

# ARM2GC: Succinct Garbled Processor for Secure Computation

Ebrahim M. Songhori  
esonghori@google.com  
Google Inc.

M. Sadegh Riazi  
mriazi@ucsd.edu  
UC San Diego

Siam U. Hussain  
siamumar@ucsd.edu  
UC San Diego

Ahmad-Reza Sadeghi  
ahmad.sadeghi@trust.tu-darmstadt.de  
TU Darmstadt

Farinaz Koushanfar  
farinaz@ucsd.edu  
UC San Diego

## ABSTRACT

We present ARM2GC<sup>1</sup>, a novel secure computation framework based on Yao’s Garbled Circuit (GC) protocol and the ARM processor. It allows users to develop privacy-preserving applications using standard high-level programming languages (e.g., C) and compile them using off-the-shelf ARM compilers (e.g., gcc-arm). The main enabler of this framework is the introduction of SkipGate, an algorithm that dynamically omits the communication and encryption cost of the gates whose outputs are independent of the private data. SkipGate greatly enhances the performance of ARM2GC by omitting costs of the gates associated with the instructions of the compiled binary, which is known by both parties involved in the computation. Our evaluation on benchmark functions demonstrates that ARM2GC not only outperforms the current GC frameworks that support high-level languages, it also achieves efficiency comparable to the best prior solutions based on hardware description languages. Moreover, in contrast to previous high-level frameworks with domain-specific languages and customized compilers, ARM2GC relies on standard ARM compiler which is rigorously verified and supports programs written in the standard syntax.

## KEYWORDS

Privacy-Preserving Computation, Yao’s Garbled Circuit, Secure Processor, ARM

## 1 INTRODUCTION

Secure Function Evaluation (SFE) allows two or more parties to compute an arbitrary function on their respective inputs such that they learn the function’s output without revealing their private data. The first and one of the most powerful methods for two-party SFE is the Yao’s Garbled Circuit (GC) protocol proposed by Andrew Yao in 1986 [47]. Upon arrival, Yao’s protocol immediately attracted significant attention from the cryptographic community and it has been the core enabler of many secure and privacy-preserving applications including but not limited to genomic data analysis [32], text search [36], and stable matching [37]. The protocol requires representing the underlying function as a Boolean circuit. It has been shown that the bottleneck of the GC protocol execution time is the communication between the two parties [7]. Therefore, the non-trivial challenge of utilizing GC is to generate the Boolean circuit such that its secure evaluation requires the minimum inter-party communication.

The challenge of the GC circuit optimization is partially addressed by TinyGarble [41]. This work shows that the GC-optimized circuit generation can be viewed as an atypical instance of the conventional logic synthesis task. This approach outperforms previous methods for generating Boolean circuit using custom compilers or custom libraries [2, 4, 21, 23, 30]. In TinyGarble, however, the highest efficiency and scalability can only be achieved when the function is described in a Hardware Description Language (HDL), e.g., Verilog; while most users prefer to develop their applications in high-level programming languages, e.g., C.

In order to facilitate the deployment of the secure computation frameworks, many researchers have designed Domain Specific Languages (DSL) and/or custom designed compilers for secure computation [2, 6, 16, 23, 25, 30]. These compilers enable users to write the program in a high-level language and compile them into a Boolean/Arithmetic circuit representation such that it can be evaluated by a secure computation protocol. Despite providing a user-friendly solution, the DSL and customized compilers exhibit many limitations compared to the standard high-level languages (e.g., C/C++). For example, they have specialized complex syntax, limited built-in data types, and certain rules on the programming style. Moreover, these compilers do not support many of the typical advanced code optimizations due to their customized design. Last but not the least, the recent analysis by Mood et al. [25] demonstrates that the current state-of-the-art high-level compilers for secure computation crashed on programs that were compiled correctly or generated incorrect compiled programs in some cases.

In this paper, we introduce ARM2GC, a novel *garbled processor* that supports developing privacy preserving applications using any (unmodified) high-level language and an off-the-shelf standard compiler (e.g., arm-gcc) without significant performance ramifications. In a garbled processor [42, 45], the underlying Boolean circuit is that of a general purpose processor. The compiled binary is loaded into the processor’s instruction memory and the private data of the users is loaded into the data memory. Then, the circuit (processor) is garbled/evaluated through a GC back-end. However, such a straightforward garbling approach results in a massive overhead compared to describing the program in HDLs or DSLs. For example, a single addition operation can be securely computed with a minimal number of gates when the task is described in an HDL [41]. Performing the same task using a garbled processor requires garbling/evaluating all of the processor’s components such as control path, register files, and the entire ALU. Garbling these gates are not required to ensure privacy since they operate on the compiled binary of the function which is known to both parties.

<sup>1</sup>a short version of this paper to appear in Desgin Automation Conference (DAC) 2019

An earlier work by Wang et al. [45] suggested a garbled processor based on the standard MIPS instructions. It incurred a high overhead compared with the recent DSL-based solutions. The reason for this inefficiency can be traced back to its *instruction-level* pruning of the processor circuit instead of *gate-level* optimization. (A more detailed description of this approach is discussed in Section 6.) ARM2GC benefits from the first *dynamic fine-grained gate-level optimization* on the garbled processor such that only the gates associated with the private data incur garbling cost. The outputs of the gates associated with the instructions of the compiled binary of the function are computed locally by each party without communication or encryption. Moreover, the gates that do not contribute to the final output are dynamically skipped. This is enabled by the development of a novel algorithm called SkipGate that wraps around the GC protocol. The algorithm dynamically computes the gate outputs that can be calculated without communication and marks the redundant gates for skipping.

The primary objective of SkipGate is to minimize the communication, the bottleneck of GC [7], at the expense of a small increase in local computation. Several secure computation compilers support somewhat similar approaches like *constant propagation* and *dead gate elimination* [2, 25, 28]. However, as we show in Section 3 and Section 5, SkipGate is superior to these solutions since it operates at gate-level and does not require flattening the circuit for the entire computation; that is, it dynamically detects and removes gates that can be skipped from the garbling. Moreover, the static circuit simplification method [29] that removes gates with constant inputs at compile time is not required by the ARM2GC framework, since the Boolean circuit of the ARM processor is generated by industrial circuit synthesis tools which take care of this task. Note that SkipGate avoids unnecessary garbling costs and is different from the cryptographic improvements of GC such as free-XOR [15], Row Reduction [27], and Half Gate [49] that reduce the garbling cost of an individual gate. SkipGate’s operation is orthogonal to these methods; the underlying GC protocol in ARM2GC already benefits from these cryptographic improvements.

In contrast to the earlier custom high-level GC compilers, which employed ad-hoc verification techniques [6, 16, 21, 48], ARM2GC inherits available ARM compilers. These compilers go through rigorous verification as they are used by a large community of programmers in different fields. Therefore, it does not suffer from the reliability issues exposed by Frigate [25] in most of the state-of-the-art GC compilers. Moreover, it readily supports trivial simplifications such as  $a = a \& a$ , that is only supported by two of the most recent frameworks Frigate [25] and CBMC-GC [2]. Moreover, as our evaluation demonstrates, ARM2GC outperforms both these frameworks on the common benchmark functions.

#### **Contributions.**

- We introduce SkipGate, the first algorithm that can dynamically optimize the sequential description of a garbled circuit to allow efficient secure evaluation of functions with publicly known inputs. SkipGate locally computes the output of the gates when it is independent of secret values. The algorithm also skips any gate which does not contribute to the final output.

- We develop the ARM2GC framework based on the SkipGate algorithm and the ARM processor. In this framework, users can efficiently develop SFE applications in a high-level language like C/C++. It enables them to benefit from the available thoroughly verified compilers of ARM. We use the ARM architecture (without affecting the instruction set) to make it most effective for the GC protocol with SkipGate.
- We provide extensive experimental results and show that ARM2GC is 156 times more efficient compared to prior garbled processors [42, 45]. The ARM2GC framework demonstrates comparable performance to HDL synthesis approach of TinyGarble [41]. ARM2GC also outperforms the state-of-the-art high-level GC compilers [2, 25] in terms of communication while utilizing unmodified programming languages and compilers.

## 2 PRELIMINARIES

### 2.1 Security Model

Consistent with the earlier relevant literature [4, 10, 21, 25, 30], we assume an *honest-but-curious* adversary model where the participating parties follow the agreed upon protocol but may attempt to learn about the other parties’ input from the information at hand [1]. This model can be generalized to more advanced adversary models that are typically addressed by multiple runs of the basic honest-but-curious model [18, 20].

### 2.2 Oblivious Transfer

Oblivious Transfer (OT) [26] is a cryptographic protocol based on public key encryption executed between Alice (sender) and Bob (receiver) where Bob selects one of the messages provided by Alice without revealing his selection. Bob also does not learn anything about the unselected messages. In an important special case of 1-out-of-2 OT protocol ( $OT_1^2$ ), Alice holds a pair of messages  $(m_0, m_1)$ ; Bob holds a selection bit  $b \in \{0, 1\}$  and obtains  $m_b$  without revealing  $b$  to Alice and learns nothing about  $m_{1-b}$ .

### 2.3 Garbled Circuit

Yao’s Garbled Circuit protocol [47] allows two parties Alice (*garbler*) and Bob (*evaluator*) to jointly compute a function  $c = f(a, b)$  on their private inputs ( $a$  from Alice and  $b$  from Bob) such that none of them reveal their inputs to each other. In the end, one or both of them learn the output  $c$ . The function  $f$  is represented as a Boolean circuit consisting of 2-input gates. For each wire  $w$  in the circuit, Alice assigns two  $k$ -bit random keys, called *labels*,  $X_w^0$  and  $X_w^1$  corresponding to 0 and 1 Boolean values respectively.  $k$  is the security parameter—typically  $k = 128$  [1]. For each gate, Alice encrypts the output label in each row of the truth table with the corresponding input labels. The resulting table containing the encrypted output labels is then randomly rearranged and called a *garbled table*. She sends the garbled tables of all gates along with the labels corresponding to her input values to Bob. Bob obtains the labels corresponding to his input values obliviously through the OT protocol from Alice. He uses these input labels to decrypt the garbled tables gate by gate. In the end, Bob learns the labels for the final output wire and Alice has its mapping to 0 and 1 so that the actual value of the output can be determined.

The cost of communicating the garbled tables in the GC protocol is its performance bottleneck [7]. Throughout the years, Yao’s GC protocol has gone through a number of optimizations that reduce its communication cost. We describe the most important optimizations here. A significant optimization of the GC protocol is ***free-XOR*** [15] that removes the communication cost for XOR gates. In this optimization, for any wire  $w$ , Alice only generates the label  $X_w^0$  and computes the label corresponding to 1 as  $X_w^0 \oplus (R \parallel 1)$  where  $\parallel$  represents bit concatenation and  $R$  is a global random  $(k - 1)$ -bit value known only to Alice. With this convention, the label for the output of an XOR gate with inputs  $a, b$  and output  $c$  can simply be computed as  $X_c = X_a \oplus X_b$ . Thus, it does not need any encryption or transfer of garbled tables, meaning the XOR gate is *free*. As a result, the optimization goal for circuit generation is to minimize the number of non-XOR gates.

The **Row Reduction** [27] lessens the communication cost of the AND gates by 25% by generating the labels of the output wire as a function of the labels of the input wires and thus making one row of the garbled table all zeros. The **Half Gate** method [49] utilizes both free-XOR and row reduction and reduces the cost of AND gates by an additional 25%.

Earlier GC protocols support only combinational circuit description of the logic functions. Along with the use of logic synthesis for circuit generation, TinyGarble introduced the concept of the **Sequential Garbled Circuits** [41]. Sequential circuits are cyclic graph representation of the Boolean circuits and allow for a compact representation of the functionality. Sequential circuits include memory elements (flip-flops) in addition to logic gates and run for multiple clock cycles. In sequential GC, in each clock cycle, all the gates in the circuit are garbled/evaluated. At the end of each cycle, the labels for the input wire of each flip-flop are simply transferred (copied) to its output wire to be used in the next cycle. At the first clock cycle, the output wires of flip-flops are treated as (either Alice or Bob’s) inputs depending on the function.

Utilizing sequential circuits drastically reduces the memory footprint during garbling and evaluation. For example, a 32-bit summation can be performed using a 1-bit full-adder circuit that outputs 1-bit of the result at each clock cycle. Another example of a sequential circuit is a *processor* that fetches the instruction, performs the corresponding computation, and stores the result.

### 3 SKIPGATE ALGORITHM

SkipGate is a set of novel algorithms that automatically identifies gates that should be garbled given private and public inputs to the circuit. Any gate that can be evaluated based on the public values is *skipped* for garbling and is evaluated in plaintext instead. For example, consider a Multiplexer where both inputs are private and generated by some sub-circuits, whereas, the selection signal is public and known to both parties. In this scenario, one can skip the garbling of a sub-circuit that is not connected. However, standard garbling methodologies require the entire circuit to be garbled and to the best of our knowledge there is no systematic solution that can identify minimal set of gates that is necessary to be garbled.

SkipGate is developed to complement the GC protocol for sequential circuits. As explained in Section 2.3, GC allows secure computation of a function in the form  $c = f(a, b)$ . However, for a

generic function, in addition to the private inputs  $a$  and  $b$ , there can be public inputs (known to both parties). For example, in case of RSA, the encryption key is public. A more practical scenario is garbling a general purpose processor as we explain in Section 4. In general, a processor will have two types of inputs: instruction and data, where the first input is known to both parties unless they want to keep the program private. If the GC framework does not distinguish between public and private inputs, garbling a processor will incur a massive cost for redundant garbling. Previous work [41, 42, 45] proposed generating customized netlists for limited instruction sets. However, they fail to achieve the optimal optimization due to the coarse grain nature of their approach, i.e., instruction level as opposed to gate level.

In SkipGate, we introduce a notion  $p$  to incorporate the public inputs from both parties. It allows secure evaluation of functions in the form of  $c = f(a, b, p)$  where  $p$  is the public input known to both parties and  $a$  and  $b$  are the private inputs. The goal of SkipGate is to reduce the circuit of  $c = f(a, b, p)$  into a simpler circuit  $c = f_p(a, b)$  utilizing the knowledge of public input  $p$ . Secure evaluation of  $f_p(a, b)$  requires less number of garbled tables than that of  $f(a, b, p)$  when using the standard GC protocol and treating  $p$  as a private input. SkipGate removes communication cost of garbling for a gate when its output can either be computed independently by Alice and Bob or has no effect on the final output. In other words, it reduces the communication between the parties when it can be replaced by less costly local computation. The cost reduction is especially significant in a garbled processor where the control path is public and independent of the private inputs. Before presenting SkipGate, let us introduce the following notations and definitions.

In a classic Boolean circuit, each wire  $w$  carries a value ( $x_w \in \{0, 1\}$ ), whereas, in a garbled circuit, each wire carries a pair of labels ( $X_w^0$  and  $X_w^1$ ) on Alice’s side and one label ( $X_w \in \{X_w^0, X_w^1\}$ ) on Bob’s. If  $X_w = X_w^0$ , the actual Boolean value is 0 and if  $X_w = X_w^1$ , the value is 1. This, in turn, means that the information is shared between the two parties. In our scheme, we combine the notion of Boolean and garbled circuits. Each wire either carries a Boolean value known to both parties independently (*public* wire) or it carries a (pair of) label(s) (*secret* wire).

**Illustrative Example:** Assume a sequential circuit that has a 2-to-1 MUX whose inputs come from two sub-circuits  $f_0$  and  $f_1$  connecting to MUX inputs 0 and 1, respectively. At a certain clock cycle, if the “select wire” of the MUX ( $x$ ) is public, say equal to 1, both parties know that the gates in the sub-circuit  $f_0$  do not need to be garbled/evaluated since they have no effect on the final output. The gates in the MUX itself act as wires and pass the input 1 (output of  $f_1$ ) to the MUX output, thus they do not need to be garbled/evaluated in that clock cycle either. However, in the conventional GC protocol where public wire  $x$  is treated as a secret value, the entire circuit has to be garbled/evaluated. In what follows, we explain how the SkipGate algorithm identifies such gates.

It is worth mentioning that in a sequential garbled circuit [41], the Boolean value of a wire can change at every clock cycle. A wire may also switch between being secret and public from one clock cycle to another. The SkipGate algorithm is executed once for every sequential cycle. SkipGate’s decision on each gate depends on the status of the gate’s inputs (public or secret) on that cycle.

### 3.1 Gate Categories

The SkipGate algorithm classifies the gates into four categories in terms of the parties' knowledge about the inputs of a given gate:

- i *Gate with two public inputs*: In this case, the output is public and can be computed locally by each party.
- ii *Gate with one public input*: Depending on the gate type, the output becomes either public or secret. For example, for an AND gate with public value 0 at one input, the output becomes 0. This means that if the gate's secret input is not connected to any other gate, the gate generating the secret wire can be skipped for garbling/evaluation. If the public input is 1, then the AND gate acts as a wire and the output wire carries the label of the secret input.
- iii *Gate with secret inputs that have identical (or inverted) labels*: This indicates that the two secret inputs have identical (or inverted) Boolean values. Depending on the gate type, the output becomes either public or secret. For example, the output of an XOR gate with two inverted inputs (either secret or public) is always 1 (public). Similar to Category ii, the gate generating the inputs, if not connected to any other gate, can be skipped for garbling/evaluation.
- iv *Gate with unrelated secret inputs*: The output is always secret. The gate has to be garbled/evaluated conventionally according to the GC protocol. However, if its output does not have any effect on the circuit output, the gate is skipped, i.e., the corresponding garbled table is not transferred.

### 3.2 Algorithm

Algorithm 1 and Algorithm 2 show the SkipGate algorithm for Alice and Bob sides, respectively. Lines 2-5 of Algorithm 1 and Lines 2-4 of Algorithm 2 are similar to the GC protocol label generation and transfer for both sides. The SkipGate algorithm has two main phases: In Phase 1, the outputs of the gates with public input(s) (Categories i-ii) are computed. In Phase 2, the gates with private inputs (Categories iii-iv) are garbled/evaluated. For each round of sequential cycle, Alice executes Phase 1 and 2 of SkipGate and sends the generated garbled tables to Bob. Bob receives the tables and executes two phases in order to evaluate the gates. However, this does not affect the parallelism of the operation. When Bob is evaluating the gates in cycle  $c$ , Alice is garbling the gates for cycle  $c+1$ . In Line 14 of Algorithm 1 and Line 13 of Algorithm 2, the labels associated with the input of flip-flops are copied to their output for the next cycle [41]. Similar to conventional GC, at the end of the protocol, Alice learns pairs of labels for each output wire and Bob has one of the labels; they share this information to learn the output  $c$ . For example, in the case where Alice intends to learn the final output, she receives Bob's output label and together with her output labels finds the real output value (Line 16-17 of Algorithm 1 and Line 16 of Algorithm 2).

In SkipGate, an integer called `label_fanout` is associated with each gate and indicates the number of times the gate's output label is used (either as a circuit's output or an input to other gates). At the beginning of each cycle (Line 8 of Algorithm 1 and Line 7 of Algorithm 2), the `label_fanout` is set to the gate fanout in the circuit<sup>2</sup>.

<sup>2</sup>*Fanout* of a gate, borrowed from hardware design, is the number of subsequent gates (and circuit outputs) dependent on the gate's output.

---

**Algorithm 1:** SkipGate, Alice's side.

---

**Inputs:** Sequential circuit of  $c = f(a, b, p)$ , Alice's input  $a$ , public inputs  $p$ , number of clock cycles  $cc$ .  
**Outputs:** Output  $c$ .

```

1: SkipGate_Alice (circuit, a, p, cc):
2: generate random labels →  $(X_A^0, X_A^1, X_B^0, X_B^1)$ 
3: send Alice labels  $\in \{X_A^0, X_A^1\}$  based on her input  $a$ 
4: send Bob labels  $(X_B^0, X_B^1)$  through OT
5: set wires corresponding to  $a$  and  $b$  as private
6: set wires corresponding to  $p$  as public
7: for cid from 1 to cc do
8:   initialize labels' fanout
9:   // Algorithm 3, process gate categories i-ii
10:  perform Phase 1
11:  // Algorithm 4, process gate categories iii-iv
12:  perform Alice Phase 2 → garbled tables
13:  send garbled tables
14:  copy flip flops labels
15: end for
16: receive Bob output labels →  $X_C$ 
17: compute output value based on output labels  $(X_C^0, X_C^1)$  and received labels  $(X_C)$  →  $c$ 
```

---



---

**Algorithm 2:** SkipGate, Bob's side.

---

**Inputs:** Sequential circuit of  $c = f(a, b, p)$ , Bob's input  $b$ , public input  $p$ , number of clock cycles  $cc$ .  
**Outputs:** Output labels  $X_C$ .

```

1: SkipGate_Bob(circuit, b, p, cc):
2: receive Alice's labels →  $X_A$ 
3: receive Bob labels →  $X_B$  through OT
4: set wires corresponding to  $a$  and  $b$  as private
5: set wires corresponding to  $p$  as public
6: for cid from 1 to cc do
7:   initialize labels' fanout
8:   // Algorithm 3, process gate categories i-ii
9:   perform Phase 1
10:  receive garbled tables
11:  // Algorithm 5, process gate categories iii-iv
12:  perform Bob Phase 2
13:  copy flip flops labels
14: end for
15: compute circuit output labels →  $X_C$ 
16: send output labels  $(X_C)$ 
```

---

`label_fanout` of a gate may decrease, e.g., a gate whose output is connected to an AND gate with 0 at the other input (Category ii). If `label_fanout` reaches 0, it means that gate's output label does not have any effect on the rest of the circuit and final output. The gates with `label_fanout` = 0 are subsequently marked for skipping, which in turn decreases the `label_fanout` of the gates connected to the input of the marked gates. Note that this step is recursive in nature, i.e., when decreasing a gate's `label_fanout`,

---

**Algorithm 3:** Phase 1 in SkipGate for both Alice and Bob sides.

---

```

1: SkipGate_phase1()
2: for g in circuit do
3:   if both inputs of g are public then
4:     //Category i
5:     compute output of g based on its type
       and inputs
6:     set g label fanout to 0
7:   else if one of the g inputs is public then
8:     //Category ii
9:     compute output of g based on its type, private, and public
       inputs
10:    if output of g is public then
11:      set g label fanout to 1 // will become zero in
          recursive_reduction()
12:      recursive_reduction(g)
13:    end if
14:  end if
15: end for

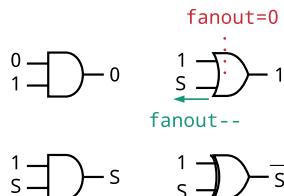
```

---

it might reach zero and subsequently call the gates who are providing the input to this gate and so on (see Figure 3). Finally, the gates in Category iv that have not been marked for skipping are garbled/evaluated.

Algorithm 3 illustrates the Phase 1 of SkipGate in which Alice and Bob find and compute the gates that belong to Categories i-ii. `label_fanout` of the gates in Category i are set to zero. For gates in Category ii, if the output becomes public, SkipGate decreases the `label_fanout` of the secret input's originating gate recursively by invoking `recursive_reduction` (Algorithm 6). Figure 1 shows four different examples in Phase 1. Bob does not receive any information from Alice about the gates in Category i-ii because he can locally evaluate Phase 1 just like Alice. An alternative approach is that Alice sends the result of Phase 1 to Bob. This approach has two main disadvantages: First, it makes the protocol altered if one wants to enhance the security of the protocol to be secure against malicious adversaries [18]. Second, it increases the communication overhead which is the bottleneck of the GC protocol [7].

Algorithm 4 shows the Phase 2 of SkipGate for Alice's side in which she performs the same task for Category iii. She then generates garbled tables for gates with non-zero `label_fanout` in



**Figure 1:** Four examples in Phase 1 where gates are replaced by zero, one, wire, or inverter. The top-left gate is in Category i and the rest are in Category ii. `label_fanout` is set to zero for the skipped gate.

---

**Algorithm 4:** Phase 2 in SkipGate, Alice's side.

---

**Output:** list of garbled tables.

```

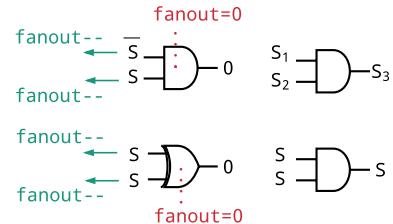
1: SkipGate.phase2_Alice()
2: for g in circuit where label_fanout > 0 do
3:   if g's input labels are equal or inverted then
4:     //Category iii
5:     compute g's output based on its type
6:     if g's output label is public then
7:       set g's label_fanout to 1 // will become zero in
          recursive_reduction()
8:       recursive_reduction(g)
9:     end if
10:   else
11:     //Category iv
12:     garble g // table = null for XOR gates
13:     if g is non-XOR then
14:       add garbled table to the list
15:     end if
16:   end if
17: end for
18: remove garbled tables where gates's fanout is 0

```

---

Category iv. Figure 2 shows four different examples in this phase. By the end of Phase 2, due to the recursive nature of the fanout reduction, `label_fanout` of some gates that have already been garbled may become 0. In Line 18 of Algorithm 4, Alice filters the garbled tables that have non-zero `label_fanout` to be sent to Bob.

Algorithm 5 shows the Phase 2 for Bob's side. Bob evaluates the gates that belong to Category iii and iv. In Line 18 of Algorithm 5, Bob generates and assigns new unique labels for gates that were filtered by Alice. Bob knows that the `label_fanout` of these gates will eventually become 0. Therefore, he produces new labels for them only to keep track of these secret variables that are used to compute the output of the gates in Category iii. He can generate these labels randomly or use a monotonic counter that increases by one for each newly generated label. To distinguish valid GC labels from his generated labels, he keeps a single bit flag along with each label that indicates the label is generated by him and is not valid for the GC evaluation.



**Figure 2:** Four examples of replacing and computing gates in Phase 2. The top-right example is in Category iv, and the rest are in Category iii. `label_fanout` is set to zero for the skipped gates.

---

**Algorithm 5:** Phase 2 in SkipGate, Bob's side.

---

**Input:** list of garbled tables.

```

1: SkipGate.phase2_Bob(garbled_tables):
2: for g in circuit where label_fanout > 0 do
3:   if g's input labels are equal or inverted then
4:     //Category iii
5:     compute g's output based on its type
6:     if g's output label is public then
7:       set g's label_fanout to 1 // will become zero in
      recursive_reduction()
8:       recursive_reduction(g)
9:     end if
10:   else
11:     //Category iv
12:     if g is an XOR gate then
13:       compute output label based on input labels
14:     else if g is top of the garbled tables list then
15:       remove the garbled table from the list → gt
16:       compute output label of g based on its type, input
          labels, and gt
17:     else
18:       assign g's output label to a unique random binary
          string
19:     end if
20:   end if
21: end for
```

---

**Algorithm 6:** Recursive Fanout Reduction of SkipGate.

---

**Inputs:** Gate g (where the reduction starts).

```

1: SkipGate.recursive_reduction(g):
2: if g's label_fanout is 0 then
3:   return
4: end if
5: g's label_fanout = label_fanout - 1
6: if label_fanout is 0 then
7:   if g's first input is secret then
8:     recursive_reduction(first input of g)
9:   end if
10:  if g's second input is secret then
11:    recursive_reduction(second input of g)
12:  end if
13: end if
```

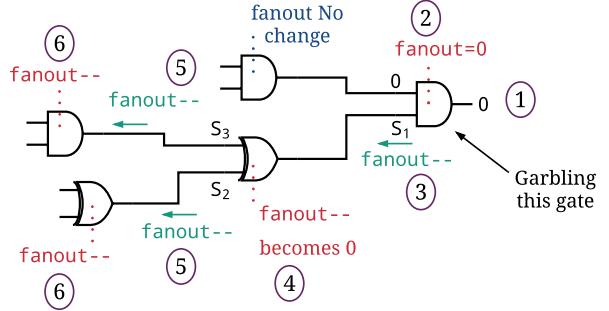
---

Algorithm 6 illustrates the pseudo-code for the recursive fanout reduction. It receives the circuit and a gate inside the circuit. It first decreases the label\_fanout of the given gate. If the label\_fanout becomes 0, it recursively calls the function with the gates that generate the corresponding secret input(s). This process is illustrated on an example circuit in Figure 3.

### 3.3 Identification of Identical and Inverted Labels

According to the GC protocol, Bob only has one label  $X_w$  for each secret wire  $w$ . Due to free-XOR [15], he does not need to modify

the label when he evaluates a NOT gate because the labels corresponding to 0 and 1 are inverted by Alice during the garbling process, flipping the secret value of  $w$  accordingly. This, in turn, means that Bob cannot tell apart an identical and inverted secret value based on the label alone. However, it is still possible for Bob to keep track of the flips by storing one bit along with the label. After evaluating a NOT gate, he simply flips the bit. This extra bit helps him to differentiate between identical and inverted secret values which are crucial during Phase 2.



**Figure 3: Recursive reduction of label\_fanout to skip unnecessary gates (Algorithm 6).**

### 3.4 Computational Complexity

The SkipGate algorithm decreases the communication cost, which is the bottleneck of GC, at the expense of increasing the local computations. The conventional GC protocol has a linear computational complexity in terms of the number of gates in the circuit for each sequential cycle. We show that, despite its recursive appearance, the SkipGate algorithm does not increase the computation complexity of the GC protocol. All parts of the SkipGate algorithm, except recursive\_reduction (Algorithm 6), is executed once per gate, thus they incur a complexity similar to the classic GC protocol. The only procedure that can potentially increase the computation complexity is recursive\_reduction function whose number of invocations depends on the underlying circuit and whether input wires are secret or public. To find the complexity of SkipGate, we present an upper bound on the number of invocations of recursive\_reduction function.

The termination condition in recursive\_reduction is the fanout reaching zero (Lines 2 of Algorithm 6). Thus, the worst case scenario is when the function reduces the fanout of all the gates to zero. In this case, the number of execution of the fanout decrement (Line 5) should be at most the sum of all the initialized fanouts. label\_fanout is initialized with the gate fanout in the circuit. The upper bound on the sum of fanouts of all the gates in the circuit is

$$F = \sum_{i=1}^n g[i].fanout \leq 2n - m + q,$$

where  $n$  is the number of gates,  $q$ , and  $m$  are the number of circuit output and inputs, respectively. Each gate has two inputs, as required by the GC protocol, and each input creates a fanout in previous gates unless it is a circuit input. Also, each output wire

incurs the fanout of one. Both  $q$  and  $m$  are typically less than or at most in the order of  $n$ . Thus,  $F$  and subsequently the number of invocation of `recursive_reduction` function are  $O(n)$ . This shows that SkipGate does not increase the overall linear computational complexity of the GC protocol.

### 3.5 Correctness and Security Proof

**Correctness:** Given the correctness of Yao’s GC protocol, we show that GC protocol with SkipGate is also correct. In SkipGate, the topology of the circuit is not changed, thus the dependencies of the values remain the same. Therefore, we only prove that processing a single gate remains correct in SkipGate.

The operations for gates in Category i are merely based on the Boolean operation of the gates and are clearly correct. For gates in Categories ii-iii, the secret input is considered as an *unknown* variable. Either the label at the secret input of the gate is passed to its output or the output is set to a public value. Therefore, the functionality of the gate is not changed. Gates in Category iv with non-zero `label_fanout` are garbled/evaluated according to the GC protocol. For the rest of the gates in Category iv, `label_fanout` = 0 indicates that their secret output does not have any effect on the final output of the circuit. Therefore, they can be safely skipped. As such, we conclude that the SkipGate algorithm with the GC protocol results in a logically correct output.

**Security:** The GC protocol is proved to be secure under honest-but-curious adversary model for any two-input Boolean function  $f(a, b)$  where  $a$  and  $b$  are private inputs from Alice and Bob, respectively [1, 19]. In this work, we extend the function to the form of  $f(a, b, p)$  to include a public input  $p$  that is known to both parties. The SkipGate algorithm reduces the Boolean circuit of  $f(a, b, p)$  to a two-input circuit of  $f_p(a, b)$  where, for a given  $p$ ,  $f_p(a, b) = f(a, b, p)$  for any  $a$  and  $b$ .  $f_p(a, b)$  consists of the gates in Category iv with non-zero `label_fanout` evaluated by the GC protocol. The process of skipping gates from  $f(a, b, p)$  only utilizes the public input  $p$  which is already known to both parties. In the process, the private values are treated as unknown Boolean variables. In other words, Alice and Bob do not access their inputs in the SkipGate algorithm for reducing  $f(a, b, p)$  to  $f_p(a, b)$ . Thus, no information about the private inputs  $a$  and  $b$  is accessed/revealed by the SkipGate algorithm. The garbling/evaluation of the two-input Boolean function of  $f_p(a, b)$  is passed to the original GC protocol. Therefore, the security proof of SkipGate is identical to that of the GC protocol.

## 4 ARM2GC

In this section, we present ARM2GC, a GC framework based on a garbled ARM processor and the SkipGate algorithm. The framework aims to simplify the development of privacy-preserving applications while keeping the garbling cost as low as the best optimized garbled circuits. We first describe the overview of ARM2GC and its API for GC development. Then, we explain how ARM’s unique architecture helps to decrease garbling overhead. Next, the effect of SkipGate in reducing the garbling cost is discussed. Finally, we discuss why we do not employ Oblivious RAM for ARM register files.

### 4.1 Global Flow

The ARM2GC framework allows users to write a two-party SFE program in C/C++ (or any language that can be compiled to the ARM binary code). Figure 4 shows the overview of the framework. The SFE program is compiled using an ARM cross-compiler, e.g., `gcc-arm-linux-gnueabi`. The compiled binary code is fed to the SkipGate algorithm as the public input  $p$ . The Boolean circuit that is going to be garbled/evaluated is the synthesized ARM processor circuit. The ARM2GC framework supports the following API:

```
void gc_main()
    const int *a, // Alice's input
    const int *b, // Bob's input
    int *c) // output array
    // The user's code goes here.
}
```

The entry function, `gc_main`, receives three arguments: pointers to Alice’s input, Bob’s input, and the output. The framework has five separate memory elements (consisting of flip-flops and MUXes) to store: Alice’s inputs, Bob’s inputs, output, stack, and instructions. The flip-flops in the instruction memory are initialized with the compiled binary code that is known to both parties (the public input  $p$ ). The flip-flops in Alice’s and Bob’s memories are initialized with the labels corresponding to their private inputs  $a$  and  $b$ , respectively. The other flip-flops in the stack, output, pipeline registers, and the register file are initialized to zero. The ARM circuit is garbled following the sequential garbling process [41] for a pre-specified number of clock cycles.

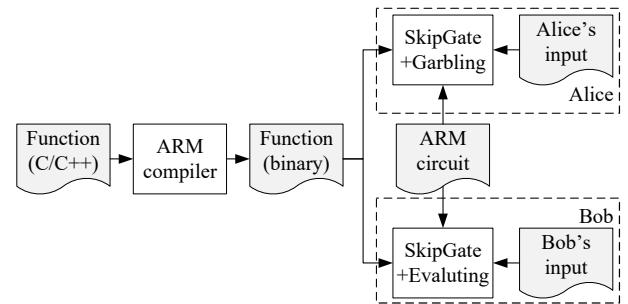


Figure 4: Overview of the ARM2GC framework.

### 4.2 ARM as a Garbled Processor

In this work, we choose ARM as our garbled processor which is a more ubiquitous and sophisticated processor compared to MIPS [41, 42, 45]. ARM has two main advantages: (1) Pervasiveness: the compilers and toolsets of ARM are under constant scrutiny, updating, and probably, more optimized as a result. (2) Conditional Execution: Designed to improve performance and code density, conditional execution in ARM allows each instruction to be executed only if a specific condition is satisfied [40].

ARM compilers tend to replace conditional branches with conditional instructions to make the flow of the program predictable, and thus, lower the cost of branch misprediction. Similarly, in a garbled processor, the main design effort is to make sure that the flow of the

<pre> 1: cmp \$8, \$9 2: bne L0 3: mov \$1, #10 4: b L1 5: L0: 6: mov \$2, #20 7: L1: ... </pre>	<pre> 1: cmp \$8, \$9 2: moveq \$1, #10 3: movne \$2, #20 4: L1: ... </pre>
--	---

(a) Without Conditional Execution      (b) With Conditional Execution

**Figure 5: An example illustrating how conditional execution in ARM can reduce the code size and make the program flow predictable.**

program is predictable so that the next instruction remains public. Replacing conditional branches with conditional instructions in garbled ARM generates a code with a predictable flow. Figure 5 shows an example function compiled into assembly with and without the conditional execution. For the code without the conditional execution, the program counter becomes dependent on the results of the comparison. If one of the compared values are secret, the program counter becomes secret as well. For code with the conditional execution, the program counter goes serially through all the commands serially, irrespective of the result of the comparison operation. Thus, it always remains public. We also modify the ARM controller such that conditional instructions always take the same number of cycles regardless of their condition (taken or not taken). Otherwise, the program flow will be dependent on the secret condition. Having a secret program counter makes the SkipGate algorithm less effective on ARM2GC and therefore reduces the efficiency of the execution.

We modify and remove a few features from the ARM processor such as interrupts, co-processors, and performance-related components including cache and pipeline. These components do not bring any performance advantages in the GC protocol, as the circuit is garbled/evaluated gate by gate (serially). Note that unlike in hardware, the performance of GC does not increase by parallelizing the gates in the circuit. In the GC protocol, the total number of garbled non-XOR gates is the *only* factor affecting the performance. The user does not need to pass any flag to the ARM compiler because of the removed components since such blocks only enhance the performance of the processor internally. Therefore, the compiled instructions do not have to be modified because of this modification.

Implementation of the ARM processor results in a complex and large netlist ( $\approx 5$  times larger than that of a MIPS processor). Thus, using ARM instead of MIPS in the earlier garbled processor approaches [42, 45] would incur an even higher cost. However, the majority of the components of the ARM processor remain idle during execution of an instruction. In the next section, we describe how SkipGate utilizes this characteristic to minimize the cost of garbling the processor.

### 4.3 Effect of SkipGate on ARM2GC

As explained above, the instruction memory of the ARM processor is initialized with public values (compiled program). Therefore, if

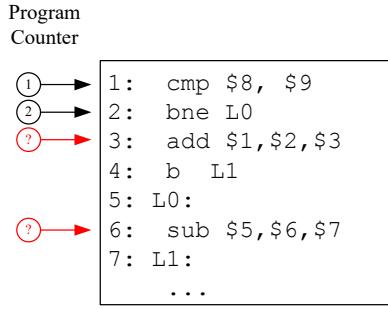
the program counter (the address of the next instruction) is public, the content of the next instruction becomes public as well. As a result, the control path also becomes public and SkipGate can easily detect the idle components to mark them for skipping. Moreover, due to SkipGate, the gates of the active components that are only transporting data between memory, register file, and ALU act as wires and do not incur any cost. According to SkipGate’s notation, the ARM Boolean circuit is a 3-input function  $c = f(a, b, p)$  where  $p$  is the public binary code and  $a$  and  $b$  are the parties’ private inputs. SkipGate reduces the ARM circuit into a smaller circuit of  $c = f_p(a, b)$  where  $f_p$  is able to perform the exact operation required by the public binary code  $p$ , e.g.,  $c = a + b$ . Therefore, the main garbling cost is paid only for the actual computation on the secret values. As explained in the previous section, SkipGate performs these optimizations at the gate level, in contrast to instruction level as in [42, 45].

### 4.4 Why not Sub-linear Oblivious RAM?

As mentioned in Section 4.1, we use an array of MUXs and flip-flops to implement the register file in the ARM circuit. This means that the cost of accessing the register file, when performed obliviously, is linear with respect to its size. One natural question would be why we did not employ Oblivious RAM (ORAM) that enables oblivious access to memories in the GC protocol with sub-linear cost [46, 50]. The reason is that, in most cases, the access to the register file is not required to be oblivious. Since the instructions come from a publicly known instruction memory, both parties know which register is accessed. The SkipGate algorithm utilizes this information to skip garbling of the gates in the MUXs of the register file, thus, no garbling cost is required for such accesses. With ORAM, all the accesses to the register file would be the costly oblivious access.

In rare occasions where two or more instructions should be garbled at a time, accessing a register would not be free using MUXs. These cases only happen when ARM compiler fails to replace a conditional branch on a secret value with conditional instructions. The user can typically alter the program in a way that the compiler avoids such branches and replaces it with conditional instructions instead. However, in these cases, the SkipGate algorithm removes most of the gates in the register file. Currently, state-of-the-art ORAM constructions such as Circuit ORAM [44], SR-ORAM [50], or Floram [5] start outperforming the linear scan (MUXs and flip-flops) from memory size of 8KB (512-bit block size), 8KB (32-bit block size), 2KB (32-bit block size), respectively. ARM’s total register file has 16 registers, each containing a 32-bit value, thus, the total size of the register file is 64B which is smaller than the break-even points of ORAMs.

Figure 6 shows an example where after execution of a branch on a secret value, the next instruction becomes secret and unknown to parties. In this example, the program counter can either be 3 or 6 depending on the outcome of the comparison in Line 1. Thus, two instructions add \$1, \$2, \$3 (\$3 = \$1 + \$2) and sub \$5, \$6, \$7 (\$5 = \$6 - \$7) have to be garbled/evaluated at the same time. For fetching the second instruction from the register file, we only have two choices: \$2 and \$6. This means that, instead of having a complete oblivious access to the register file with 16 choices, we only have to obliviously select between 2 of the 16 registers. This



**Figure 6: Failure to replace a secret branch with conditional instructions, makes the program counter secret. Thus, the instruction becomes secret.**

costs far less than using ORAMs. The cost of oblivious access using MUXs and SkipGate to a *subset* of a memory is equal to an oblivious access to a memory with the size of the subset.

The rationale for using an array of MUXs in the register file also applies to the code, data, and stack memories where the access is almost always public and known to both parties. In the worst case, only a subset of memory is accessed obliviously, thus making the cost of memory access below the threshold of switching to ORAMs. The integration of the SkipGate algorithm and garbled processor introduces an unusual use case for oblivious memory where oblivious access is performed only on a varying subset of the memory. The subset can be different from one access to the other. The current sub-linear ORAM protocols cannot address this scenario efficiently. Thus, an interesting research question is raised:

**Is it possible to *obliviously* access (read/write) a varying subset of the memory with a *sub-linear* cost in terms of the subset size?**

Note that programs which contain secret terminate conditions in the for-loops prohibit the protocol to be terminated. The underlying reason is that when the number of loop executions is secret, it is not known when the program is going to finish. This is, in fact, a fundamental constraint for any high-level secure computation frameworks and it is not a shortcoming of ARM2GC. By definition, securely computing for-loops for *secret* number of times, requires that the protocol's behavior is not dependent on the value of the secret condition. Since the value of the secret condition can be arbitrarily large, the protocol should not terminate until after performing dummy operations for the maximum possible number of loop executions which is prohibitively expensive.

## 5 EVALUATION

### 5.1 Evaluation Setup

We use Synopsis Design Compiler (DC) H-2013.03-SP4 [3] along with TinyGarble [41] synthesis and technology libraries to generate the netlists for the benchmark circuits and the ARM processor.

For the ARM2GC framework, we use the Amber ARM project, an open-source implementation of ARM v2a ISA on opencores [39]. The ARM circuit is modified as explained in Section 4.2. Synthesizing the ARM processor with Synopsis DC takes a few hours. However, the process is done only once for a given memory size and it can be used for any set of functions and inputs afterward. The

**Table 1: Improvement by the SkipGate algorithm on sequential circuits generated by TinyGarble [41]. These functions do not have public inputs. SkipGate benefits from public initial values of the small number of flip-flops to reduce their garbling cost.**

Function	# of Garbled Non-XOR		# of Skipped	Improv.
	w/o SkipGate	w/ SkipGate		
<b>Sum 32</b>	32	31	1	3.13%
<b>Sum 1024</b>	1,024	1,023	1	0.10%
<b>Compare 32</b>	32	32	0	0.00%
<b>Compare 16,384</b>	16,384	16,384	0	0.00%
<b>Hamming 32</b>	160	145	15	9.38%
<b>Hamming 160</b>	1,120	1,092	28	2.50%
<b>Hamming 512</b>	4,608	4,563	45	0.98%
<b>Mult 32</b>	2,048	2,016	32	1.56%
<b>MatrixMult3x3 32</b>	25,947	25,668	279	1.08%
<b>MatrixMult5x5 32</b>	120,125	119,350	775	0.65%
<b>MatrixMult8x8 32</b>	492,032	490,048	1,984	0.40%
<b>SHA3 256</b>	40,032	38,400	1,632	4.08%
<b>AES 128<sup>†</sup></b>	15,807	6,400	9,407	59.51%

<sup>†</sup>The missing key expansion module to AES 128 of [41] is added here.

benchmark functions for ARM2GC are implemented in C and compiled using GNU gcc-arm-linux-gnueabi (Ubuntu/Linaro 5.3.1-14ubuntu2). We used -Os compiler optimization flag in order to reduce the number of instructions. We modified the header assembly code to change the addresses of stack, code, and data memories in the compiled binary. We do not apply any optimization on the binary code. Thus, similar to a normal software compilation, it takes less than a second to compile the majority of the functions into the ARM binary codes.

### 5.2 Benchmark Functions and Metrics

We use the benchmark functions that have frequently been used for evaluation in the GC literature [2, 25, 41]. The most important metric to compare the cost of garbling is the total number of garbled non-XOR gates. This metric encompasses both the cost of computation (encrypting/decrypting garbled tables) and the cost of communication (transferring garbled tables) in the GC protocol due to the free-XOR optimization [15].

### 5.3 Effect of SkipGate on Sequential GC

As described in Section 3, the SkipGate algorithm avoids redundant garbling/evaluation of gates in sequential circuits with public wires. In the sequential benchmark circuits reported in TinyGarble [41], the flip-flops were initialized with known values but their output wires were treated as secret. We applied SkipGate to the same benchmark functions to demonstrate the cost reduction even for a small number of public values. In Table 1, we compare the cost of garbling for circuits generated by TinyGarble [41] with and without applying the SkipGate algorithm. As can be seen, cost reduction of SkipGate can be as high as 59.5% for AES and as little as 0% in Compare function.

The degree of improvement depends on the structure of the circuit and whether or not the registers are connected to non-XOR gates. For example, in AES, garbling of the controller part of the sequential circuit (including a counter keeping track of the AES

**Table 2: Comparison of the number of garbled non-XOR gates of ARM2GC with the HDL synthesis approach of TinyGarble [41]. Both frameworks benefit from SkipGate.**

Function	# of Garbled Non-XOR		Overhead
	TinyGarble [41] (Verilog)	ARM2GC (C)	
Sum 32	31	31	0.00%
Sum 1024	1,023	1,023	0.00%
Compare 32	32	32	0.00%
Compare 16,384	16,384	16,384	0.00%
Hamming 32	145	57	-60.69%
Hamming 160	1,092	247	-77.38%
Hamming 512	4,563	1,012	-77.82%
Mult 32	2,016	993	-50.74%
MatrixMult3x3 32	25,668	27,369	6.63%
MatrixMult5x5 32	119,350	127,225	6.60%
MatrixMult8x8 32	490,048	522,304	6.58%
SHA3 256	38,400	37,760	-1.67%
AES 128 <sup>†</sup>	6,400	6,400	0.00%

<sup>†</sup>The missing key expansion module to AES 128 of [41] is added here.

round and MUXs connecting to it) is avoided by SkipGate because both parties know the AES control path in advance. Note that the functions in Table 1 do not have any public known inputs that are the main target of SkipGate. Nevertheless, SkipGate reduces the cost of GC by leveraging the public initial value of the small number of flip-flops in the circuits.

**Comparison with Garbled MIPS [45].** Even though the approach of ARM2GC is similar to the garbled MIPS presented in [45], it outperforms that work by a long margin. For example, to compute the Hamming distance between 32 32-bit integers<sup>3</sup>, [45] needs 481K garbled gates, whereas ARM2GC needs only 3073- and improvement by 156×.

#### 5.4 ARM2GC vs HDL Synthesis

Table 2 compares the cost of garbling of (i) functions devised in Verilog HDL and constructed by the hardware synthesis technique of TinyGarble [41] with (ii) functions developed in C and constructed by the ARM2GC framework. SkipGate is applied in both cases. As expected, ARM2GC incurs only a small overhead (at most 6.6% for MatrixMult8x8) compared to hardware synthesis method. In the case of Hamming distance function, ARM2GC results in even less number of non-XOR gates (up to 77.8% improvement). Note that we use an efficient binary tree-based method [11] for Hamming distance realization in C.

#### 5.5 ARM2GC vs GC Frameworks Supporting High-level Languages

Table 3 reports the cost of garbling for the benchmark functions constructed by ARM2GC and the prior-art GC frameworks Frigate [25] and CBMC-GC [2]. The last column compares ARM2GC with the best of these two. In all cases, ARM2GC is either equal or better than the earlier frameworks in terms of garbling cost. It shows significant improvements in hamming distance and AES, 44% and

<sup>3</sup>as reported in [45], this is different from the common approach of computing Hamming distance where the inputs are binary

**Table 3: Comparison of the number of garbled non-XOR gates of ARM2GC with the best prior art solution supporting high-level languages. We choose CBMC-GC and Frigate for comparison as they outperform previous frameworks for these benchmarks. The improvement is shown w.r.t. the best of these two.**

Function	Number of non-XORs			Improv.
	CBMC-GC [6]	Frigate [25]	ARM2GC	
Sum 32	-	31	31	0.00%
Sum 1024	-	1,025	1,023	0.20%
Compare 32	-	32	32	0.00%
Compare 16,384	-	16,386	16,384	0.01%
Hamming 160	449	719	247	44.99%
Mult 32	-	995	993	0.20%
MatrixMult5x5 32	127,225	128,252	127,225	0.00%
MatrixMult8x8 32	522,304	-	522,304	0.00%
AES 128	-	10,383	6,400	38.36%
$a = a \text{ op}^{\ddagger} a$	0	0	0	0.00%
SHA3 256	-	-	37,760	-

<sup>†</sup>op represents any Boolean operation (+, &, ⊕, etc.)

38% respectively. Moreover, as shown in the table, software-level optimizations such as  $a = a \& a$  are automatically performed by the ARM compiler. Such operations can result in compile time or runtime errors in several state-of-the-art frameworks as reported in [25]. Note that we choose to compare with these two frameworks as they outperform the earlier frameworks like Obliv-C [48] and OblivM [21]. Even though the approach of ARM2GC is similar to the garbled MIPS presented in [45], it outperforms that work by a long margin. For example ARM2GC requires 8.4E3 and 49E3 times less garbled non-XOR gates for computing Hamming distances with 32 and 512-bit inputs, respectively.

#### 5.6 Effect of SkipGate on ARM

Table 4 shows the cost of garbling an ARM processor for the benchmark functions using conventional GC compared to GC with the SkipGate algorithm. Since the instruction memory is known to both parties in ARM, SkipGate omits a significant number of non-XOR gates in the circuits. The circuit of ARM has 126,755 non-XOR gates and for computing a function, for example, Hamming 160, it takes 1,909 clock cycles. It means with the conventional GC protocol, garbling/evaluation of  $1,909 \times 126,755 = 241,975,295$  non-XORs is required. SkipGate reduces the circuit into a smaller circuit with only 247 non-XORs (almost seven orders of magnitude less). In the case of AES, we achieve more than six orders of magnitude improvement over the garbled processor based on the conventional GC without the SkipGate algorithm. The algorithm transforms the impractical cost of garbling an ARM processor into the near-optimal cost of the reduced circuit. These dramatic improvements are due to a large number of public inputs in the ARM processor originating from the instruction memory that allows SkipGate to skip garbling/evaluation most of the gates in the ARM circuit.

#### 5.7 Complex Functions

We develop a number of complex functions, as described below, with the ARM2GC framework. In each of these functions, the input

**Table 4: Improvement by SkipGate on ARM2GC.**

Function	# of Garbled Non-XOR		Improv. (1000X)
	w/o SkipGate	w/ SkipGate	
Sum 32	3,817,680	31	123
Sum 1024	76,483,260	1,023	75
Compare 32	4,072,192	130	31
Compare 16,384	1,047,095,280	16,384	64
Hamming 32	67,063,912	57	1,177
Hamming 160	242,931,704	247	984
Hamming 512	863,559,216	1,012	853
Mult 32	4,199,448	993	4
MatrixMult3x3 32	72,790,432	27,369	3
MatrixMult5x5 32	286,071,488	127,225	2
MatrixMult8x8 32	1,079,894,416	522,304	2
SHA3 256	29,354,783,052	37,760	777
AES 128	54,621,701,856	6,400	8,535

is XOR-shared between two parties. Table 5 shows the improvement for these functions by SkipGate over the state-of-the-art GC.

**Bubble-Sort:** This function receives a list of 32 32-bit integers, sorts the list using Bubble Sort algorithm, and then writes the sorted list in the output memory.

**Merge-Sort:** This function receives a list of 32 32-bit integers, sorts the list using Merge Sort algorithm, and then writes the sorted list in the output memory.

**Dijkstra:** This function receives the adjacency matrix of a directed graph with 64 weighted edges (described as a 32-bit integer), finds the shortest path between a source and other nodes using Dijkstra algorithm, and then writes the corresponding distances in the output memory.

**CORDIC:** COordinate Rotation DIgital Computer receives a degree and a 2D vector described as 32-bit fixed-points (2-bit decimal and 30-bit fraction), computes trigonometric, hyperbolic, or exponential functions according to Universal CORDIC algorithm [43], and then writes the final 2D vector in the output memory. The output vector in CORDIC algorithm converges one bit per iteration; thus, it requires 32 iterations in our case. The addition, shift, and non-oblivious table lookup are the only required operations in this algorithm. Universal CORDIC has two modes for updating vector: rotational and vectoring and three modes for lookup table: circular, linear, and hyperbolic. Combining these two modes allows the user to compute trigonometric, hyperbolic, exponential, square root, multiplication, or division functions. Among these functions, square root and division have previously been reported in [12] and require 12,733 and 12,546 non-XOR gates respectively, almost three times more than ARM2GC.

**Table 5: Improvement by SkipGate on ARM2GC for the complex functions.**

Function (bit)	# of Garbled Non-XOR		Improv. (1000X)
	w/o SkipGate	w/ SkipGate	
Bubble-Sort32 32	1,366,390,620	65,472	21
Merge-Sort32 32	981,712,458	540,645	2
Dijkstra64 32	1,493,339,886	59,282	25
CORDIC 32	228,847,596	4,601	50

## 6 RELATED WORK

The idea of designing a custom programming language to describe and efficiently compile functions for secure evaluation dates back to Fairplay, the first GC compiler [23]. Fairplay introduces a custom language, namely, the Secure Function Definition Language (SFDL). SFDL compiles to Secure Hardware Description language (SHDL). More powerful languages and compilers were later presented [8, 17, 30]. The introduction of a custom programming language is neither user-friendly nor versatile when compared with conventional programming languages like C.

Another approach adopted in FastGC [9, 11], VMCRYPT [22], and ABY [4] for GC circuit generation is to design a library containing implementations of GC optimized sub-circuits in a general-purpose high-level language like Java. This method requires the user to have a thorough understanding of the circuit description of the secure function as the circuits and their decomposition into sub-circuits has to be specified manually.

The first GC implementation supporting a general purpose language is CBMC-GC [10] which supports ANSI-C. However, it supports only a subset of ANSI-C that is not compatible with many important primitives, and therefore, not compatible with legacy code. The main drawback of [10] is the compile-time loop unrolling that makes it scale poorly with the input size. To cope with this problem, the compiler presented in [16] introduces loops that are specified manually within the code and not unrolled until the GC evaluation. The circuit is stored as a Portable Circuit Format (PCF). This compiler supports a more general version of C language. However, in [10] and [16], the code had to be compiled with their custom compiler. As a result, users cannot benefit from the optimizations provided by general purpose compilers. Moreover, these compilers are less scrutinized and more prone to bugs. In contrast, ARM2GC supports any general purpose ARM compiler and thus benefits from all the state-of-the-art optimizations, supports legacy codes, and is fully verified.

The TinyGarble framework [41] allows a user to describe the function with a Hardware Description Language (HDL) like Verilog or VHDL. It presents custom GC-optimized libraries which enable synthesis of the HDL code with standard logic synthesis tools, thus, benefiting from the standard hardware optimizations. TinyGarble also suggests using sequential circuits for GC to solve the scalability issue. Unlike [16], it allows to infer loops automatically and to optimize across multiple sub-circuits. However, TinyGarble limits the programmer to a hardware level language which is less user-friendly than a high-level compiler. Our work utilizes TinyGarble’s methodology to generate the most optimized Boolean circuit for the ARM processor. The big advantage of ARM2GC is that the function to be evaluated securely can be written in any programming language and compiled with any ARM compiler of choice.

The work in [45] accepts a function as a MIPS machine code, which allows the programmer to describe the function in a language of her choice and compile the function with a standard compiler. They design a MIPS emulator to securely execute the code. To avoid emulating a large number of instructions supported by the MIPS machine, they perform a data independent static analysis before execution of the program to build a small instruction bank and ALU circuit tailored for each processor cycle. In contrast, our approach

performs this optimization with bit-precision instead of instruction-precision. Moreover, this is done in the runtime while the circuit remains the same for each cycle.

To solve the problem of secure conditional branches, Wang et al. [45] propose to pad `nop` instruction to parallel branches so that their lengths become equal. This way when the code exits either of the branches, it ends up in the same instruction and the process can continue with less cost. However, this approach increases the cost for conditional branches. To mitigate this problem, we propose to use the ARM processor which supports conditional execution and can replace these branches with conditional instructions (see Section 4.2). In rare cases where the ARM compiler fails to replace the conditional branch, we adopted their approach in padding the parallel branches with `nop` instruction. Overall, our evaluation shows that ARM2GC outperforms their MIPS framework, for example by 4 orders of magnitudes for Hamming distance function, mostly thanks to the SkipGate algorithm and its bit-precision optimization.

Recently, Mood et al. [25] performed extensive research on the efficiency and reliability of the current frameworks and found out that most of them suffer from reliability issues. For example, they reported that PAL, KSS, CMBC, Obliv-C, OblivVM, and PCF crashed on programs that should have been compiled correctly. Moreover, KSS, OblivVM, and PCF generated incorrect netlists. As they discuss in the paper, there are serious limitations for formal verification and due to its impracticality, they limit their analysis to validation by testing. This type of testing does not detect all possible flaws in the compilation process. While many of the issues were later taken care of by the respective developers, this research exposed a serious reliability issue regarding the usage of these compilers.

Frigate [25] introduces a new C-style language for SFE and the corresponding compiler. Whereas in our work, we utilize C language with standard ARM cross compiler. Our work also supports any programming language and its corresponding ARM compiler. As of now, Frigate only supports three different types (`uint_t`, `int_t`, and `struct_t`). The user can add her own types but it requires a good understanding of the internal structure of the compiler. Since these three types have a specific bit length, the final computation is not bit-level efficient. Frigate divides the program into different functions and creates the circuit by calling the corresponding functions and as a result prohibits the overall circuit optimization. In contrast, our ARM circuit is optimized globally using state-of-the-art hardware synthesis techniques. Therefore, our overall platform is based on very well-developed and debugged tools that have been used in industry for many years. Also, if any new update becomes available for these tools, they can effortlessly be incorporated into our framework.

It is worth mentioning that SkipGate is different from the “constant propagation” and “dead gate elimination” techniques introduced in [16] and [17], respectively. These solutions eliminate parts of the code that do not contribute to the output (or can be computed) at the compile time using static analysis. In contrast, SkipGate performs gate-level optimization dynamically at the runtime to reduce the number of non-XOR gates to close to the optimal value (compared to state-of-the-art HDL frameworks [41]). Indeed, this is the reason why ARM2GC outperforms [16, 17]. For example, for 160-bit Hamming distance, [16] reports 880 number of non-XOR gates

**Table 6: High-level characteristics of secure computation frameworks, their programming languages, and compilers.** “Cust.” indicates custom designed-compilers. CP: support for Constant Propagation. DCE: support for Dead Code Elimination. DGE: support for Dynamic Gate Elimination.

Framework	Lang.	Compiler	CP	DCE	DGE
CBMC-GC [2]	ANSI-C	Cust.	yes	yes	no
KSS [17]	DSL	Cust.	no	yes	no
PCF [16]	ANSI-C	Cust.	yes	yes	no
OblivVM [21]	DSL	Cust.	no	no	no
Obliv-C [48]	DSL	Cust.	yes	yes	no
TinyGarble [41]	HDL	HW Synth.	no	yes	no
Frigate [25]	DSL	Cust.	yes	yes	no
ARM2GC	C/C++ <sup>†</sup>	ARM	yes	yes	yes

any language with supported ARM compiler

while ARM2GC garbles only 247. Table 6 shows a high-level comparison of ARM2GC to the prior secure computation frameworks.

In [14] and [31], authors address an interesting yet orthogonal problem to ours. They compute what information can be obtained from computation output and each party’s private input, whereas, we compute what information can be revealed based on private and public inputs from both parties to avoid garbling/evaluating selected gates. Their approach is inapplicable when only one party is provided with final output and function is required to be evaluated without revealing intermediate values. They do not use a standard verified compiler and cannot garble sequential circuits.

There has been extensive research on secure computation frameworks for machine learning [13, 24, 33–35, 38]. These frameworks are customized for the operations that are frequently performed in the machine learning applications such as vector-dot-product and certain non-linear functionalities. In contrast, the focus of this work is to create a generic high-level secure computation framework.

## 7 CONCLUSION

This paper introduces the novel SkipGate algorithm for Yao’s Garbled Circuit protocol. The algorithm dynamically omits the communication cost for gates with outputs independent of private data and also the gates not affecting the final output. Based on the SkipGate algorithm and the ARM processor architecture, we create ARM2GC, a simple-to-use and verified garbled circuit framework. Users can develop secure functions in high-level languages and compile them using standard ARM cross-compilers. As a result of SkipGate, only the gates associated with private data in the ARM circuit incur communication and encryption cost. Evaluations on a host of benchmark functions show that the ARM2GC framework achieves efficiency close to that of HDL-level synthesis methods.

## REFERENCES

- [1] Mihir Bellare, Viet Tung Hoang, Sriram Keelvedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*. IEEE, 2013.
- [2] Niklas Büscher, Martin Franz, Andreas Holzer, Helmut Veith, and Stefan Katzenbeisser. On compiling boolean circuits optimized for secure multi-party computation. *Formal Methods in System Design*, 51(2):308–331, 2017.
- [3] Design Compiler. Synopsys inc. <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler,2000>.

- [4] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.
- [5] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*. ACM, 2017.
- [6] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI-C compiler for secure two-party computations. In *Compiler Construction*. Springer, 2014.
- [7] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In *CCS*. ACM, 2015.
- [8] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for automating secure Two-partY computations. In *CCS*. ACM, 2010.
- [9] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *ASIACCS*. ACM, 2013.
- [10] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *CCS*. ACM, 2012.
- [11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, volume 201, 2011.
- [12] Siam U Hussain and Farinaz Koushanfar. Privacy preserving localization for smart automotive systems. In *DAC*. ACM, 2016.
- [13] C Juvekar, V Vaikuntanathan, and A Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.
- [14] Florian Kerschbaum. Automatically optimizing secure computation. In *CCS*. ACM, 2011.
- [15] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*. Springer, 2008.
- [16] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin RB Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Security*. USENIX, 2013.
- [17] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Security*. USENIX, 2012.
- [18] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*. Springer, 2007.
- [19] Yehuda Lindell and Benny Pinkas. A proof of security of YaoâZs protocol for two-party computation. *Journal of Cryptology*, 22(2):161â188, 2009.
- [20] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 2012.
- [21] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P*. IEEE, 2015.
- [22] Lior Malka. VMCrypt: modular software architecture for scalable secure computation. In *CCS*. ACM, 2011.
- [23] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay-secure two-party computation system. In *Security*. USENIX, 2004.
- [24] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 38th IEEE Symposium on Security and Privacy (SP)*, pages 19â38. IEEE, 2017.
- [25] Benjamin Mood, Debaryan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *Euro S&P*. IEEE, 2016.
- [26] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. In *Journal of Cryptology*. Springer, 2005.
- [27] Moni Naor, Benny Pinkas, and Reuben Sumner. Privacy preserving auctions and mechanism design. In *CEC*. ACM, 1999.
- [28] Annika Paus, Ahmad-Reza Sadeghi, and Thomas Schneider. Practical secure evaluation of semi-private functions. In *International Conference on Applied Cryptography and Network Security*, pages 89â106. Springer, 2009.
- [29] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *ASIACRYPT*. Springer, 2009.
- [30] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. WYSTERIA: A programming language for generic, mixed-mode multiparty computations. In *S&P*. IEEE, 2014.
- [31] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A Hammer. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 3â14. ACM, 2013.
- [32] M Sadegh Riazi, Neeraj KR Dantu, LN Vinay Gattu, and Farinaz Koushanfar. Gen-match: Secure dna compatibility testing. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 248â253. IEEE, 2016.
- [33] M Sadegh Riazi, Mojgan Javaheripi, Siam U Hussain, and Farinaz Koushanfar. MPCircuits: Optimized circuit generation for secure multi-party computation. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 198â207. IEEE, 2019.
- [34] M Sadegh Riazi, Bita Darvish Rouhani, and Farinaz Koushanfar. Deep learning on private data. In *IEEE Security and Privacy Magazine*. IEEE, 2019.
- [35] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, volume 2019.
- [36] M Sadegh Riazi, Ebrahim M Songhori, and Farinaz Koushanfar. Prisearch: Efficient search on private data. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1â6. IEEE, 2017.
- [37] M Sadegh Riazi, Ebrahim M Songhori, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Toward practical secure stable matching. *Proceedings on Privacy Enhancing Technologies*, 2017(1):62â78, 2017.
- [38] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707â721. ACM, 2018.
- [39] Conor Santfort. Amber ARM-compatible core. *OpenCores.org*, 2010.
- [40] Andrew Sloss, Dominic Symes, and Chris Wright. *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.
- [41] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*. IEEE, 2015.
- [42] Ebrahim M Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. GarbledCPU: A MIPS processor for secure computation in hardware. In *DAC*. IEEE, 2016.
- [43] Jack E Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, (3), 1959.
- [44] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS*. ACM, 2015.
- [45] Xiao Wang, S Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. In *European Symposium on Research in Computer Security*, pages 99â117. Springer, 2016.
- [46] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. SCORam: oblivious ram for secure computation. In *CCS*. ACM, 2014.
- [47] A. Yao. How to generate and exchange secrets. In *FOCS*. IEEE, 1986.
- [48] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. *IACR Cryptology ePrint Archive*, 2015:1153, 2015.
- [49] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*. Springer, 2015.
- [50] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square root ORAM: Efficient random access in multi-party computation. In *S&P*. IEEE, 2016.