ARM2GC: Simple and Efficient Garbled Circuit Framework by Skipping

Ebrahim M. Songhori
UC San Diego

M. Sadegh Riazi
UC San Diego

Siam U. Hussain
UC San Diego

Ahmad-Reza Sadeghi
Technische Universitat Darmstadt

Farinaz Koushanfar
UC San Diego

ABSTRACT

We present ARM2GC, a novel secure function evaluation framework based on Yao's Garbled Circuit (GC) protocol and the ARM processor. It allows users to develop privacy-preserving applications using high-level programming languages (e.g., C) and compile them using standard ARM compilers (e.g., gcc-arm). In our framework, the underlying Boolean circuit is that of an ARM processor to which the compiled binary of the function is input as a non-private instruction code. The main enabler of this construction is the introduction of SkipGate, an algorithm that omits the communication and encryption cost of a Boolean gate when its output is independent of the private data. SkipGate greatly enhances the performance of ARM2GC by omitting costs of the gates associated with the instructions in the compiled binary, thus making it practical and efficient. 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 results which were achieved using conventional logic synthesis tools and hardware description language.

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 promising methods for two-party SFE is Yao's Garbled Circuit (GC) protocol proposed by Andrew Yao in 1986 [1]. Yao's protocol immediately attracted a significant attention from the cryptographic community but was believed to be of limited practical usage for many years. The protocol requires representing the underlying function as a Boolean circuit. The non-trivial challenge of GC is to generate this Boolean circuit such that its secure evaluation requires the minimum inter-party communication, thus, optimizing the overall performance.

The challenge of the GC circuit optimization was partially addressed by TinyGarble [2]. The work showed 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 [3–7]. A major disadvantage of TinyGarble, however, is that peak 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 in high-level programming languages, e.g., C. In this paper, our goal is to combine the efficiency of hardware synthesis with the versatility of high-level languages.

A rather recent development in this field is the introduction of a *garbled processor* where the underlying Boolean circuit is that of a general-purpose processor [8, 9]. Users of a garbled processor develop the function in a high-level language and feed its compiled binary code to the processor along with their private inputs. In the garbled processor's case, the binary code which resembles the functionality, unlike the parties' inputs, is not private and is publicly known to both parties. However, until now, this publicly known input has been treated as a secret input, hence, significantly increasing the communication cost of the secure evaluation. To overcome this inefficiency, performing instruction-level optimization has been suggested for garbled processors. That is, at each cycle, a custom processor is generated such that it only supports the instruction(s) to be executed at that cycle. However, due to the coarse-grain nature of this optimization, the garbling costs were unacceptably high even compared to GC frameworks supporting high-level languages.

In this work, we introduce a methodology to perform a fine-grain gate-level optimization on the garbled processor such that only the gates associated with the private inputs incur garbling cost. Our key observation is that in GC with publicly known inputs, there are gates whose outputs are independent of the secret values (and thus known to both parties). The outputs of these gates can be computed locally by each party without communication or encryption. The other major observation is that the gates that do not contribute to the final output can be skipped. Our observations lead to the development of a novel algorithm called SkipGate that wraps around the GC protocol. Unlike the current GC frameworks, the publicly known inputs and wires are not treated as secret values in SkipGate. Using these public values, the algorithm 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, at the expense of a small increase in local computation. Please note that SkipGate avoids unnecessary garbling and is different from the cryptographic improvements of GC such as free-XOR [10] and Half Gate [11] that reduce the garbling cost of individual gates. SkipGate operates on top of these methods, and we suppose the underlying GC protocol in SkipGate already benefits from these cryptographic improvements. SkipGate also differs from the static

circuit simplification method [12] that removes gates with constant inputs (see Section 3).

SkipGate is most effective for reducing the garbling cost of sequential circuits [2] containing known control paths. An example of such a circuit is the garbled processor where the control path depends on the input binary code of the function that is known to both parties. By utilizing this property, we develop a high-level GC framework called ARM2GC built upon the ARM instruction set and the SkipGate algorithm. Users can develop functions in high-level languages, e.g., C/C++ and compile them using standard ARM cross-compilers, e.g., gcc-arm. In contrast to the earlier custom high-level compilers which called for new ad-hoc verification techniques [7, 13–15], ARM2GC inherits the ARM's available fully verified compilers. In fact, the most recent framework, Frigate [16] performed extensive research on the efficiency and reliability of the current compilers and found out that most of them suffer from reliability issues. One major advantage of ARM2GC is that the ARM compiler goes through rigorous verification and hence does not suffer from the reliability issues reported by [16]. Thanks to SkipGate, ARM2GC incurs a garbling cost comparable to the HDL synthesis approach while allowing users to develop SFE applications in a high-level language.

Our work leverages ARM as the general purpose processor, instead of the earlier MIPS-based garble processors [2, 8, 9]. ARM's pervasiveness and, most importantly, conditional execution are the two main advantages of ARM over MIPS. The latter simplifies the framework by reducing conditional branches and making the program flow predictable for both parties to take the full advantage of the SkipGate algorithm. We adapt the standard ARM architecture (without affecting the instruction set) such that it incurs less cost in GC. ARM's interrupts, co-processors, and performance-related components including cache and pipeline are removed or modified because they only increase the number of gates in the ARM circuit without providing any performance advantages in GC.

The source-code of the SkipGate algorithm and the ARM2GC framework is integrated into the TinyGarble repository[1].

Contributions.

- We introduce the novel SkipGate algorithm that can be added to the GC protocol 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 fully verified compilers of ARM. We adapt the ARM architecture (without affecting the instruction set) to make it most effective for the GC protocol with SkipGate.
- We perform extensive experiments to evaluate the SkipGate algorithm and the ARM2GC framework. The ARM2GC framework demonstrates comparable performance to HDL synthesis approach of TinyGarble [2]. Its overhead is negligible for most of the benchmark functions. ARM2GC outperforms the state-of-the-art garbled processors [8, 9] and high-level GC compilers [4, 16].

[1] https://github.com/esonghori/TinyGarble

2 PRELIMINARIES

2.1 Security Model

Consistent with the earlier relevant literature [4–7, 16], 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 [17]. This model can be generalized to more advanced adversary models that are typically addressed by multiple runs of the basic honest-and-curious model [18, 19].

2.2 Oblivious Transfer

Oblivious Transfer (OT) [20] is a cryptographic protocol based on public key encryption executed between Alice (sender) and Bob (receiver) where Bob selects one from a set of messages provided by Alice without revealing his selection. 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 [1] 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 $k = 128$ [17]. 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 *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 [21]. 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*** [10] 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*** [22] 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 [11] utilizes both free-XOR and row reduction and reduces the cost of AND gates by an additional 25%.

Sequential GC: 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 circuit for the GC protocol [2]. Sequential circuits run for multiple clock cycles and include memory elements (flip-flops) in addition to logic gates. 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 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.

3 SKIPGATE ALGORITHM

SkipGate is developed to work with 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, the previous GC framework did not address the fact that some part of the inputs a and b may be public, i.e., known to both parties. For example, if the function is RSA, the encryption key is public. A more practical scenario is garbling a general purpose processor as we explain in detail later in Section 4. In general, the processor will have two types of inputs: instruction and data, where the first one 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 [2, 8, 9] proposed generating customized netlists for limited instruction sets. However, they fail to achieve the maximum possible optimization due to the coarse grain nature (instruction level as opposed to gate level) of their approach.

In SkipGate, we introduce a new variable 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 $f(a, b, p)$ into a simpler circuit of $c = f_p(a, b)$ with the same logic for a given public input p. Secure evaluation of $f_p(a, b)$ costs less than that of $f(a, b, p)$ using the conventional GC protocol where p is treated 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 sequential circuit 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 $X_w = X_w^1$, it is 1. This means that the information is shared between two parties. In our scheme, we combine these notions 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 input 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 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 was treated as a secret value, the entire circuit had to be garbled/evaluated. In the following subsection, we explain how the SkipGate algorithm identifies such gates to reduce the garbling cost in circuits with public wires.

It is worth noting that in a sequential garbled circuit [2], the Boolean value of a wire can change at every clock cycle. A wire may also alter between being secret and public. The SkipGate algorithm is executed once for each sequential cycle. SkipGate's decision on each gate (locally computing, garbling/evaluating, or skipping) depends on the status of the gate's inputs (public or secret) on that cycle. Thus, SkipGate is fundamentally different compared to offline circuit simplification methods such as the one introduced in [12] which remove gates with known constant inputs. The constant gates are already removed in our circuits that are generated by the conventional HDL synthesis tools.

3.1 Gate Categories

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

i *Gate with two public inputs.* In this case, the output is public.

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 0 at one input, the output becomes 0. This means that if the secret input is not connected to any other gate, the gate generating it 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 swapped) labels.* This indicates that the two secret inputs have identical (or inverted) Boolean values. (We will explain shortly how Bob identifies the swapped case.) 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 gates, 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

Algorithm 1 SkipGate, Alice's side.

Inputs: Sequential `circuit` of $f(a, b, p)$, Alice's input a, public input p, number of clock cycles cc.
Outputs: Pairs of output labels X_c^0 and X_c^1.

1: **SkipGate_alice (circuit, a, p, cc):**
2: $(X_a^0, X_a^1, X_b^0, X_b^1)$ = generate_random_labels()
3: send_alice_labels(a, X_a^0, X_a^1)
4: send_bob_labels(X_b^0, X_b^1) *// through OT*
5: circuit.set_private_input$(X_a^0, X_a^1, X_b^0, X_a^1)$
6: circuit.set_public_input(p)
7: **for** cid in $[0 \ldots cc-1]$ **do**
8: circuit.initial_label_fanout()
9: circuit.phase1()
10: garbled_tables = circuit.phase2_alice()
11: send_garbled_tables(garbled_tables)
12: circuit.transfer_flip_flops_labels()
13: **end for**
14: (X_c^0, X_c^1) = circuit.get_output_label()
15: X_c = receive_bob_output_label()
16: c=get_output_value(X_c^0, X_c^1, X_c)

Algorithm 2 SkipGate, Bob's side.

Inputs: Sequential `circuit` of $f(a, b, p)$, Bob's input b, public input p, number of clock cycles cc.
Outputs: Output label X_c.

1: **SkipGate_bob(circuit, b, p, cc):**
2: X_a = receive_alice_labels()
3: X_b = receive_bob_labels(b) *//through OT*
4: circuit.set_private_input(X_a, X_b)
5: circuit.set_public_input(p)
6: **for** cid in $[0 \ldots cc-1]$ **do**
7: circuit.initial_label_fanout()
8: circuit.phase1()
9: garbled_tables = receive_garbled_tables()
10: circuit.phase2_bob(garbled_tables)
11: circuit.transfer_flip_flops_labels()
12: **end for**
13: X_c = circuit.get_output_label()
14: send_output_label(X_c)

corresponding garbled table is not transferred from Alice to Bob.

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 (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. In Line 12 of Algorithm 1 and Line 11 of Algorithm 2, the labels associated with the input of flip-flops are transferred to their output for the next cycles [2]. Similar to conventional GC, at the end, Alice learns pairs of labels for each output wire and Bob has one of the pairs; 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 input labels finds the real output value (Line 15-16 of Algorithm 1 and Line 14 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 Algorithm 2), the `label_fanout` is set to the gate fanout in the circuit[2]. `label_fanout` of a gate may decrease if its output label is not needed anymore, 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 final output. The gates with `label_fanout=0` are subsequently marked for skipping, which in turn decreases the `label_fanout` of their input gates recursively. Finally, the gates in Category iv that have not been marked for skipping are garbled/evaluated.

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

1: **circuit.phase1():**
2: **for** g in circuit **do**
3: **if** g.i0 is public and g.i1 is public *//Category i* **then**
4: g.o = public_calculate(g.type, g.i0, g.i1)
5: g.label_fanout = 0
6: **else if** g.i0 is public or g.i1 is public *//Category ii* **then**
7: g.o = g.half_public_calculate(g.type, g.i0, g.i1)
8: **if** g.o is public **then**
9: g.label_fanout = 1 *//will become zero in recursive_reduction()*
10: circuit.recursive_reduction(g)
11: **end if**
12: **end if**
13: **end for**

Algorithm 3 illustrates Phase 1 of SkipGate in which Alice and Bob find and compute the gates that belong to Categories i-ii. `label_fanouts` 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 gates 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 disadvantageous: First, it makes the protocol complicated if one wants to enhance the security of the protocol to be secure against malicious adversaries. Second, it increases the communication overhead which is the bottleneck of the GC protocol.

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

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

Output: `garbled_tables` queue.

```
1:  circuit.phase2_alice():
2:  for g in circuit where g.label_fanout > 0 do
3:    if (g.i0.label is equal g.i1.label or
      g.i0.label is inverted g.i1.label) //Category iii then
4:      g.o = related_secret_calculate(g.type, g.i0, g.i1)
5:      if g.o is public then
6:        g.label_fanout = 1 //will become zero in recur-
          sive_reduction()
7:        circuit.recursive_reduction(g)
8:      end if
9:    else
10:     //Category iv
11:     (g.o, g.table) = garble(g.type, g.i0, g.i1) //table=null for XOR
12:     if g is non-XOR then
13:       garbled_tables.enqueue(g.id, g.table)
14:     end if
15:   end if
16: end for
17: garbled_tables.filter(t : circuit[t.id].label_fanout > 0)
```

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

Input: `garbled_tables` queue.

```
1:  circuit.phase2_bob(garbled_tables):
2:  for g in circuit where g.label_fanout > 0 do
3:    if (g.i0.label is equal g.i1.label or
      g.i0.label is inverted g.i1.label) //Category iii then
4:      g.o = related_secret_calculate(g.type, g.i0, g.i1)
5:      if g.o is public then
6:        g.label_fanout = 1 //will become zero in recur-
          sive_reduction()
7:        circuit.recursive_reduction(g)
8:      end if
9:    else
10:     //Category iv
11:     if g is XOR then
12:       g.o = g.eval_XOR(g.i0, g.i1)
13:     else if g.id is garbled_tables.top().id then
14:       gt = garbled_tables.dequeue().table
15:       g.o = g.eval(g.type, g.type, g.i0, g.i1, gt)
16:     else
17:       g.o = next_unique_label() //generate a unique label.
18:     end if
19:   end if
20: end for
```

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 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 17 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 17 of Algorithm 5, Bob generates and assigns new unique labels (`next_unique_label`) 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 beside each label that indicates the label is generated by him and is not valid for GC evaluation.

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 itself with the gates that generate the 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 has only one label X_w for each secret wire w. Due to free-XOR [10], he does not need to do anything with the label when he evaluates a NOT gate because the labels corresponding to 0 and 1 are switched by Alice during the garbling

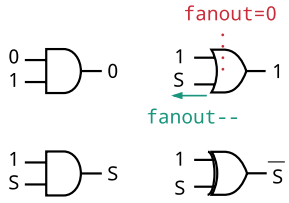

Figure 1: Four examples of replacing gates in Phase 1 by zero, one, wire, or inverter. `label_fanout` is decreased for the unnecessary gate. The top-left example is in Category i and the rest are in Category ii.

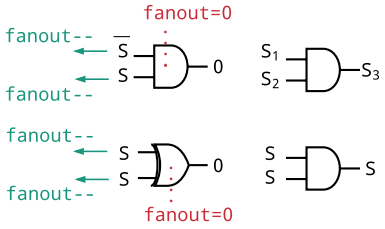

Figure 2: Four examples of replacing and computing gates in Phase 2. `label_fanout` is decreased for the unnecessary gates. The top-right example is in Category iv, and the rest are in Category iii.

Algorithm 6 Recursive Fanout Reduction of SkipGate.

Inputs: Gate g (where the reduction starts).

```
1:  circuit.recursive_reduction(g):
2:  if g.label_fanout is 0 then
3:      return
4:  end if
5:  g.label_fanout = g.label_fanout - 1
6:  if g.label_fanout is 0 then
7:      if g.i0 is secret then
8:          circuit.recursive_reduction(circuit[g.i0])
9:      end if
10:     if g.i1 is secret then
11:         circuit.recursive_reduction(circuit[g.i1])
12:     end if
13: end if
```

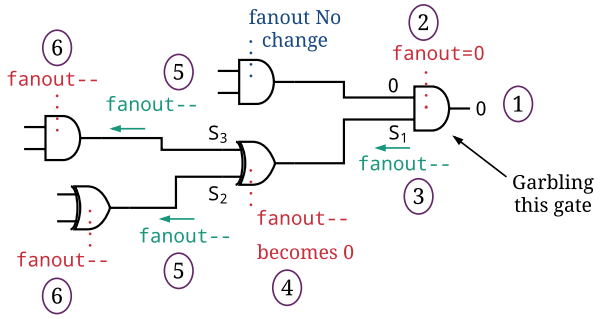

Figure 3: Recursive reduction of `label_fanout` to skip unnecessary gates in Phase 1.

of the gate. This collectively flips the secret value of w. This means that Bob cannot tell apart an identical and inverted secret value just by looking at the label. 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. The extra bit helps him to differentiate between identical and inverted secret values which are crucial for Phase 2.

3.4 Computational Complexity

The SkipGate algorithm decreases the communication cost 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* procedure (Algorithm 6), is executed once per gate, thus they incur a complexity similar to the classic GC protocol. The only possible source of complexity increase 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 compute 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 and 6 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 is the number of circuit output, and m is the number of circuit inputs. 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 have to 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, if we can prove that the operation of SkipGate on a single gate is correct, the entire algorithm is proved to be logically correct.

The operations for gates in Category i are 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. Since this operation is performed based on the Boolean logic of the pertinent gate, the output remains logically correct. 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 algorithm with 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 [17, 23]. 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 SkipGate algorithm for reducing $f(a, b, p)$ to $f_p(a, b)$. Thus, no information about the private inputs a and b is revealed by 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 the GC protocol still holds for SkipGate.

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 ARM2GC.

4.1 Global Flow

The ARM2GC framework allows users to write two-party SFE program in C/C++ (or any language that can be compiled to 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 and the synthesized ARM processor circuit are fed to the SkipGate algorithm as the public input p and the Boolean circuit, respectively.

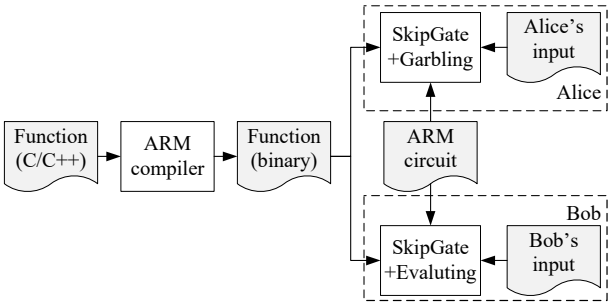

Figure 4: Overview of the ARM2GC framework.

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 MUXs) 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 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 using sequential garbling process [2] for a pre-specified number of clock cycles cc.

```
1:  cmp $8, $9
2:  bne L0
3:  mov $1, #10
4:  b  L1
5: L0:
6:  mov $2, #20
7: L1:
    ...
```

(a) Without Conditional Execution

```
1:  cmp $8, $9
2:  moveq $1, #10
3:  movne $2, #20
4: L1:
    ...
```

(b) With Conditional Execution

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

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 [2, 8, 9]. 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 [24].

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 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. Moreover, we 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 and as a result, program flow itself will become secret which in turn reduces the efficiency of the execution.

We modify and remove a few features from the ARM processor like interrupts, co-processors, and performance-related components including cache and pipeline. The last group does 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 non-XOR gates is the only factor affecting the performance.

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 [8, 9] would incur even a 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 ARM processor.

4.3 How SkipGate Helps

As explained above, the instruction memory of the ARM processor is initialized with public values. Therefore, if the program counter (the address of the next instruction) is public, 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 in the public binary code p, e.g., $c = a + b$. Therefore, the main garbling cost is paid only for the actual computation of the secret values. As explained in the previous section, SkipGate performs these optimizations at the gate-level, in contrast to instruction-level of [8, 9].

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 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 [25, 26]. 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 of the register file is read or written. The SkipGate algorithm utilizes this 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 of ORAM.

In rare occasions where two or more instructions should be garbled at a time, accessing a register would not be free using MUXs and SkipGate. 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. Since the cost of fetching instructions remains smaller than that of break-even points of sub-linear ORAMs, using ORAM would not improve the efficiency for this case either.

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 costs far less than 1-out-of-16 oblivious access. The cost of oblivious

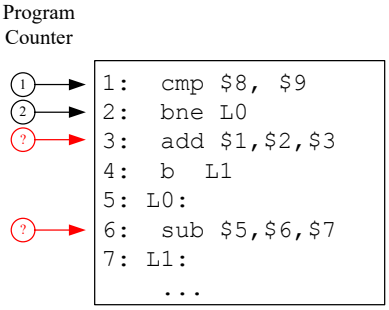

Figure 6: In case of compiler failure to replace a secret branch with conditional instructions, the parties do not know which instruction is executed after the branch. Thus, the instruction becomes secret.

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 mixture 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?

5 EVALUATION

5.1 Evaluation Setup

We use Synopsis Design Compiler (DC) H-2013.03-SP4 [27] along with TinyGarble [2] 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 [28]. The ARM circuit is modified as explained in Section 4.2. Synthesizing the ARM processor with Synopsis DC takes 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 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 few seconds to compile a function into an ARM binary code.

Table 1: SkipGate algorithm improvement on sequential circuits generated by TinyGarble (TG) [2]. These functions do not have public inputs. SkipGate benefits from the small number of flip-flops initial values that are public to reduce their garbling cost.

Function (bit)	# of garbled non-XOR		# of skipped non-XOR	Improv.
	TG [2]	SkipGate		
Sum 32	32	31	1	3.1%
Sum 1024	1,024	1,023	1	0.1%
Compare 32	32	32	0	0.0%
Compare 16,384	16,384	16,384	0	0.0%
Hamming 32	160	145	15	9.4%
Hamming 160	1,120	1,092	28	2.5%
Hamming 512	4,608	4,563	45	1.0%
Mult 32	2,048	2,016	32	1.6%
MatrixMult3x3 32	25,947	25,668	279	1.1%
MatrixMult5x5 32	120,125	119,350	775	0.6%
MatrixMult8x8 32	492,032	490,048	1,984	0.4%
SHA3 256	40,032	38,400	1,632	4.1%
AES 128†	15,807	6,400	9,407	59.5%

†We add the missing key expansion module to AES 128 of [2] here.

Table 2: The number of garbled non-XOR gates for the benchmark functions. Comparing ARM2GC to TinyGarble's hardware synthesis [2].

Function (bit)	TinyGarble (Verilog) [2]	ARM2GC (C)	Overhead
Sum 32	31	31	0.0%
Sum 1024	1,023	1,023	0.0%
Compare 32	32	32	0.0%
Compare 16,384	16,384	16,384	0.0%
Hamming 32	160	57	-64.4%
Hamming 160	1,120	247	-77.9%
Hamming 512	4,608	1,012	-78.0%
Mult 32	1,023	993	-2.9%
MatrixMult3x3 32	27,369	27,369	0.0%
MatrixMult5x5 32	120,125	127,225	5.9%
MatrixMult8x8 32	492,032	522,304	6.2%
SHA3 256	38,400	37,760	-1.7%
AES 128†	6,400	6,400	0.0%

†Here, we added the cost of missing key expansion in AES 128 to the reposted result in [2].

5.2 Benchmark Functions and Metrics

We use the benchmark functions that have been frequently used for evaluation in the GC literature [2, 4, 8]. The benchmarks are as follows:

- Sum adds two integers.
- Compare compares two integers.
- Hamming finds the Hamming distance between two integers.
- Mult calculates the product of two integers.
- MatrixMult$N \times N$ computes matrix multiplication of two $N \times N$ matrices.
- SHA3 finds the SHA3-256 hash of a string.
- AES computes AES-128 encryption given two 128-bit numbers as the key and the message.

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 [10].

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 [2], 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 [2] with and without applying the SkipGate algorithm. The total number of non-XOR gates to be garbled is cc×#non-XORs in the sequential circuit and is shown in the second column. The table also reports the cost of garbling of the same circuits by employing the SkipGate algorithm (third column) and their percentage improvement (fifth column). 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 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 functions.

5.4 ARM2GC vs HDL Synthesis

Table 2 compares the cost of garbling of functions devised in Verilog HDL and constructed by the hardware synthesis technique of TinyGarble [2] with functions developed in C and constructed by the ARM2GC framework. As expected, ARM2GC incurs only a small overhead (at most 6.2% 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 78% improvement). Note that we use an efficient binary tree-based method [29] 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 the prior-art GC frameworks [2, 4, 16] and garbled processors [8, 9] along with the ARM2GC framework. The corresponding programming language is shown in parentheses. Note that this is not an exhaustive list and only includes the most recent GC frameworks that report the best results on the benchmark functions. The garbling cost of ARM2GC is compared with the best previous work in the right most column. In all cases, ARM2GC outperforms the earlier frameworks in terms of garbling cost. For

Table 3: Number of garbled non-XOR gates for the benchmark functions. Comparing ARM2GC to previous work.

Function (bit)	ANSI-C (C) [4]	TinyGarble HLS (C → Verilog) [2]	TinyGarble MIPS (C) [2]	SFE MIPS (C) [8]	GarbledCPU (C) [9]	Frigate (C) [16]	ARM2Garble (C)	Improvement*
Sum 32	32	288	-	-	-	-	31	3.2%
Sum 1024	-	9,216	-	-	-	1,025	1,023	0.2%
Compare 32	65	102	-		-	-	32	50.8%
Compare 16,384	-	52,224	-	-	-	16,386	16,384	0%
Hamming 32	601	253	3,762,725	481,000	2,860,590	-	57	77.5%
Hamming 160	3,003	1,264	18,456,485	-	14,302,950	719	247	65.6%
Hamming 512	9,610	4,045	58,864,325	49,600,000	45,769,440	-	1,012	75.0%
Mult 32	1,741	-	-	-	-	995	993	0.2%
MatrixMult3x3 32	47,583	-	-	-	-	-	27,369	42.5%
MatrixMult5x5 32	220,825	-	-	-	-	128,252	127,225	0.8%
MatrixMult8x8 32	905,728	-	-	-	-	-	522,304	42.3%
SHA3 256	-	-	-	-	-	-	37,760	-
AES 128	-	-	-	-	198,789,506	10,383	6,400	38.4%

*Compared to the best previous method

example, ARM2GC results in 12.2×, 5.1×, 74, 000×, 57, 000×, and 2.9× less number of non-XOR gates for 160-bit Hamming distance compared to ANSI-C [4], TinyGarble high-level synthesis (HLS) [2], PF-SFE MIPS [2], GarbledCPU [9], and Frigate [16] respectively. ARM2GC also results in 38.3% less non-XOR gates compared to Frigate [16] for AES function.

5.6 Effect of SkipGate on ARM

Table 4: SkipGate algorithm improvement on the ARM sequential circuit.

Function (bit)	# of non-XOR gates		Improvement (1000X)
	Conventional GC+ARM	ARM2GC	
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

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. On the other hand, 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 conventional GC without the SkipGate algorithm. The algorithm transforms the impracticable 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 that allows SkipGate to skip garbling/evaluation most of the gates in the ARM circuit.

Comparing the result of Table 1 and Table 4 shows that the extent of SkipGate's impact highly depends on the structure of the circuit, as well as the degree of presence of public values in the circuit.

5.7 Complex Functions

Table 5: SkipGate algorithm improvement on the ARM sequential circuit for the complex functions.

Function (bit)	# of non-XOR gates		Improvement (1000X)
	Conventional GC+ARM	ARM2GC	
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

We developed a number of complex functions, as described below, with the ARM2GC framework. In each of these functions, the input is XOR-shared between two parties. Table 5 shows the improvement for these functions by SkipGate over state-of-the-art GC.

Bubble-Sort32: 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-Sort32: 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): This function 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 [30], 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 operations are only required. 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 in each combination. Among these functions, square root and division have previously been reported in [31] and required 12,733 and 12,546 non-XOR gates respectively, almost three times more than ARM2GC.

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 [3]. Fairplay introduces a custom language created specifically to describe functions, namely the Secure Function Definition Language (SFDL). SFDL was compiled to Secure Hardware Description language (SHDL). More powerful languages and compilers were later presented [5, 32, 33]. 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 [29, 34], VMCRYPT [35], and ABY [6] 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 presented in [4], 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 codes. The main drawback of [4] is compile-time loop unrolling that makes it unscalable with input size. To cope with this problem, the compiler presented in [13] introduces loops that are specified manually within the code and not rolled out until the GC evaluation. This compiler supports a more general version of C language. However, in [4] and [13], 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 therefore more prone to bugs. In contrast, the ARM2GC framework supports any general purpose ARM compiler and thus benefit from all the state-of-the-art optimizations, supports legacy codes, and is fully verified.

The TinyGarle framework [2] allows a user to describe the function with a Hardware Description Language (HDL) like Verilog or VHDL. It presents customized 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 [13], it allows to infer loops automatically and to optimize across multiple sub-circuits. TinyGarble also 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 [8] accepts a function as a MIPS machine code, which allows the programmer to describe the function in a language of her choice and compile 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, they 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.

The most recent framework, Frigate [16] 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, ObliVM, and PCF crashed on programs that should have been compiled correctly. Moreover, KSS, ObliVM, 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 [16] 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 corresponding 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. For example for a 9-bit comparison, Frigate

needs to do the comparison for a given bit length of `int_t`. On the contrary, the ARM2GC framework eliminates unnecessary gates and evaluates the circuit only up to the number of bits needed. 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.

7 CONCLUSION AND FUTURE WORK

This paper introduces the novel SkipGate algorithm for Yao's Garbled Circuit protocol. The algorithm 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 efficient 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 massive 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.

As future work, we plan to investigate the integration of already implemented and optimized circuits as *co-processors* in ARM. The SkipGate algorithm will remove the overhead of co-processors when they are not used. Optimized co-processors may reduce the garbling cost for functions that cannot be implemented efficiently or easily using high-level languages, e.g., functions involving bit-manipulation and AES S-Boxes.

REFERENCES

[1] A. Yao, "How to generate and exchange secrets," in *FOCS*. IEEE, 1986.
[2] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *S&P*. IEEE, 2015.
[3] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay-secure two-party computation system." in *Security*. USENIX, 2004.
[4] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *CCS*. ACM, 2012.
[5] A. Rastogi, M. A. Hammer, and M. Hicks, "WYSTERIA: A programming language for generic, mixed-mode multiparty computations," in *S&P*. IEEE, 2014.
[6] D. Demmler, T. Schneider, and M. Zohner, "ABY-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*. The Internet Society, 2015.
[7] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *S&P*. IEEE, 2015.
[8] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of MIPS machine code," 2015. [Online]. Available: http://eprint.iacr.org/2015/547
[9] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, "GarbledCPU: A MIPS processor for secure computation in hardware," in *DAC*. IEEE, 2016.
[10] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *ICALP*. Springer, 2008.
[11] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *EUROCRYPT*. Springer, 2015.
[12] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *ASIACRYPT*. Springer, 2009.
[13] B. Kreuter, A. Shelat, B. Mood, and K. R. Butler, "PCF: A portable circuit format for scalable two-party secure computation." in *Security*. USENIX, 2013.
[14] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ANSI-C compiler for secure two-party computations," in *Compiler Construction*. Springer, 2014.
[15] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1153, 2015.
[16] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Euro S&P*. IEEE, 2016.
[17] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *S&P*. IEEE, 2013.
[18] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *EUROCRYPT*. Springer, 2007.
[19] ——, "Secure two-party computation via cut-and-choose oblivious transfer," *Journal of Cryptology*, 2012.
[20] M. Naor and B. Pinkas, "Computationally secure oblivious transfer," in *Journal of Cryptology*. Springer, 2005.
[21] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas, "Fast garbling of circuits under standard assumptions," in *CCS*. ACM, 2015.
[22] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *CEC*. ACM, 1999.
[23] Y. Lindell and B. Pinkas, "A proof of security of yaoâĂŹs protocol for two-party computation," *Journal of Cryptology*, vol. 22, no. 2, pp. 161–188, 2009.
[24] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.
[25] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "Scoram: oblivious ram for secure computation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 191–202.
[26] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square root ORAM: Efficient random access in multi-party computation." in *S&P*. IEEE, 2016.
[27] D. Compiler, "Synopsys inc," http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler, 2000.
[28] C. Santifort, "Amber arm-compatible core," *OpenCores.org*, 2010.
[29] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits." in *Security*, vol. 201, no. 1, 2011.
[30] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computers*, no. 3, 1959.
[31] S. U. Hussain and F. Koushanfar, "Privacy preserving localization for smart automotive systems," in *DAC*. ACM, 2016.
[32] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for automating secure Two-partY computations," in *CCS*. ACM, 2010.
[33] B. Kreuter, A. Shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries." in *Security*. USENIX, 2012.
[34] W. Henecka and T. Schneider, "Faster secure two-party computation with less memory," in *ASIACCS*. ACM, 2013.
[35] L. Malka, "Vmcrypt: modular software architecture for scalable secure computation," in *CCS*. ACM, 2011.