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

Benjamin Mood
University of Florida
bmood@ufl.edu

Debayan Gupta
Yale University
debayan.gupta@yale.edu

Kevin R. B. Butler
University of Florida
butler@ufl.edu

Joan Feigenbaum
Yale University
joan.feigenbaum@yale.edu

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 while maintaining input and output privacy. The two-party variant, known as 2P-SFE, was first introduced by Yao in the 1980s [39] and was largely a theoretical curiosity. Developments in recent years have made 2P-SFE vastly more efficient [18, 27, 38]. 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, which can happen for non-trivial input sizes and algorithms [8]. One way to allow mobile devices to perform SFE is to use a server-aided computational model [8, 22], allowing the majority of an SFE computation to be "outsourced" to a more powerful device while still preserving privacy. Past approaches, however, have not considered the ways in which mobile computation differs from the desktop. Often, the mobile device is called upon to perform *incremental* operations that are continuations of a previous computation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660285.

Consider, for example, a "friend-finder" application where the location of users is updated periodically to determine whether a contact is in proximity. Traditional applications disclose location information to a central server. A privacy-preserving friend-finder could perform these operations in a mutually oblivious fashion. However, every incremental location update would require a full re-evaluation of the function with fresh inputs in a standard SFE solution. Our examination of an outsourced SFE scheme for mobile devices by Carter et al. [8] (hereon CMTB), determined that the cryptographic consistency checks performed on the inputs to an SFE computation *themselves* can constitute the greatest bottleneck to performance.

Additionally, many other applications require the ability to save state, a feature that current garbled-circuit implementations do not possess. 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 (we use a cut-and-choose garbled circuit protocol) even if one is restricted to primitives that are part of standard garbled circuits. Our system, PartialGC, which is based on CMTB, 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, which requires either generating inside the circuit, or inputting, a very large one-time pad, both complex operations. Through the use of improved input validation mechanisms proposed by shelat and Shen [38] (hereon sS13) and new methods of *partial input* gate checks and evaluation, we improve on previous proposals. There are other

approaches to the creation of reusable garbled circuits [13, 10, 5], and previous work on reusing encrypted values in the ORAM model [30, 11, 31], but these earlier schemes have not been implemented. By contrast, we have implemented our scheme and found it to be both practical and efficient; we provide a performance analysis and a sample application to illustrate its feasibility (Section 6), as well as a simplified example execution (Appendix C).

By breaking a large program into smaller pieces, our system allows interactive I/O throughout the garbled circuit computation. To the best of our knowledge this is the first practical protocol for performing interactive I/O in the middle of a cut-and-choose garbled circuit computation.

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.

Figure 1 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 values from previous executions. In order to reuse these values, information from both parties – the generator and the evaluator – has to be saved. In our protocol, it is the cloud – rather than the evaluator – that saves information. This allows multiple distinct evaluators to participate in a large computation over time by saving state in the cloud between 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 2). 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. Our friend finder application, built for an Android device, reflects this usage model and allows multiple friends to share their intermediate values in a cloud. Other friends 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.

By incorporating our optimizations, we give the following 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.
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 know about each other.

The remainder of our paper is organized as follows: Section 2 provides some background on SFE. Section 3 introduces the concept of partial garbled circuits in detail. The PartialGC protocol and its implementation are described in Section 4, while its security is analyzed in Section 5. Section 6 evaluates PartialGC and introduces the friend finder application.

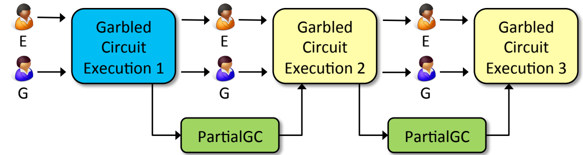


Figure 1: PartialGC Overview. E is evaluator and G is generator. The blue box is a standard execution that produces partial outputs (garbled values); yellow boxes represent executions that take partial inputs and produce partial outputs.

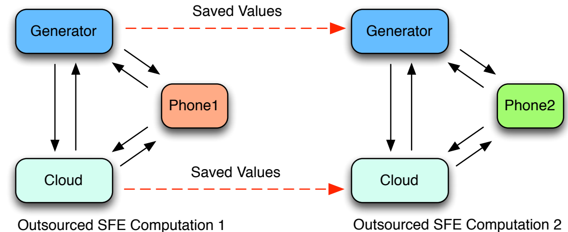


Figure 2: Our system has three parties. Only the cloud and generator have to save intermediate values - this means that we can have different phones in different computations.

Section 7 discusses related work and Section 8 concludes.

2. BACKGROUND

Secure function evaluation (SFE) addresses scenarios where two or more mutually distrustful parties $P_1, \ldots, P_n$, with private inputs $x_1, \ldots, x_n$, want to compute a given function $y_i = f(x_1, \ldots, x_n)$ (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 – if there was, 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 [39]. The area has been studied extensively by the cryptography community, leading to the creation of the first general purpose platform for SFE, Fairplay [32] in the early 2000s. Today, there exist many such platforms [6, 9, 16, 17, 26, 37, 40].

The classic platforms for 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, 20, 34, 35] 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* [32].

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.

Selective failure of input If the generator does not offer up correct input wires to the evaluator, and the evaluator selects the wire that was not created properly, the generator can learn up to a single bit of information based on whether the computation produced correct outputs.

Input consistency If either party's input is not consistent across all circuits, then it might be possible for extra information to be retrieved.

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.

2.1.1 Non-collusion

CMTB assumes non-collusion, as quoted below:
"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. [22]. Non-collusion does not mean the parties are trusted; it only means the two parties are not working together (i.e. both malicious). In CMTB, any individual party that attempts to cheat to gain additional information will still be caught, but collusion between multiple parties could leak information. For instance, the generator could send the cloud the keys to decrypt the circuit and see what the intermediate values are of the garbled function.

3. PARTIAL GARBLED CIRCUITS

We introduce the concept of *partial garbled circuits* (PGCs), which allows 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 transforms 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.

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 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 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. During the evaluation and check phase, E must be able to verify the mapping G sent. G must have either 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

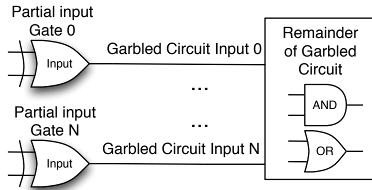


Figure 3: This figure shows how we create a single *partial input gate* for each input bit for each circuit and then link the *partial input gates* to the remainder of the circuit.

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 full details of the outsourced oblivious transfer [8] and the generator's input consistency check [38] subprotocols that we use as primitives in our protocol.

Our system operates in the same threat model as CMTB (see Section 2.1.1): we are secure against a malicious adversary under the assumption of non-collusion. A description of the CMTB protocol is available in Appendix A.

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_i - The ith circuit.

$CKey_i$ - Circuit key used for the free XOR optimization [25]. The key is randomly generated and then used as the difference between the 0 and 1 wire labels for a circuit C_i.

$CSeed_i$ - This value is created by the generator's PRNG and is used to generate a particular circuit C_i.

$POut\#_{i,j}$ - The *partial output* values are the encrypted wire values output from an SFE computation. These are encrypted garbled circuit values that can be reused in another garbled circuit computation. # is replaced in our protocol description with either a 0, 1, or x, signifying whether it represents a 0, 1, or an unknown value (from the cloud's point of view). i denotes the circuit the $POut$ value came from and j denotes the wire of the $POut_i$ circuit.

$PIn\#_{i,j}$ - The *partial input* values are the re-entered $POut$ values after they have been obfuscated to remove the circuit key from the previous computation. These values are input to the *partial input gates*. #, i, and j, are the same as above.

$GIn\#_{i,j}$ - The *garbled circuit input* values are the results of the partial input gates and are input into the remaining garbled circuit, as shown in Figure 3. #, i, and j, are the same as above.

Partial Input Gates - These are garbled gates that take in PIn values and output GIn values. Their purpose is to transform the PIn values into values that are under $CKey_i$ for the current circuit.

4.2 Protocol

Each computation is self-contained; other than what is explicitly described as saved in the protocol, each value or property is only used for a single part of the computation (*i.e.* randomness is different across computations).

Algorithm 0: PartialComputation

Input : Circuit_File, Bit_Security, Number_of_Circuits, Inputs, Is_First_Execution
Output: Circuit File Output
`Cut_and_Choose(`*is_First_Execution*`)`
Eval_Garbled_Input ← `Evaluator_Input(`*Eval_Select_Bits, Possible_Eval_Input*`)`
`Generator_Input_Check(`*Gen_Input*`)`
Partial_Garbled_Input ← `Partial_Input(`$Partial_Output_{time-1}$`)`
Garbled_Output, Partial_Output ←
`Circuit_Execution(`*Garbled_Input (Gen, Eval, Partial)*`)`
`Circuit_Output(`*Garbled_Output*`)`
`Partial_Output(`*Partial_Output*`)`

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

Private Inputs: The evaluator's input *evlInput* and generator's input *genInput*.

Outputs: The evaluator and generator can both receive garbled circuit outputs.

Phase 1: Cut-and-choose

We modify the cut-and-choose mechanism described in sS13 as we have an extra party involved in the computation. In this cut-and-choose, the cloud selects which circuits are evaluation circuits and which circuits are check circuits,

$$circuitSelection = rand()$$

where *circuitSelection* is a bit vector of size S; N evaluation circuits and $S-N$ check circuits are selected where $N = \frac{2}{5}S$. The generator does not learn the circuit selection.

The generator generates garbled versions of his input and circuit seeds for each circuit. He encrypts these values using unique 1-time XOR pad keys. For $0 \leq i < S$,

$$\begin{aligned} CSeed_i &= rand() \\ garbledGenInput_i &= garble(genInput, rand()) \\ checkKey_i &= rand() \\ evlKey_i &= rand() \\ encSeedIn_i &= CSeed_i \oplus evlKey_i \\ encGarbledIn_i &= garbledGenInput_i \oplus checkKey_i \end{aligned}$$

The cloud and generator perform an oblivious transfer where the generator offers up decryption keys for his input and decryption keys for the circuit seed for each circuit. The cloud can select the key to decrypt the generator's input or the key to decrypt the circuit seed for a circuit but not both. For each circuit, if the cloud selects the decryption key for the circuit seed in the oblivious transfer, then the circuit is used as a check circuit.

Algorithm 1: Cut_and_Choose

```
Input  : is_First_Execution
if is_First_Execution then
    circuitSelection ← rand() // bit-vector of size S
N ← (2/5)S // Number of evaluation circuits
//Generator creates his garbled input and circuit seeds for each circuit
for i ← 0 to S do
    CSeed_i ← rand()
    garbledGenInput_i ← garble(genInput, rand())
    //generator creates or loads keys
    if is_First_Execution then
        checkKey_i ← rand()
        evlKey_i ← rand()
    else
        loadKeys();
        checkKey_i ← hash(loadedCheckKey_i)
        evlKey_i ← hash(loadedEvlKey_i)
    // encrypts using unique 1-time XOR pads
    encSeedIn_i ← CSeed_i ⊕ evlKey_i
    encGarbledIn_i ← garbledGenInput_i ⊕ checkKey_i
if is_First_Execution then
    // generator offers input OR keys for each circuit seed
    selectedKeys ←
    OT(circuitSelection, {evlKey, checkKey})
else
    loadSelectedKeys()
for i ← 0 to S do
    genSendToEval(hash(checkKey_i),
    hash(evaluationKey_i))
for i ← 0 to S do
    cloudSendToEval(hash(selectedKey_i), isCheckCircuit_i)
// If all values match, the evaluator learns split, else abort.
for i ← 0 to S do
    j ← isCheckCircuit_i
    correct ← (recievedGen_{i,j} == recievedEvl_i)
    if !correct then
        abort()
```

$$selectedKeys = OT(circuitSelection, \{evlKey, checkKey\})$$

If the cloud selects the key for the generator's input then a given circuit is used as an evaluation circuit. Otherwise, the key for the circuit seed was selected and the circuit is a check circuit. The decryption keys are saved by both the generator and cloud in the event a computation uses saved values from this computation.

The generator sends the encrypted garbled inputs and check circuit information for all circuits to the cloud. The cloud decrypts the information he can decrypt using its keys.

The evaluator must also learn the circuit split. The generator sends a hash of each possible encryption key the cloud could have selected to the evaluator for each circuit as an ordered pair. For $0 \leq i < S$,

$$genSend(hash(checkKey_i), hash(evaluationKey_i))$$

The cloud sends a hash of the value received to the evaluator for each circuit. The cloud also sends bits to indicate which circuits were selected as check and evaluation circuits to the evaluator. For $0 \leq i < S$,

$$cloudSend(hash(selectedKey_i), isCheckCircuit_i)$$

The evaluator compares each hash the cloud sent to one of the hashes the generator sent, which is selected by the circuit selection sent by the cloud. For $0 \leq i < S$,

$$j = isCheckCircuit_i$$
$$correct = (receivedGen_{i,j} == receivedEvl_i)$$

If all values match, the evaluator uses the $isCheckCircuit_i$ to learn the split between check and evaluator circuits. Otherwise the evaluator safely aborts.

We only perform the cut-and-choose oblivious transfer for the initial computation. For any subsequent computations, the generator and evaluator hash the saved decryption keys and use those hashes as the new encryption and decryption keys. The circuit split selected by the cloud is saved and stays the same across computations.

Phase 2: Oblivious Transfer

Algorithm 2: Evaluator_Input

```
Input  : Eval_Select_Bits, Possible_Eval_Input
Output: Eval_Garbled_Input
// cloud gets selected input wires // generator offers both
possible input wire values for each input wire; evaluator selects
its input
outSeeds = BaseOOT(bitsEvl, possibleInputs).
// the generator sends unique IKey values for each circuit to the
evaluator
for i ← 0 to S do
    genSendToEval(IKey_i)
// the evaluator sends IKey values for all evaluation circuits to
the cloud
for i ← 0 to S do
    if !isCheckCircuit(i) then
        EvalSendToCloud(IKey_i)
// cloud uses this to learn appropriate inputs
for i ← 0 to S do
    for j ← 0 to len(evlInputs) do
        if !isCheckCircuit(i) then
            inputEvl_ij ← hash(IKeys_i, outSeeds_j)
return inputEvl
```

We use the base outsourced oblivious transfer (OOT) of CMTB. In this transfer the generator inputs both possible input wire values for each evaluator's input wire while the evaluator inputs its own input. After the OOT is performed, the cloud has the selected input wire values, which represent the evaluator's input.

As with CMTB, which uses the results from a single OOT as seeds to create the evaluator's input for all circuits, the cloud in our system also uses seeds from a single base OT (called "BaseOOT" below) to generate the input for the evaluation circuits. The cloud receives the seeds for each input bit selected by the evaluator.

$$outSeeds = BaseOOT(evlInputSeeds, evlInput).$$

The generator creates unique keys, $IKey$, for each circuit and sends each key to the evaluator. The evaluator sends the keys for the evaluation circuits to the cloud. The cloud then uses these values to attain the evaluator's input. For $0 \leq i < S$, for $0 \leq j < len(evlInputs)$ where $!isCheckCircuit(i)$,

$$inputEvl_{ij} = hash(IKey_i, outSeeds_j)$$

Phase 3: Generator's Input Consistency Check

We use the input consistency check of sS13. In this check, a universal hash is used to prove consistency of the generator's input across each evaluation circuit. Simply put, if the hash is different in any of the evaluation circuits, we know the generator did not enter consistent input. More formally, a hash of the generator's input is taken for each circuit. For $0 < i < S$ where $!isCheckCircuit(i)$,

$$t_i = UHF(garbledGenInput_i, C_i)$$

Algorithm 3: Generator_Input_Check

Input : Generator_Input
// The cloud takes a hash of the generator's input or each evaluation circuit **for** $i \leftarrow 0$ **to** S **do**
  if *isCheckCircuit(i)* **then**
    $t_i \leftarrow UHF(garbledGenInput_i)$
//If a single hash is different then the cloud knows the generator tried to cheat.
$correct \leftarrow ((t_0 == t_1)\&(t_0 == t_2)\& \ldots \&(t_0 == t_{N-1}))$
if *!correct* **then**
  abort()

The results of these universal hashes are compared. If a single hash is different then the cloud knows the generator tried to cheat and safely aborts.

$$correct = ((t_0 == t_1)\&(t_0 == t_2)\& \ldots \&(t_0 == t_{N-1}))$$

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

Generation

For $0 \leq i < S$, for $0 \leq j < len(savedWires)$, the generator creates a *partial input gate*, which transforms a wire's saved values, $POut0_{i,j}$ and $POut1_{i,j}$, into wire values that can be used in the current garbled circuit execution, $GIn0_{i,j}$ and $GIn1_{i,j}$. For each circuit, C_i, the generator creates a pseudorandom transformation value R_i, to assist with the transformation.

For each set of $POut0_{i,j}$ and $POut1_{i,j}$, the generator XORs each value with R_i. Both results are then hashed, and put through a function to determine the new permutation bit, as hashing removes the old permutation bit.

$$t0 = hash(POut0_{i,j} \oplus R_i)$$
$$t1 = hash(POut1_{i,j} \oplus R_i)$$
$$PIn0_{i,j}, PIn1_{i,j} = setPPBitGen(t0, t1)$$

This function, *setPPBitGen*, pseudo-randomly finds a bit that is different between the two values of the wire and notes that bit to be the permutation bit. *setPPBitGen* is seeded from $CSeed_i$, allowing the cloud to regenerate these values for the check circuits.

For each $PIn0_{i,j}, PIn1_{i,j}$ pair, a set of values, $GIn0_{i,j}$ and $GIn1_{i,j}$, are created under the master key of C_i, $CKey_i$, – where $CKey_i$ is the difference between 0 and 1 wire labels for the circuit. In classic garbled gate style, two truth table values, $TT0_{i,j}$ and $TT1_{i,j}$, are created such that:

$$TT0_{i,j} \oplus PIn0_{i,j} = GIn0_{i,j}$$
$$TT1_{i,j} \oplus PIn1_{i,j} = GIn1_{i,j}$$

The truth table, $TT0_{i,j}$ and $TT1_{i,j}$, is permuted so that the permutation bits of $PIn0_{i,j}$ and $PIn1_{i,j}$ tell the cloud which entry to select. Each *partial input gate*, consisting of the permuted $TT0_{i,j}$, $TT1_{i,j}$ values and the bit location from *setPPBitGen* is sent to the cloud. Each R_i is also sent to the cloud.

Check

For $0 \leq i < S$ where $isCheckCircuit(i)$, for $0 \leq j < len(savedWires)$, the cloud receives the truth table information, $TT0_{i,j}, TT1_{i,j}$, and bit location from *setPPBitGen*, and proceeds to regenerate the gates based on the check circuit information. The cloud uses R_i (sent by the generator), $POut0_{i,j}$ and $POut1_{i,j}$ (saved during the previous execution), and $CSeed_i$ (recovered during the cut-and-choose) to

Algorithm 4: Partial_Input

Input : Partial_Output
Output: Partial_Garbled_Input
// Generation: the generator creates a *partial input gate*, which transforms a wire's saved values, $POut0_{i,j}$ and $POut1_{i,j}$, into values that can be used in the current garbled circuit execution, $GIn0_{i,j}$ and $GIn1_{i,j}$.
for $i \leftarrow 0$ **to** S **do**
  $R_i \leftarrow PRNG.random()$
  for $j \leftarrow 0$ **to** $len(savedWires)$ **do**
    $t0 \leftarrow hash(POut0_{i,j} \oplus R_i)$
    $t1 \leftarrow hash(POut1_{i,j} \oplus R_i)$
    $PIn0_{i,j}, PIn1_{i,j} \leftarrow setPPBitGen(t0, t1)$
    $GIn0_{i,j} \leftarrow TT0_{i,j} \oplus PIn0_{i,j}$
    $GIn1_{i,j} \leftarrow TT1_{i,j} \oplus PIn1_{i,j}$
    GenSendToCloud(Permute([$TT0_{i,j}$, $TT1_{i,j}$]), permute_bit_locations)
  GenSendToCloud(R_i)
// Check: The cloud checks the gates to make sure the generator didn't cheat
for $i \leftarrow 0$ **to** S **do**
  if *isCheckCircuit(i)* **then**
    for $j \leftarrow 0$ **to** $len(savedWires)$ **do**
      // the cloud has received the truth table information, $TT0_{i,j}, TT1_{i,j}$, bit locations from *setPPBitGen*, and R_i
      $correct \leftarrow (generateGateFromInfo() == receivedGateFromGen())$
      // If any gate does not match, the cloud knows the generator tried to cheat.
      if *!correct* **then**
        abort();
// Evaluation
for $i \leftarrow 0$ **to** S **do**
  if *!isCheckCircuit(i)* **then**
    for $j \leftarrow 0$ **to** $len(savedWires)$ **do**
      //The cloud, using the previously saved $POutx_{i,j}$ value, and the location (point and permute) bit sent by the generator, creates $PInx_{i,j}$
      $PInx_{i,j} \leftarrow setPPBitEval(hash(R_i \oplus POutx_{i,j}), location)$
      // Using $PInx_{i,j}$, the cloud selects the proper truth table entry $TTx_{i,j}$ from either $TT0_{i,j}$ or $TT1_{i,j}$ to decrypt
      // Creates $GInx_{i,j}$ to enter into the garbled circuit
      $GInx_{i,j} \leftarrow TTx_{i,j} \oplus POutx_{i,j}$
return GIn;

generate the *partial input gates* in the same manner as described previously. The cloud then compares these gates to those the generator sent. If any gate does not match, the cloud knows the generator tried to cheat and safely aborts.

Evaluation

For $0 \leq i < S$ where $!isCheckCircuit(i)$, for $0 \leq j < len(savedWires)$ the cloud receives the truth table information, $TTa_{i,j}, TTb_{i,j}$ and bit location from *setPPBitGen*. a and b are used to denote the two permuted truth table values. The cloud, using the previously saved $POutx_{i,j}$ value, creates the $PInx_{i,j}$ value:

$$PInx_{i,j} = setPPBitEval(hash(R_i \oplus POutx_{i,j}), location)$$

location is the location of the point and permute bit sent by the generator. Using the point and permute bit of $PInx_{i,j}$, the cloud selects the proper truth table entry $TTx_{i,j}$ from either $TTa_{i,j}$ or $TTb_{i,j}$ to decrypt, creates $GInx_{i,j}$ and then enters $GInx_{i,j}$ into the garbled circuit.

$$GInx_{i,j} = TTx_{i,j} \oplus POutx_{i,j}$$

Phase 5: Circuit Generation and Evaluation

```
Algorithm 5: Circuit_Execution
Input  : Generator_Input, Evaluator_Input, Partial_Input
Output: Partial_Output, Garbled_Output
// The generator generates each garbled gate and sends it to the
cloud. Depending on whether the circuit is a check or evaluation
circuit, the cloud verifies that the gate is correct or evaluates the
gate.
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← genGate(C_i, j)
        send(g)
// the cloud receives all gates for all circuits, and then checks
OR evaluates each circuit
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← recvGate()
        if isCheckCircuit(i) then
            if ! verifyCorrect(g) then
                abort()
        else
            eval(g)
return Partial_Output, Garbled_Output
```

Circuit Generation

The generator generates each garbled gate for each circuit and sends them to the cloud. Since the generator does not know the check and evaluation circuit split, nothing changes for the generation for check and evaluation circuits. For $0 \leq i < S$, For $0 \leq j < len(circuit)$,

$$g = genGate(C_i, j), send(g)$$

Circuit Evaluation and Check

The cloud receives each garbled gate for all circuits. For evaluation circuits the cloud evaluates those garbled gates. For check circuits the cloud generates the correct gate, based on the circuit seed, and is able to verify it is correct.

For $0 \leq i < S$, For $0 \leq j < len(circuit)$, $g = recvGate()$, $if(isCheckCircuit(j))\ verifyCorrect(g)\ else\ eval(g)$

If a garbled gate is found not to be correct, the cloud informs the evaluator and generator of the incorrect gate and safely aborts.

Phase 6: Output and Output Consistency Check

```
Algorithm 6: Circuit_Output
Input  : Garbled_Output
// a MAC of the output is generated inside the garbled circuit,
and both the resulting garbled circuit output and the MAC are
encrypted under a one-time pad.
outEvlComplete = outEvl||MAC(outEvl)
result = (outEvlMAC == MAC(outEvl))
if !result then
    abort() // output check fail
```

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.

$$outEvlComplete = outEvl||MAC(outEvl)$$

Both the resulting garbled circuit output and the MAC are encrypted under a one-time pad. The generator can also have output verified in the same manner. The cloud sends the corresponding encrypted output to each party.

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

$$result = (outEvlMAC == MAC(outEvl))$$

Phase 7: Partial Output

```
Algorithm 7: Partial_Output
Input  : Partial_Output
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        //The generator saves both possible wire values
        GenSave(Partial_Output0_{i,j})
        GenSave(Partial_Output1_{i,j})
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        if  isCheckCircuit(i) then
            EvlSave(Partial_Output0_{i,j})
            EvlSave(Partial_Output1_{i,j})
        else
            // circuit is evaluation circuit
            EvlSave(Partial_OutputX_{i,j})
```

The generator saves both possible wire values for each partial output wire. For each evaluation circuit the cloud saves the partial output wire value. For check circuits the cloud saves both possible output values.

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.

We modified the KSS12 [27] compiler to allow for the saving of intermediate wire labels and loading wire labels from a different SFE computation. 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 compare 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. The fixes 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 [38], we focus on our additions, and defer to the original papers for detailed proofs of those protocols. Due to space constraints, we do not provide a formal proof here; a complete proof will be provided in the technical report.

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 [38], 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 [15]. 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 obviates 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 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 show that other devices, such as server-class machines, can act as evaluators, 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 programs for the Generator and Outsourcing Proxy,

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 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 under the same 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. 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 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. Other than varying the number of circuits our system parameters are set for 80-bit security.

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 (Levenstein) Distance: This program 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 multiplications 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.

	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 circuits, the difference 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*.

Reusing Values

For a test of our system's wire saving capabilities we tested a dynamic programming problem, longest common substring, in both PartialGC and CMTB. This program determines the length of the longest common substring between two strings. Rather than use a single computation for the solution, our version incrementally adds a single bit of input to both strings each time the computation 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 where it is faster to save the intermediate state and add to it after seeing an intermediate result than rerun the entire computation many times after seeing the result.

For our testing, PartialGC uses our technique to reuse wire values. In CMTB, we save each desired internal bit under a one-time pad and re-enter them into the next computation, as well as the information needed to decrypt the ciphertext. We use a MAC (the AES circuit of KSS12) to verify that the party saving the output bits did not modify them. We also use AES to generate a one-time pad inside the garbled circuit. We use AES as this is the only cryptographically secure function used in CMTB. Both parties enter private keys to the MAC functions. This program is labeled *CMTB-Inc*, for CMTB *incremental*. The size of this program represents the size of the total strings. We also created a circuit that computes the complete longest common substring in one computation labeled *CMTB-Full*.

The resulting size of the PartialGC and CMTB circuits are shown in Table 1, and the results are shown in Figure 4. This result shows that saving and reusing values in PartialGC is more efficient than completely rerunning the computation. The input consistency check adds considerably to the memory use on the phone for *CMTB-Inc* and in the case of input bit 512, the *CMTB-Inc* program will not complete. 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 multiple phones can have a shared private key. We do

	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.

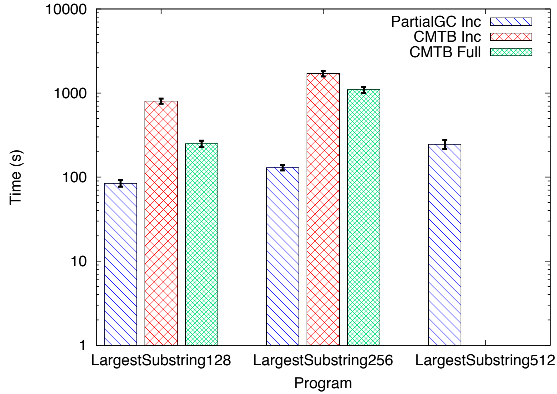


Figure 4: Results from testing our largest common substring (LCS) programs for PartialGC and CMTB. This shows when changing a single input value is more efficient under PartialGC than either CMTB program. CMTB crashed on running LCS Incremental of size 512 due to memory requirements. We were unable to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

not provide a full program due to space requirements.

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. Even with a more efficient AES function, *CMTB-Inc* would not be faster as the bottleneck is the input, not the size of the circuit.

The overhead of saving and reusing values is discussed further in Appendix B.

Outsourcing to a Server Process

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 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 improvements 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.

	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.

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 making 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 use a consistent generator (a server application) and outsource computation to a cloud. The generator must be the same for all computations; the cloud must be the same for each computation. The cloud and generator are two different parties. 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:

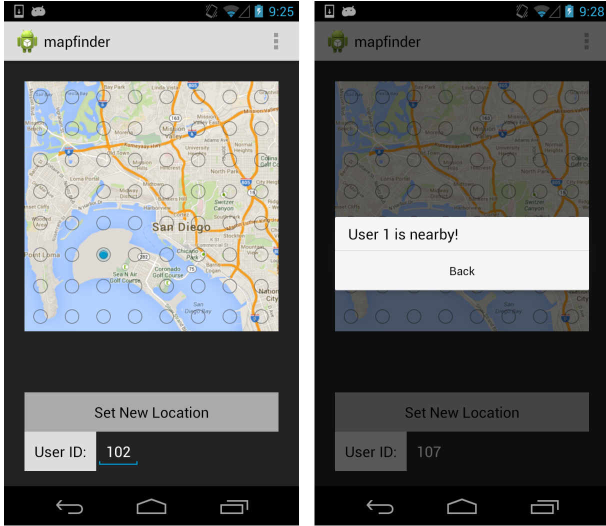
MapStart - The three parties (generator, evaluator, cloud) create a "blank" map region, where all locations in the map are blank and remain that way until some mobile party sets a location to his or her ID.

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.

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

Table 4: Timing results from outsourcing the garbled circuit evaluation from a single server process. Results in seconds.



(a) Location selected. (b) After computation.

Figure 5: Screenshots from our application. (a) shows the map with radio buttons a user can select to indicate position. (b) 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.

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 the cell is updated 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 6 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.

Example - As an example, two friends initiate a friend finder computation using Amazon as the cloud and Face-

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

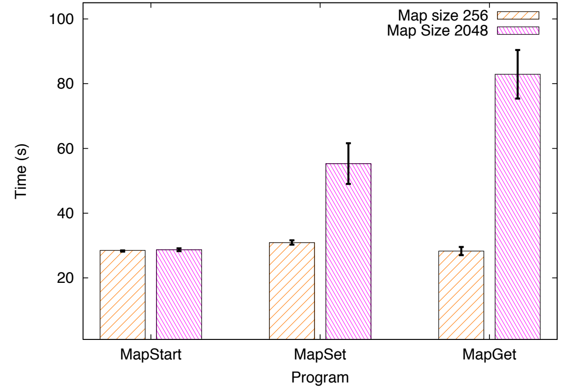


Figure 6: Run time comparison of our map programs with two different map sizes.

book as the generator. The first friend goes out for a coffee at a café. The second friend, riding his bike, gets a message that his friend is nearby and looks for a few minutes and finds him in the café. Using this application prevents either Amazon or Facebook from knowing either user's location while they are able to learn whether they are nearby.

6.4 Discussion

Analysis of improvements

We analyzed our results and found the improvements came from three places: the improved sS13 consistency check, the saving and reusing of values, and the fixed oblivious transfer. In the case of the sS13 consistency check, there are two reasons for the improvement, first there is less network traffic and second it does not use exponentiations. In the case of saving and reusing values, we save time by the faster input consistency check and not requiring a user to recompute a circuit multiple times. Lastly, we reduced the runtime and bandwidth by fixing parts of the OOT. The previous outsourced oblivious transfer performed the primitive OT S times instead of a single time, which turn forced many extra exponentiations. Each amount of improvement varies depending upon the circuit.

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 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 [39] on the subject. The first general purpose platform for SFE, Fairplay [32], was created in 2004. Fairplay had both a compiler for creating garbled circuits, and a run-time 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, 16, 17, 40] and malicious settings [26, 37].

Optimizations for garbled circuits include the free-XOR technique [25], garbled row reduction [36], rewriting computations to minimize SFE [23], and pipelining [18]. Pipelining allows the evaluator to proceed with the computation while the generator is creating gates.

KSS12 [27] included both an optimizing compiler and an efficient run-time system using a parallelized implementation of SFE in the malicious model from [37].

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. PAL [33] improved such systems by using a simple template circuit, reducing memory usage by orders of magnitude. PCF [26] built from this and used a more advanced representation to reduce the disk space used.

Other methods for performing MPC involve homomorphic encryption [3, 12], secret sharing [4], and ordered binary decision diagrams [28]. A general privacy-preserving computation protocol that uses homomorphic encryption and was designed specifically for mobile devices can be found in [7]. There are also custom protocols designed for particular privacy-preserving computations; for example, Kamara et al. [21] showed how to scale server-aided Private Set Intersection to billion-element sets with a custom protocol.

Previous reusable garbled-circuit schemes include that of Brandão [5], which uses homomorphic encryption, Gentry *et al.* [10], which uses attribute-based functional encryption, and Goldwasser *et al.* [13], which introduces a succinct functional encryption scheme. These previous works are purely theoretical; none of them provides experimental performance analysis. There is also recent theoretical work on reusing encrypted garbled-circuit values [30, 11, 31] in the ORAM model; it uses a variety of techniques, including garbled circuits and identity-based encryption, to execute the underlying low-level operations (program state, read/write queries, etc.). Our scheme for reusing encrypted values is based on completely different techniques; it enables us to do new kinds of computations, thus expanding the set of things that can be computed using garbled circuits.

The Quid-Pro-Quo-tocols system [19] allows fast execution with a single bit of leakage. The garbled circuit is executed twice, with the parties switching roles in the latter execution, then running 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%. Future work will consider the generality of the encryption re-use scheme to other SFE evaluation systems and 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. It is also supported in part by the U.S. National Science Foundation under grant numbers CNS-1118046 and CNS-1254198. 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, NSF, or the U.S. Government.

9. 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, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In *Advances in Cryptology–EUROCRYPT 2014*, pages 405–422. Springer Berlin Heidelberg, 2014.

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

[13] 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, 2013.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] S. Lu and R. Ostrovsky. How to garble ram programs. In *Advances in Cryptology–EUROCRYPT 2013*, pages 719–734. Springer Berlin Heidelberg, 2013.

[31] S. Lu and R. Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

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

[33] 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.

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

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

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

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

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

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

[40] 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.

APPENDIX

A. 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 Goyal et al.'s [14] random seed check, which was implemented by Kreuter et al. [27] to combat generation of incorrect circuits. This technique uses a cut-and-choose style protocol to determine whether 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 [20] to reduce the number of OTs.

CMTB uses an encoding technique from Lindell and Pinkas [29], 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. [27] 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 [29], 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 [37].

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.

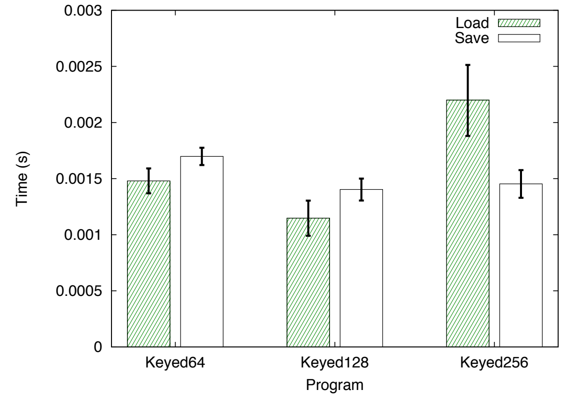


Figure 7: The amount of time it takes to save and load a bit in PartialGC when using 256 circuits.

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 [24] 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. [27].

B. 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 7). 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.

C. EXAMPLE PROGRAM

In this section we describe the execution of an *attendance application*. Imagine a building where the host wants each user to sign in from their phones to keep a log of the guests, but also wants to keep this information secret.

This application has three distinct programs. The first program initializes a counter to a number input by the evaluator. The second program, which is used until the last program is called, takes in a name and increments the counter by one. The last program outputs all names and returns the count of users.

For this application, users (rather, their mobile phones) assume the role of evaluators in the protocol (Section 4).

First, the host runs the initial program to initialize a

database. We cannot execute the second program to add names to the log until this is done, lest we reveal that there is no memory saved (*i.e.*, there is no one else present).

Protocol in Brief: In this first program, the cut-and-choose OT is executed to select the circuit split (the circuits that are for evaluation and generation). Both parties save the decryption keys: the cloud saves the keys attained from the OT and the generator saves both possible keys that could have been selected by the cloud. The evaluator performs the OOT with the other parties to input the initial value into the program. There is no input by the generator so the generator's input check does not execute. There is no partial input so that phase of the protocol is skipped. The garbled circuit to set the initial value is executed; while there is no output to the generator or evaluator, a partial output is produced: the cloud saves the garbled wire value, which it possesses, and the generator saves both possible wire values (the generator does not know what value the cloud has, and the cloud does not know what the value it has saved actually represents). The cloud also saves the circuit split.

Saved memory after the program execution (when the evaluator inputs 0 as the initial value):

Count
0
Saved Guests

Guest 1 then enters the building and executes the program, entering his name ("Guest 1") as input.

Protocol in Brief: In this second program, the cut-and-choose OT is not executed. Instead, both the generator and cloud load the saved decryption key values, hash them, and use those values for the check and evaluation circuit information (instead of attaining new keys through an OT, which would break security). The new keys are saved, and the evaluator then performs the OOT for input. The generator does not have any input in this program so the check for the generator's input is skipped. Since there exists a partial input, the generator loads both possible wire values and creates the partial input gates. The cloud loads the attained values, receives the partial input gates from the generator, and then executes (and checks) the partial input gates to receive the garbled input values. The garbled circuit is then executed and partial output saved as before (although there is more data to save for this program as there is a name present in the database).

After executing the second program the memory is as follows:

Count
1
Saved Guests
Guest1

Guest 2 then enters the dwelling and runs the program. The execution is similar to the previous one (when Guest 1 entered), except that it's executed by Guest 2's phone.

At this point, the memory is as follows:

Count
2
Saved Guests
Guest1
Guest2

Guest 3 then enters the dwelling and executes the program as before. At this point, the memory is as follows:

Count
3
Saved Guests
Guest1
Guest2
Guest3

Finally, the host runs the last program that outputs the count and the guests in the database. In this case the count is 3 and the guests are $Guest1$, $Guest2$, and $Guest3$.