ARM2GC: Succinct Garbled Processor for Secure Computation

Ebrahim M. Songhori
esonghori@google.com
Google

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, 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 a 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 of the compiled binary, which is known by both parties involved in the computation. Our evaluation on benchmark functions demonstrates that ARM2GC outperforms the prior best solution by 156×.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

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 [24]. Upon arrival, Yao's protocol immediately attracted significant attention from the cryptographic community. The protocol requires representing the underlying function as a Boolean circuit. Therefore, the non-trivial challenge of utilizing GC is to generate the Boolean circuit such that its secure evaluation requires the minimum inter-party computation and communication.

The challenge of the GC circuit optimization is partially addressed by TinyGarble [19]. 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, 10, 11, 17]. 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, 9, 11, 12, 17]. 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. [12] 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). In a garbled processor [20, 22], 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 [19]. 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.

An earlier work by Wang et al. [22] 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*

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317777

pruning of the processor circuit instead of *gate-level* optimization. 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 similar approaches like *constant propagation* and *dead gate elimination* [2, 12, 15]. However, SkipGate is superior to these solutions since it operates at gate-level and does not require flattening the circuit; it dynamically detects and removes gates that can be skipped from the garbling. Moreover, the static circuit simplification method [16] 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.

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.
- 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 provide extensive experimental results and show that ARM2GC is 156 times more efficient compared to prior garbled processors [20, 22]. ARM2GC also outperforms the state-of-the-art high-level GC compilers [2, 12] in terms of communication while utilizing unmodified programming languages and compilers.

2 PRELIMINARIES

Security Model. Consistent with the earlier relevant literature [12, 17, 22], we assume an *honest-but-curious* Security model where the parties follow the agreed upon protocol but may attempt to learn about the other parties' input from the information at hand [1].

Oblivious Transfer. Oblivious Transfer (OT) [13] 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.

Garbled Circuit. Yao's Garbled Circuit protocol [24] 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. The function f should be 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* corresponding to 0 and 1 Boolean values. 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 OT 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].

Our framework supports all major GC optimizations. The *free-XOR* [8] optimization removes the communication cost for XOR gates. The *Row Reduction* [14] decreases the communication cost of the AND gates by 25%. The *Half-Gate* method [25] utilizes both free-XOR and row reduction and reduces the cost of AND gates by an additional 25%. TinyGarble enables garbling *Sequential Circuits* [19].

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.

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).

Gate Categories. SkipGate 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. 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

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 $\rightarrow (X_A^0, X_A^1, X_B^0, X_B^1)$
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 $\rightarrow$ garbled tables
13:     send garbled tables
14:     copy flip flops labels
15: end for
16: send output labels $\rightarrow X_C$
17: compute output value based on output labels $(X_C^0, X_C^1)$ and
    received labels $(X_C) \rightarrow c$
```

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 eliminated.

Algorithms. Algorithms 1 and 1 show the SkipGate algorithm for Alice and Bob sides, respectively. The input to these algorithms is the sequential description of the function $c = f(a, b, p)$ where c is the output, a is Alice's input, b is Bob's input, and p is the public input. 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. Alice (respectively Bob) garbles (evaluates) the sequential circuit for cc clock cycles (Lines 7-15 of Algorithm 1 and Lines 6-14 of Algorithm 2). 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 based on the approach presented in [19]. 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 (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).

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 $\rightarrow X_A$
3:  receive Bob labels $\rightarrow X_B$ 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 $\rightarrow X_C$
16: send output labels $(X_C)$
```

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 0 in
                recursive_reduction()
12:             recursive_reduction(g)
13:         end if
14:     end if
15: end for
```

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[1]. In SkipGate, `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`, it might reach zero and subsequently call the gates who are providing the input to this gate and so on (see

[1]*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: 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
```

Fig 1). 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). 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.

Algorithm 4 shows the Phase 2 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. 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 where he evaluates the gates in Category iii and iv. In Line 18 of Algorithm 5, Bob generates and assigns new unique labels for gates that were filtered by Alice. 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

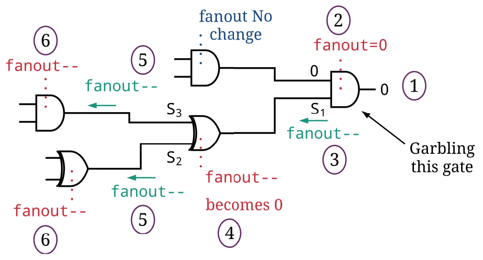

Figure 1: Recursive reduction of `label_fanout`.

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
```

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.

Algorithm 6 illustrates the pseudo-code for the recursive fanout reduction. It receives the circuit and a gate. 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 corresponding input(s). This is illustrated by an example in Fig 1.

Security of SkipGate. 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`. 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. 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 [24].

4 ARM2GC FRAMEWORK

In this section, we present ARM2GC, a GC framework based on a garbled ARM processor and the SkipGate algorithm. The framework enables the development of privacy-preserving applications using high-level languages while keeping the garbling cost as low as the best optimized garbled circuits.

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). Fig 2 shows the overview of the framework. The SFE program is compiled using an *unmodified* 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.

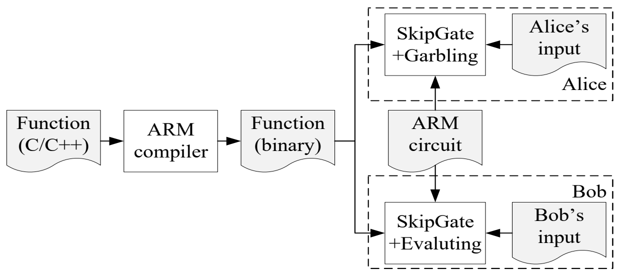

Figure 2: Overview of the ARM2GC framework.

The framework has five memory elements to store: Alice's inputs, Bob's inputs, output, stack, and instructions. The instruction memory is initialized with the compiled binary code (the public input p). Alice's and Bob's memories are initialized with the labels corresponding to their private inputs a and b, respectively. The stack, output, pipeline registers, and the register file are initialized to zero. The ARM circuit is garbled following the sequential garbling process [19] for a pre-specified number of clock cycles.

In this work, we choose ARM as our garbled processor which is a more ubiquitous and sophisticated processor compared to MIPS [19, 20, 22]. 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: conditional execution in ARM allows each instruction to be executed only if a specific condition is satisfied.

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. 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.

Synthesis of the ARM processor results in a large netlist (≈ 5 times compared to the MIPS processor). Thus, using ARM instead of MIPS in the earlier garbled processor approaches [20, 22] 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.

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 not secret, 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. 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 [20, 22].

Oblivious RAM. One natural suggestion is to employ Oblivious RAM (ORAM) that enables oblivious access to memories in the GC protocol with sub-linear cost [23, 26]. Our framework supports utilization of ORAMs for data memory. However, for registers, we use linear scan using MUXs and flip-flops because: (i) 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. (ii) Current state-of-the-art ORAM constructions [5, 21, 26] start outperforming the linear scan 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.

5 EXPERIMENTAL RESULTS

We use Synopsis Design Compiler (DC) H-2013.03-SP4 [3] along with TinyGarble [19] 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 [18]. The benchmark functions for ARM2GC are implemented in C and compiled using GNU `gcc-arm-linux-gnueabi`.

5.1 Effect of SkipGate on ARM2GC

Table 1 shows the cost of garbling an ARM processor for the benchmark functions using conventional GC compared to GC with the SkipGate algorithm. Since the instructions are known to both parties, SkipGate omits a significant number of non-XOR gates. The ARM circuit has 126,755 non-XOR gates and for computing a function, for example, Hamming 160, it takes 1,909 clock cycles. It

Table 1: Improvement by the SkipGate on ARM2GC.

Function	# of Garbled Non-XOR		Improv.
	w/o SkipGate	w/ SkipGate	(1000X)
Sum 1024	76,483,260	1,023	75
Compare 16,384	1,047,095,280	16,384	64
Hamming 512	863,559,216	1,012	853
Mult 32	4,199,448	993	4
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
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

means with the conventional GC protocol, garbling/evaluation of $1,909 \times 126,755 = 241,975,295$ non-XORs is required. SkipGate reduces the ARM circuit into a smaller circuit with only 247 non-XORs. Besides the benchmark functions, Table 1 also shows the evaluations on a number of practical problems.

5.2 Comparison of ARM2GC with prior-art

Garbled MIPS [22]. Even though the approach of ARM2GC is similar to the garbled MIPS presented in [22], it outperforms that work by a long margin. For example, to compute the Hamming distance between 32 32-bit integers [2], [22] needs 481K garbled gates, whereas ARM2GC needs only 3073- and improvement by 156×.

High-level GC compilers. To demonstrate the effectiveness of this work, we compare the cost of garbling for the benchmark functions with the most efficient high-level GC frameworks at present: Frigate [12] and CBMC-GC [2] in Table 2. In all cases, ARM2GC is either equal or better than the present frameworks in terms of the garbling cost. It shows significant improvements in Hamming distance and AES, 44% and 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 [12].

Table 2: Comparison of the number of garbled non-XOR gates of ARM2GC with the best prior high-level GC framework. The improvement is shown w.r.t. the best of these two.

Function	Number of non-XORs			Improv.
	CBMC-GC [6]	Frigate [12]	ARM2GC	
Sum 1024	-	**1,025**	1,023	0.20%
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%
AES 128	-	**10,383**	6,400	38.36%
a = a op† a	**0**	**0**	0	0.00%

†op represents any Boolean operation (+, &, ⊕, *etc.*)

6 CONCLUSION

This paper introduces the novel SkipGate algorithm for Yao's GC protocol. The algorithm dynamically omits the communication cost for the gates whose outputs are independent of the private data and 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 the benchmark functions show that the ARM2GC framework achieves 156× performance improvement over the best prior art.

[2]as in [22], different from more common Hamming distance between binary inputs

ACKNOWLEDGMENTS

This work is supported in part by NSF/SRC (1619261/2016-TS-2690), ONR N00014-17-1-2500, and MURI (FA9550-14-1-0351) grants.

REFERENCES

[1] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, 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*, 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] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*. Springer, 2008.
[9] 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.
[10] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P*. IEEE, 2015.
[11] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay-secure two-party computation system. In *Security*. USENIX, 2004.
[12] Benjamin Mood, Debayan 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.
[13] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. In *Journal of Cryptology*. Springer, 2005.
[14] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *CEC*. ACM, 1999.
[15] 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 89–106. Springer, 2009.
[16] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *ASIACRYPT*. Springer, 2009.
[17] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. WYSTERIA: A programming language for generic, mixed-mode multiparty computations. In *S&P*. IEEE, 2014.
[18] Conor Santifort. Amber ARM-compatible core. *OpenCores.org*, 2010.
[19] 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.
[20] 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.
[21] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS*. ACM, 2015.
[22] 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 99–117. Springer, 2016.
[23] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. SCORam: oblivious ram for secure computation. In *CCS*. ACM, 2014.
[24] A. Yao. How to generate and exchange secrets. In *FOCS*. IEEE, 1986.
[25] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*. Springer, 2015.
[26] 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.