \pm 0.9\%$	5.7x

Table 4: Timing results from outsourcing the garbled circuit evaluation from a single server process to a group of 32 processors. Results in seconds.



(a) Map with a point selected.

(b) Map with a point selected and "set new location" pressed with a user present.

Figure 6: Screenshots from our application. The first image shows the map with radio buttons a user can select to indicate position. The second image show the result after "set new position" is pressed when a user is present. The application is set to use 64 different map locations. Map image from Google Maps.

generator (a server application) and can outsource computation to the same cloud. After each computation, the map is updated when PartialGC saves the current state of the map as wire labels. Without PartialGC outsourcing values to the cloud, the wire labels would have to be transferred directly between mobile devices, making a multi-user application difficult or impossible.

We define three privacy-preserving operations that comprise the application's functionality:

MapStart - The three parties (generator, evaluator, cloud) create a "blank" map region, where all locations in the map are blank and remain that way until some mobile party sets a location to his or her ID.

MapSet - The mobile party sets a single map cell to a new value. This program takes in partial values from the generator and cloud and outputs a location selected by the mobile party.

MapGet - The mobile party retrieves the contents of a single map cell. This program retrieves partial values from the generator and cloud and outputs any ID set for that cell to the mobile.

In the application, each user using the *Secure Friend Finder* has a unique ID that represents them on the map. We divide the map into 'cells', where each cell is a set amount of area. When the user presses "Set New Location', the program will first look to determine if that cell is occupied. If the cell is occupied, the user is informed he is near a friend. Otherwise it will update the cell to contain his user ID and remove his ID from his previous location. We assume a maximum of 255 friends in our application since each cell in the map is 8 bits.

Figure 7 shows the performance of these programs in the malicious model with a 2^{-80} security parameter (evaluated over 256 circuits). We consider map regions containing both 256 and 2048 cells. For maps of 256 cells, each operation takes about 30 seconds.[1] As there are three operations for each "Set New Location" event, the total execution time is about 90 seconds, while execution time for 2048 cells is about 3 minutes. The bottleneck of the 64 and 256 cell maps is the outsourced oblivious transfer, which is not affected by the number of cells in the map. The vastly larger circuit associated with the 2048-cell map makes getting and setting values slower operations, but these results show such an application is practical for many scenarios.

6.4 Discussion

Output check

Although the garbled circuit is larger for our output check, this check performs less cryptographic operations for the outsourcing party, as the evaluator only has to perform a MAC on the output of the garbled circuit. We use this check to demonstrate using a MAC can be an

[1] Our 64-cell map, as seen the application screenshots, also takes about 30 seconds for each operation.

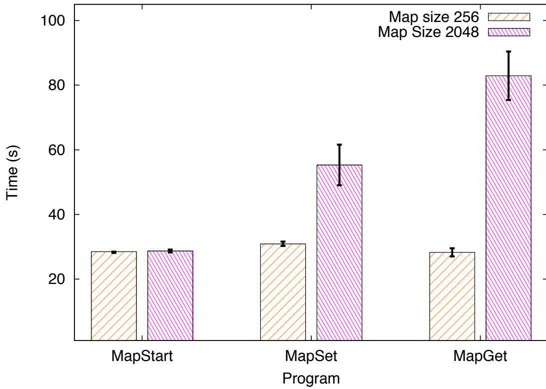


Figure 7: Run time comparison of our map programs with two different map sizes.

efficient output check for a low power device when the computational power is not equivalent across all parties.

Commit Cut-and-Choose vs OT Cut-and-Choose

Our results unexpectedly showed that the sS13 OT cut-and-choose used in PartialGC is actually slower than the KSS12 commit cut-and-choose used in CMTB in our experimental setup. Theoretically, sS13, which requires fewer cryptographic operations, as it generates the garbled circuit only once, should be the faster protocol. The difference between the two cut-and-choose protocols is the network usage – instead of $\frac{2}{5}$ of the circuits (CMTB), *all* the circuits must be transmitted in sS13. The sS13 cut-and-choose is required in our protocol so that the cloud can check that the generator creates the correct gates.

7 Related Work

SFE was first described by Yao in his seminal paper [36] on the subject. The first general purpose implementation of SFE, Fairplay [29], was created in 2004. Fairplay had both a compiler for creating garbled circuits, and an evaluation system for executing them. Computations involving three or more parties have also been examined; one of the earliest examples is FairplayMP [2]. There have been multiple other implementations since, in both semi-honest [6, 9, 15, 16, 37] and malicious settings [25, 34].

Optimizations for garbled circuits include the free-XOR technique [24], garbled row reduction [33], rewriting computations to minimize SFE [22], and pipelining [17]. Pipelining allows the evaluator to proceed with the computation while the generator is creating gates.

KSS12 [26] included both an optimizing compiler and efficient execution system using a parallelized implementation of SFE in the malicious model from [34]. The compiler generally created better circuits than Fairplay, and was much faster and required far fewer resources.

The creation of circuits for SFE in a fast and efficient manner is one of the central problems in the area. Previous compilers, from Fairplay to KSS12, were based on the concept of creating a complete circuit and then optimizing it. The PAL compiler [30] improved such systems by using a simple template circuit, which reduced the amount of memory used by orders of magnitude. As work related to that compiler, another was created [25], that used a more advanced intermediate representation to reduce the amount of disk space used.

Other branches of privacy-preserving computation include homomorphic encryption [3, 11], which performs operations on encrypted data, secret sharing [4], and ordered binary decision diagrams [27]. Another work using privacy-preserving computation using homomorphic encryption was created specifically for mobile devices [7]. There also exist custom protocols for privacy-preserving computations; recently, Kamara et al. [20] showed how to scale server-aided Private Set Intersection to sets with a billion elements using a custom protocol. Previous attempts at creating reusable garbled circuits include theoretical work by Brandão [5], which uses homomorphic encryption, Gentry *et al.* [10], which uses attribute-based functional encryption, and Goldwasser *et al.* [12], which introduces a succinct functional encryption scheme. All of these protocols are very inefficient; none of them provides a performance analysis.

The Quid-Pro-Quo-tocols system [18] allows fast execution with a single bit of leakage. In their system, the garbled circuit is executed twice, with the parties switching roles in the latter execution. In the end, the two parties run a secure protocol to ensure that the output from both executions are equivalent; if this fails, a single bit may be leaked due to the selective failure attack.

8 Conclusion

This paper presents PartialGC, a server-aided SFE scheme allowing the reuse of encrypted values to save the costs of input validation and to allow for the saving of state, such that the costs of multiple computations may be amortized. Compared to the server-aided outsourcing scheme by CMTB, we reduce costs of computation by up to 96% and bandwidth costs by up to 98%. In future work we will consider the generality of the encryption re-use scheme to other SFE evaluation systems and consider large-scale systems problems that benefit from the addition of state, which can open up new and intriguing ways of bringing SFE into the practical realm.

Acknowledgements: This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under contracts FA8750-11-2-0211 and FA8750-13-2-0058. The U.S. Government is authorized to reproduce and distribute reprints

for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government

References

[1] M. Bellare and S. Micali. Non-Interactive Oblivious Transfer and Applications. In *Proceedings of CRYPTO*, 1990.

[2] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the ACM conference on Computer and Communications Security*, 2008.

[3] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-Homomorphic Encryption and Multiparty Computation. In *Proceedings of EUROCRYPT*, 2011.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, 2008.

[5] L. T. A. N. Brandão. Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique. Technical report, University of Lisbon, 2013.

[6] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[7] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. In *Journal of Security and Communication Networks (SCN)*, To appear 2014.

[8] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Proceedings of the USENIX Security Symposium*, 2013.

[9] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] C. Gentry, S. Gorbunov, S. Halevi, V. Vaikuntanathan, and D. Vinayagamurthy. How to compress (reusable) garbled circuits. Cryptology ePrint Archive, Report 2013/687, 2013. `http://eprint.iacr.org/`.

[11] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *Proceedings of CRYPTO*, 2012.

[12] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, STOC '13.

[13] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques annual international conference on Advances in cryptology*, 2008.

[14] S. Halevi, Y. Lindell, and B. Pinkas. Secure Computation on the Web: Computing without Simultaneous Interaction. In *CRYPTO'11*, 2011.

[15] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the ACM conference on Computer and Communications Security*, 2010.

[16] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

[17] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[18] Y. Huang, J. Katz, and D. Evans. Quid-Pro-Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. *IEEE Symposium on Security and Privacy*, (33rd), May 2012.

[19] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proceedings of CRYPTO*, 2003.

[20] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. Technical Report MSR-TR-2013-63, Microsoft Research, 2013.

[21] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[22] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Conference on Data and Application Security and Privacy*, 2013.

[23] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction. In *Proceedings of Symposium on Information Theory in the Benelux*, 2006.

[24] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the international colloquium on Automata, Languages and Programming, Part II*, 2008.

[25] B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of the USENIX Security Symposium*, 2013.

[26] B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[27] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2006.

[28] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[29] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[30] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[31] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the annual ACM Symposium on Theory of Computing (STOC)*, 1999.

[32] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2001.

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

[34] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of EUROCRYPT*, 2011.

[35] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In *Conference on Computer and Communications Security (CCS)*, 2013.

[36] A. C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[37] Y. Zhang, A. Steele, and M. Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *Proceedings of the ACM Conference on Computer Communications Security (CCS)*, 2013.