RICE UNIVERSITY

TinyGarble: Efficient, Scalable, and Versatile Privacy-Preserving Computation Through Sequential Garbled Circuit

by

Ebrahim M. Songhori

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

[Signatures]

Farinaz Koushanfar, Thesis Director
Professor of Electrical and Computer Engineering
University of California, San Diego, CA

Joseph R. Cavallaro, Chair
Professor of Electrical and Computer Engineering and Computer Science
Rice University, Houston, TX

Aydin Babakhani
Assistant Professor of Electrical and Computer Engineering
Rice University, Houston, TX

Dan Wallach
Professor of Computer Science and of Electrical and Computer Engineering
Rice University, Houston, TX

Houston, Texas
April, 2017
ABSTRACT

TinyGarble: Efficient, Scalable, and Versatile Privacy-Preserving Computation Through Sequential Garbled Circuit

by

Ebrahim M. Songhori

Privacy-preserving computation is a standing challenge central to several modern-world applications which require computing on sensitive data. Secure Function Evaluation (SFE) refers to provably secure techniques aiming to address this problem by enabling multiple parties to compute an arbitrary function jointly on their private inputs. The most promising two-party SFE method is called the Garbled Circuit (GC) protocol introduced by Andrew Yao. The protocol relays on representing the function as a Boolean circuit and encrypting/communicating at the logic gate level. Despite several significant improvements in GC, efficiency, scalability and ease-of-use of the available methods are limited by the naive circuit representation as a directed acyclic graph, ad-hoc logic optimizations, and custom compilers.

In this thesis, we proposed a holistic solution to enhance the efficiency, scalability, and simplicity of the GC protocol. Our approach has three main pillars to address these key challenges: GC synthesis, sequential GC, and garbled processor. The GC synthesis is a novel automated methodology based on logic synthesis techniques for generating optimized Boolean circuits for the GC protocol. Using sequential GC, we achieve an unprecedented level of compactness and scalability using sequential circuit descriptions. We combine GC synthesis and sequential GC in an open-source frame-
work called TinyGarble. The preliminary implementation of benchmark functions using TinyGarble demonstrates a high degree of memory-footprint compactness as well as improvement in overall efficiency compared to results of existing tools.

Our sequential description also enables us, for the first time, to design and realize a garbled processor to reduce the problem of private function evaluation to a conventional SFE problem. In addition, the garbled processor allows users to develop SFE applications in high-level languages (e.g., C) and eliminates the need for Boolean circuit generation. We present ARM2GC, a garbled processor framework based on TinyGarble and the ARM processor. It allows users to develop GC applications using high-level programming languages with comparable efficiency to the best previous results. The primary enabler to make this construction practical and efficient is the introduction of SkipGate, a new algorithm that omits the communication cost of a Boolean gate when its output is independent of the private data. Benchmark evaluations demonstrate efficiency and usability of ARM2GC compared with the prior art in high-level GC compilation.
Acknowledgements

The completion of this thesis would not have been possible without the contribution of several individuals who in one way or another have provided their invaluable guidance and assistance.

My deepest gratitude to my advisor Professor Farinaz Koushanfar for her invaluable support, encouragement, and advice. Her dedication, fondness, and motivation towards doing novel research, as well as exceptional care for her students have taught me valuable lessons and greatly influenced my life at academic and especially personal levels.

I would also like to thank Professors Joseph R. Cavallaro, Aydin Babakhani, and Dan Wallach for serving as my committee members. I also want to thank them for their brilliant comments and suggestions on the thesis. Special thanks to Professors Ahmad-Reza Sadeghi and Thomas Schneider of TU Darmstadt for their collaboration with our lab at Rice and later UCSD and their great academic and scientific supports.

I would also like to thank my collages: Siam U. Hussain, M. Sadegh Riazi, Ghada Dessouky, and Shaza Zeitouni for their endless help; this work would not have been possible without your assistance, seriously. Thank you!

Special thanks to my family. Words cannot express how grateful I am to my mother, father, and two brothers for all of the sacrifices that you have made on my behalf. In the end, I would like express appreciation to my beloved fiancée, Zeinab, who was always my support in the moments when there was no one to answer my queries.
## Contents

Abstract .............................................. ii
List of Illustrations ................................. x
List of Tables ......................................... xiv

1 Introduction ........................................ 1
  1.1 Challenges ....................................... 2
    1.1.1 Efficiency .................................... 2
    1.1.2 Scalability .................................... 3
    1.1.3 Ease-of-use .................................... 4
  1.2 Our Approach ..................................... 4
    1.2.1 Garbled Circuit Synthesis ..................... 5
    1.2.2 Sequential Garbled Circuit ..................... 6
    1.2.3 Garbled Processor ............................. 7
    1.2.4 Garbled Processor for PF-SFE ................. 7
    1.2.5 Garbled Processor for SFE ..................... 8
  1.3 Global Flow ..................................... 10
    1.3.1 TinyGarble .................................... 10
    1.3.2 ARM2GC ...................................... 11
  1.4 Contributions ................................... 14
  1.5 Organization ................................... 15

2 Preliminaries and Background ..................... 16
  2.1 Secure Multi-party Computation ................. 16
    2.1.1 Secure Function Evaluation ................... 17
2.2 Two-Party Secure Function Evaluation ........................................ 18
  2.2.1 Adversary Model .............................................................. 19
  2.2.2 Oblivious Transfer .......................................................... 19
  2.2.3 Yao’s Garbled Circuit Protocol ......................................... 19
  2.2.4 Recent Improvements on the GC Protocol ............................ 20
2.3 Boolean Circuits ........................................................................ 22
  2.3.1 Combinational Boolean Circuits ......................................... 22
  2.3.2 Sequential Boolean Circuits ................................................. 23

3 Garbled Circuit Synthesis .......................................................... 25
  3.1 Background on Logic Synthesis ............................................ 25
  3.2 TinyGarble GC Synthesis ...................................................... 27
  3.3 Synthesis Flow ....................................................................... 27
  3.4 Synthesis Library .................................................................... 29
  3.5 Technology Library ............................................................... 30
  3.6 Offline Circuit Synthesis ....................................................... 31
  3.7 Limitation ............................................................................... 32

4 Sequential Garbled Circuit .......................................................... 33
  4.1 Sequential Circuits ................................................................... 33
  4.2 Synthesis Sequential Circuit ................................................... 37
  4.3 Overhead of Sequential Circuit ............................................... 38

5 SkipGate: Reducing Sequential Overhead ..................................... 41
  5.1 Notations ............................................................................... 42
  5.2 Motivational Example ........................................................... 42
  5.3 Gate Categories ...................................................................... 43
  5.4 Algorithm .............................................................................. 44
    5.4.1 Identification of Identical and Inverted Labels .................... 54
### 5.5 Computational Complexity

55

### 5.6 Correctness Proof

56

### 5.7 Security Proof

57

### 6 Garbled Processor

58

#### 6.1 Private Function Evaluation

59

##### 6.1.1 Arithmetic Logic Unit

59

##### 6.1.2 Memory

61

#### 6.2 Garbled Processor for PF-SFE

62

##### 6.2.1 Global Flow

62

##### 6.2.2 MIPS

63

#### 6.3 Garbled Processor for SFE

65

##### 6.3.1 Garbled Processor for Public Functions

66

##### 6.3.2 Garbled Processor for Semi-Private Functions

66

##### 6.3.3 Garbled Processor for Private Functions

67

##### 6.3.4 Hardware Implementation of Garbled Processor

67

#### 6.4 ARM2GC: Garbled ARM for SFE

70

##### 6.4.1 Global Flow

71

##### 6.4.2 ARM as a Garbled Processor

73

##### 6.4.3 How SkipGate Helps

75

##### 6.4.4 Why not Sub-linear ORAM?

75

### 7 Evaluation

79

#### 7.1 TinyGarble GC Synthesis

79

##### 7.1.1 Experimental Setup

80

##### 7.1.2 Performance Metrics

81

##### 7.1.3 Benchmark Functions

82

##### 7.1.4 Combinational Garbled Circuit

84

##### 7.1.5 Sequential Garbled Circuit

86
7.1.6 Effect of Folding on Garbling Time 89
7.1.7 High-level Synthesis Tools 91

7.2 Evaluation of MIPS for PF-SFE 92
7.2.1 MIPS Implementation 92
7.2.2 Benchmark Function 93

7.3 Evaluation of MIPS for SFE 93
7.3.1 Experimental Setup 93
7.3.2 Benchmark Function 95
7.3.3 Synthesis of MIPS for Various ISAs 95
7.3.4 Performance of the Hardware GC Evaluator 99

7.4 Evaluation of ARM2GC 103
7.4.1 Experimental Setup 104
7.4.2 Effect of SkipGate on Sequential GC 104
7.4.3 ARM2GC vs. Logic Synthesis 105
7.4.4 ARM2GC vs. GC Frameworks Supporting High-level Languages 108
7.4.5 Effect of SkipGate on ARM 108
7.4.6 Complex Functions 112

8 Related Work 115
8.1 Compiler for Garbled Circuits 115
8.2 Libraries for Garbled Circuits 118
8.3 Garbled Processor 119
8.4 Logic Synthesis for Other SFE Circuit-based Protocols 120
8.5 GC Implementations with Hardware Accelerators 121
8.6 GC Implementations on Embedded Devices 122
8.7 Securing GC against Malicious Adversary 122
8.7.1 Discussion 124

9 Conclusion 126
Illustrations

1.1 The global flow of the TinyGarble framework. The framework consists of GC synthesis flow and GC engine. The GC synthesis flow generates the optimized Boolean circuit given a function description. The GC engine allows two parties (Alice and Bob) to compute the function securely according to Yao’s GC protocol. 12

1.2 The global flow of the ARM2GC framework. The framework consists of GC synthesis flow of TinyGarble and its GC engine with the SkipGate algorithm. The GC synthesis flow generates the circuit description of the ARM processor $f(\cdot, \cdot, \cdot)$. The function to be securely evaluated ($g(\cdot, \cdot)$) is compiled using an ARM cross-compiler. The GC engine allows two parties (Alice and Bob) to securely compute the ARM circuit $f(a, b, p)$ where $p$ is the binary code of $g(\cdot, \cdot)$. In the end, they will receive the output of $f(a, b, p) = g(a, b)$ 13

2.1 Emulation of a trusted third-party in secure multi-party computation (schematic based on [43]). 17

2.2 The real model for two-party Secure Function Evaluation (SFE). 18

2.3 (a) Combinational circuit where outputs are functions of only inputs. (Combinational Logic (CL) represents Combinational Logic.) (b) Sequential circuit where outputs are functions of inputs and present states. 23

2.4 Schematic of a D-FF with initial value. 24
3.1 TinyGarble GC synthesis flow. The inputs can be either a C/C++ program (translatable to HDL via a standard HLS tool) or a direct HDL description. The output is a scheduled circuit description ready to be garbled/evaluated.

3.2 Sample files at the different steps of TinyGarble GC synthesis flow for Hamming distance function.

4.1 The combinational and sequential design of a 4-bit Adder. (a) The HA circuit. (b) The FA circuit. (c) The combinational 4-bit Adder using one HA and three FAs. (d) The sequential 4-bit Adder using one FA.

4.2 Functionally equivalent unrolled sequential circuit corresponding to Figure 2.3b.

4.3 The combinational and sequential representations of a function with an asymmetric structure. The sequential implementation has to be evaluated for six cycles and eventually needs more gates to be garbled/evaluated compared to the combinational one.

5.1 Motivational example: two sub-circuits connecting to a MUX with a public select signal.

5.2 Four examples of replacing gates in Phase 1 by zero, one, wire, or inverter. We decrease the label \texttt{fanout} of the unnecessary gate accordingly. The top-left example is in Category i, and the rest are in Category ii.

5.3 Four examples of replacing and computing gates in Phase 2. We decrease the label \texttt{fanout} of the unnecessary gate accordingly. The top-right example is in Category iv, and the rest are in Category iii.
5.4 Recursive reduction of label_fanout to skip unnecessary gates in Phase 1. ................................. 54
5.5 SkipGate macro effect on the example in Figure 5.1. Only sub-circuit $f_1$ is garbled/evaluated. ................... 55

6.1 Lite MIPS architecture. Alice’s and Bob’s inputs and the output are shown. ...................................... 64
6.2 Architecture of our hardware GC Evaluator. ......................... 69
6.3 Overview of the ARM2GC framework. ................................. 72
6.4 An example code that shows how conditional execution in ARM can reduce the code size and make the program flow predictable. ............... 74
6.5 In a case when the compiler fails 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. .... 76
6.6 Bit-precise details of how source register is fetched while executing two instructions add $1$, $2$, $3$ and sub $5$, $6$, $7$ at the same time after a branch on a secret condition value $S$. ..................... 77

7.1 Garbling 32,768-bit Sum function. The CPU time in term of the number of cycles and the approximate memory footprint in KBytes (y-axis) versus cc (x-axis) are shown. ......................... 90
7.2 Hamming distance assembly code. ............................................... 114

A.1 The combinational circuit for 1-NN. It consists of $n$ Hamming distance and $n - 1$ MIN modules. ...................... 136
A.2 The sequential circuit for 1-NN. It consists of one Hamming distance and one MIN module. For a dataset of size $n$, the circuit is required to be garbled/evaluated $n$ times. ......................... 138
A.3 The sequential circuit for $k$-NN. It consists of one Hamming distance, $k$ MIN, and $k-1$ MAX modules. It requires to be evaluated $n$ times where $n$ is the size of the dataset $S$. 

139
### Tables

7.1 Comparison of TinyGarble combinational circuits with PCF. In the case of AES 128, we compare the result with Frigate [27].

7.2 Comparison of TinyGarble sequential circuits with PCF and TinyGarble combinational circuits. In the case of AES 128, we compare the result with Frigate [27].

7.3 Comparison of performance of the circuits generated using C input to HLS and a direct Verilog input to the logic synthesis tool.

7.4 The number of total gates and non-XOR gates in the MIPS implementation. The global optimization of TinyGarble reduces the overall number of gates compared to that of the sum of individual modules.

7.5 The number of required cycles to compute the benchmark functions.

7.6 Synthesis results of application-specific ISA.

7.7 Synthesis results of restricted and full ISA.

7.8 Resource allocation and utilization of our hardware GC evaluator on a Xilinx Virtex-7 FPGA.

7.9 The performance of evaluating MIPS for application-specific ISAs with different memory sizes at 100MHz clock frequency on FPGA.

7.10 The performance of evaluating MIPS for restricted and full ISAs with different memory sizes at 100MHz clock frequency on FPGA.

7.11 Comparing our GC evaluator implementation with other works’ estimation for MIPS with 64-word memory.
7.12 The SkipGate algorithm improvement on sequential circuits generated by TinyGarble (TG) synthesis flow (Section 7.1.5). 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. 106
7.13 The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC to TinyGarble’s GC synthesis (Chapter 3) . . . 107
7.14 The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC with previous high-level custom compiler methods and TinyGarble HLS approach . . . . . . . . . . . . . . . . . . . . 109
7.15 The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC with the previous garbled processor approaches. 110
7.16 The SkipGate algorithm improvement on the ARM sequential circuit. 111
7.17 The SkipGate algorithm improvement on the ARM sequential circuit for the complex functions . . . . . . . . . . . . . . . . . . . . . . . . . . 112

A.1 Circuit size and timing evaluation for 1-NN search . . . . . . . . 141
A.2 Circuit size and timing evaluation for k-NN search . . . . . . . . 142

C.1 The result of integer division and remainder functions . . . . . . 154
C.2 The result of floating-point functions . . . . . . . . . . . . . . . . 155

D.1 Comparison of circuit generation performance between the commercial Synopsys DC and Yosys+ABC open source logic synthesizer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
To my fiancée, Zeinab.
Chapter 1

Introduction

Secure Function Evaluation (SFE) allows two or more mutually suspicious parties to evaluate an arbitrary function on their private data. The parties learn the output of the function without revealing any information about their private data. An important special case of SFE is its two-party version. Formally in two-party SFE, Alice and Bob wish to find the output of $f(a,b)$ where $a$ is Alice’s private input, $b$ is Bob’s, and $f(\cdot,\cdot)$ is an arbitrary function. By far the most promising and efficient solution for two-party SFE is Garbled Circuit (GC) protocol proposed by Andrew C. Yao [1]. Yao’s GC protocol works on a Boolean circuit representation of $f(\cdot,\cdot)$. The Boolean circuit should consist of two-input logic gates such as AND, OR, XOR. Alice encrypts (garbles) the gates in the circuit, sends them to Bob, and then Bob decrypts (evaluates) them to learn the output. The intermediate values in the circuit are encrypted such that the parties’ inputs remain private. Secure evaluating of a Boolean circuit can also be generalized to multi-party SFE [2, 3].

A host of privacy-preserving and security-critical applications can directly benefit from a practical and efficient realization of SFE, including but not limited to: biometrics matching, face recognition, image/data classification, electronic auctions and voting, remote diagnosis, secure search, and stable matching [4, 5, 6, 7, 8, 9, 10, 11].
1.1 Challenges

While the GC protocol was considered to be prohibitively expensive and practically infeasible a decade ago, today we are witnessing a surge of theoretical, algorithmic, and tool developments. These studies have significantly improved the efficiency and practicality of this protocol, see [12, 13, 14, 15, 16, 17, 18, 19]. However, three main challenges have yet to be addressed for a wide adoption of the GC protocol in the digital world: efficiency, scalability, and ease-of-use.

1.1.1 Efficiency

Although the GC protocol has a constant complexity for communication rounds that makes it superior to other SFE protocols, GC is still considered a communication-intensive protocol. That means the cost of transferring encrypted data between the parties far exceeds the cost of GC’s encryption/decryption (computation) [20]. In GC, the communication stems mainly from transferring encrypted (garbled) gates of the Boolean circuit from Alice to Bob. To improve the overall efficiency, we have to decrease the communication through reducing either the communication cost of a gate or the total number of gates in a function’s Boolean circuit.

We classified the research on the efficiency of the GC protocol into two broad categories: (i) Optimizations of cryptographic constructs and protocols such as [13, 14, 21, 16, 22, 17]; out of which, the two most prominent are Free XOR [13] and Half Gate [17]. Free XOR proposed a cryptographic scheme in which XOR gates do not incur communication cost. Subsequently, the Half Gate technique reduces the required communication for a non-free (non-XOR) gate to the minimum possible cost using symmetric encryption. (Further communication reduction may only be achieved using more costly asymmetric encryptions.) (ii) Engineering techniques including but
not limited to [23, 15, 24, 25, 26, 27]. The primary objective of these techniques was to create a framework that, given a function, generates an optimized Boolean circuit that has a minimum number of non-free gates. They either employ library-based or compiler-based approaches.

The library-based methods devolve on building a library for a general purpose programming language such as Java along with routines for emitting the optimized sub-circuits, e.g., [15, 28, 24]. For better usability, their libraries typically include the manually optimized sub-circuits of frequently used operations such as adder and multiplier. One of the drawbacks of the library-based approaches is that they do not perform global circuit optimization.

The compiler-based approaches provide users with a new domain-specific programming language to describe the function. In addition to this, a new custom compiler is introduced to translate the described function into a Boolean circuit, e.g., [12, 29, 25, 26].

1.1.2 Scalability

For executing the GC protocol, the parties have to store the encrypted input and intermediate values in their memory. The size of this data increases linearly with the number of gates in the Boolean circuit. Both library-based and compiler-based approaches suffer from complex memory management when the number of gates is large, thereby affecting their performance and scalability [24, 25]. A few frameworks have been recently proposed to address the scalability problem of previous methods such as [28, 30, 29, 25]. Unfortunately, they do not provide a generic solution that addresses both efficiency and scalability at once.
1.1.3 Ease-of-use

As mentioned earlier, GC receives the function in a Boolean circuit format. Usually, users find it cumbersome to describe a relatively simple function by connecting Boolean gates to make the circuit. The library-based frameworks provide libraries of optimized sub-circuits to facilitate the circuit generation, so users can manually connect them together to make the circuit. However, for creating an optimized circuit, users need to have a thorough understanding of digital design and the low-level structure of the circuit.

To further simplify developing applications for SFE, some compiler-based frameworks provide users with high-level languages to develop the function [30, 29, 25, 19]. Recently, a few frameworks support standard high-level languages like C as well [31, 26, 18, 27]; however, they do not exhibit the efficiency comparable to competitor approaches.

Despite numerous efforts in improving efficiency, scalability, and simplicity of the GC protocol, none of the existing methods addressed these three challenges at once.

1.2 Our Approach

We proposed a holistic solution to enhance the efficiency, scalability, and simplicity of the GC protocol in this thesis. Our approach has three main pillars to address those major challenges: Garbled Circuit synthesis, sequential Garbled Circuit, and garbled processor.
1.2.1 Garbled Circuit Synthesis

Our Garbled Circuit (GC) synthesis solution simply views the optimized circuit generation for GC as an atypical logic synthesis task that, if properly defined, can still be addressed by conventional hardware synthesis tools. By modeling the circuit generation for Yao’s protocol as a hardware synthesis problem, we naturally benefit from the elegant algorithms and powerful techniques already incorporated in existing logic synthesis solutions, see, [32, 33, 34, 35]. This view provides a radically different perspective on this important problem in contrast to the earlier work in this area. Previous work attempted to generate circuits by building new libraries for general purpose languages such as Java [15, 28], custom compilers such as [25, 26], or introduction of new programming languages such as [12, 36].

We introduce new techniques for minimizing the number of non-XOR gates which directly results in reducing communication cost for the GC protocol. We do so by integrating the cost function in the new custom libraries that we design and use within our logic synthesis flow. This way, we can gain up to 80% improvement in the number of non-XOR gates for benchmark circuits compared to PCF [25]. Our methodology is automated, i.e., we can achieve the savings for many functions synthesized by this method, regardless of their sophistication.

Our logic synthesis method can also apply to other circuit-based SFE protocols with different cost functions, e.g., Goldreich-Micali-Wigderson (GMW) [2], and Beaver-Micali-Rogoway (BMR) [37]. For example, Demmler et al. (2015) show how our GC synthesis approach can adopt and apply for the GMW protocol where the cost depends on both the depth and size of the circuit [38]. In this thesis, we will focus only on the GC protocol, and we leave the adaptation of the logic synthesis approaches for other circuit-based protocols to the future work.
1.2.2 Sequential Garbled Circuit

To address the scalability, we introduce sequential Garbled Circuit (GC) which differentiates our methodology from the previous work. Sequential GC allows expressing the function in a very compact format, namely as a sequential logic. The earlier work in this area mainly described the function in a format where the value of the output is determined entirely by the circuit input. This input/output relationship is expressed by a Directed Acyclic Graph (DAG) of logic gates also known as a combinational circuit in digital design jargon. The sequential circuit description, on the other hand, allows having feedback from the output to the input by adding the notion of a state (memory). At each sequential cycle, the current state of the circuit along with the input determine the output of the circuit. For each particular sequential cycle, a combinational circuit determines the relationship between the output and the input for a given state.

The only previous work, we are aware of, which implicitly hinted at the possibility of having a more compact representation is PCF [25]. It does so by embracing loops and unrolling them only at runtime. A sequential circuit, however, goes far beyond the loop embracing performed at the software level. Not only does our approach embrace the high-level loops, but it also enables the user to further compact the functions by folding the implementation up to its core elements. For example, using our method, the user can compress the 1024-bit addition function into only a 1-bit adder.

An notable advantage of our sequential representation is providing a new degree of freedom to the user to fold the functions to simpler computing elements. In other words, the user has the freedom to choose the number of sequential cycles needed for evaluation of the function—the size of the combinational logic path between the
states/inputs and the outputs. The user can manage the number of gates in the sequential circuit by varying the number of cycles. The memory footprint of the GC operation is directly related to the number of gates in the sequential circuit; at any moment during garbling/evaluation, only the information corresponding to the current cycle needs to be stored. Compact sequential circuits yield a small enough memory footprint that can fit mostly on a typical processor cache. This low memory usage allows us to avoid costly cache misses while accessing the wire labels during the GC protocol. Indeed, our sequential approach can enable the realization of SFE applications on embedded systems with limited memory. In Appendix A, we implement a scalable solution for private nearest neighbor search as an application of the sequential GC.

1.2.3 Garbled Processor

1.2.4 Garbled Processor for PF-SFE

The sequential representation enables, for the first time, implementation of a universal processor for private function evaluation where the function is known only to one party. We reduce Private-Function SFE (PF-SFE) to general SFE by secure evaluation of a general-purpose processor like MIPS. PF-SFE is useful for scenarios where the function is proprietary or classified, e.g., credit checking or private database queries. Formally, the processor \( f(\cdot, \cdot) \) accepts a binary code of Alice’ private function \( g_{\text{Alice}}(\cdot) \) along with Bob’s private input \( b_{\text{Bob}} \). The processor then computes the private function given the private input: \( o = f(b_{\text{Bob}}, g_{\text{Alice}}(\cdot)) = g_{\text{Alice}}(b_{\text{Bob}}) \). The private function \( g_{\text{Alice}}(\cdot) \) can be written in a high-level language and compiled using standard compiler into the binary code. Since a processor is inherently a sequential circuit, the garbled processor was infeasible to be realized with previous GC frame-
works.

1.2.5 Garbled Processor for SFE

The garbled processor approach, besides providing a scalable solution for PF-SFE, also makes developing SFE applications easier and more accessible to non-expert users. Garbled processors allow users to avoid representing the function as a Boolean circuit; users can describe the function in a high-level language and compile it into binary code using a standard compiler. This development flow is more similar to the conventional software development flow compared to that of the previous library- and compiler-based approaches.

Unfortunately, the cost of using a garbled processor for SFE where the function is not private is staggering larger than the one incurred by the conventional SFE methods. The reason behind the massive cost is that garbled processor always hides the function by garbling/evaluating the entire processor. Thus, using garbled processor incurs an unnecessary overhead for numerous applications in which a private function is not necessary. To reduce this overhead, we extend the garbled MIPS processor for SFE by relaxing the privacy of the function. We further improve garbled processor by proposing ARM2GC a framework based on the ARM processor that closes the gap between the cost of conventional SFE methods (e.g., GC synthesis) and that of garbled processors.

MIPS for SFE

We extend garbled MIPS to provide a generalized support for SFE of varying flavors of privacy, beyond PF-SFE, to allow for more relaxed privacy demands and hence an improved performance. More explicitly, the parties can evaluate a private,
semi-private or public function by revealing none, partial or all information about
the function respectively while still benefiting from the simplicity of programming
a processor. We provide a coarse-grain optimization in which both parties decide
first which subset of MIPS Instruction Set Architecture (ISA) they are willing to
use which determines the level of privacy. The looser privacy of the function results
in less number of non-XOR gates in the circuit of MIPS processor. The function
g(\cdot, \cdot) is described in a high-level language, e.g., C and compiled into a binary code
of MIPS. Next, the MIPS processor f(\cdot, \cdot, \cdot) is garbled/evaluated given the parties’
private inputs a and b and the compiled binary code of g(\cdot, \cdot) to compute the output
o = f(a, b, g(\cdot, \cdot)) = g(a, b).

**ARM for SFE**

We also introduce a methodology to perform 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 the gates whose outputs are independent
of the private data (and thus known to both parties) can either be computed without
communication or simply skipped. This observation gives birth to the novel SkipGate
algorithm. The algorithm wraps around the GC protocol to compute the gate outputs
that can be calculated without communication and to mark the redundant gates for
skipping*.

SkipGate is mostly effective for reducing the garbling cost of sequential circuits
containing known control paths. An example of such a circuit is the garbled processor
where the control path depends on the binary code of the function that is known to

---

*SkipGate avoids garbling redundant gates and is orthogonal to cryptographic methods such as
both parties. By utilizing this property, we develop a high-level GC framework called ARM2GC built upon the ARM ISA and the SkipGate algorithm. Users can develop the secure function in high-level languages, e.g., C and compile it using standard ARM cross-compilers. In contrast to the previous high-level compiler-based approaches which called for new ad-hoc verification techniques [36, 39, 19, 27], ARM2GC inherits the ARM’s available fully verified compilers. Thanks to SkipGate, ARM2GC incurs a garbling cost comparable to the GC synthesis approach while allowing users to develop SFE applications in a high-level language.

ARM2GC leverages ARM as the general purpose processor, instead of MIPS because of ARM’s pervasiveness and, most importantly, conditional execution. 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.

1.3 Global Flow

1.3.1 TinyGarble

We combined our GC synthesis and sequential GC approaches into a framework called TinyGarble. TinyGarble accepts inputs in two different formats: a standard Hardware Description Language (HDL), or a higher-level language as long as it is compatible with the existing High-Level Synthesis (HLS) tools (e.g., C for SPARK [40] and Xilinx Vivado [41], or Python for PandA [42], that converts the high-level language to an HDL). Besides user’s manual optimization, TinyGarble performs various optimizations through standard logic synthesis tools to generate an optimized netlist, i.e., list of gates. TinyGarble then translates the netlist into the Simple Se-
sequential Circuit Description (SSCD) format used in our implementation of the GC protocol.

Figure 1.1 illustrates the global flow of the TinyGarble framework. The framework consists of two main parts (i) GC synthesis flow and (ii) GC engine. The GC synthesis flow receives a sequential description of the function \( f(\cdot, \cdot) \) in HDL and generates a Boolean circuit that can be efficiently evaluated by the GC protocol. The GC synthesis can create combinational circuits as well as sequential ones.

The GC engine is an implementation of the GC protocol allows two parties, Alice and Bob, to securely evaluate the function \( f(\cdot, \cdot) \) given the function description generated by the GC synthesis flow and parties’ private inputs (\( a \) and \( b \)). By the end of the protocol, both parties learn the output \( o = f(a, b) \) without learning anything about the other parties’ input. The GC engine supports garbling/evaluating of sequential circuits along with the most recent GC’s cryptographic optimizations [13, 16, 17]. We discuss TinyGarble GC engine in more details in Appendix B.

1.3.2 ARM2GC

The ARM2GC framework relies on the TinyGarble framework. The circuit of ARM processor \( f(\cdot, \cdot, \cdot) \) is synthesized using the TinyGarble GC synthesis flow. The user describes the function \( g(\cdot, \cdot) \) in a high-level language (e.g., C) and compiles it using an ARM cross-compiler (e.g., gcc-arm) to produce the binary code. Then, the TinyGarble GC engine that supports SkipGate is used to securely evaluate the circuit of the ARM processor given the publicly known binary code of the function \( p = g(\cdot, \cdot) \), and parties inputs (\( a \) and \( b \)). By the end of the protocol, both parties receive the output of \( o = f(a, b, p) = g(a, b) \) without learning anything about the other parties’ input.
Figure 1.1: The global flow of the TinyGarble framework. The framework consists of GC synthesis flow and GC engine. The GC synthesis flow generates the optimized Boolean circuit given a function description. The GC engine allows two parties (Alice and Bob) to compute the function securely according to Yao’s GC protocol.
Figure 1.2: The global flow of the ARM2GC framework. The framework consists of GC synthesis flow of TinyGarble and its GC engine with the SkipGate algorithm. The GC synthesis flow generates the circuit description of the ARM processor \( f(\cdot, \cdot, \cdot) \). The function to be securely evaluated \( g(\cdot, \cdot) \) is compiled using an ARM cross-compiler. The GC engine allows two parties (Alice and Bob) to securely compute the ARM circuit \( f(a, b, p) \) where \( p \) is the binary code of \( g(\cdot, \cdot) \). In the end, they will receive the output of \( f(a, b, p) = g(a, b) \)
1.4 Contributions

In brief, the main contributions of this thesis are as follows:

- We address the efficiency challenge of the GC protocol by introducing GC synthesis that adapts the traditional logic synthesis techniques to compile a function into an optimize Boolean circuit.

- We address the scalability challenge of the GC protocol by introducing sequential GC that allows us to achieve an unprecedented compactness in function representation and memory footprint at the same time benefiting from GC synthesis optimization.

- We implement the first scalable emulation of a universal processor for private function evaluation (PF-SFE) where the number of instruction invocations is not limited by the memory required for garbling. The sequential GC was the primary enabler of this unique design. Our design is a secure general-purpose processor based on MIPS that receives the private function from one party and the data from the other.

- We propose a MIPS-based garbled processor solution for SFE that allows leveraging the trade-off between privacy and performance: application-specific ISA for SFE, restricted ISA for semi-private SFE, and full ISA for PF-SFE.

- We introduce the novel SkipGate algorithm that avoids excessive garbling by utilizing the mutual knowledge between the two parties.

- We develop the ARM2GC framework based on the SkipGate algorithm and ARM processor. In this framework, users can efficiently develop SFE appli-
cations in a high-level language like C. It enables them to benefit from the available fully verified compilers of ARM.

1.5 Organization

The thesis is organized as follows. We review the preliminaries and background of the SFE, GC, and Boolean circuits in Chapter 2. We discuss GC synthesis and its limitation in Chapter 3. Sequential GC and its overhead is described in Chapter 4. We introduce SkipGate algorithm to remove the sequential overhead in Chapter 5. Next, we present garbled processors for PF-SFE, SFE and the ARM2GC framework in Chapter 6. Lastly, we review the related work in Chapter 8 and conclude the thesis in Chapter 9.
Chapter 2

Preliminaries and Background

In this chapter, we provide a short background about secure multi-party computation and the generalized problem of Secure Function Evaluation (SFE). Next, we introduce an important special case of two-party SFE problem and how Yao’s Garbled Circuit (GC) protocol can solve it. Then, we explain basics about combinational and sequential Boolean circuits that are used in the GC protocol to describe functions.

2.1 Secure Multi-party Computation

In a secure multi-party computation problem, a number of mutually suspicious parties wish to compute a joint function of their private attributes. For example, a group of people wants to find their average salary while keeping their salary secret. The parties have to use some distributed protocols to evaluate and learn the output of the function. Besides the information that parties can deduce from the output, no other information should leak about the private attributes through the computation.

If there was a third party that all the involving parties trust, they could send their attributes to her, and she computes the function and then returns the result to the parties. This model is called the ideal model for secure multi-party computation. Of course, such a trusted party is not available in real scenarios of distributed systems. Thus at best, the trusted third party can be emulated using a cryptographic protocol (the real model) [43]. Figure 2.1 illustrates the idea model with a trusted third party
Figure 2.1: Emulation of a trusted third-party in secure multi-party computation (schematic based on [43]).

and its emulation in the real model using a secure multi-party computation protocol.

2.1.1 Secure Function Evaluation

Secure Function Evaluation (SFE) is a generic formulation of secure multi-party computation in which parties can evaluate any arbitrary function on their attributes. Formally in SFE, there are $p$ parties that have access to secure communication channels between each other. Each party has an attribute denoted by $i_k$ where $k$ is party identifier. The parties are interested to collectively compute function $o = f(i_1, i_2, ..., i_p)$ where $f$ is an arbitrary function and $o$ is a tuple $(o_1, o_2, ..., o_p)$ such that $o_k$ belongs to the $k^{th}$ party. After the computation of $f$, the parties should gain no information about other parties’ attribute, other than the one they can deduce from the output.

One can also describe the function $f$ as a tuple of functions $f = (f_1, f_2, ..., f_p)$ where the output of $o_k = f_k(i_1, i_2, ..i_p)$ belongs to $k^{th}$ party. A common special case
is where \( f = f_1 = f_2 = \ldots = f_p \), meaning all parties receive the same output. For example in the case of the average of salaries, the result of the average function is evaluated and shared with all the participating parties.

### 2.2 Two-Party Secure Function Evaluation

Two-party SFE is an important special case of general secure multi-party computation that was introduced and solved by Andrew Yao in his seminal work [1]. In this problem, there are two parties: Alice and Bob; each has an attribute (input) \( a \) and \( b \) and they want to compute \((o_1, o_2) = (f_1(a, b), f_2(a, b))\) where \( o_1 \) belong to Alice and \( o_2 \) belongs to Bob and \( f_1 \) and \( f_2 \) are arbitrary polynomial-time computable functions. Figure 2.2 illustrates the real model for two-party SFE.

In the following, we explain how the problem of two-party SFE can be solved using Yao’s Garbled Circuit (GC) protocol. First, we introduce the adversary model supposed for the GC protocol. Next, we describe an important cryptographic primitive called Oblivious Transfer used in the GC protocol. Then, we explain Yao’s GC protocol and its recent improvements.
2.2.1 Adversary Model

We assume that all the parties are semi-honest which means that they follow the protocol exactly as specified without deviation or malicious behavior but they try to learn as much as possible about the other party’s input from the information they receive during the protocol. This model is often called Honest-But-Curious (HBC) adversary model in the literature and it is the basis for building a stronger security protocol. This model can be generalized to more advanced adversary models that are typically addressed by cut-and-chose methods through multiple runs of the basic HBC model [44, 45, 46].

2.2.2 Oblivious Transfer

Oblivious Transfer (OT) [47] is a cryptographic protocol based on asymmetric cryptography executed between two parties: Alice (sender) and Bob (receiver). Through OT, Bob selects and receives a number of messages from a set provided by Alice without revealing his selection to her. In a 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 $m_{1-b}$.

2.2.3 Yao’s Garbled Circuit Protocol

Yao’s Garbled Circuit protocol [1] allows two parties Alice, called the Garbler and Bob, called the Evaluator, to jointly compute a function $(o_{Alice}, o_{Bob}) = f(a, b)$ on their private attributes (inputs) $a$, provided by Alice and $b$, provided by Bob such that none of them reveal their inputs to each other. In the end, Alice learns $o_{Alice}$ and Bob learns $o_{Bob}$. The steps of Yao’s protocol are described below.
i. The function $f$ to be computed is represented as a Boolean circuit, called netlist, consisting of 2-input 1-output logic gates (a four entry truth table).

ii. For each wire $w$ in the netlist, Alice assigns two $k$-bit random strings, called labels, $X_w^0$ and $X_w^1$, respectively corresponding to 0 and 1 Boolean values. $k$ is the security parameter – typically $k = 128$ is sufficient [16].

iii. For each gate, she encrypts the output label with the corresponding input labels and creates a encrypted table with four rows. The table is then randomly rearranged and called the garbled table.

iv. She then sends the garbled tables along with the labels corresponding to her input values to Bob. Bob obtains the labels corresponding to his input values obliviously through 1-out-of-2 OT protocol.

v. Bob uses these input labels to decrypt the garbled tables gate by gate. The label for the output wire of one gate are the label(s) for the input wire(s) of the subsequent gate(s).

vi. In the end, Bob learns the labels for the final output wire and Alice has its mapping to 0 and 1. Alice sends the mapping for $o_{Bob}$ to Bob and Bob sends the labels for $o_{Alice}$ to her, so they can determine the actual value of outputs.

2.2.4 Recent Improvements on the GC Protocol

Throughout the last decade, the cryptographic scheme of the GC protocol has gone through numerous developments. We describe the most consequential of these improvements here.
**Row Reduction**

The Row Reduction technique reduces the size of a garbled table by 25%, i.e., only three rows of a garbled table are needed to transferred between the parties [9]. Here, instead of randomly generating a label for the output wire of a gate, the output label is produced as a function of the labels of the inputs. Alice generates the output label such that the first entry of the garbled table becomes all 0 and no longer needs to be sent.

**Free XOR**

The most significant optimization to GC so far is Free XOR proposed in [13]. In this optimization, for any wire $w$, Alice only generates the label $X^0_w$ and computes the label corresponding to 1 as $X^0_w \oplus (R \parallel 1)$ 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 $o$ can simply be computed as $X_o = X_a \oplus X_b$. Thus it does not need computation or transfer of the garbled tables and the XOR gate becomes free.

**Garbling with a Fixed-key Block Cipher**

This method allows to efficiently garble and evaluate non-XOR gates using fixed-key Advanced Encryption Standard (AES) symmetric encryption [16]. In this garbling scheme which is compatible with the Free XOR and Row Reduction techniques, the output label $X_o$ is encrypted with the input label $X_a$ and $X_b$ using the encryption function $E(X_a, X_b, T, X_o) = \pi(K) \oplus K \oplus X_o$, where $K = 2X_a \oplus 4X_b \oplus T$, $\pi$ is a fixed-key block-cipher (e.g., instantiated with AES), and $T$ is a unique-per-gate number (e.g., gate identifier) called *tweak*. A comprehensive proof of security for this method
is provided in [16].

**Half Gate**

Zahur et al. (2015) proposed the Half Gate technique that utilizes reduces the cost of non-XOR gates by an additional 25%, i.e., two rows of garbled tables are needed to transferred between parties[17]. They also prove that this two rows is the lowest cost possible with symmetric encryption, e.g., AES. Fewer than two row may only be possible using costly asymmetric encryption.

### 2.3 Boolean Circuits

In SFE, the parties are able to evaluate an arbitrary function on their private attributes. In a number of SFE protocols including Yao’s GC, Goldreich-Micali-Wigderson (GMW) [2], and Beaver-Micali-Rogoway (BMR) [37], the function has to be represented as a Boolean circuit. In this context, a Boolean circuit can be defined as a directed graph where nodes are 2-input Boolean gates (e.g., AND, OR, NOT, XOR, etc.), edges are wires connecting the gates, and their direction is from the output of a gate to the input of another gate.

#### 2.3.1 Combinational Boolean Circuits

The circuits that are used in GC and other circuit-based SFE protocols are acrylic Boolean circuits. In digital circuit theory, such a circuit is called *combinational circuit* and defined as a memory-less circuit in which outputs are functions only of inputs, see Figure 2.3a.
2.3.2 Sequential Boolean Circuits

Another class of circuits in digital circuit theory are sequential circuits in which unlike in combinational, the circuit outputs are functions of both inputs and circuit states. The circuit states are kept in memory elements such as Flip-Flop (FF). The states can change at the end of each clock cycle*. In this thesis, we use only Data Flip-Flop (D-FF), a type of FF that has four inputs: D (data), I (initial), clk (clock), and rst (reset) and one output: Q (latched data).

As seen in Figure 2.3b, a sequential circuit can be represented as an ensemble of a combinational circuit and feedback loops with memory elements. At each cycle, circuit inputs as well as the present states are fed to the combinational part. Then, it generates the outputs and next states which will be stored in the memory elements for

---

*The clock signal oscillates between a low and a high state and its (rising) edge is typically utilized to coordinate the memory updates.
the next cycle. The initial value of the memory elements are either a known constant value (0 or 1) or determined by an initial input value (I). In a digital hardware, FF initialization is usually done by reset or set signals. In TinyGarble, signal I of an FF can be connected to a constant value or input wire (in this cased called init wire).
Chapter 3

Garbled Circuit Synthesis

In this chapter, first, we review the background of the logic synthesis for digital circuits. Next, we explain Garbled Circuit (GC) synthesis flow that generates optimized combinational circuits for the GC protocol using logic synthesis. Then, we discuss the limitation of logic synthesis techniques and tools for making large combinational circuits for GC. A version of this chapter has been published in 2015 IEEE Symposium on Security and Privacy (S&P) [48].

3.1 Background on Logic Synthesis

Logic synthesis refers to the process of translating an abstract form of function (circuit) presentation to the gate-level logic implementation using a series of sophisticated optimizations, transformations, and mapping [32, 33, 34, 35]. A logic synthesis tool is a computer program which typically accepts the input circuit in some algorithmic form, logic equation, or even a table, and outputs an implementation suitable for the target hardware platform. Classic commercial/open-source logic synthesis tools accept the input functions in the Hardware Description Language (HDL) format, e.g., Verilog or VHDL [49, 50, 51, 52, 42, 53] but newer ones also accept high-level format, e.g., C/C++ [40, 41]. The common target hardware platforms for the synthesized logic include Field-Programmable Gate Array (FPGA), Programmable Array Logic (PAL), and Application-Specific Integrated Circuits (ASIC).
Typical practical implementations of a Boolean function utilize a multi-level network of logic elements. The tools translate the input to the implementation in two steps: (i) Logic minimization; and (ii) logic optimization. Logic minimization simplifies the function by combining the separate terms into larger ones containing fewer variables. The best-known algorithm for logic minimization is the ESPRESSO algorithm [54]; although the resulting minimization is not guaranteed to be the global minimum, it provides a very close approximation of the optimal, while the solution is always free from redundancy. This algorithm has been incorporated as a standard Boolean function minimization step in virtually any contemporary logic synthesis tool.

Logic optimization takes this minimized format, further processes it, and eventually maps it onto the available basic logic cells or library elements in the target technology (e.g., Lookup tables in FPGA and basic Boolean gates in ASIC).

Mapping is limited by factors such as the available gates (logic functions or standard cells) in the technology library, as well as the drive sizes, delay, power, and area characteristics of each gate.

Newer generations of synthesis programs, referred to as High-Level Synthesis (HLS) tools, accept other forms of input in a higher level programming language [55, 56, 57], e.g., ANSI C, C++, SystemC, or Python. HLS tools are also available in both open-source and commercial forms, e.g., [41, 53, 42]. The limitation of the higher level languages is that the behavior of the function is typically decoupled from the timing. The HLS tools handle the micro-architecture and transform the untimed or partially timed functional code into a fully timed HDL implementation, which in turn can be compiled by a conventional synthesis tool. It is well-known that the performance of the circuits resulting from automatically compiled HLS code
3.2 TinyGarble GC Synthesis

As explained in Section 2.2.3, Yao’s GC protocol requires a Boolean circuit representation of the underlying function. Previous work like FairPlay [12] and WYSTERIA [36] used custom-made languages to describe a function and generate the circuit for GC computation. In the TinyGarble framework, the user can write the function in a standard HDL like Verilog or VHDL. (She may also write the function in a high-level language like C and convert it to HDL using an HLS tool.) TinyGarble uses existing logic synthesis tools to map an HDL to a list of basic binary gates. In digital circuit theory, this list is called a *netlist*. The netlist is generated based on various constraints and objectives such that it is functionally equivalent to the HDL/HLS input function. Exploiting synthesis tools helps to reduce the number of non-XOR gates in the circuit and as a result the total garbling time and communication while also making the framework easily accessible. In Appendix C, we provide a library of optimized circuits for complex mathematical/logical operations made by our GC synthesis approach.

3.3 Synthesis Flow

Figure 3.1 shows the global flow of TinyGarble GC synthesis. It consists of the following four steps:

1. The input to TinyGarble GC synthesis is a file that describes a function written
Figure 3.1: TinyGarble GC synthesis flow. The inputs can be either a C/C++ program (translatable to HDL via a standard HLS tool) or a direct HDL description. The output is a scheduled circuit description ready to be garbled/evaluated.

in an HDL like Verilog or VHDL. The function can also be written in a high-level language like C/C++ and automatically translated to HDL using an HLS tool.

2. A standard logic synthesis tool compiles the HDL to generate the netlist file. The synthesis tool optimizes the netlist based on the user defined objectives/constraints and customized library. We set the constraints and library such that the logic synthesis tool produces a netlist optimized for evaluation in the GC protocol.

3. The netlist is parsed and topologically sorted. Then, we store the sorted netlist in a format readable by TinyGarble GC engine.

Figure 3.2 shows examples of files at different steps of TinyGarble GC synthesis flow for Hamming distance function. The `hamming.c` file contains the description of the function in the C language. The user inputs this function to an HLS tool to generate the corresponding description in Verilog. The resulting Verilog file is functionally similar to the `hamming.v` file shown in the figure, but it may look more
module hamming(clk, rst, a, b, h);
input clk, rst, a, b;
output reg [15:0] h;
always @ (posedge clk or poledge rst)
if (rst)
h <= 0;
else
h <= h + a^b;
endmodule

Figure 3.2 : Sample files at the different steps of TinyGarble GC synthesis flow for Hamming distance function.

complicated and be less efficient since it is generated by an automated tool. A user can also write the description directly in Verilog to have more control of the circuit and therefore a more optimized netlist. We provide the hamming.v file to a logic synthesis tool along with the TinyGarble custom libraries to generate netlist hamming_netlist.v. The netlist describes the same function as hamming.c and hamming.v but uses the logic cells provided in the technology library. The technology library contains 2-input-1-output logic cells to be compatible the GC protocol.

In the following, we describe the details of the synthesis steps and how we manipulate the synthesis tools in each of the steps to generate optimized netlists for the GC protocol.

### 3.4 Synthesis Library

The first phase in the synthesis flow is to convert arithmetic and conditional operations like add, multiply, and if-else to their logical representations that fit best to the user’s constraints. For example, one can replace the sum of two N-bit numbers with an N-bit ripple carry adder in the case of area optimization or an N-bit carry look-ahead adder for timing optimization. A library that consists of these various
implementations is called a *synthesis library*. We develop our synthesis library that includes implementations customized for the GC protocol. In this library, we build the arithmetic operations based on a full adder with one non-XOR gate [58] and conditional operations based on a 2-to-1 Multiplexer (MUX) with one non-XOR gate [13].

### 3.5 Technology Library

The next step is to map the structural representation onto a *technology library* to generate the netlist. A technology library contains basic units available on the target platform. For example, tools targeting Field-Programmable Gate Array (FPGA) like Xilinx ISE or Quartus provide Lookup table (LUT) and Flip-Flop (FF) in their technology libraries, which form the architecture of an FPGA. On the other hand, tools targeting Application-Specific Integrated Circuits (ASIC) like Synopsys DC, Cadence, and ABC, may contain a more diverse collection of elements starting from basic gates like XOR and AND to more complex units like FFs. The technology library includes logical descriptions of these units along with their performance parameters like delay and area. The goal of the synthesis tool in this step is to generate a netlist of library components that best fit the given constraints. For logic synthesis, we use tools targeting ASICs as they allow more flexibility in their input technology library.

We design a custom technology library that contains two-input gates as they incur minimum garbled tables in the GC protocol compared to gates with more inputs. We set the area of XOR gates to zero and that of non-XOR gates to a non-zero value, e.g., one or a large positive number. We choose area minimization as the only optimization goal, so the synthesis tool produces a netlist with the minimum possible number of non-XOR gates. Reducing non-XOR gates translates directly into decreasing the
communication cost of the GC protocol due to the Free XOR optimization [13].

An additional feature of our custom technology library is that it contains non-standard gates (other than basic gates like NOT, AND, NAND, OR, NOR, XOR, and XNOR) to increase the flexibility of mapping process. For example, the logical functions $F = A \land B$ and $F = (\neg A) \land B$ requires same effort in garbling/evaluation. However, using only standard gates, the second function would need two gates (a NOT gate and an AND gate) and store one extra pair of labels for $\neg A$ in the memory. We include four such non-standard gates with an inverted input in our custom library. XOR, XNOR, and NOT gates are free, so we set their area to zero. We assign the area of all other gates, similar to that of non-XOR, to one. To be consistent with the previous work in the literature, we use “number of non-XOR gates” to refer to the number of non-free gates in a circuit throughout this thesis. Due to the Free XOR optimization, this metric directly corresponds to time and communication cost of garbling/evaluating the circuit.

### 3.6 Offline Circuit Synthesis

In TinyGarble GC synthesis, we use logic synthesis tools in an offline manner to generate a circuit for a given functionality. This offline synthesis followed by a topological sort provides a ready-to-use circuit description for any GC implementations including our TinyGarble GC engine. This approach, unlike online circuit generation, does not require misspending time for producing the circuit during garbling/evaluation. It also enables employing useful synthesis optimization techniques that were previously infeasible for the online generation. Moreover, the synthesis tools have a global view of the circuit, unlike previous work that manually optimized small modules of the circuit. This overall view allows logic synthesis tools to provide more efficient
optimization for any arbitrary function and set of constraints.

3.7 Limitation

The synthesis approach has certain limitations when it comes to generating combinational circuits for extremely large functions. The amount of memory and computation resources required by logic synthesis tools considerably increase for producing large combinational circuits. Synthesis tools are not designed to support such large scale combinational circuits. Thus, the GC synthesis approach cannot scale, at least automatically, for generating combinational circuits for large functions.

However, most of these large functions include one or more recurring computation blocks (a.k.a, loops). In combinational representation, these loops have to be unfolded and represented as repeated blocks in the circuits. In hardware design, these functions are folded around their loops and are described using sequential circuits. A sequential circuit has feedback loops and is evaluated for multiple times (sequential cycles). Thus, it is the most suited format to represent the recurring blocks in large functions.

In the next chapter, we will discuss the possibility of using the sequential circuit for the GC protocol. We will show how one can benefit from the loops in sequential circuits to fold large functions while still benefiting from the optimization of synthesis tools.
Chapter 4

Sequential Garbled Circuit

Sequential circuits can be used as a very compact Boolean circuit description. We use the sequential description to overcome the scalability limitation of the synthesis approach for combinational circuits. In the following chapter, we first describe the concept of sequential circuits using a motivational example. Next, we discuss synthesizing sequential circuit for the GC protocol. Then, we explain the modifications required to garble/evaluate sequential circuits. A version of this chapter has been published in 2015 IEEE Symposium on Security and Privacy (S&P) [48].

4.1 Sequential Circuits

In the context of the GC protocol, we can define a sequential circuit as a folded version of a combinational circuit that needs to be evaluated for multiple cycles. Unlike combinational circuits where the output depends only on the inputs, the output of sequential circuit depends on both input and the state of the circuit stored in its memory. Usually, intermediate values during computation are stored as the state of the circuit and are used in later cycles to complete computation of the function. In following, we illustrate the difference between combinational and sequential circuit using a simple 4-bit adder as a function.

Figure 4.1 demonstrates an example of a combinational and a sequential circuit for a 4-bit Adder function with inputs $x = x_3x_2x_1x_0$ and $y = y_3y_2y_1y_0$, producing sum
Figure 4.1: The combinational and sequential design of a 4-bit Adder. (a) The HA circuit. (b) The FA circuit. (c) The combinational 4-bit Adder using one HA and three FAs. (d) The sequential 4-bit Adder using one FA.
\[ s = s_3s_2s_1s_0 = f(x, y) = x + y. \]

Figure 4.1a and 4.1b show the internal combinational circuit of a half Adder (HA) and a full Adder (FA) respectively. In Figure 4.1c a combinational Adder is built by cascading three FAs and one HA. Figure 4.1d represents a sequential implementation of a 4-bit Adder which uses one FA and a one bit FF to save the carry bit from the previous cycle. The circuit should be evaluated for four cycles. At the first cycle, the carry bit is \( z_0 = 0 \). Note that, in the combinational circuit we use three FAs and one HA whereas, in the sequential circuit, we have to use one FA for four sequential cycles. This asymmetry in the loop of Addition function introduces a small overhead in GC computation and communication time as an HA circuit has fewer gates compared to an FA circuit. We will discuss the overhead of garbling sequential circuits and its sources in more detail in Section 4.3.

However, the total number of gates for representing the function is reduced approximately by a factor of four when using a sequential circuit (one FA for sequential compared with three FA and one HA for combinational). This reduction in the number of gates helps to limit the memory footprint for garbling and evaluation required for storing circuit description and wire labels (\(k\)-bit per wire, see Section 2.2.3). In a sequential circuit, the number of labels that we need to store in the memory at any moment is proportional to the number of gates in the circuit. We simply overwrite the wire labels with the ones for the next sequential cycle. Only labels corresponding to FFs are kept for the next cycle.

Nearly all commercial circuits used in digital hardware are designed in the sequential format. There are multiple reasons for preferring sequential circuit description over combinational including the reduction in complexity, area, power, and cost, as well as natural mapping of finite state machine control functions into a sequential format. Some of these reasons also provide a rationale for a sequential description
of a function in GC, including: (i) reduction in size and memory footprint achieved by introducing the state elements and the feedback loop from output to input; (ii) removing the need to perform costly compile-time/runtime loop unrolling by embracing loops within the sequential feedback loop; (iii) providing a new degree of freedom for folding by the placement of memory elements in the long combinational paths—the placement can be done in accordance with the user’s objective; (iv) solving the limitation of synthesis techniques and tools for large function (see Section 3.7) by describing circuits in compact sequential format.

During the evaluation of a sequential circuit, the combinational block is evaluated $cc$ times where $cc$ is the number of sequential cycles that the circuit operates. We can visualize this process as the unrolled combinational representation of the sequential circuit as shown in Figure 4.2. $cc$ also shows the folding factor of the sequential circuit ($cc = 1$ means the circuit is combinational). The inputs/outputs of the unrolled circuit are the inputs/outputs of the combinational block in all the cycles. The present states at each cycle $cid$, where $0 \leq cid < cc$, are equal to the next states in the previous cycle ($cid - 1$). The present states at $cid = 0$ are equal to the initial input value.
In digital hardware, at the edge of a cycle, the value of D of an FF is latched and then transferred to the Q in the next sequential cycle. In garbling/evaluating of a sequential circuit, an FF operates as a simple wire connection between the unrolled combinational circuit in two consecutive cycles. Thus, it can be dealt with latching the labels from the flip-Flop input D in cid to output Q in cid + 1. We simply ignore clk and rst signals of FFs in the sequential GC.

Garbling/evaluation of the sequential circuit is equivalent of garbling its gates for cc times. To make sure that the security and correctness of the GC protocol still hold for the sequential circuits, we use a combination of cycle index and gate identifier for the encryption tweak T (see Section 2.2.4). In other words, we assign a unique tweak for the same gates in different cycles. This trick makes the process similar to garbling an unrolled combinational circuit.

In Appendix B, we discuss our implementation of the GC protocol for sequential circuits in TinyGarble GC engine. We also explain in more detail about how we set the encryption tweak and how it affects the security and correctness of the GC protocol.

4.2 Synthesis Sequential Circuit

Fortunately, adopting TinyGarble GC synthesis for sequential circuits is straightforward since the synthesis tools support sequential circuits. We add memory element of sequential circuits into the technology library of TinyGarble (see Section 3.5).

We implement these elements using FFs. Although in ASIC design FFs are typically as costly as four NAND gates, as seen above in GC, FFs do not have any impact on the garbling/evaluation process as they require no cryptographic operations. Therefore, we set the area of FFs to zero to show its lack of impact on the
communication cost of the GC protocol. Moreover, we modify our FFs such that they can accept an initial value. The initial value helps us to remove unnecessary MUXs in standard FF design for initialization.

Describing functions using sequential circuits allows us to overcome the scalability limitations of the synthesis tool. Sequential circuits are radically smaller than combinational ones with the same functionality. This property allows synthesis tools to not only generates the circuit with fewer recourses, but to find a more optimized circuit. Moreover, the compatibility of our sequential descriptions with standard synthesis tools simplifies the workflow of circuit generation for SFE applications.

4.3 Overhead of Sequential Circuit

In the sequential circuit, we divide the function into smaller steps such that we can compute each step at one cycle. If one can split a function into identical steps, then she can implement the sequential circuit using only one step. Evaluating the sequential circuit for multiple cycles ensures that the function is computed completely. However, many functions do not have such a symmetrical structure, and they cannot be divided perfectly into identical steps. The sequential circuits of such functions include different sub-circuits that each corresponds to one of the steps in the function.

Figure 4.3a shows an example of such function represented as a combinational circuit. We divide the computation of the function can into six non-identical steps. The computation in the first and last steps are different from the rest. U1 sub-circuit represents the computation in the first step, U2 the middle four steps, and U3 the last step. Figure 4.3b illustrates the sequential circuit representation of the example function. A MUX selects the output of the sub-circuits according to the cycle id (cid). In the digital circuit design, MUX and similar components that control the
Figure 4.3: The combinational and sequential representations of a function with an asymmetric structure. The sequential implementation has to be evaluated for six cycles and eventually needs more gates to be garbled/evaluated compared to the combinational one.

The unselected sub-circuits remain idle in that cycle since their output is not used. At each cycle, the output of the selected sub-circuit is stored in the FF to be read in the following cycles.

The number of gates that have to be garbled/evaluated in the combinational circuit is $C_{U1} + 4 \times C_{U2} + C_{U3}$, where $C_{Ui}$ is the number of gates in $U_i$ sub-circuit. Since the sequential circuit has to be garbled/evaluated for $cc = 6$ times, the total gates for the sequential circuit are $6 \times (C_{U1} + C_{U2} + C_{U3} + C_{MUX})$, where $C_{MUX}$ is the number of non-XOR in the MUX. The difference between the cost of the combinational and
the sequential circuits is $5 \times C_{U1} + 2 \times C_{U1} + 5 \times C_{U3} + 6 \times C_{MUX}$). We call this difference the sequential \textit{overhead} for garbling an asymmetrical function.

The overhead is caused by treating cid as a secret value in the GC protocol while both parties know its value at all cycles. In the next chapter, we will discuss how to reduce this overhead using a novel algorithm on top of sequential GC protocol. The algorithm skips unnecessary garbling/evaluation of idle sub-circuits and allows local computation of the gates in the control path.
Chapter 5

SkipGate: Reducing Sequential Overhead

SkipGate is developed to work with the Garbled Circuit (GC) protocol to reduce the overhead of sequential circuits. It allows secure evaluation of functions in the form of $o = f(a, b, p)$ 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 $o = f_p(a, b)$ 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. For doing so, 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, SkipGate reduces the communication between the parties when when less costly local computation can replace it. The cost reduction is especially significant in a sequential circuit where the control path is public and independent of the private inputs.

In this chapter, first, we introduce the notation used in the SkipGate algorithm. Next, a motivational example is provided to present the sequential overhead. Next, we discuss how gates in a circuit are categorized in SkipGate. The pseudo-code of the SkipGate algorithm is then provided with its complexity analysis and correctness and security proofs.
5.1 Notations

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 ($X^0_w$ and $X^1_w$) on Alice’s side and one label ($X_w \in \{X^0_w, X^1_w\}$) on Bob’s. If $X_w = X^0_w$, the actual Boolean value is 0 and if $X_w = X^1_w$, it is 1. This construction 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).

5.2 Motivational Example

Figure 5.1 illustrates a sequential circuit that has a control path with a 2-to-1 Multiplexer (MUX) whose inputs come from two sub-circuits $f_0$ and $f_1$ connecting to MUX input 0 and 1 respectively. At a certain sequential 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, the gates in the control path do not need to be garbled/evaluated in that sequential 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 presented in Chapter 4, the Boolean value of a wire can change at every sequential 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 [14] which remove gates with known constant inputs. Since we use TinyGarble to create the circuits, the logic synthesis tool already removed the constant gates in it.

### 5.3 Gate Categories

The SkipGate algorithm classifies the gates into four categories regarding the parties’ knowledge about their inputs:

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

2. **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 like 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. (In
Section 5.4.1, we will explain 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., Alice does not send the corresponding garbled table to Bob.

5.4 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 parties compute the outputs of the gates with
public input (Categories i-ii). In Phase 2, they garble/evaluate the gates with private
inputs (Categories iii-iv). 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 to evaluate the gates. In Line 12 of Algorithm 1
Algorithm 1 SkipGate, Alice’s side.

Inputs: Sequential circuit of $f(a, b, p)$, Alice’s input $a$, public input $p$, number of sequential cycles $cc$.

Outputs: $o = f(a, b, p)$.

1: **SkipGate_alice** (circuit, a, p, cc):
2: $(X^0_a, X^1_a, X^0_b, X^1_b) = \text{generate_random_labels}()$  
3: send_alice_labels(a, $X^0_a, X^1_a$)  
4: send_bob_labels($X^0_b, X^1_b$)  // through OT  
5: circuit.set_private_input($X^0_a, X^1_a, X^0_b, X^1_b$)  
6: circuit.set_public_input(p)  
7: for cid in $[0...cc - 1]$ 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: $(X^0_c, X^1_c) = \text{circuit.get_output_label}()$  
15: $X_c = \text{receive_bob_output_label}()$  
16: $o = \text{get_output_value}(X^0_o, X^1_o, X_o)$
Algorithm 2 SkipGate, Bob’s side.
Inputs: Sequential circuit of \( f(a, b, p) \), Bob’s input \( b \), public input \( p \), number of sequential cycles \( cc \).

1: \textbf{SkipGate\textunderscore bob}(circuit, b, p, cc):
2: \( X_a = \text{receive\_alice\_labels()} \)
3: \( X_b = \text{receive\_bob\_labels}(b) // \text{through OT} \)
4: circuit.set\_private\_input(\( X_a, X_b \))
5: circuit.set\_public\_input(\( p \))
6: \textbf{for} cid in [0...cc - 1] \textbf{do}
7: \hspace{1em} circuit.initial\_label\_fanout()
8: \hspace{1em} circuit.phase1()
9: \hspace{1em} garbled\_tables = \text{receive\_garbled\_tables()} \)
10: \hspace{1em} circuit.phase2\_bob(garbled\_tables)
11: \hspace{1em} circuit.transfer\_flip\_flops\_labels()
12: \textbf{end for}
13: \( X_c = \text{circuit.get\_output\_label()} \)
14: send\_output\_label(\( X_o \))
and Line 11 of Algorithm 2, the labels associated with the input of flip-flops are transferred to their output for the next cycles as required by sequential GC protocol (see Section 4.1). Similar to conventional GC, in 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 $o$. 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 $\text{label\_fanout}$ 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 as an input to other gates). At the beginning of each cycle (Line 8 of Algorithm 1 and Algorithm 2), the parties set the $\text{label\_fanout}$ to the gate fanout in the circuit*. A gate’s $\text{label\_fanout}$ may decrease if its output label is not needed anymore, e.g., a gate whose output is connected an AND gate with 0 at the other input (Category ii). If $\text{label\_fanout}$ reaches zero, it means that gate’s output label does not have any effect on the final output. The gates with $\text{label\_fanout}=0$ are subsequently designated for skipping, which in turn decreases the $\text{label\_fanout}$ of their input gates recursively. Finally, the gates in Category iv that the parties have not marked for skipping are garbled/evaluated.

Algorithm 3 illustrates Phase 1 of SkipGate in which Alice and Bob find and compute the gates that belong to Categories i-ii. They set the $\text{label\_fanouts}$ of the gates in Category i to zero. For gates in Category ii, if the output becomes public, SkipGate decreases the $\text{label\_fanout}$ of the secret input’s originating gates recursively by invoking $\text{recursive\_reduction}$ (Algorithm 6). Figure 5.2 shows four

*Fanout of a gate, borrowed from hardware design, is the number of subsequent gates (and circuit outputs) dependent on the gates output.
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
Figure 5.2: Four examples of replacing gates in Phase 1 by zero, one, wire, or inverter. We decrease the label fanout of the unnecessary gate accordingly. The top-left example is in Category i, and the rest are in Category ii.

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 method 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.

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 5.3 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 zero. 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
**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
4:      g.i0.label is inverted g.i1.label) //Category iii then
5:     g.o = related_secret_calculate(g.type, g.i0, g.i1)
6:   if g.o is public then
7:     g.label_fanout = 1 //will become zero in recursive_reduction()
8:     circuit.recursive_reduction(g)
9:   end if
10:  else
11:     //Category iv
12:     (g.o, g.table) = garble(g.type, g.i0, g.i1) //table=null for XOR
13:     if g is non-XOR then
14:       garbled_tables.enqueue(g.id, g.table)
15:     end if
16:   end if
17: end for
18: 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
4:     g.i0.label is inverted g.i1.label) //Category iii then
5:     g.o = related_secret_calculate(g.type, g.i0, g.i1)
6:   if g.o is public then
7:     g.label_fanout = 1 //will become zero in recursive_reduction()
8:     circuit.recursive_reduction(g)
9:     end if
10: else
11:   //Category iv
12:     if g is XOR then
13:       g.o = g.eval_XOR(g.i0, g.i1)
14:     else if g.id is garbled_tables.top().id then
15:       gt = garbled_tables.dequeue().table
16:       g.o = g.eval(g.type, g.type, g.i0, g.i1, gt)
17:     else
18:       g.o = next_unique_label() //generate a unique label.
19:     end if
20:     end if
21: end for
Figure 5.3: Four examples of replacing and computing gates in Phase 2. We decrease the label fanout of the unnecessary gate accordingly. The top-right example is in Category iv, and the rest are in Category iii.

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 zero. 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. Bob can generate these labels randomly or use a monotonic counter that increases by one for each newly generated label. To distinguish valid GC labels from his generated labels, he keeps a single bit flag along with each label that indicates he generated the label and it is not valid for GC evaluation.

Algorithm 6 illustrates the pseudo-code for the recursive fanout reduction. It receives the circuit and a gate of the circuit. It first decreases the label fanout of the given gate. If the label fanout becomes zero, it recursively calls itself with the gates that generate the secret input(s). Figure 5.4 illustrates this process on a sample circuit.
**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
5.4.1 Identification of Identical and Inverted Labels

According to the GC protocol, Bob receives only one label $X_w$ for each secret wire $w$. Due to Free XOR [13], he does not need to do anything with a label when it passes a NOT gate because Alice will flip the labels associated with the Boolean value. Thus, he cannot tell apart an identical and inverted secret values just by storing labels. 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 is crucial for Phase 2.

Figure 5.5 illustrates the effect of SkipGate on the example in Figure 5.1. The gates in the sub-circuit $f_0$ will all be skipped for garbling because their $\text{label}\_\text{fanout}$ will be zero. The gates in the MUX also will be bypassed since they belong to Groups (i-iii).
Figure 5.5: SkipGate macro effect on the example in Figure 5.1. Only sub-circuit $f_1$ is garbled/evaluated.

5.5 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 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. The parties initialize $\text{label}$.fanout the gate fanout in the circuit. The upper bound on the sum of fanouts of all the gates in the circuit is

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

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 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 linear complexity shows that SkipGate does not increase the overall computational complexity of the GC protocol.

### 5.6 Correctness Proof

Given the correctness of Yao’s GC protocol, we have to show that GC protocol with SkipGate is also correct. SkipGate does not alter the topology of the circuit. 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 rely on the Boolean operation of the gates and are clearly correct. For gates in Categories ii-iii, we consider the secret input as an unknown variable. The parties either pass the secret label to the gate’s output, or they set the output to a public value. Since they perform this operation based on the Boolean logic of the pertinent gate, the output remains logically correct. Gates
in Category iv with non-zero $\text{label\_fanout}$ are garbled/evaluated according to the GC protocol. For the rest of the gates in Category iv, $\text{label\_fanout}=0$ 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.

### 5.7 Security Proof

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 [59, 16]. In this thesis, 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 $\text{label\_fanout}$ evaluated by the GC protocol. The process of skipping gates from $f(a, b, p)$ only utilizes the public input $p$ which is already known to both parties. In the process, the private values are treated as unknown Boolean variables. In other words, Alice and Bob do not access their inputs in the SkipGate algorithm for reducing $f(a, b, p)$ to $f_p(a, b)$. Thus, the SkipGate algorithm reveals no information about the private inputs $a$ and $b$. 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.
Chapter 6

Garbled Processor

Supporting sequential circuits in the Garbled Circuit (GC) protocol enables us to evaluate a general processor as a function in two-party Secure Function Evaluation (SFE). The Boolean circuit of a general processor is a sequential circuit that receives a function (more pressingly the compiled binary code of a function) and data in its memory, then computes the function and writes the result back in the memory. In this chapter first, we explain the problem of Private-Function SFE (PF-SFE) and how a processor can solve it. Next, we describe how we used MIPS processor, a simple text-book processor, to resolve the PF-SFE problem. Next, we study how a garbled processor can be modified to employ for the SFE problem as a mean to make it easy for users to develop privacy-preserving applications. Then, we explain how ARM, a more sophisticated processor, can be well suited for solving SFE development if its secure evaluation accompanied by the SkipGate algorithm. Versions of Section 6.1 and Section 6.2 of this chapter has been published in 2015 IEEE Symposium on Security and Privacy (S&P) [48]. A version of Section 6.3.1 of this chapter has been published in 2016 Proceedings of the 53rd Annual Design Automation Conference (DAC) [60].
6.1 Private Function Evaluation

Two-party Private-Function SFE (PF-SFE) allows secure computation of a function \( g_{Alice}(\cdot) \) held by one party (Alice) operating on another party’s data \( b_{Bob} \) (Bob) while both the data and the function are kept private. This setting is in contrast to the usual requirement of SFE where both parties know the function. PF-SFE is especially useful when the function is proprietary or classified.

It is well known that PF-SFE can be reduced to regular SFE by securely evaluating a Universal Circuit (UC) \cite{61}. UC is a Boolean circuit capable of simulating any Boolean circuit (function) \( g(\cdot) \) given the description of \( g(\cdot) \) as input \cite{62, 63}:

\[
UC(b_{Bob}, g_{Alice}(\cdot)) = g_{Alice}(b_{Bob}).
\]

Secure evaluation of UC completely hides the functionality and the topology of the Boolean circuit of \( g_{Alice}(\cdot) \). Subsequent works have shown how to allow PF-SFE while avoiding the overhead of UCs \cite{64, 65}.

A UC is similar to a Universal Turing Machine (UTM) \cite{66, 67} that receives a Turing machine description \( g_{Alice}(\cdot) \) and applies it to the input data \( b_{Bob} \) on its tape \cite{68}. One party provides the description, and the other one provides the input data. After UTM completes the operations, it writes the output \( g_{Alice}(b_{Bob}) \) on the tape. A general purpose processor is a special realization of a UTM. It receives a list of instructions \( g_{Alice}(\cdot) \) (the compiled binary code of \( g_{Alice}(\cdot) \)) and applies them to the input data \( b_{Bob} \).

6.1.1 Arithmetic Logic Unit

The core of conventional processors is the Arithmetic Logic Unit (ALU) which receives two operands and an opcode indicating the desired operation. ALU supports a set of
operations, for example, addition, multiplication, and XOR. The ALU circuit consists of multiple sub-circuits for these operations and a Multiplexer (MUX) which selects one of their outputs. Secure evaluation of an ALU, where the opcode comes from one party and operands come from the other party, keeps the operations private. Thus, we can think of ALU as an emulator of a simple UC in which the input function $g_{\text{Alice}}(\cdot)$ is limited to a single operation.

One can combine a number of ALUs to make a more comprehensive UC that can support functions consisting of multiple operations. Unfortunately, this approach is not practical as the complexity of the circuit grows linearly with the number of operations. On the other hand, in conventional processors, ALUs are combined with arrays of Flip-Flop (FF)s, a.k.a., registers, to store the intermediate values for supporting functions with an arbitrarily large number of operations. Since none of the earlier implementations of GC explicitly supported memory elements such as FFs, the ways to connect the feedback loop around the ALU were rather limited. However, an explicit sequential description supported by TinyGarble allows us to leverage conventional processor architectures. Therefore, the TinyGarble methodology not only provides a powerful method for generating compact circuits with a low overhead for SFE but also paves the way for systematically building scalable sequential circuits used for PF-SFE.

The idea of using an ALU or a universal next-instruction circuit in the GC protocol can also be found in [69]. The objective of that work was improving the efficiency of SFE where the function is known by both parties, unlike PF-SFE where the function is private. Nonetheless, instead of ALU they eventually decided to use an instruction-specific circuit which leaks information about the function but results in less effort for non-private function evaluation.
6.1.2 Memory

The processor accesses the memory while executing an instruction to read the instruction and data and write the data back. If the parties evaluate the memory along with the processor in the GC protocol, they cannot learn about the access patterns of the memory. On the other hand, if they evaluate the memory separately and outside of GC, they may learn the access patterns that in turn could reveal information about the function to Bob and the data to Alice. For example, the instruction read pattern discloses the branching decisions in the function which may leak information about the data. Because of TinyGarble sequential methodology, the memory can be easily implemented using MUX and arrays of FFs. Thus, it can be included in the processor circuit to be evaluated securely using the GC protocol. However, the inclusion of MUXs and FFs increases the operation time and communication linearly with respect to the memory size.

One alternative approach for hiding memory access patterns is the use of Oblivious Random-Access Machine (ORAM) protocols [70] which allow oblivious load/store operations with amortized poly-logarithmic overhead at the expense of increasing the round complexity of the GC protocol [71, 69, 72, 73]. For the sake of simplicity, we do not use ORAM in this thesis. However, one can simply connect our implementation of PF-SFE to an ORAM to benefit from its lower amortized complexity. As another alternative, [74] shows that algorithms can sometimes be rewritten to use data structures such as stacks, queues, or associative maps for which they give compact circuit constructions of poly-logarithmic size.
6.2 Garbled Processor for PF-SFE

6.2.1 Global Flow

We assume Alice provides the private function $g_{\text{Alice}}(\cdot)$ and Bob provides private data $b_{\text{Bob}}$. At the end of the operation, only Bob learns the output $g_{\text{Alice}}(b_{\text{Bob}})$. Note that we are not considering the case where both parties learn the output as that would allow Alice to learn Bob's private data with an identity function ($g \equiv I$). The protocol is as follows:

1. Alice and Bob agree on an Instruction Set Architecture (ISA), its implementation (i.e., the processor circuit), the maximum number of sequential cycles, and the configuration of data $b_{\text{Bob}}$ in the memory.

2. Alice compiles the function $g_{\text{Alice}}(\cdot)$ according to the ISA. Her input is the compiled binary of the function.

3. Bob prepares his input based on the agreed configuration to initialize the processor memory.

4. Using any secure GC framework, Alice garbles the processor circuit for the maximum number of sequential cycles and Bob, after receiving his inputs with Oblivious Transfer (OT), evaluates the garbled processor circuit for the same number of cycles.

5. Alice shares the output labels with Bob such that he learns the value of the output $g_{\text{Alice}}(b_{\text{Bob}})$ stored in memory. Alice shares the labels only for the agreed memory locations containing the outputs such that Bob does not learn intermediate values in the memory.
Because of secure evaluation using the GC protocol in Step 4, no information about values in the circuit leaks except the output. Without knowing internal values in the processor circuit, none of the parties can distinguish instructions or memory access patterns. In the following, we demonstrate an implementation of a processor supporting the MIPS ISA, as an example of a garbled processor for securely evaluating private functions.

### 6.2.2 MIPS

MIPS is a text-book Reduced Instruction Set Computing (RISC) ISA [75]. The RISC ISA consists of a small set of simplified assembly instructions in contrast to Complex Instruction Set Computing (CISC), e.g., x86 ISA, which includes more complex multi-step instructions [76]. We choose a RISC ISA processor instead of CISC for the following main reasons: (i) lower number of non-XOR gates, (ii) simple and straightforward implementation, and (iii) availability and diversity of open-source implementations. Moreover, we choose a single-cycle MIPS architecture (i.e., one instruction per sequential cycle). Other architectures (i.e., multi-cycle and pipelined) increase the performance of the processor by parallelization. However, the GC protocol does not benefit from such low-level parallelization. The only important factor for GC is the total number of non-XORs which is smaller in the single-cycle MIPS.

We follow the Harvard Architecture which has distinct Instruction Memory (IM) and Data Memory (DM) to separate the parties’ inputs. The IM is a Read-Only Memory (ROM) that stores Alice’s instructions. The DM is a Random Access Memory (RAM) initialized with Bob’s input. The parties’ inputs are connected to the initial signals (I) of FFs in the memories. Bob’s outputs are connected to the outputs (Q) of FFs in the specified address of the DM. The output address in the DM is part of the agreed
Figure 6.1: Lite MIPS architecture. Alice’s and Bob’s inputs and the output are shown.

memory configuration.

Figure 6.1 shows the overall architecture of our 32-bit MIPS processor. We borrowed the circuit from the Plasma project in opencores [77]. We modified the circuit such that the IM and the DM are separated. The original Plasma processor supports all the MIPS-I ISA except unaligned memory access. In our implementation, we also omit division instructions because of their large overhead. Any arbitrary function described in C can be easily compiled to the MIPS-I assembly code using a cross-platform compiler, e.g., GNU gcc.

In 32-bit MIPS, the Program Counter (PC) is a 32-bit register (array of FFs) that points to the instruction that the processor is executing at the current cycle. The processor fetches the instruction from the IM based on the current PC value. The controller unit is responsible for setting signals to perform the instruction. In 32-bit MIPS, the register file consists of 32 registers of 32-bit each. In each cycle, at most two registers can be read and at most one register can be written back. The ALU receives the read register(s) or a sign extended immediate as operands. The
ALU also receives an opcode from the controller unit. The output of the ALU will be either written back to the register file or fed to the DM as an address for load/store. The loaded data from the DM is written back to the register file. In each cycle, the processor increments the PC by 4 to point to the next instruction in the IM or changes it according to a branch or jump instruction.

6.3 Garbled Processor for SFE

In the previous section, we discuss the idea of garbling a processor as a solution for hiding the function in PF-SFE. Besides enabling PF-SFE, another advantage of a garbled processor is usability for non-expert users since it can be programmed using high-level languages, whereas other frameworks for the GC protocol require tedious Boolean circuit construction. However, garbling and evaluating the entire processor incurs a tremendous cost compared to SFE solutions due to stronger privacy requirements in PF-SFE.

In this section, we expand the garbled processor introduced in Section 6.2 and introduce a framework for secure computation that provides scalable support for generalized SFE. The framework provides three options: a high performance with a relaxed privacy setting, the more security-demanding PF-SFE with a higher cost (similar to the one in Section 6.2), and a flavor in-between.

To avoid information leakage about the function (i.e., PF-SFE), we employ the MIPS circuit with its full Instruction Set Architecture (ISA), which incurs a substantial overhead due to garbling and evaluating of the entire ISA. We can also compile the function using only a subset of the ISA: restricted ISA (i.e., semi-private function). A third alternative is public function mode in which the function is compiled using only an application-specific subset of the ISA required for executing the function. In the
following, we discuss these modes of function evaluation and the trade-off between privacy and performance further.

6.3.1 Garbled Processor for Public Functions

Employ a general-purpose processor supporting its entire ISA for SFE incurs a significant computation and communication cost. However, this cost seems unnecessary since both parties know the executed function instructions and they are only interested in learning the output value. Hence, garbling a limited application-specific ISA for executing each instruction is sufficient to achieve the desired privacy. To further reduce the ISA, assuming, for example, a function that consists of 10 instructions, we could theoretically generate $2^{10} - 1$ netlists (netlists of ISA with different combinations of the ten instructions, excluding the netlist with zero instructions). At run-time, one of these netlists is plugged in (garbled and evaluated) at each instruction step depending on the expected instructions. However, to make it more reasonable (generate fewer netlists), for functions with control flow independent of private data, we know in advance which instruction will be executed at each step. Thus, we need only the netlist of the processor implementing ISA with that specific instruction, restricting the required netlists in this case to 10. For functions with control flow dependent on private data, a simple static analysis can be used to specify the combination of possible instructions at each step, and hence the required ISA netlist as proposed in [78].

6.3.2 Garbled Processor for Semi-Private Functions

The main cost for garbling a processor with its entire ISA results from garbling circuits for expensive instructions like multiplication and division. Most compilers can avoid
these costly instructions and replace them with cheaper loops of shifts, addition, and subtraction instructions. These replacements would eliminate the need for the Mult/Div unit in the processor and reduce the cost of garbling per instruction on the one hand. However, on the other hand, one expensive instruction will be replaced with multiple cheap instructions, thus increasing the total number of instructions. For example, multiplying two 32-bit numbers with the MULT instruction in MIPS requires 15 cycles and a circuit of 13,257 non-XOR gates\(^*\), while it requires at least 31 cycles and a circuit of 9.676 non-XOR gates when using a conditional loop over an ADD instruction. We call this mode “semi-private” since it only reveals partial information about instructions executed in the program (that the program does not use division/multiplication) and increases the probability of guessing an instruction by reducing the subset of possible instructions (restricted ISA).

### 6.3.3 Garbled Processor for Private Functions

In the standard 2-party PF-SFE, Alice provides the function \(g_{\text{Alice}}(\cdot)\), Bob provides the input data \(b_{\text{Bob}}\), and the output is \(g_{\text{Alice}}(b_{\text{Bob}})\). Similar to Section 6.2, the garbled processor receives a list of instructions of compiled \(g_{\text{Alice}}(\cdot)\) and applies them to the input data \(b_{\text{Bob}}\) in memory, and eventually writes the output back to the memory. To avoid information leakage about the private function, we use a general-purpose processor with its entire ISA (full ISA).

### 6.3.4 Hardware Implementation of Garbled Processor

To the best of our knowledge, the fastest implementation of GC in hardware was [79]. However, their performance is far less than a JustGarble [16], a software implemen-

\(^*\)XOR gates are evaluated free of cost in GC according to Free XOR [13].
tation. The reason for this gap is that JustGarble utilizes a more efficient fixed-key AES encryption for garbling instead of an expensive hash function. Thus, it is possible that a hardware implementation leveraging the latest GC optimizations including fixed-key AES garbling would outperform the software implementation. Furthermore, a processor is essentially a sequential circuit, and its evaluation requires sequential GC which no hardware implementation of GC supports.

We implement our GC evaluator based on the most recent optimizations listed in Section 2.2.4. Its architecture is shown in Figure 6.2 and consists of: (1) Simple Sequential Circuit Description (SSCD) memory: a read-only memory that stores the information about gates in the MIPS circuit in SSCD format (see Appendix B.2). (2) GC Label memory: a read-write random-access memory that stores GC labels of all wires in the corresponding MIPS circuit. (3) Garbled Tables (GT) memory: a read-write random-access memory that stores the garbled tables of each non-XOR gate in the MIPS circuit that are generated by Alice (garbler). (4) Sequential Handler: a controller that supports evaluation of the sequential circuits with the GC protocol. (5) Evaluator Core: the implementation of the core functionality of Yaos GC protocol’s evaluation, its most recent optimizations [13, 16, 17], and the sequential garbling presented in Chapter 4.

As shown in Figure 6.2, Bob’s input labels in the Label memory are initialized by the OT protocol with Alice. The rest of the labels in the Label memory and the Garbled Tables memory are received from Alice.

**Pipelined Evaluator Core and Gate Dependency**

To maximize the performance of our hardware GC evaluator, we use a 20-stage pipelined AES implementation [80] inside our Evaluator Core module. It increases
the throughput of the module by increasing the maximum operating clock frequency of the core. We also add one stage for the rest of the GC evaluation functionality.

Due to Free XOR [13], evaluating an XOR gate requires only XORing the input labels while evaluating a non-XOR gate requires two AES encryptions [17]. Therefore, the Evaluator Core can finish the evaluation of an XOR gate in one clock cycle of the pipeline. The different timing for XOR and non-XOR gates introduces a challenge for handling dependencies of gates’ inputs and output. A gate cannot enter the evaluation pipeline if its inputs are another gate’s output which is not yet evaluated. Stalling pipeline is a naive approach for resolving this dependency and degrades the overall performance To mitigate this, we push XOR gates to the latest empty stage of the pipeline such that the subsequent dependent gates can enter the pipeline as soon as possible.
Extending Hardware Prototype

In this thesis, we only use on-chip memory for as a proof-of-concept implementation. However, this prototype can be extended to support interfacing with off-chip memory which would store garbled tables and labels of larger garbled processor circuits and functions. It can also interface with another hardware emulator of the garbler which generates the garbled tables and labels and streams them to our evaluator. A wide range of scenarios is now feasible owing to our current hardware platform and state-of-the-art optimized GC evaluator.

Such extensions would incur additional area and performance overheads but would allow upscaling of our implementation to support garbled processor circuits and benchmark functions in the Gigabytes range. We emphasize that we provide in this thesis a proof-of-concept prototype to motivate further research in this direction to bring garbled processors some steps closer to the realm of efficient and practical implementations.

6.4 ARM2GC: Garbled ARM for SFE

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 Application Program Interface (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 ORAM for ARM2GC.
6.4.1 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 6.3 shows the overview of the framework. The framework benefits from the SkipGate algorithm to reduce the cost of garbling the ARM processor. As described in Chapter 5, the SkipGate algorithm supports secure evaluation of circuits in the form of \( f(a, b, p) \) where \( a \) and \( b \) are Alice’s and Bob’s inputs, and \( p \) is a public input known to both parties. In the ARM2GC framework, the circuit \( f(\cdot, \cdot, \cdot) \) is the circuit of ARM processor and the public input \( p \) is the compiled binary code of the user’s SFE program. To avoid confusion with ARM circuit, we denote the user’s function with \( g \). Similar to other two-party SFE functions, \( g(\cdot, \cdot) \) has two inputs, one from Alice and one from Bob. The high-level code of \( g(\cdot, \cdot) \) is compiled using an ARM cross-compiler, e.g., gcc-arm-linux-gnueabi. The compiled binary code of \( g(\cdot, \cdot) \) is then passed to the ARM circuit as the public input \( p \). The parties’ private inputs \( (a \) and \( b \) are passed directly to ARM circuit: \( f(a, b, p) \). The SkipGate algorithm then securely evaluate \( f(a, b, p) \) by reducing the circuit into the simpler circuit of \( f_p(a, b) = f(a, b, p) \). The ARM circuit computes the function \( g(\cdot, \cdot) \) on the private inputs and returns its output: \( o = f(a, b, p) = g(a, b) \).

The ARM2GC framework supports the following API in C language for the user’s function \( o = g(a, b) \):

```c
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 circuit of our ARM processor has five separate memory elements (consisting of FFs and MUXs) to store: Alice’s inputs, Bob’s inputs, output, stack, and instructions. The FFs 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 FFs in the stack, output, pipeline registers, and the register file are initialized to zero. The ARM circuit is garbled using sequential garbling process of TinyGarble GC engine (see Appendix B) for a pre-specified number of sequential cycles \( cc \).

A signal called `terminate` is produced by the ARM circuit that indicates if `gc_main` function is returned. The signal can be revealed to the parties once in \( K \) cycles (predetermined by parties) to reduce the total number of cycles of garbling the ARM
circuit. For $K = 1$, the parties instantly identify the termination, but the exact number of cycles the function evaluated for the given inputs is revealed. A larger $K$ would reduce this information leakage, but increase the garbling cost. Appendix B provides more details about the support of the terminate signal in TinyGarble GC engine. Eventually, when the function is executed, the parties reveal the content of the output memory to each other to learn output $o = g(a, b)$.

### 6.4.2 ARM as a Garbled Processor

In this thesis, we choose ARM as the garbled processor which is a more ubiquitous and sophisticated processor compared to MIPS. ARM has two primary 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 the performance and code density, conditional execution allows ARM to execute each instruction only if a specific condition is satisfied [81].

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 mis-prediction. Similarly, in the 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 6.4 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, the program flow itself will become secret which in turn reduces the efficiency of the execution.
Figure 6.4: An example code that shows how conditional execution in ARM can reduce the code size and make the program flow predictable.

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 gates in the circuit. In the GC protocol, the total number of non-XOR gates is the only factor affecting the performance, not the circuit’s topology.

Implementation of the ARM processor results in a complex and large circuit (containing almost five times more gates than the MIPS processor). Thus, using ARM instead of MIPS 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.
6.4.3 How SkipGate Helps

As explained above, we initialize the instruction memory of the ARM 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 \( o = f(a, b, p) \) where \( p \) is equal to the compiled binary code of \( g(a, b) \) and \( a \) and \( b \) are the parties’ private inputs. SkipGate reduces the ARM circuit into a smaller circuit of \( o = f_p(a, b) \) where \( f_p \) can perform the exact operation required in the function \( g(\cdot, \cdot) \). 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 the approaches in Section 6.3 and [78].

6.4.4 Why not Sub-linear ORAM?

As mentioned in Section 6.4.1, we use an array of MUXs and FFs to implement the register file in ARM circuit. This structure 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 Random-Access Machine (ORAM) that enables oblivious access to memories in the GC protocol with sub-linear cost [82, 83]. The reason is that, in most cases, the access to the register file is not required to be oblivious. Since the instructions come from the publicly known instruction memory, both parties know which register of the register file is read or
Figure 6.5: In a case when the compiler fails 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.

written. The SkipGate algorithm utilizes this to skip garbling of the gates in the MUXs of the register file. Thus, no 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 6.5 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 be either 3 or 6 depending on the outcome of the comparison in Line 1. Thus, two instructions \( \text{add } \$1, \$2, \$3 \) and \( \text{sub } \$5, \$6, \$7 \) have to be garbled/evaluated at the same time. For fetching the second register in instruction from the register file, we only have two choices: \( \$2 \) and \( \$6 \). This limited choice means that, instead of having a complete oblivious access to the register file with 16 options, we only have to select obliviously between 2 of the 16 registers. This cost is far less than 1-out-of-16 oblivious access. The cost of oblivious access using MUXs and SkipGate to a subset of memory is equal to an oblivious access to a memory with the size of the subset.

Figure 6.6 illustrates the details of the above example where bit \( S \) is the secret bit that separates the branches of computation that each is leading into a different instruction. The address of the source register is computed by a MUX over the two
options in the two instructions: \( \text{MUX}(\$2, \$6, S) = \text{MUX}(0010b, 0110b, S) = 0S10b \). The most and least significant bits are, in either case, zero, and thus it will remain zero in the output. The bit 1 is one in both cases and remains one. The bit 2 is zero if \( S==0 \) and is one if \( S==1 \), and thus this bit is equal to \( S \). Now, let us look at the MUXs in the registers files that fetch the source register. The MUXs connected to the known bit can be evaluated separately by both parties, and thus SkipGate will avoid garbling/evaluating them. Within the MUXs connected to \( S \), only the one that selects between \([\$2]\) and \([\$6]\) remains for garbling, and SkipGate will skip the rest due to non-positive fanout.

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?**
Chapter 7

Evaluation

This chapter is dedicated to the evaluation of the proposed frameworks and algorithms in the thesis. We first discuss the assessment of TinyGarble Garbled Circuit (GC) synthesis flow for combinational and sequential circuits by comparing its result with that of the prior art (Chapter 3 and Chapter 4). We next discuss our experiments for evaluating the SkipGate algorithm and its effect on the sequential circuits. We then present the result of secure evaluation of MIPS processor for Private-Function SFE (PF-SFE) and Secure Function Evaluation (SFE). Finally, we report the result of evaluating the ARM2GC framework. Versions of Section 7.1 and Section 7.2 have been published in 2015 IEEE Symposium on Security and Privacy (S&P) [48]. A version of Section 7.3 of this chapter has been published in 2016 Proceedings of the 53rd Annual Design Automation Conference (DAC) [60].

7.1 TinyGarble GC Synthesis

We use a variety of benchmark functions to evaluate the performance and practicability of TinyGarble GC synthesis of combinational and sequential circuits. In this section, we first describe our experimental setup and metrics for quantifying the performance of TinyGarble. Next, we outline the performance comparison of TinyGarble (with a logic synthesis tool and our custom libraries) on combinational benchmark functions with PCF [25], one of the best known earlier automated methodologies to
generate circuits for garbling. We next demonstrate TinyGarble’s performance in generating sequential circuits for benchmark functions using a standard logic synthesis tool. Next, we report the CPU time for various numbers of sequential cycles to show the effect of memory footprint reducing garbling time. Lastly, we discuss the difference between TinyGarble’s performance using an High-Level Synthesis (HLS) tool (input written in C) and using a conventional logic synthesis tool (input given in Verilog).

We also compare the performance of the commercial logic synthesis tool with an academic open-source tool in Appendix D. We show that in most cases, the performance of the open-source tool is comparable to the commercial tool.

### 7.1.1 Experimental Setup

The circuit generations in this section are done on a system with Linux RedHat Server 5.6, 8 GB of memory, and Intel Xeon X5450 CPU @ 3 GHz. We use another system with Ubuntu 14.10 Desktop, 12.0 GB of memory, and Intel Core i7-2600 CPU @ 3.4GHz to assess the timing performance of the sequential garbling scheme in Section 7.1.6.

Two sets of logic synthesis toolchains are used in our experiments: one commercial and one open-source (Appendix D). Our commercial logic synthesis tool is Synopsys DC 2010.03-SP4 [49]. We also use the Synopsys Library Compiler from the DC package to interpret our custom technology library. In Section 7.1.7, we utilize Xilinx Vivado HLS [41], a commercially available HLS tool whose inputs are in C/C++ programming language. We emphasize that TinyGarble can operate with any commercial or open-source logic synthesis tool, as long as it is capable of performing state-of-the-art logic optimization and mapping algorithms.
7.1.2 Performance Metrics

We use the following metrics to measure the efficiency of TinyGarble for generating garbled circuits:

- **Memory Footprint Efficiency (MFE):**

  \[ MFE = \frac{q_0}{q}, \]

  where \( q_0 \) is the total number of gates in the reference circuit and \( q \) is the total number of gates in the circuit under evaluation. The maximum number of labels that the parties need to store at any point during garbling/evaluation is directly proportional to the number of gates in both sequential and combinational circuits. Thus, the total number of gates is a perfect approximation of the memory footprint.

- **Number of Garbled Tables (#GT):**

  \[ #GT = \#\text{nonXOR} \times cc, \]

  where \( \#\text{nonXOR} \) is the number of non-XOR gates in a circuit and \( cc \) is the number of sequential cycles that the circuit needs to be garbled/evaluated. In Free XOR-based GC schemes, each non-XOR gate requires a garbled table to be generated by the garbler and sent to the evaluator at each sequential cycle. This metric encompasses both the cost of computation (encrypting/decrypting garbled tables) and the cost of communication (transferring garbled tables) in the GC protocol [13].

- **Garbled Tables Difference (GTD) (%)**:  

  \[ GTD = \frac{#GT - #GT_0}{#GT_0} \times 100, \]
where $\#GT_0$ is the total number of garbled tables for the reference circuit, and $\#GT$ is the total number of garbled tables for the circuit under evaluation. When comparing a sequential with a combinational circuit, positive GTD shows an overhead (caused by folding a circuit with an asymmetric loop, see Section 4.3) in total computation and communication time resulting from an excessive number of garbled tables generated in the sequential circuits. However, in general, negative GTD shows improvement in the number of non-XOR gates and generated garbled tables. This effect is the result of logic synthesis optimization.

### 7.1.3 Benchmark Functions

We evaluate TinyGarble’s circuit generation method on various benchmark functions. Several of these functions have been used in previous works, e.g., PCF [25]. In the following, we introduce our benchmark and explain how we fold them into a sequential representation.

**Sum.** This function receives two $N$-bit inputs and outputs an $N$-bit sum. We implement the sum function in $N$ steps of one-bit sums by keeping the carry bit. Thus, it can be folded up to $N$ times without any significant overhead in Number of Garbled Tables ($\#GT$).

**Hamming Distance.** This function receives two $N$-bit inputs and outputs the $\log_2(N)$-bit Hamming distance between them. The Hamming distance between two numbers is the number of positions at which the corresponding bits are different. A possible combinational implementation of the $N$-bit Hamming distance uses a binary tree of adders that sums all 1-bit values from the bit differences to a final Hamming distance consisting of $\log_2(N)$ bits [58]. We cannot easily fold this implementation.
However, we can fold this function into $N$-cycles of one XOR and one $\log_2(N)$-bit adder. This folding causes an overhead compared to the combinational circuit.

*Compare (Millionaires’ Problem).* This function receives two $N$-bit unsigned input values and outputs a greater than signal consisting of one bit that indicates if the first input is greater than the second one. We can realize the comparison function can in $N$ steps of subtraction by keeping the carry bit [84]. Thus, it can be folded up to $N$ times without any significant overhead.

*Multiplication.* This function receives two unsigned $N$-bit inputs and outputs their unsigned $N$-bit product. The multiplication function consists of $N$ additions and shifts. The shift operations result in an asymmetric structure in the circuit. Thus, folding it up to $N$ times may increase the overhead.

*Matrix Multiplication.* This function receives two $N \times N$ matrices consisting of 32-bit unsigned numbers and outputs an $N \times N$ matrix equal to the product of the input matrices. The $N \times N$ matrix multiplication function consists of three $N$-cycle nested loops with a symmetric structure. It can be folded up to $N^3$ times without any significant overhead.

*AES-128.* This function receives a 128-bit plaintext and 128-bit round keys and outputs a 128-bit ciphertext based on the Rijndael algorithm. The AES-128 function consists of ten rounds with almost symmetric structure. Ideally, it can be folded up to 10 times without any significant overhead.

*SHA3.* This function receives 576-bit inputs and outputs a 1600-bit number equal to the SHA3 hash of the input. We implement the Keccak-f permutations [1600] procedure for realizing this function. The SHA3 function consists of 24 steps, each with a symmetric structure. It can be folded 24 times without any significant overhead.
7.1.4 Combinational Garbled Circuit

To show the performance gain of using our custom libraries, we compare TinyGarble combinational circuits with circuits reported in PCF [25]. We choose PCF because, among the automated GC tools available at the time, it shows better results for most of the benchmark functions. In some other work like FastGC [15], a number of benchmark circuits have been more aggressively improved (compared to PCF) using ad-hoc and mostly manual optimizations, but without a generalizable methodology.

Table 7.1 shows the comparison. We compute the garbled tables difference GTD (see Section 7.1.2) of various benchmark functions using circuits reported in PCF as the reference (GTD_{PCF}). We can see that the combinational circuits generated by TinyGarble have non-positive GTD_{PCF} which means that the number of garbled tables is less than or equal to that of PCF circuits. We also compare the memory footprint by computing the memory footprint efficiency MFE with PCF as the reference (MFE_{PCF}). We observe that MFE_{PCF} is larger than 1 (up to 9.3) that means even without using sequential circuits, TinyGarble GC synthesis can reduce the memory footprint can by almost an order of magnitude.

In the case of Hamming distance, TinyGarble shows, on average, 80% improvement in the number of garbled tables. Another automated tool CBMC-GC [26] reports better result compared to PCF for Hamming 160 (non-XOR 4,738, total gates 20,356). However, TinyGarble shows 66% improvement in the number of garbled tables compared to CBMC-GC in the case of 256-bit and 1024-bit Multiplication, and 8 × 8 and 16 × 16 Matrix Multiplication, because of the huge (impractical) sizes, Synopsys DC was unable to generate the entire combinational circuit. Please note that Synopsys DC is a tool developed for commercial applications. The real-life applications are almost always written sequentially, otherwise, the design would not be scalable or...
Table 7.1: Comparison of TinyGarble combinational circuits with PCF. In the case of AES 128, we compare the result with Frigate [27].

<table>
<thead>
<tr>
<th>Function</th>
<th>PCF[25] (*Frigate[27])</th>
<th>TinyGarble Combinational</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-XOR</td>
<td>Total gates</td>
<td>Non-XOR</td>
</tr>
<tr>
<td>Sum 128</td>
<td>345</td>
<td>1,443</td>
<td>127</td>
</tr>
<tr>
<td>Sum 256</td>
<td>721</td>
<td>2,951</td>
<td>255</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>2,977</td>
<td>11,999</td>
<td>1,023</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>880</td>
<td>4,368</td>
<td>158</td>
</tr>
<tr>
<td>Hamming 1600</td>
<td>6,375</td>
<td>32,912</td>
<td>1,597</td>
</tr>
<tr>
<td>Hamming 16000</td>
<td>97,175</td>
<td>389,312</td>
<td>15,994</td>
</tr>
<tr>
<td>Compare 16384</td>
<td>32,229</td>
<td>97,733</td>
<td>16,384</td>
</tr>
<tr>
<td>Mult 64</td>
<td>24,766</td>
<td>105,880</td>
<td>3,925</td>
</tr>
<tr>
<td>Mult 128</td>
<td>100,250</td>
<td>423,064</td>
<td>16,046</td>
</tr>
<tr>
<td>Mult 256</td>
<td>400,210</td>
<td>1.66E+06</td>
<td>-</td>
</tr>
<tr>
<td>Mult 1024</td>
<td>6.37E+06</td>
<td>2.56E+07</td>
<td>-</td>
</tr>
<tr>
<td>MatxMult 3x3</td>
<td>27,369</td>
<td>92,961</td>
<td>27,369</td>
</tr>
<tr>
<td>MatxMult 5x5</td>
<td>127,225</td>
<td>433,475</td>
<td>127,225</td>
</tr>
<tr>
<td>MatxMult 8x8</td>
<td>522,304</td>
<td>1.78E+06</td>
<td>-</td>
</tr>
<tr>
<td>MatxMult 16x16</td>
<td>4.19E+06</td>
<td>1.43E+07</td>
<td>-</td>
</tr>
<tr>
<td>AES 128* [27]</td>
<td>10,383</td>
<td>34,889</td>
<td>6,400</td>
</tr>
<tr>
<td>SHA3 1600</td>
<td>-</td>
<td>-</td>
<td>38,400</td>
</tr>
</tbody>
</table>
even amenable to offline compilation onto a hardware circuit (see Section 3.7). We emphasize that our sequential circuit ($cc > 1$) provides the same functionality while having a very small memory footprint compared with the reference circuit.

**Comparison with Hand-Optimized Circuits**

The netlists generated by the automated flow of TinyGarble show similar performance as the hand-optimized netlists in many cases. For example, [84] describe an $N$-bit sum circuit with $5N$ gates of which $N$ gates are non-XOR and an $N$-bit comparison circuit with $4N$ gates of which $N$ gates are non-XOR. The circuits generated by TinyGarble have about the same number of gates for these two functions. Note that one can always add any hand-optimized module to the synthesis library of TinyGarble.

**7.1.5 Sequential Garbled Circuit**

As described in Chapter 4, the user has the degree of freedom to fold a combinational circuit and convert it to a sequential one to reduce the memory footprint. $cc$ denotes the number of sequential cycles required to garble/evaluate the circuit. This value demonstrates the amount of folding that we performed before the circuit is input to the logic synthesis tool. The user defines the value of $cc$ and writes her input function in an Hardware Description Language (HDL) or a higher-level language such that the function is evaluated in $cc$ sequential cycles.

We use Memory Footprint Efficiency (MFE), to assess the reduction in memory requirement. We use TinyGarble combinational circuits ($cc = 1$) as the reference. The ideal MFE for a circuit with $cc$ sequential cycles is $cc$. We also compare the memory footprints of sequential circuits with combinational circuits reported in PCF ($MFE^{PCF}$).
As explained in Section 4.3, using sequential circuits may introduce some overhead on the total number of garbled tables. To assess this overhead, we compute the GTD of the sequential circuit using TinyGarble combinational circuits as the reference. The ideal GTD is 0%, which means that the total number of garbled tables should be equal to those for a functionally equivalent combinational circuit. We also compare the number of garbled tables of sequential circuits with combinational circuits reported in PCF (GTD_{PCF}) to show that even with the incurred overhead, the number of garbled tables for sequential circuits is still less than that of PCF for most cases.

Table 7.2 demonstrates the number of total gates, non-XOR gates, MFE, GTD, MFE_{PCF}, and GTD_{PCF} of the benchmark circuits for various input widths. MFE, GTD are computed with TinyGarble combinational circuits (with cc = 1) as the reference. MFE_{PCF} and GTD_{PCF} use the circuits reported in PCF as the reference. In the case of AES 128, we compare our implementation with the circuit reported in Frigate [27] because PCF did not report it directly.

We provide a few highlights from Table 7.2. TinyGarble can decrease the size of the sum of two 1024-bit numbers by 1,022.8 times (i.e., more than three orders of magnitude) without affecting the number of garbled tables (GTD) compared with its combinational circuit. For Hamming 16000, TinyGarble can reduce the memory footprint by 7,345.5 times (i.e., about four orders of magnitude) while reducing the number of garbled tables by 47.3% in comparison with the circuit reported in PCF. In the case of Mult 1024, TinyGarble shrinks the memory footprint by a factor of 2,504.4 while minimizing the number of garbled tables by 79.4% when compared with the result in PCF. For a 16 × 16 matrix multiplication, a 4,434.1 more compact TinyGarble solution with 6% less garbled tables compared with PCF is available. By folding AES-128 10 times, we reduce the total number of gates by a factor of 9.2
Table 7.2: Comparison of TinyGarble sequential circuits with PCF and TinyGarble combinational circuits. In the case of AES 128, we compare the result with Frigate [27].

<table>
<thead>
<tr>
<th>Function</th>
<th>TinyGarble Sequential</th>
<th>Comparison with PCF [25]</th>
<th>Comparison with Combinational</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cc</td>
<td>Non-XOR</td>
<td>Total gates</td>
</tr>
<tr>
<td>Sum 128</td>
<td>128</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Sum 256</td>
<td>256</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>1,024</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>32</td>
<td>10</td>
<td>41</td>
</tr>
<tr>
<td>Hamming 1600</td>
<td>320</td>
<td>13</td>
<td>47</td>
</tr>
<tr>
<td>Hamming 16000</td>
<td>3,200</td>
<td>16</td>
<td>53</td>
</tr>
<tr>
<td>Compare 16384</td>
<td>16,384</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Mult 64</td>
<td>16</td>
<td>316</td>
<td>561</td>
</tr>
<tr>
<td>Mult 128</td>
<td>32</td>
<td>636</td>
<td>1,137</td>
</tr>
<tr>
<td>Mult 256</td>
<td>64</td>
<td>1,276</td>
<td>2,539</td>
</tr>
<tr>
<td>Mult 1024</td>
<td>256</td>
<td>5,116</td>
<td>10,219</td>
</tr>
<tr>
<td>MatxMult 3x3</td>
<td>27</td>
<td>961</td>
<td>3,227</td>
</tr>
<tr>
<td>MatxMult 5x5</td>
<td>125</td>
<td>961</td>
<td>3,227</td>
</tr>
<tr>
<td>MatxMult 8x8</td>
<td>512</td>
<td>961</td>
<td>3,227</td>
</tr>
<tr>
<td>MatxMult 16x16</td>
<td>4,096</td>
<td>961</td>
<td>3,227</td>
</tr>
<tr>
<td>AES 128</td>
<td>10</td>
<td>640</td>
<td>3,810</td>
</tr>
<tr>
<td>SHA3 1600</td>
<td>24</td>
<td>1,668</td>
<td>6,788</td>
</tr>
</tbody>
</table>
compared to the Frigate circuit and the number of non-XOR gates by 38.4%. Observe that the savings are typically more for larger bit-widths while extreme foldings can introduce an increased overhead in the number of garbled tables due to the resulting asymmetry.

Because of the TinyGarble superior scalability, we can implement functions that have never been reported before, such as SHA-3, which can be represented using 344,059 and 6,788 gates respectively.

### 7.1.6 Effect of Folding on Garbling Time

So far, we have only reported the overhead regarding the number of garbled tables (GTD) that is a function of the number of non-XOR gates. As explained in [16], if we see garbling as a cryptographic primitive, its computation time (without considering communication) will also be interesting. In practice, smaller circuits which can fit entirely in the processor cache result in fewer cache misses and therefore, consume less CPU clock cycles for garbling. To better observe the impact of cache speed-up for the compact circuits resulting from TinyGarble, Figure 7.1 depicts the CPU Time (left y-axis) and the memory footprint of wire labels (right y-axis) versus \( cc \) (x-axis) for the 32,768-bit Sum function. As mentioned earlier, the memory footprint is directly proportional to the total number of gates in the sequential circuit.

This experiment is done using TinyGarble GC engine, our implementation of sequential GC protocol, that includes using Free XOR, Row Reduction, and Fixed-key Block-cipher, and Half Gate garbling techniques (see Appendix B for TinyGarble GC engine). We use an Intel Core i7 CPU @ 3.40GHz which supports Intel AES New Instructions (AES-NI) instruction set. We measure the CPU cycle as the average of 10,000 trials using RDTSC instruction. For security parameter \( k = 128 \) (the bit-
width of wire label, see Section 2.2.3), we store 128-bit per label. For garbling, we store two labels, two 32-bit input indexes, and an 8-bit gate-type per gate. Thus, the memory footprint is approximately 328-bit per gate in garbling operation. Folding the circuit by a factor of $cc \in [1 : 32,768]$ constantly decreases the memory footprint while the computation effort remains almost constant. Interestingly, as can be seen from the figure, the number of CPU cycles sharply decreases by $1.6 \times$ just when we fold four times ($cc = 4$) compared to $cc = 1$. The sudden decrease is because, for $cc \geq 4$, the memory space required for garbling fits completely in the CPU cache. The minimum CPU cycle per gate happens at $cc = 2,048$ for 3.2 KB memory footprint. This effect signifies the fact that even for large functions, we can use the sequential approach to fit the corresponding memory space requirement into the cache and avoid the penalty of cache misses, thus achieving a significant reduction in garbling time.
Table 7.3: Comparison of performance of the circuits generated using C input to HLS and a direct Verilog input to the logic synthesis tool.

<table>
<thead>
<tr>
<th>Function</th>
<th>c</th>
<th>C→Verilog</th>
<th>Verilog</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Non-XOR</td>
<td>Total gates</td>
<td>Non-XOR</td>
</tr>
<tr>
<td>16384-bit Compare</td>
<td>1,024</td>
<td>51</td>
<td>70</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4,096</td>
<td>15</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>6</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>160-bit Hamming</td>
<td>1</td>
<td>1,264</td>
<td>2,142</td>
<td>371</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>91</td>
<td>146</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>45</td>
<td>71</td>
<td>10</td>
</tr>
<tr>
<td>1024-bit Sum</td>
<td>1</td>
<td>3,067</td>
<td>5,115</td>
<td>1,023</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>195</td>
<td>292</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>1,024</td>
<td>9</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

7.1.7 High-level Synthesis Tools

The design automation community has been working on tools that work with higher-level languages and abstractions than Hardware Description Language (HDL). While a host of commercial and academic High-Level Synthesis (HLS) tools are available [41, 42, 53, 40], we selected the Xilinx Vivado HLS for compiling C code to HDL which can then be synthesized using a conventional logic synthesis tool. The HLS engine in the Vivado suite is built upon the xPilot project [55].

Table 7.3 demonstrates a comparison between the performance of the circuits generated using C input to the HLS tool (C→Verilog) and a direct Verilog input. As can be seen from the table, the resulting memory footprint could increase by a factor between 1 and 4, while the number of garbled tables varies in a range of 3 to 9 times.
It is well known that writing the HDL level code which contains the time information and more detailed structural/behavioral description would yield much more efficient circuits than the code written in a higher-level language.

7.2 Evaluation of MIPS for PF-SFE

We implement the general purpose processor for PF-SFE using the single-cycle MIPS-I architecture where one party provide function description in assembly, and the other provides the data. Support of sequential circuits in TinyGarble enables us to use the MIPS circuit description in Plasma project [77] without major modifications. In the following, we provide the result of MIPS implementation and its number of non-XOR and total gates. Lastly, we present implementation of Hamming distance with variable input length as a benchmark of private function application on MIPS. We implement more benchmark functions and compare their results with MIPS for Secure Function Evaluation (SFE) in Section 7.3

7.2.1 MIPS Implementation

We used TinyGarble to generate the netlist for the MIPS sequential circuit. Table 7.4 shows the total number of gates and non-XOR gates for each module of the MIPS processor with $64 \times 32$bit DM and IM. The sum of non-XORs for each module is 14,997. However, when the modules are combined to form the entire MIPS processor, the synthesis tool optimizes the circuit and reduces the total number of non-XOR gates by 14.95% to reach 12,755. The memory footprint for storing labels during garbling MIPS is approximately the size of two labels times the total number of gates which is $2 \times 128 \times 31,719$bit = 991 KB for label bit-width $k = 128$. The communication load between parties for invocation of one instruction (one sequential
cycle) is approximately the size of two labels times the number of non-XOR gates which is \(2 \times 128 \times 12,755\text{bit} = 398\text{ KB}\) with the Half Gate optimization.

### 7.2.2 Benchmark Function

We implemented the Hamming distance function as a proof-of-concept for our secure MIPS for PF-SFE. It counts the number of different elements in two arrays \(A\) and \(B\) with variable length \(l\). For the hand-optimized assembly code shown in Figure 7.2, the function requires at most \(7 + 9l\) sequential cycles (instructions) to evaluate. Thus, based on Table 7.4, this function requires overall \(12,755 \times (7 + 9l)\) non-XOR gates. It has only 16 instructions and is stored in \(16 \times 32\text{bit}\) of the IM. The function requires that \(l, A,\) and \(B\) are stored in addresses \(0, [2 : l + 1]\), and \([l + 2 : 2l + 1]\) of Data Memory (DM) respectively. It will store the Hamming distance of \(A\) and \(B\) in address 1.

### 7.3 Evaluation of MIPS for SFE

In this section, we present the evaluation of our approach for high-level SFE framework using MIPS processor. We first provide the experimental setup and benchmark function used for evaluating the framework in Section 7.3.1 and Section 7.3.2. Next, we present the synthesis result of MIPS processor for various ISAs in Section 7.3.3. Lastly, we discuss the performance of our hardware GC evaluator for the various ISAs and its comparison with the previous work in Section 7.3.4.

#### 7.3.1 Experimental Setup

We create different instances of a single-cycle MIPS architecture with specific, restricted, and full ISA to support a trade-off between efficiency and privacy. The
Table 7.4: The number of total gates and non-XOR gates in the MIPS implementation. The global optimization of TinyGarble reduces the overall number of gates compared to that of the sum of individual modules.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Total gates</th>
<th>Non-XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>509</td>
<td>470</td>
</tr>
<tr>
<td>Bus</td>
<td>603</td>
<td>590</td>
</tr>
<tr>
<td>ALU</td>
<td>651</td>
<td>346</td>
</tr>
<tr>
<td>Shifter</td>
<td>1,362</td>
<td>1,092</td>
</tr>
<tr>
<td>Mult</td>
<td>2,147</td>
<td>1,792</td>
</tr>
<tr>
<td>Reg File</td>
<td>8,880</td>
<td>3,023</td>
</tr>
<tr>
<td>IM</td>
<td>6,048</td>
<td>2,016</td>
</tr>
<tr>
<td>DM</td>
<td>13,779</td>
<td>5,423</td>
</tr>
<tr>
<td>PC</td>
<td>309</td>
<td>245</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>34,288</strong></td>
<td><strong>14,997</strong></td>
</tr>
<tr>
<td><strong>MIPS</strong></td>
<td><strong>31,719</strong></td>
<td><strong>12,755</strong></td>
</tr>
<tr>
<td><strong>Global optimization</strong></td>
<td>7.49%</td>
<td>14.95%</td>
</tr>
</tbody>
</table>
result of full ISA MIPS is presented in previous section (Section 7.2) and is used here for comparison. The different MIPS instances are synthesized using Synopsys DC H-2013.03-SP4 to generate optimized sequential Boolean circuits. We then evaluate these circuits using our hardware GC evaluator implemented in Vivado 2014.4.1 on a Xilinx Virtex-7 Field-Programmable Gate Array (FPGA).

7.3.2 Benchmark Function

As benchmark functions, we used Hamming distance, Private Set Intersection (PSI), and AES functions. We compile these functions from high-level C to MIPS binary using a MIPS cross-compiler. For some functions, assembly code manipulation allows reducing the number of sequential cycles required. To assure correctness of both functions and ISA under test, we simulate the resulting binary file using the ModelSim simulator and calculate the number of required sequential cycles to compute each of the functions, reported in Table 7.5, for accurate performance measurements. For Hamming distance, the number of cycles depends on the size of the input strings. In the PSI function, we compute a variant of PSI called PSI Cardinality (PSI-CA) where only the number of common elements is revealed. The sets can have different sizes where each element is 32-bit. For AES, we assume that one party holds a 128-bit message and the other party holds eleven round keys each of 128-bit length to avoid unnecessary garbling and evaluation of the round key generation function.

7.3.3 Synthesis of MIPS for Various ISAs

We synthesize the MIPS architecture, shown in Figure 6.1, with Synopsys DC for different ISAs and memory sizes: 32 to 512 32-bit words for instruction and data memories. Generating these Boolean circuits is a one-time process, and we can re-
use the circuits without incurring further compilation costs. Table 7.6 shows the synthesis time and number of non-XOR gates of the Hamming distance-, PSI-, and AES-specific ISA with different sizes of memories. Table 7.7 illustrates the result for restricted and full ISA with various sizes of memories.

- **Application-specific ISA for public functions:** We synthesized three variants of the application-specific ISA where the selected instructions include only the ones used by a particular function, for various memory sizes. We create the application-specific ISA for the three benchmark functions: Hamming distance, PSI, and AES. Instructions required for Hamming distance function are LW, SW, ADD, SUB, XOR, NOP, SLL and BEQ. Instructions required for PSI are LW, SW, ADD, SUB, NOP, SLL, BEQ, BNE and SLT. Instructions required for AES are LW, LB, SW, SB, ADD, SUB, AND, XOR, OR, NOP, SLL, SRL, BEQ, BNE, JAL, JR and SLT.
<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>Synthesis Time (seconds)</th>
<th># of Combinatorial non-XOR gates</th>
<th># of Sequential gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM, IM = 32</td>
<td>19.648 s</td>
<td>6,715</td>
<td>2,021</td>
</tr>
<tr>
<td>DM, IM = 64</td>
<td>31.692 s</td>
<td>9,830</td>
<td>3,046</td>
</tr>
<tr>
<td>DM, IM = 128</td>
<td>62.212 s</td>
<td>16,062</td>
<td>5,095</td>
</tr>
<tr>
<td>DM, IM = 256</td>
<td>167.398 s</td>
<td>28,493</td>
<td>9,192</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>589.186 s</td>
<td>53,374</td>
<td>17,385</td>
</tr>
</tbody>
</table>

Hamming distance-specific ISA

PSI-specific ISA

<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>Synthesis Time (seconds)</th>
<th># of Combinatorial non-XOR gates</th>
<th># of Sequential gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM, IM = 32</td>
<td>18.735 s</td>
<td>6,751</td>
<td>2,021</td>
</tr>
<tr>
<td>DM, IM = 64</td>
<td>31.117 s</td>
<td>9,866</td>
<td>3,046</td>
</tr>
<tr>
<td>DM, IM = 128</td>
<td>61.551 s</td>
<td>16,097</td>
<td>5,095</td>
</tr>
<tr>
<td>DM, IM = 256</td>
<td>163.564 s</td>
<td>28,529</td>
<td>9,192</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>591.145 s</td>
<td>53,410</td>
<td>17,385</td>
</tr>
</tbody>
</table>

AES-specific ISA

<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>Synthesis Time (seconds)</th>
<th># of Combinatorial non-XOR gates</th>
<th># of Sequential gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM, IM = 256</td>
<td>169.498 s</td>
<td>32,177</td>
<td>9,214</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>594.047 s</td>
<td>61,570</td>
<td>17,406</td>
</tr>
</tbody>
</table>
Table 7.7: Synthesis results of restricted and full ISA

<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>Synthesis Time (seconds)</th>
<th># of Combinatorial non-XOR gates</th>
<th># of Sequential gates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU&amp;Shift ISA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM, IM = 32</td>
<td>21.681 s</td>
<td>9,676</td>
<td>2,046</td>
</tr>
<tr>
<td>DM, IM = 64</td>
<td>34.588 s</td>
<td>12,702</td>
<td>3,070</td>
</tr>
<tr>
<td>DM, IM = 128</td>
<td>65.873 s</td>
<td>19,694</td>
<td>5,118</td>
</tr>
<tr>
<td>DM, IM = 256</td>
<td>170.974 s</td>
<td>34,071</td>
<td>9,214</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>593.945 s</td>
<td>66,238</td>
<td>17,406</td>
</tr>
<tr>
<td><strong>ALU-only ISA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM, IM = 32</td>
<td>19.599 s</td>
<td>8,136</td>
<td>2,046</td>
</tr>
<tr>
<td>DM, IM = 64</td>
<td>31.970 s</td>
<td>11,696</td>
<td>3,070</td>
</tr>
<tr>
<td>DM, IM = 128</td>
<td>62.599 s</td>
<td>18,816</td>
<td>5,118</td>
</tr>
<tr>
<td>DM, IM = 256</td>
<td>164.894 s</td>
<td>33,041</td>
<td>9,214</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>598.986 s</td>
<td>65,183</td>
<td>17,406</td>
</tr>
<tr>
<td><strong>Full ISA (Section 7.2)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM, IM = 32</td>
<td>38.331 s</td>
<td>13,257</td>
<td>2,110</td>
</tr>
<tr>
<td>DM, IM = 64</td>
<td>50.446 s</td>
<td>16,818</td>
<td>3,134</td>
</tr>
<tr>
<td>DM, IM = 128</td>
<td>82.863 s</td>
<td>23,899</td>
<td>5,182</td>
</tr>
<tr>
<td>DM, IM = 256</td>
<td>189.157 s</td>
<td>38,118</td>
<td>9,278</td>
</tr>
<tr>
<td>DM, IM = 512</td>
<td>616.750 s</td>
<td>69,423</td>
<td>17,470</td>
</tr>
</tbody>
</table>
• **Restricted ISA for semi-private functions:** We synthesized two variants of the restricted ISA: one without the Mult/Div unit and another without Mult/Div and Shift units. Since the difference between the two depends mainly on reducing the control logic and select lines of multiplexers, the numbers of non-XOR gates for both are different. However, the number of FFs are the same.

• **Full ISA for private functions:** This variant is an extension to the result reported in Section 7.2 for different memory sizes.

### 7.3.4 Performance of the Hardware GC Evaluator

#### Area

Table 7.8 shows the resource allocation and utilization of our hardware GC evaluator on a Xilinx Virtex-7 FPGA. Note that the FPGA utilization does not vary for different memory sizes and instances of the MIPS processor since the evaluator logic remains unaltered. For various memory sizes and ISA instances, only the non-XOR gate count varies. This only impacts the garbled labels and tables memory which significantly affects the off-chip memory utilized for storing the garbled tables, and the Block Random-Access Memory (BRAM) resources utilization only to a small extent.

#### Performance

Table 7.9 presents the runtime required to evaluate MIPS for one instruction in terms of FPGA clock cycles and μs for application-specific ISAs. Table 7.10 reports the same results for restricted and full ISAs. Our GC evaluator operates at 100MHz on the FPGA. This clock cycle frequency is used to compute an average evaluation runtime of 1.1 FPGA clock cycles per gate for our pipelined GC evaluator which translates
Table 7.8: Resource allocation and utilization of our hardware GC evaluator on a Xilinx Virtex-7 FPGA.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Estimation</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip-Flop</td>
<td>22,035</td>
<td>2.54</td>
</tr>
<tr>
<td>Slice LUT</td>
<td>21,229</td>
<td>4.90</td>
</tr>
<tr>
<td>BRAM</td>
<td>354</td>
<td>24</td>
</tr>
<tr>
<td>BUFG</td>
<td>2</td>
<td>6.25</td>
</tr>
</tbody>
</table>

to an average of 11ns per gate in our FPGA implementation. The reported runtime can be further improved by providing tighter timing constraints.

**Comparison with the Prior Art**

Table 7.11 shows a comparison with other GC evaluator implementations [79, 16]. However, for fairness, we are leveraging GC optimizations that were not available at the time for [79]. We compare with our two implementations, the 21-stage pipelined evaluator, and un-pipelined variant to show the effect of pipelining in improving our performance by a factor of 7.8. Table 7.11 compares our results with interpolated results estimated for other works. Results indicate that our pipelined GC evaluator FPGA implementation takes $51 \times$ fewer clock cycles compared to the fastest software implementation JustGarble [16]. Although the CPU clock frequency (3.0GHz) is $30 \times$ faster than that of our Virtex-7 FPGA (100MHz), our pipelined implementation would still be almost $2 \times$ faster than JustGarble in terms of absolute time. Note that our implementation is just a prototype on a reconfigurable FPGA as opposed to a custom design of Intel AES-NI in CPU. Implementing our hardware GC evalu-
Table 7.9: The performance of evaluating MIPS for application-specific ISAs with different memory sizes at 100MHz clock frequency on FPGA.

<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming distance-ISA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of non-XOR gates</td>
<td>6,715</td>
<td>9,830</td>
<td>16,062</td>
<td>28,493</td>
<td>53,374</td>
</tr>
<tr>
<td>Time per inst. (cc)</td>
<td>7,118</td>
<td>10,813</td>
<td>17,829</td>
<td>30,773</td>
<td>57,644</td>
</tr>
<tr>
<td>Time per inst. (µs)</td>
<td>71.18</td>
<td>108.13</td>
<td>178.29</td>
<td>307.72</td>
<td>576.44</td>
</tr>
<tr>
<td>Avg. Time per gate (cc)</td>
<td>1.06</td>
<td>1.10</td>
<td>1.11</td>
<td>1.08</td>
<td>1.08</td>
</tr>
<tr>
<td>PSI-ISA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of non-XOR gates</td>
<td>6,751</td>
<td>9,866</td>
<td>16,097</td>
<td>28,529</td>
<td>53,410</td>
</tr>
<tr>
<td>Time per inst. (cc)</td>
<td>7,426</td>
<td>10,952</td>
<td>18,029</td>
<td>30,811</td>
<td>57,149</td>
</tr>
<tr>
<td>Time per inst. (µs)</td>
<td>74.26</td>
<td>109.52</td>
<td>180.29</td>
<td>308.11</td>
<td>571.49</td>
</tr>
<tr>
<td>Avg. Time per gate (cc)</td>
<td>1.10</td>
<td>1.11</td>
<td>1.12</td>
<td>1.08</td>
<td>1.07</td>
</tr>
<tr>
<td>AES-specific ISA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of non-XOR gates</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32,177</td>
<td>61,570</td>
</tr>
<tr>
<td>Time per inst. (cc)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>35,717</td>
<td>68,343</td>
</tr>
<tr>
<td>Time per inst. (µs)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>357.17</td>
<td>683.43</td>
</tr>
<tr>
<td>Avg. Time per gate (cc)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.11</td>
<td>1.11</td>
</tr>
</tbody>
</table>
Table 7.10: The performance of evaluating MIPS for restricted and full ISAs with different memory sizes at 100MHz clock frequency on FPGA.

<table>
<thead>
<tr>
<th>Memory Size (words)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU&amp;Shift-ISA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of non-XOR gates</td>
<td>9,676</td>
<td>12,702</td>
<td>19,694</td>
<td>34,071</td>
<td>66,238</td>
</tr>
<tr>
<td>Time per inst. (cc)</td>
<td>10.644</td>
<td>13.972</td>
<td>22.057</td>
<td>36.115</td>
<td>71.537</td>
</tr>
<tr>
<td>Time per inst. (µs)</td>
<td>106.44</td>
<td>139.72</td>
<td>220.57</td>
<td>361.15</td>
<td>715.37</td>
</tr>
<tr>
<td>Avg. Time per gate (cc)</td>
<td>1.10</td>
<td>1.10</td>
<td>1.12</td>
<td>1.06</td>
<td>1.08</td>
</tr>
</tbody>
</table>

| **ALU-only ISA**     |      |      |      |      |      |
| # of non-XOR gates  | 8,136| 11,696| 18,816| 33,041| 65,183|
| Time per inst. (cc) | 8,624| 12,866| 21,074| 35,684| 73,657|
| Time per inst. (µs) | 86.24| 128.66| 210.74| 356.84| 736.57|
| Avg. Time per gate (cc) | 1.06 | 1.10 | 1.12 | 1.08 | 1.13 |

| **Full ISA (Section 7.2)** |      |      |      |      |      |
| # of non-XOR gates  | 13,257| 16,818| 23,899| 38,118| 69,423|
| Time per inst. (cc) | 14,848| 18,668| 25,811| 40,786| 77,060|
| Time per inst. (µs) | 148.48| 186.68| 258.11| 407.86| 770.60|
| Avg. Time per gate (cc) | 1.12 | 1.11 | 1.08 | 1.07 | 1.11 |
Table 7.11: Comparing our GC evaluator implementation with other works’ estimation for MIPS with 64-word memory.

<table>
<thead>
<tr>
<th>Method</th>
<th>Total time (cc)</th>
<th>cc/gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Järvinen et al. (SoC) [79]</td>
<td>37,329,233</td>
<td>2,219.6</td>
</tr>
<tr>
<td>Järvinen et al. (Stand-Alone FPGA) [79]</td>
<td>4,291,954</td>
<td>255.2</td>
</tr>
<tr>
<td>JustGarble (CPU) [16]</td>
<td>948,535</td>
<td>56.4</td>
</tr>
<tr>
<td>Our work w/o pipeline</td>
<td>144,635</td>
<td>8.6</td>
</tr>
<tr>
<td>Our work w/ pipeline</td>
<td>18,500</td>
<td>1.1</td>
</tr>
</tbody>
</table>

ator as ASIC would improve its performance in terms of absolute time even further. Moreover, our implementation is two orders of magnitude faster than the previously fastest hardware implementation of [79].

7.4 Evaluation of ARM2GC

In this section, we present the evaluation result of the SkipGate algorithm and the ARM2GC framework. First, we explain the experimental setup of the evaluation. Next, we discuss the effect of the SkipGate algorithm on removing sequential overhead from the benchmark functions. We next compare the result of the ARM2GC framework and that of TinyGarble GC synthesis and high-level custom compiler approaches of the previous work. We then report the effect of the SkipGate algorithm on the ARM circuit. Lastly, we provide a number of complex functions that has been implemented using the ARM2GC framework.
7.4.1 Experimental Setup

We use Synopsys DC H-2013.03-SP4 [49] along with TinyGarble synthesis flow 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 [85]. The ARM circuit is modified as explained in Section 6.4.2. Synthesizing the ARM processor with Synopsys DC takes few hours. However, the process is done only once for a given memory size, and we can use it 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 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.

7.4.2 Effect of SkipGate on Sequential GC

As described in Chapter 5, the SkipGate algorithm avoids redundant garbling/evaluation of gates in sequential circuits with public wires. In the sequential benchmark circuits reported in Section 7.1.5 for TinyGarble sequential synthesis, 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 7.12, we compare the cost of garbling for circuits generated by TinyGarble with and without applying the SkipGate algorithm. The results without SkipGate are the same as the ones in second and third
columns of Table 7.2 in Section 7.1.5. The second column shows the total number of non-XOR gates to be garbled that is $cc \times \# \text{non-XORs}$ in the sequential circuit. 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 Multiplexer (MUX)s connecting to it) is avoided by SkipGate because both parties know the AES control path in advance. Note that the functions in Table 7.12 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.

### 7.4.3 ARM2GC vs. Logic Synthesis

Table 7.13 compares the cost of garbling of functions devised in Verilog HDL and constructed by the hardware synthesis technique of TinyGarble GC synthesis (see Chapter 3) with functions developed in C and built using the ARM2GC framework. The results of TinyGarble GC synthesis are repeated here from Table 7.1 in Section 7.1.4 for comparison. 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 [15] for Hamming distance realization in C.
Table 7.12: The SkipGate algorithm improvement on sequential circuits generated by TinyGarble (TG) synthesis flow (Section 7.1.5). 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.

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th># of garbled non-XOR</th>
<th># of skipped non-XOR</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TG (Section 7.1.5)</td>
<td>SkipGate</td>
<td></td>
</tr>
<tr>
<td>Sum 32</td>
<td>32</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>1,024</td>
<td>1,023</td>
<td>1</td>
</tr>
<tr>
<td>Compare 32</td>
<td>32</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>Compare 16,384</td>
<td>16,384</td>
<td>16,384</td>
<td>0</td>
</tr>
<tr>
<td>Hamming 32</td>
<td>160</td>
<td>145</td>
<td>15</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>1,120</td>
<td>1,092</td>
<td>28</td>
</tr>
<tr>
<td>Hamming 512</td>
<td>4,608</td>
<td>4,563</td>
<td>45</td>
</tr>
<tr>
<td>Mult 32</td>
<td>2,048</td>
<td>2,016</td>
<td>32</td>
</tr>
<tr>
<td>MatrixMult3x3 32</td>
<td>25,947</td>
<td>25,668</td>
<td>279</td>
</tr>
<tr>
<td>MatrixMult5x5 32</td>
<td>120,125</td>
<td>119,350</td>
<td>775</td>
</tr>
<tr>
<td>MatrixMult8x8 32</td>
<td>492,032</td>
<td>490,048</td>
<td>1,984</td>
</tr>
<tr>
<td>AES 128†</td>
<td>15,807</td>
<td>6,400</td>
<td>9,407</td>
</tr>
<tr>
<td>SHA3 256</td>
<td>40,032</td>
<td>38,400</td>
<td>1,632</td>
</tr>
</tbody>
</table>

†The missing key expansion module to AES 128 of Table 7.2 in Section 7.1.5 is added here.
Table 7.13: The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC to TinyGarble’s GC synthesis (Chapter 3).

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th>TinyGarble (Verilog) (Section 7.1.4)</th>
<th>ARM2GC (C)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum 32</td>
<td>31</td>
<td>31</td>
<td>0.0%</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>1,023</td>
<td>1,023</td>
<td>0.0%</td>
</tr>
<tr>
<td>Compare 32</td>
<td>32</td>
<td>32</td>
<td>0.0%</td>
</tr>
<tr>
<td>Compare 16,384</td>
<td>16,384</td>
<td>16,384</td>
<td>0.0%</td>
</tr>
<tr>
<td>Hamming 32</td>
<td>160</td>
<td>57</td>
<td>-64.4%</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>1,120</td>
<td>247</td>
<td>-77.9%</td>
</tr>
<tr>
<td>Hamming 512</td>
<td>4,608</td>
<td>1,012</td>
<td>-78.0%</td>
</tr>
<tr>
<td>Mult 32</td>
<td>1,023</td>
<td>993</td>
<td>-2.9%</td>
</tr>
<tr>
<td>MatrixMult3x3 32</td>
<td>27,369</td>
<td>27,369</td>
<td>0.0%</td>
</tr>
<tr>
<td>MatrixMult5x5 32</td>
<td>120,125</td>
<td>127,225</td>
<td>5.9%</td>
</tr>
<tr>
<td>MatrixMult8x8 32</td>
<td>492,032</td>
<td>522,304</td>
<td>6.2%</td>
</tr>
<tr>
<td>AES 128†</td>
<td>6,400</td>
<td>6,400</td>
<td>0.0%</td>
</tr>
<tr>
<td>SHA3 256</td>
<td>38,400</td>
<td>37,760</td>
<td>-1.7%</td>
</tr>
</tbody>
</table>

†The missing key expansion module to AES 128 of Table 7.1 in Section 7.1.4 is added here.
7.4.4 ARM2GC vs. GC Frameworks Supporting High-levelLanguages

Table 7.14 reports the cost of garbling for the benchmark functions constructed by the prior-art GC frameworks including (TinyGarble HLS synthesis in Chapter 3 and [31, 27]). Table 7.15 reports the comparison with previous work on garbled processors approaches (MIPS for PF-SFE in Section 6.1 and SFE in Section 6.3 and [78]). We show the respective programming language between the 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. In all cases, ARM2GC outperforms the earlier frameworks in terms of garbling cost. For example, ARM2GC results 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 [31], TinyGarble HLS approach (Section 7.1.7), PF-SFE MIPS (Section 7.2), SFE MIPS (Section 7.3 and [78]), and Frigate [27] respectively. ARM2GC also results in 38.3% less non-XOR gates compared to Frigate [27] for AES function.

7.4.5 Effect of SkipGate on ARM

Table 7.16 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 sequential cycles. It means with the conventional GC protocol, garbling/evaluation of $1,909 \times 126,755 = 241,975,295$ non-XORs is required, while 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
Table 7.14: The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC with previous high-level custom compiler methods and Tiny-Garble HLS approach.

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th>ANSI-C (C) [31]</th>
<th>TG HLS (C → Verilog) Section 7.1.7</th>
<th>Frigate (C) [27]</th>
<th>ARM2GC (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum 32</td>
<td>32</td>
<td>288</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>-</td>
<td>9,216</td>
<td>1,025</td>
<td>1,023</td>
</tr>
<tr>
<td>Compare 32</td>
<td>65</td>
<td>102</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>Compare 16,384</td>
<td>-</td>
<td>52,224</td>
<td>16,386</td>
<td>16,384</td>
</tr>
<tr>
<td>Hamming 32</td>
<td>601</td>
<td>253</td>
<td>-</td>
<td>57</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>3,003</td>
<td>1,264</td>
<td>719</td>
<td>247</td>
</tr>
<tr>
<td>Hamming 512</td>
<td>9,610</td>
<td>4,045</td>
<td>-</td>
<td>1,012</td>
</tr>
<tr>
<td>Mult 32</td>
<td>1,741</td>
<td>-</td>
<td>995</td>
<td>993</td>
</tr>
<tr>
<td>MatrixMult3x3 32</td>
<td>47,583</td>
<td>-</td>
<td>-</td>
<td>27,369</td>
</tr>
<tr>
<td>MatrixMult5x5 32</td>
<td>220,825</td>
<td>-</td>
<td>128,252</td>
<td>127,225</td>
</tr>
<tr>
<td>MatrixMult8x8 32</td>
<td>905,728</td>
<td>-</td>
<td>-</td>
<td>522,304</td>
</tr>
<tr>
<td>SHA3 256</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>37,760</td>
</tr>
<tr>
<td>AES 128</td>
<td>-</td>
<td>-</td>
<td>10,383</td>
<td>6,400</td>
</tr>
</tbody>
</table>
Table 7.15: The number of garbled non-XOR gates for the benchmark functions.
Comparing ARM2GC with the previous garbled processor approaches.

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th>PF-SFE MIPS (C) (Section 7.2)</th>
<th>SFE MIPS (C) [78]</th>
<th>SFE MIPS (C) (Section 7.3)</th>
<th>ARM2GC (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1,023</td>
</tr>
<tr>
<td>Compare 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>Compare 16,384</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16,384</td>
</tr>
<tr>
<td>Hamming 32</td>
<td>3,762,725</td>
<td>481,000</td>
<td>2,860,590</td>
<td>57</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>18,456,485</td>
<td>-</td>
<td>14,302,950</td>
<td>247</td>
</tr>
<tr>
<td>Hamming 512</td>
<td>58,864,325</td>
<td>49,600,000</td>
<td>45,769,440</td>
<td>1,012</td>
</tr>
<tr>
<td>Mult 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>993</td>
</tr>
<tr>
<td>MatrixMult3x3 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>27,369</td>
</tr>
<tr>
<td>MatrixMult5x5 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>127,225</td>
</tr>
<tr>
<td>MatrixMult8x8 32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>522,304</td>
</tr>
<tr>
<td>SHA3 256</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>37,760</td>
</tr>
<tr>
<td>AES 128</td>
<td>-</td>
<td>-</td>
<td>198,789,506</td>
<td>6,400</td>
</tr>
</tbody>
</table>
Table 7.16: The SkipGate algorithm improvement on the ARM sequential circuit.

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th># of non-XOR gates</th>
<th>Improvement (1000X)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conventional GC+ARM</td>
<td>ARM2GC</td>
</tr>
<tr>
<td>Sum 32</td>
<td>3,817,680</td>
<td>31</td>
</tr>
<tr>
<td>Sum 1024</td>
<td>76,483,260</td>
<td>1,023</td>
</tr>
<tr>
<td>Compare 32</td>
<td>4,072,192</td>
<td>130</td>
</tr>
<tr>
<td>Compare 16,384</td>
<td>1,047,095,280</td>
<td>16,384</td>
</tr>
<tr>
<td>Hamming 32</td>
<td>67,063,912</td>
<td>57</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>242,931,704</td>
<td>247</td>
</tr>
<tr>
<td>Hamming 512</td>
<td>863,559,216</td>
<td>1,012</td>
</tr>
<tr>
<td>Mult 32</td>
<td>4,199,448</td>
<td>993</td>
</tr>
<tr>
<td>MatrixMult3x3 32</td>
<td>72,790,432</td>
<td>27,369</td>
</tr>
<tr>
<td>MatrixMult5x5 32</td>
<td>286,071,488</td>
<td>127,225</td>
</tr>
<tr>
<td>MatrixMult8x8 32</td>
<td>1,079,894,416</td>
<td>522,304</td>
</tr>
<tr>
<td>SHA3 256</td>
<td>29,354,783,052</td>
<td>37,760</td>
</tr>
<tr>
<td>AES 128</td>
<td>54,621,701,856</td>
<td>6,400</td>
</tr>
</tbody>
</table>

over the conventional GC without the SkipGate algorithm. The algorithm transforms the impracticable cost of garbling an ARM processor into a 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 non-XOR gates in the ARM circuit.

Comparing the result of Table 7.12 and Table 7.16 shows that the extent of SkipGate’s impact highly depends on the structure of the circuit, as well as the degree of
Table 7.17: The SkipGate algorithm improvement on the ARM sequential circuit for the complex functions.

<table>
<thead>
<tr>
<th>Function (bit)</th>
<th># of non-XOR gates</th>
<th>Improvement (1000X)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conventional GC+ARM</td>
<td>ARM2GC</td>
</tr>
<tr>
<td>Bubble-Sort32 32</td>
<td>1,366,390,620</td>
<td>65,472</td>
</tr>
<tr>
<td>Merge-Sort32 32</td>
<td>981,712,458</td>
<td>540,645</td>
</tr>
<tr>
<td>Dijkstra64 32</td>
<td>1,493,339,886</td>
<td>59,282</td>
</tr>
<tr>
<td>CORDIC 32</td>
<td>228,847,596</td>
<td>4,601</td>
</tr>
</tbody>
</table>

presence of public values in the circuit.

### 7.4.6 Complex Functions

As shown in Table 7.17, we also develop a number of more complex functions with the ARM2GC framework.

**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 on 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 on 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 (CORDIC) function receives a degree and a 2D vector described as 32-bit fixed-point (2-bit decimal and 30-bit fraction), computes trigonometric, hyperbolic or exponential functions according to Universal CORDIC algorithm [86], 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 only required operations are Addition, Shift, and a non-oblivious Lookup table. 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 [87] and required 12,733 and 12,546 non-XOR gates respectively, almost three times more than ARM2GC.
# Hamming distance

# between A and B with length of l

Hamming:

```assembly
lw $9, 0($0)  # load l into $9
sll $9, $9, 2  #$9 = $9*4
addi $2, $0, 8  #$2 := A
add $3, $2, $9  #$3 := B = A + l

# answer; no need to reset
addi $10, $0, 0

# l+=2 to compare with end of A
addi $9, $9, 8

loop:  # done if A==end of A
    beq $2, $9, done
    lw $4, 0($2)  # load *A
    lw $5, 0($3)  # load *B
    xor $6, $4, $5  #$6==0
    beq $6, $0, same  # goto A[i]!=B[i]
    addi $10, $10, 1  # answer++

same:  # A++  # B++
    addi $2, $2, 4
    addi $3, $3, 4
    j loop  # jump back to the top

done:  # store answer
    sw $10, 4($0)

end:  # while(l)
    j end
```

Figure 7.2: Hamming distance assembly code.
Chapter 8

Related Work

In this chapter, we review the related work to this thesis in the literature. First, we classify the prior art for generating a circuit for Yao’s Garbled Circuit (GC) protocol into custom compilers in Section 8.1, optimized libraries in Section 8.2, and garbled processor in Section 8.3. Next, we review the similar work on logic synthesis for other circuit-based Secure Function Evaluation (SFE) protocol in Section 8.4. We then discuss the work on GC implementations with hardware accelerators in Section 8.5, and GC realization on embedded devices in Section 8.6. Lastly, we review the work on securing the GC protocol against a malicious adversary in Section 8.7.

8.1 Compiler for Garbled Circuits

The idea of designing a custom programming language to describe and efficiently compile functions for secure evaluation dates back to Fairplay, the first practical implementation of the GC protocol introduced in 2004 [12]. Fairplay introduced a custom high-level procedural language called Secure Function Definition Language (SFDL). The user has to write their privacy-preserving applications in SFDL and translate them into Fairplay’s circuit description language, Secure Hardware Description Language (SHDL).

Henecka et al. (2010) introduced TASTY compiler [23] which allows combining garbled circuits and Homomorphic encryption. In TASTY, the user can develop an
SFE program in a new domain-specific language to be performed on private data. The compiler then translates the program into a combination of the GC protocol and Homomorphic encryption to securely evaluate the program.

Mood et al. (2012) introduced a new Pseudo-Assembly Language (PAL) for GC circuit generation [30]. They designed a new compiler that translates a program in the PAL into Fairplay SFDL to generate circuits with limited memory budget.

Kreuter et al. (2012) introduced a GC compiler that for the first time addressed the scalability issue of generating and garbling large circuits under the malicious model [29]. Their compiler can produce circuits consisting of billions of gates, e.g., a 4095x4095-bit Edit Distance circuit with almost six Billion gates.

The first GC framework supporting a general-purpose language is presented in [31], which supports ANSI-C. However, it supports only a subset of ANSI-C that is not compatible with many crucial primitives and therefore, not compatible with legacy codes. The main drawback of [31] is compile-time loop unrolling that makes it unscalable with input size.

Kreuter et al. (2013) introduced PCF compiler for the GC protocol [25]. PCF does not unroll loops in the program until the GC evaluation to reduce the memory overhead. The loops have to be marked manually in the high-level language. In contrast to PCF, TinyGarble allows to infer loops automatically and also allows to optimize across multiple sub-circuits.

Franz et al. (2014) introduced CBMC-GC, a new GC compiler in [26]. Their compiler lets users program in C language and translate the programs into a Boolean circuit. The objective of their compiler is to reduce the number of non-XOR gates in the generated Boolean circuits.

Rastogi et al. (2014) proposed WYSTERIA, a new high-level programming lan-
guage for the GC protocol in [36]. WYSTERIA, unlike previous work, allows a combination of local and private computations on the data. WYSTERIA also introduces a new abstraction for secret shared data that support generic n-party secure computations.

Liu et al. (2015) proposed ObliVM framework for secure computation in [19]. They proposed a new domain-specific language that ObliVM compiles into an efficient representation for secure computation. Most significantly, ObliVM allows users to benefit from the sub-linear cost of oblivious array accesses in the GC protocol using Circuit Oblivious Random-Access Machine (ORAM) scheme [88].

Zahur et al. (2015) proposed a new compiler for GC that relies on a standard C compiler [18]. Their compiler lets users program in normal C. They also make their compiler compatible with Path ORAM scheme [89] which reduce the cost of oblivious access to arrays.

Mood et al. (2016) provided a comprehensive study of the state-of-the-art compilers [30, 29, 25, 26, 18, 19] for performing secure function evaluation using high-level languages in [27]. They showed that the majority of such compilers are not thoroughly validated and they reported the observed flaws in six commonly used platforms. As they discussed in their paper, there are severe 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.

Mood et al. (2016) also introduced Frigate, a new C-style language for SFE and the corresponding compiler in the same paper [27]. Frigate supports three different types (uint_t, int_t, and struct_t). The user can add her 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 perform the comparison for a given bit length of \texttt{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 relies 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.

The introduction of a new custom programming language is neither user-friendly nor versatile when compared with a conventional programming language like C. Moreover, the user has to compile her code with a newly designed custom compiler in these works. As a result, the user cannot benefit from the optimizations provided by general-purpose and standard compilers. Furthermore, 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.

### 8.2 Libraries for Garbled Circuits

Instead of compiling circuits, Huang et al. (2011) proposed the FastGC framework that uses a library-based approach where circuits can be programmed and integrated into high-level applications [15].

Another library-based GC framework is VMCrypt proposed by Malka et al. (2011) in [28] to address software modularity and scalability issues in the previous GC frameworks. VMCrypt provides a modular software architecture in Java programming lan-
guage that dynamically constructs and deconstructs sub-circuits. VMCrypt removes sub-circuits from the memory when their computations are done through destructing their objects.

Henecka et al. (2013) extended the FastGC framework in [24] to re-use the same sub-circuits. They also introduce multi-threaded implementation for Oblivious Transfer (OT) and cashing circuit descriptions and network packets.

Demmler et al. (2015) proposed ABY, a library-based framework for secure computation that allows the efficient combination of three secure computation schemes: Arithmetic sharing, Boolean sharing, and Yaos GC [39]. Conversion of the private data between these secure computation schemes is supported using a pre-computed OT.

Library-based approaches suffer from the fact that user has to decompose the function into sub-circuits manually. Thus, the user needs to have a thorough understanding of the circuit description. Whereas, our methods in TinyGarble and ARM2GC are automated approaches and provide more abstraction compared to library-based frameworks.

### 8.3 Garbled Processor

Wang et al. (2016) proposed a GC framework based on MIPS processor in [78]. Their framework 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 execute the code securely. To avoid emulating a large number of instructions supported by the MIPS processor, they perform a data independent static analysis before execution of the program to build a small instruction bank and Arithmetic Logic Unit (ALU) circuit tailored for each
processor cycle. In contrast, our ARM2GC performs this optimization with bit-precision instead of instruction-precision (see Section 6.4). Moreover, this is done in the runtime while the circuit remains the same for each cycle.

To solve the problem of the secure conditional branch, Wang et al. (2016) propose to pad \texttt{nop} instruction to parallel branches so that their lengths become equal [78]. 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 ARM processor which supports conditional execution and can replace these branches with conditional instructions (see Section 6.4). In rare cases where the ARM compiler fails to replace the conditional branch, we adopted their approach in padding the parallel branches with \texttt{nop} instruction. Overall, our evaluation shows that ARM2GC outperforms their MIPS framework, for example by four orders of magnitudes for Hamming distance function, mostly thanks to the SkipGate algorithm and its bit-precision optimization.

### 8.4 Logic Synthesis for Other SFE Circuit-based Protocols

Inspired by TinyGarble’s methodology, Demmler et al. (2015) proposed a method to use industrial logic synthesis tools to optimize both size of and depth of Boolean circuits used in secure computation [38]. The depth of circuit is a crucial factor in the performance of the circuit-based Goldreich-Micali-Wigderson (GMW) protocol [2]. The round complexity of GMW depends on the number of non-XOR gates in the longest path from input to output. In GMW for each layer of non-XOR gates, an OT needs to executed between the parties. Demmler et al. (2015) proposed using timing optimization of logic synthesis tools to reduce the depth of the circuit [38]. They set the area and timing delay of non-XOR gates to a large non-zero value, and those of
XOR gates to zero and then force the synthesis tool to reduce the overall delay and area of the circuit. This approach results in a small and shallow circuit in term of non-XOR gates that in turn, reduces the cost of secure evaluation of the circuit in the GMW protocol. The authors also extend their library of the ABY framework [39] with the new optimized circuits including floating-point operations for the GMW protocol.

8.5 GC Implementations with Hardware Accelerators

The following works provide better performance by implementing garbled circuits in hardware, on Graphics Processing Unit (GPU)s, or using AES New Instructions (AES-NI) available in modern CPUs. These works can benefit from the compact representation generated by TinyGarble.

Järvinen et al. (2010) [79] proposed a generic hardware architecture for GC. They realized two Field-Programmable Gate Array (FPGA) based prototypes: a system-on-a-programmable-chip with access to a hardware cryptographic accelerator targeting smart cards and smart phones, and a stand-alone hardware implementation targeting Application-Specific Integrated Circuits (ASIC).

Several accelerations of GCs using GPUs have been proposed recently. Husted et al. (2013) implemented Yao’s GC using optimizations such as Free XOR, pipelining, and OT extension [90]. Pu et al. (2013) realized dynamic programming based on GC to solve the Edit-Distance and the Smith-Waterman problems [91]. They also used the same optimizations as [90] along with permute-and-encrypt, efficient lookup-table design, and compact circuits [91]. Frederiksen et al. (2013) implemented a secure computation protocol that is secure against malicious adversaries based on a cut-and-choose strategy and an efficient OT extension for two-party computation on
Bellare et al. (2013) propose JustGarble in which they use fixed-key AES for circuit-garbling [16]. They show their implementation using AES-NI can efficiently garble and execute a circuit far faster than any prior report.

Recently, Fang et al. (2017) proposed a generic implementation of the GC protocol on FPGA [93]. They propose a coarse-grained architecture that does not require to be reprogrammed for evaluating a new SFE application.

8.6 GC Implementations on Embedded Devices

Our approach for generating compact circuit representations is also beneficial when performing secure computation on resource-constrained embedded devices such as mobile devices which have a limited amount of main memory. Huang et al. (2011) proposed secure computation on mobile devices using garbled circuits [94]. Moreover, the protocol described in [95], which uses a smart-card installed in the embedded device, can benefit from our more compact circuit representation. In [96, 97], the mobiles no longer need to process circuits anymore as GC generation and evaluation is outsourced to cloud servers.

8.7 Securing GC against Malicious Adversary

Yao’s GC protocol discussed in this thesis is secure against an Honest-But-Curious (HBC) adversary, also known as a semi-honest or passive adversary (see Section 2.2.3). In the HBC model, the parties follow the protocol, but they wish to learn as much as possible about the other party’s private input.

In a malicious model of GC where parties can deviate from the protocol, Alice can garble a faulty circuit instead of the correct one and send it to Bob. Bob is
not able to distinguish the faulty circuit because it is garbled, i.e., encrypted. A straightforward method to make the protocol secure against a malicious adversary is to apply Goldreich-Micali-Wigderson (GMW) compiler to the protocol [2], such that Alice can prove to Bob that she garbled the circuit correctly without revealing the garbling secrets. However, this approach is not practical due to the use of costly generic zero-knowledge proofs in the GMW compiler [44].

Lindell and Pinkas (2007) proposed the first practical and efficient method to secure the GC protocol against malicious adversaries in [44]. They propose to apply a cut-and-choose strategy on the GC protocol. Alice garbles the circuit using the HBC version of the protocol for $s$ times and send the garbled circuits to Bob. $s$ is a statistical security parameter, and the cheating probability is exponentially small with respect to $s$. Bob randomly selects half of the garbled circuits and asks Alice to reveal them to him. Bob can detect if Alice cheated in garbling any of the circuits. If Bob finds out that Alice garbled the majority of the circuits correctly, then he evaluates the rest of them and chooses the most recurring output as the final output. Bob selects the majority instead of terminating the protocol in case of different outputs because Alice can create a faulty circuit that fails given a particular input. Thus, Bob’s termination can reveal his input to Alice.

The cut-and-choose method creates another security challenge that is to make sure that the parties are using the same inputs for the multiple invocations of the garbled circuit. To solve this problem, the authors propose to use a commitment scheme for input such that parties cannot change their input after the start of the protocol. The authors later improve their method by reducing number of commitments at the expense of garbling more circuits to make the overall protocol more efficient [45]. They showed that garbling $s$ circuits and opening half of them result in the cheating
probability of $2^{-0.311s}$. For example, one requires garbling $s = 128$ circuits to achieve $2^{-40}$ cheating probability. Most recently, Lindell (2016) improved the cut-and-chose strategy even further in [98]. He showed applying some twists to the cut-and-chose strategy can achieve $2^{-s}$ probability error given $s$ circuits. This means one requires garbling only $s = 40$ circuits to achieve $2^{-40}$ cheating probability.

Nielsen et al. (2009) proposed LEGO, an alternative method for securing GC in the presence of malicious adversary [46]. The main difference between LEGO and the conventional cut-and-chose methods [44, 99, 45, 100, 98] is that instead of garbling the entire circuit for multiple times, Alice garbles a large number of NAND gates in LEGO. And instead of asking to reveal the entire circuit, Bob asks her to open half of the garbled NAND gates. Bob then constructs a fault-tolerant circuit based on the original circuit using the remaining unrevealed NAND gates and evaluate the circuit. Since in this thesis, we focus on generating a Boolean function with the minimum number of non-XOR gates, LEGO cannot directly enjoy from the improvement of TinyGarble synthesis similar to the conventional cut-and-chose methods that reveal the entire circuit. However, the GC synthesis methodology presented in Chapter 3 may improve the number of NAND gates required to create the fault-tolerate circuit for LEGO.

8.7.1 Discussion

Securing GC using the cut-and-chose strategy incurs a multiplicative factor of $s$ (the statistical security parameter) to the cost of GC compared to the HBC model. Thus, reducing the cost of GC under HBC (one of the objectives of this thesis) results in an immediate reduction in the cost of GC under the malicious adversary as well.

Since in this thesis, we use the GC protocol as a black-box, one can easily apply the
cut-and-chose strategy presented in [98] to secure our frameworks against malicious adversaries. For instance, in the ARM2GC framework described in Section 6.4, the Boolean circuit is the circuit of an ARM processor, and the inputs are parties’ private inputs. Both parties create and agree on a simplified circuit of the ARM processor that is going to be garbled/evaluated using the SkipGate algorithm. Using Lindell’s method, Alice and Bob first commit their inputs [98]. Then for achieving $2^{-40}$ cheating probability, Alice generates 40 garbled circuits from the simplified ARM circuit and sends them to Bob. Bob then asks Alice to reveal half of them to Bob, and if the majority of the garbled circuit are garbled correctly, he evaluates the rest and chooses the most recurring output as the final output. Similarly, one can easily apply this to TinyGarble GC engine to make it secure against the malicious adversary.
Chapter 9

Conclusion

We present TinyGarble, an automated tool that can generate optimized and compact circuits for Yao’s Garbled Circuit (GC) protocol. We are the first to define the circuit generation for GC as a logic synthesis problem and to leverage the powerful and established logic synthesis techniques with our custom objectives. We improve the results of one of the best automated tools for GC generation, PCF [25], by several orders of magnitude. For instance, TinyGarble compacts the 1,024-bit multiplication by 2,504 times, while decreasing the number of non-XOR gates by 80%; we compress the 16,000-bit Hamming distance by a factor of 7,345 times and with 47% less non-XOR gates. Further, TinyGarble can implement functions that have never been reported before, such as SHA-3. We perform extensive benchmarking with both commercial and open source hardware synthesis tools and compare the results.

We introduce a gabled processor as a scalable solution for private function evaluation (PF-SFE). This garbled processor relies on the MIPS architecture and the private function can be compiled using ubiquitous tools, e.g., gcc. We also introduce ARM2GC, a simple-to-use and efficient garbled processor framework based on ARM and TinyGarble. Users can develop secure functions in high-level languages and compile them using standard fully verified ARM cross-compilers. Evaluations on a host of benchmark functions show that the ARM2GC framework achieves efficiency close to that of TinyGarble GC synthesis. The low cost of ARM2GC is made possible by SkipGate, a novel algorithm for the sequential GC. The SkipGate algorithm omits the
communication cost for gates with outputs independent of private data and also the gates not affecting the final output. As a result of SkipGate, only the gates associated with private information in the massive ARM circuit incur communication cost.

Our TinyGarble and ARM2GC frameworks strongly improve the previous results and pave the way for a practical, scalable, and simple-to-use secure computation with many exciting applications. For instance, TinyGarble is an enabling technology for performing GC operations on mobile platforms, which is prohibitively expensive using the prior techniques. And ARM2GC enables users with no or limited knowledge about the GC protocol or Boolean circuit to develop privacy-preserving applications in high-level languages.
Appendix A

Compacting Privacy-Preserving k-Nearest Neighbor Search using Logic Synthesis

In this appendix, we discuss solving privacy-preserving $k$-nearest neighbor ($k$-NN) search using Yao’s Garbled Circuit (GC). We use the TinyGarble GC synthesis method to implement $k$-NN in optimized and compact circuits for GC.

In this appendix, we first introduce the problem of privacy-preserving $k$-NN and the challenge of solving it using Yao’s GC. Next, we provide the related work in the literature about privacy-preserving $k$-NN and methods for generating optimized and compact circuits for $k$-NN. We then explain our approach that relies on TinyGarble sequential circuit synthesis (see Chapter 4). Lastly, we provide the evaluation results and conclude the appendix. A version of this appendix has been published in 2015 Proceedings of the 52nd Annual Design Automation Conference (DAC) [101].

A.1 Introduction and Motivation

The search for similarities has a broad range of applications in data mining, such as finding close matches in images, local features, biological and genome data, and multimedia systems [102]. The most extensively used function for similarity search is the $k$-nearest neighbor ($k$-NN). For a given dataset $S$ of $n$ points in a multi-dimensional space $w$, and a query $q$, the $k$-NN search finds a subset of $S$ with $k$ points that are closest to $q$. Numerous works have focused on the development of efficient $k$-NN search, where the underlying assumption is that the dataset $S$ and the
query point $q$ are public. For example, it is well known that one can dramatically improve the relative search speed of $k$-NN by projecting the data into smaller hash tables [103, 104]. However, such approximation methods are orthogonal to our effort. We focus on the basic textbook $k$-NN search which could also be accelerated with the known approximations.

In a number of scenarios, the data and the query are sensitive, and it is important to maintain privacy while performing the search. This requirement has motivated the development of privacy-preserving $k$-NN search [105]. The existing works assume that the parties each own a private dataset, while the query $q$ is not private; they focus on devising higher-level protocols for privacy-preserving similarity search and proofs of privacy [105, 102]. These works mostly leverage Homomorphic encryption to perform a privacy-preserving task. Homomorphic encryption enciphers the data (plaintext) in such a way that performing a mathematical function on the encrypted information, and then decrypting the result, produces the same answer as performing a similar operation on the plaintext. Since the first fully homomorphic encryption was proposed [106], numerous protocol-level and implementation-level advancements have been made. Even so, the implementations are still rather inefficient and impracticable for real applications.

In this appendix, we suggest the first efficient and scalable methodology for the privacy-preserving $k$-NN search that is implementable on embedded processors. In contrast to the existing literature that assumes private datasets and known query, we assume a more general case of both private dataset and private query. Our methodology for providing a privacy-preserving solution relies on Yao’s Garbled Circuit (GC) that is currently considered the most efficient way to preserve privacy [107, 108]. Note that the use of GC for the generic problem of privacy preserving data min-
ing has been proposed, but not implemented [109]. The GC protocol requires the function be represented as a binary circuit. It encrypts the truth tables of Boolean gates in the circuit. The input values are used as to decrypt the output value of the gates. All the garbled circuits suggested to date only support one pass, directed acyclic circuits, *a.k.a.*, *combinational circuits*. The only available implementation of the privacy-preserving similarity search using the GC protocol is for the 1-NN search, where the circuit size was linearly increasing with the dataset size [84]. This increase is due to the conventional combinational logic representation is not scalable.

To implement a GC, one needs to compile the higher-level description of the functions to the Boolean logic suitable for garbling. For this purpose, several custom compilers have been developed by the security and software/compiler communities [12, 23, 31, 25]. These compilers either use a custom library for a general purpose programming language [12] or introduce transformations for performing compile-time circuit garbling and on-the-fly gate generation [25]. Since these compilation methods are built upon the combinational logic model, they all suffer from the scalability issue.

Our work is the first to synthesize and optimize the GC for performing privacy preserving $k$-NN using a sequential representation. Instead of relying on custom compilers, we follow TinyGarble synthesis methodology proposed in Chapter 3 and Chapter 4. In those chapters, we view the compilation of the general sequential circuit as a special case of logic synthesis. By defining new custom libraries and design objective/constraints, we demonstrate that utilizing standard logic synthesizers addresses the challenges in privacy preserving $k$-NN. As a result, we can store the GC and perform the privacy preserving $k$-NN search with an unprecedented efficiency.

**Problem Statement.** Alice has a query $q$, and Bob has a dataset $S$. They want to jointly compute the $k$ nearest neighbors of $q$ in $S$ such that Bob does not
learn anything about $q$ and Alice does not learn anything about $S$ except the nearest neighbors.

**Contributions.** In brief, our contributions are as follows:

- Introducing the first efficient, practicable, and scalable methodology for privacy-preserving $k$-NN search assuming that the dataset and query are each privately held. The method relies on the Yao’s GC protocol and implementable on embedded processors.

- Proposing a sequential circuit description for privacy-preserving $k$-NN search using Yao’s Garbled Circuit protocol (instead of the known combinational representation). New transformations are created such that the sequential $k$-NN implementation is securely evaluated by interfacing with the available (combinational) cryptographic garbling schemes.

- Development of new custom libraries to generate optimized circuits for $k$-NN search using the standard logic synthesis tools. This work is the first to utilize conventional logic synthesis for secure function evaluation of $k$-NN.

- Reduction in the size of the required memory for GC from $O(nw)$ to $O(w)$ compared with the best-known GC implementation of 1-NN [84]. Our scalable implementation requires a memory in the order of $O(kw)$ for $k$-NN search. Note that $k$-NN search was impracticable earlier (for large $n$) due to the linear growth of the combinational representation.

- A Proof-of-concept implementation of privacy preserving $k$-NN utilizing the Synopsys Design Compiler on an Intel processor. For example, the circuit size for $k$-NN search with $w = 31, k = 8$ is only 41.8KB.
A.2 Related Work

The related literature in realizing privacy-preserving $k$-NN search has mainly focused on using homomorphic encryption as the enabling cryptographic primitive [105, 102]. In their protocol, two parties perform $k$-NN search locally on their respective private dataset for a public query and then privately combine their results to form the $k$-NN. In contrast with these works, we adopt a more general setting in which one party holds a private dataset and the other one provides a private query. Moreover, we only rely on Yao’s GC protocol which is known to be much more efficient two-party privacy-preserving protocol than homomorphic encryption [107, 108]. The use of GC for privacy preserving data mining has been suggested, but the existing literature focused on theoretical/protocol aspects and not implementation [109]. By leveraging our sequential description, this paper proposes the first scalable realization and a low-overhead realization of secure $k$-NN on a conventional processor.

The work on generating Boolean functions for GC can be broadly classified into three categories: cryptographic primitives such as [12, 16], transformations at the logic-level such as [13], and compiler/software techniques for mapping GC to the Boolean logic including [12, 31, 25]. Our work is orthogonal to the advances in the GC cryptographic primitives and logic-level transformations. We provide a compilation of the functional description to the Boolean logic which can be optimized and interfaced for any GC scheme. Therefore, we only describe the related work in the area of compiler/software techniques.

In the field of mapping and optimizing functions into Boolean logic for GC, the related literature has suggested developing custom compilers [12, 23, 31, 25], custom libraries for conventional high-level compilers [15, 28, 24], hardware accelerators[91, 79, 16], and mobile device implementation [30].
Following the introduction of the first compiler for GC, called Fairplay [12], a number of researchers have focused on providing a custom compiler to interpret high-level procedural language and map it to a circuit description language [23, 31]. The most scalable existing compiler is PCF, which introduces loops that, if given manually in the high-level language, are kept until the GC evaluation [25]. Our sequential description of the $k$-NN function that we input in the Hardware Description Language (HDL) format is much more compact than the high-level (software) loop embracing in PCF. Note that PCF also relies on combinational descriptions.

Another class of proposed GC compilation methods leverages a library-based technique along with standard software compilers. Some examples include FastGC [15], VMCRYPT [28], and FastGC extension to re-usable sub-circuits [24]. We are the first to adapt a hardware description language and conventional logic synthesis for the important problem of privacy preserving of $k$-NN. Our method is automated, while it also benefits from custom logic-level libraries.

A number of researchers have suggested the development of hardware accelerators for GC, including GPUs [90, 91], FPGAs [79], or using the AES-NI available in modern CPUs [16]. Our work is orthogonal to this domain and can benefit from building accelerators for our sequential representation.

Using the GC for secure computing on resource-constrained devices such as mobile/embedded platforms was suggested in [94]. Recent work in this area described a protocol for GC that relies on a smart card embedded in the mobile device [95]. These implementations can significantly benefit from the scalable and efficient $k$-NN methodology introduced in this thesis. To overcome the limitations of resource-constrained devices, a set of relevant work in this area suggested outsourcing the GC generation and evaluation to cloud servers [96, 97]. Our work demonstrates the
feasibility of on-device GC implementation.

A.3 Nearest Neighbors Search Circuit

In this section, we present implementation of privacy-preserving $k$-NN search using Yao’s GC protocol. First, we describe the optimized circuit generation for GC using logic synthesis tools. Next, we outline the implementation of 1-NN search using conventional GC based on the combinational circuit. Lastly, we discuss the compact realizations of 1-NN and $k$-NN search based on the sequential circuit.

Without any loss of generality, we assume the distance function for finding the nearest neighbors is the Hamming distance in the following description.

A.3.1 Circuit Generation

We customize the flow of the standard logic synthesis tools to generate circuits optimized for the GC protocol. As mentioned in Chapter 2 for GC with Free XOR optimization, there is a need to minimize the number of non-XOR gates in the Boolean representation. We perform two major customizations in the synthesis flow. First, we create a new synthesis library to aid the conversion of the arithmetic and conditional operations to GC-optimized logical modules. Second, we develop a technology library to guide the mapping of the logic to the circuit netlist.

Synthesis Library

To realize the $k$-NN search, a set of basic arithmetic and conditional operations consisting of comparator, multiplexer, and Hamming distance are required. We create a custom synthesis library that includes the minimum non-XOR implementations of these operations. A $w$-bit comparator ($COMP_w$) is implemented using only $w$ non-
XOR gates [84]. A $w$-bit multiplexer ($MUX_w$) is realized using $w$ non-XOR gates [13]. A $w$-bit Hamming distance ($HAMMING_w$) is devised using $w - \lceil \log_2(w) \rceil$ non-XOR gates where $w = 2^k - 1$, $k \in \mathbb{N}$ [58]. In all these modules, the total number of gates is $O(w)$.

**Technology Library**

The technology library includes logical descriptions of basic units and their parameters like delay and area. The synthesis tool uses the technology library to generate a circuit optimized for given objectives and constraints. We design a custom technology library that contains 2-input gates (according to the requirement of the GC protocol). We set the area of XOR gates to zero and the area of non-XOR gates to one. We synthesize the circuits with the area constraint set to zero so that the synthesis tool’s objective becomes minimizing the number of non-XOR gates in the generated circuit.

An additional feature of this library is the inclusion of non-standard gates (other than basic gates like NOT, AND, NAND, OR, NOR, XOR, and XNOR) to increase the flexibility of the mapping process. For example, the logical functions $F = A \lor B$ and $F = (\sim A) \lor B$ requires equal time in garbling/evaluation. However, using only standard gates, the second function will need a NOT gate and an OR gate. We include four such non-standard gates which have an inverted input.

**A.3.2 Combinational Garbled Circuit**

As stated in Chapter 2, all previous implementations of the GC protocol use a combinational description. To start our implementation for the special case of 1-NN search, we look for the closest point ($o$) to the query point ($q$) in the dataset ($S$). In the privacy-preserving setting, we need to compare the query point to all the points in the
Figure A.1: The combinational circuit for 1-NN. It consists of $n$ Hamming distance and $n - 1$ MIN modules.

dataset because one cannot utilize the (private) intermediate search values to bound the search, e.g., binary search.

Figure A.1 shows the combinational circuit for 1-NN. The implementation uses $n$ Hamming distance modules, and $n - 1$ MIN modules (consisting of one COMP and two MUXs) to find the nearest point. One MUX selects the smaller distance for later comparison while the other one finds the point corresponding to that distance.

The total number of gates in the 1-NN combinational circuit is as follows:

$$\# \text{ of gates} = n \times HAMMING_w$$

$$+ (n - 1) \times (COMP_{\log_2(w)})$$

$$+ MUX_w + MUX_{\log_2(w)}$$

$$\Rightarrow \# \text{ of gates} \in O(nw).$$
The circuit should be garbled/evaluated only once. Thus, the time complexities of garbling/evaluation is $O(nw)$.

### A.3.3 Sequential Garbled Circuit

One can use sequential circuits as a very compact circuit description for both real hardware and the GC protocol. A sequential circuit is composed of a combinational circuit and a set of registers that stores the intermediate values. We modify the garbling scheme such that for each sequential cycle, it garbles/evaluates the combinational part and stores the garbling keys for the registers. The stored keys are used as inputs in the next cycle. To ensure security, each gate should have a unique identifier for each time that it is garbled/evaluated. Since in the sequential circuit each gate is garbled/evaluated multiple times, we use the combination of gate index and cycle index as a unique identifier for each gate invocation. Thereby, the proof of security provided in [59, 16] also applies to our garbling scheme. We now describe the sequential 1-NN search implementation followed by $k$-NN implementation.

#### Sequential 1-NN

Our 1-NN search sequential circuit is implemented with only one Hamming distance and one MIN module. Figure A.2 illustrates the sequential circuit for 1-NN search. In each cycle $c$, the circuit computes the distance between $q$ and $S[c]$. Next, it compares the resulting distance with the stored minimum distance in the register (reg). It then stores the minimum distance along with the nearest point until cycle $c$. The total number of cycles required to compute 1-NN is $n$. 
Figure A.2: The sequential circuit for 1-NN. It consists of one Hamming distance and one MIN module. For a dataset of size $n$, the circuit is required to be garbled/evaluated $n$ times.

The total number of gates in the 1-NN sequential circuit is as follows:

$$
\# \text{ of gates} = HAMMING_w \\
+ (COMP_{\lceil \log_2(w) \rceil} \\
+ MUX_w + MUX_{\lceil \log_2(w) \rceil}) \\
\Rightarrow \# \text{ of gates} \in \mathcal{O}(w).
$$

The circuit should be garbled/evaluated $n$ times. Thus, the time complexities of garbling/evaluation are the same as the combinational circuit and equal to $\mathcal{O}(nw)$.

Sequential k-NN

In $k$-NN search, the goal is to find the $k$ nearest points to the query in the dataset. We expand the sequential circuit for the 1-NN to store the $k$ nearest points. For this purpose, we implement a priority queue with the depth of $k$ which receives one point at each cycle. The priority of each point is equal to its distance to the query. Figure A.3 shows the sequential circuit for the $k$-NN search. The circuit has one Hamming distance, $k$ MIN, and $k - 1$ MAX modules. The MAX module, like MIN,
Figure A.3: The sequential circuit for $k$-NN. It consists of one Hamming distance, $k$ MIN, and $k-1$ MAX modules. It requires to be evaluated $n$ times where $n$ is the size of the dataset $S$.

The total number of gates in the 1-NN sequential circuit is as follows:

$$\# \text{ of nonXORs} = HAMMING_w + (2k - 1) \times COMP_{\log_2(w)} + (2k - 1) \times (MUX_w + MUX_{\log_2(w)})$$

$$\Rightarrow \# \text{ of nonXORs} \in O(kw).$$

The circuit should be garbled/evaluated $n$ times. Thus, the time complexities of garbling/evaluation are the same as the combinational circuit and equal to $O(nkw)$. Note that due to the unscalability of combinational $k$-NN search, we did not include its implementation.
A.4 Evaluation

We generate the circuits using Synopsys Design Compiler (DC) 2010.03-SP4. We also use the Synopsys Library Compiler from DC package to interpret our custom technology libraries. For garbling and evaluation, we use the TinyGarble GC engine (see Appendix B). The GC engine exploits the cryptographic permutations realizable by fixed-key AES for GC operations. We run the framework on a system with Ubuntu 14.10 Desktop, 12GB of memory, and Intel Core i7-2600 CPU at 3.4GHz to assess the timing performance.

The metrics used to evaluate the performance of our implementations are as follows:

- The Circuit Size ($CS$) in Bytes is computed as

$$CS = 24 \times q,$$

where $q$ is the total number of gates. We store two indices (16B) and one type (8B) for each gate in circuit description file.

- Circuit Size Efficiency ($CSE$) is defined as

$$CSE = \frac{CS_C}{CS_S},$$

where $CS_C$ is the size of the combinational circuit and $CS_S$ is that of the sequential circuit.

- The garbling time is calculated as

$$T = \# \text{ of non-XOR} \times T_{\text{nonXOR}} + \# \text{ of XOR} \times T_{\text{XOR}},$$

where $T_{\text{nonXOR}}$ is the execution of a non-XOR gate and $T_{\text{XOR}}$ is the execution of a XOR gate, which are 164 clock cycles (cc) and 62cc respectively on the
Table A.1: Circuit size and timing evaluation for 1-NN search.

<table>
<thead>
<tr>
<th>W</th>
<th>N</th>
<th>7</th>
<th>15</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Total Gates</th>
<th>Non-XOR</th>
<th>CS(B)</th>
<th>T(cc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational</td>
<td>9044</td>
<td>2160</td>
<td>217056</td>
<td>781048</td>
</tr>
<tr>
<td></td>
<td>18128</td>
<td>4336</td>
<td>435072</td>
<td>1566208</td>
</tr>
<tr>
<td></td>
<td>36304</td>
<td>8688</td>
<td>871296</td>
<td>3137024</td>
</tr>
<tr>
<td></td>
<td>19637</td>
<td>4325</td>
<td>471288</td>
<td>1658644</td>
</tr>
<tr>
<td></td>
<td>39349</td>
<td>8677</td>
<td>944376</td>
<td>3324692</td>
</tr>
<tr>
<td></td>
<td>40981</td>
<td>8530</td>
<td>983544</td>
<td>3410882</td>
</tr>
<tr>
<td></td>
<td>82069</td>
<td>17106</td>
<td>1969656</td>
<td>6833090</td>
</tr>
<tr>
<td>Sequential</td>
<td>62</td>
<td>17</td>
<td>1488</td>
<td>713984</td>
</tr>
<tr>
<td></td>
<td>136</td>
<td>34</td>
<td>3264</td>
<td>1427968</td>
</tr>
<tr>
<td></td>
<td>283</td>
<td>67</td>
<td>6792</td>
<td>2855936</td>
</tr>
<tr>
<td></td>
<td>1523200</td>
<td>3046400</td>
<td>3120640</td>
<td>6241280</td>
</tr>
</tbody>
</table>

Comparison CSE 145.9 292.4 585.5 144.4 289.3 144.8 290

specified system. We measured these values as the mean of 10,000 garbling trials.

Table A.1 reports the circuit size and timing evaluation for 1-NN search for different values of $w$ and $n$. For the same $w$, the circuit size remains constant for sequential implementation. Therefore, the CSE increases linearly with $n$. As can be seen, the garbling times are almost equal for both combinational and sequential circuits. The small improvements in garbling time for the sequential circuit is due to the more efficient optimization in the synthesis tool when the circuit is small (sequential). For example where $n = 256$, $w = 31$, the time of garbling using the combinational circuit is $6.83 \times 10^6$cc (2.01ms in our evaluation setup) while the one using sequential circuit is $6.24 \times 10^6$cc (1.84ms) which is 8.7% faster than the combinational circuit.

The circuit size and timing evaluation for $k$-NN search is reported in Table A.2. Since its combinational evaluation is not practical, we do not compare it in the way we
Table A.2: Circuit size and timing evaluation for $k$-NN search.

<table>
<thead>
<tr>
<th>W</th>
<th>7</th>
<th>15</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Total Gates</td>
<td>146</td>
<td>270</td>
<td>518</td>
</tr>
<tr>
<td>Non-XOR</td>
<td>44</td>
<td>92</td>
<td>188</td>
</tr>
<tr>
<td>CS(B)</td>
<td>3504</td>
<td>6480</td>
<td>12432</td>
</tr>
<tr>
<td>T(cc)</td>
<td>13540</td>
<td>26124</td>
<td>51292</td>
</tr>
</tbody>
</table>

did for 1-NN. The $CS$ for the largest circuit in this thesis ($w = 31, k = 8$) is 41.8KB which will fit easily in embedded systems. As an example where $n = 128, w = 31, k = 8$, the garbling time would be $128k \times 173848cc = 22.8 \times 10^9cc (6.7s)$.

### A.5 Conclusion

In this appendix, we present a methodology for generation of highly compact and scalable privacy-preserving $k$-nearest neighbor ($k$-NN) search using GC. We are the first to suggest a sequential description of $k$-NN, which enables generation of a compact Boolean GC. Our newly created custom libraries allow us to leverage the established powerful logic synthesis tools to optimize the ($k$-NN) Boolean expressions for GC. We show the first-of-a-kind implementation of privacy preserving $k$-NN using the (). It requires only 41.8KB storage for the 32-bit 8-NN search. Our implementation shows the practicability, efficiency, and scalability of the suggested methods.
Appendix B

Tinygarble GC Engine

In this section, we explain TinyGarble GC Engine, an implementation of the Yao’s GC protocol for sequential circuits. First, we describe the implementation of sequential GC protocol in TinyGarble GC engine. Next, we describe Simple Sequential Circuit Description (SSCD), a format for storing and processing sequential Boolean circuits in our GC engine and how Verilog netlist is translated into SSCD. We also explain the communication steps and pipelining of the GC protocol for sequential circuits. We then introduce the terminate signal that helps the user to identify when the circuit output is ready to reduce the garbling cost. We implemented the GC engine in C++, and its source code is available in the TinyGarble repository.*

B.1 Yao’s GC Protocol Implementation for Sequential Circuits

TinyGarble GC engine is an implementation of Yao’s protocol for garbling and evaluation of sequential circuits. We build it based on JustGarble’s implementation of the GC protocol [16]. We choose JustGarble because it has a number of advantages over other available implementations: simplicity and efficiency (use of AES block cipher and AES-NI hardware accelerator). However, JustGarble’s realization comes with some shortcomings. (i) It only realizes the garbling/evaluation procedures and does

*https://github.com/esonghori/\gls{tinygarble}
not include the communication capability between the parties. Most importantly, it
does not contain the implementation of OT that is a crucial part of communication
in the GC protocol. (ii) It does not include Half Gate, the recent optimization of the
GC protocol [17]. Half Gate shrinks the GC protocol cost by 33% by reducing the
communication required for an non-XOR gate from three labels to two.

Our GC engine supports both garbling/evaluation and communication between
the two parties including OT (see Section B.4). Our implementation of OT also in-
cludes its extended version [90] that reduced the computation cost by transferring the
bulk of messages together. We use OpenSSL library to handle big integer and modular
exponentiation in OT [110]. We also add Half Gate method to the implementation on
top of other optimizations including Free XOR and Fixed-Key Block-Cipher. Similar
to JustGarble, we benefit from AES-NI, a hardware implementation of AES in Intel
processors.

One of the prominent features of TinyGarble GC engine is its support for gar-
bling/evaluation of sequential circuits. The gates in a sequential circuit are gar-
bled/evaluated for multiple sequential cycles. The parties agree on the total number
of cycles before starting of the protocol. To ensure security, the encryption tweaks
$T$ (see Section 2.2.4) for each gate has to be unique in each cycle [24, Sect. 3.4]. We
set tweak $T$ for each gate to be the concatenation of the cycle index ($cid$) and the
unique gate identifiers ($gid$) in the combinational part of the sequential circuit, i.e.,
$T = cid || gid$. An alternative method could be to use a monotonic counter in the gar-
bling/evaluation routines that increases by one for each gate. As in previous work,
security and correctness of the GC garbling/evaluation follow from the uniqueness
of the tweak $T$ and the existing proofs of security and correctness are still valid for
TinyGarble (see [59, 16, 17]).
B.2 Simple Sequential Circuit Description

JustGarble [16] developed the Simple Circuit Description (SCD) format to represent combinational Boolean circuits. We introduce a similar format called Simple Sequential Circuit Description (SSCD) that additionally include information about the FFs and the newly added terminate signal in sequential circuits.

Besides clock (clk) and reset (rst) ports that are not relevant to the GC protocol, FFs has three other ports: initial (I), input (D), output (Q). The initial wires (I) are treated as inputs and can belong to either Alice or Bob. We denote Alice’s initial values as $g_{\text{init}}$ (g for Garbler) and Bob’ as $e_{\text{init}}$ (e for Evaluator). The input of the FFs at any sequential cycle are fed by their outputs at the previous cycle. The inputs of the FFs (D) are treated as the output of the circuit. Similarly, the output of the FFs (Q) are considered as the input of the circuit. The Q wires at the first sequential cycles are set to their corresponding I’s (provider by either Alice or Bob) and at the rest of cycles are set to their corresponding D’s. We denote Alice’s and Bob’s inputs as $g_{\text{input}}$ and $e_{\text{input}}$ respectively.

In the SSCD format, we index wires according to the following order: (1) $g_{\text{init}}$ (2) $e_{\text{init}}$ (3) $g_{\text{input}}$ (4) $e_{\text{input}}$ (5) gates’ output. SSCD is stored in a human-readable ASCII format. An SSCD file consists of seven lines: (1) $g_{\text{init}}$ size, $e_{\text{init}}$ size, $g_{\text{input}}$ size, $e_{\text{input}}$ size, $\text{diff}$ size, output size, terminate (an optional 1-bit output indicating that the circuit is done, see Section B.5), and the number of gates. (2) gates’ first input index. (3) gates’ second input index. (4) gates’ type. (5) outputs index. (6) Flip-Flops’ D index. (7) Flip-Flops’ I index (chosen from $g_{\text{init}}$ or $e_{\text{init}}$).
B.3 Translating Verilog Netlist to SSCD

The output of TinyGarble synthesis flow is an optimized gate level Verilog code called netlist. The netlist format needs to be translated into SSCD format to be readable by TinyGarble GC engine. We developed a tool called V2SCD that converts a Verilog netlist into an SSCD file. V2SCD first parses the netlist file and detect the gates and their ports. Next, it topologically sorts the gates based on their dependencies through wire connections. Finally, it creates a human-readable SSCD file.

B.4 Communication

JustGarble’s implementation of the GC protocol lacks the communication features required by the GC protocol. We implemented and integrated these features into our GC engine. The communication and computation in TinyGarble GC engine follow the steps below:

1. Alice generates the garbled tables.

2. Alice sends her input labels to Bob.

3. Alice sends Bob’s input labels to him through OT.

4. Alice sends garbled tables to Bob.

5. Bob evaluates the circuit.

6. Alice sends the Least Significant Bit (LSB) of labels of the output labels to Bob such that he can learn the output. We can change this to let Alice learn some or all bits of the output. To do so, Bob has to send the LSB of the selected output labels to Alice.
By default, the steps are the same for sequential circuits as well. Alice has to generate the garbled circuit for \( cc \) sequential cycles where \( cc \) is a predetermined number set by the parties. Alice sends labels corresponding to her initial values (\( g_{\text{init}} \)) to Bob as well as the ones corresponding to her inputs (\( g_{\text{input}} \)) for all \( cc \) cycles. The initial labels are sent only for the first cycle. The same goes for Bob’s initial and input values, but through OT.

\subsection*{B.4.1 Pipelining and Low Memory Mode}

In the simple communication scheme described above the memory required to store the labels is not scalable in terms of the number of sequential cycles \( cc \). Both Alice and Bob has to save all the labels in the memory for all \( cc \) cycles. To avoid such large memory footprint, we design a communication mode for sequential circuits that reduces the memory footprint by pipelining the garbling/evaluation. Pipelining for the GC protocol was first proposed in [90] for combinational circuits where the circuit is divided into multiple chunks and communication is done between garbling each chunk. However, the pipelining is more fit for sequential circuits where garbling of the circuit in each cycle creates a natural chunk for the pipeline.

The communication and computation in TinyGarble GC engine for a sequential circuit in low memory mode follows the steps below: (1) Alice generates garbled tables for one sequential cycle. (2) If it is the first cycle, Alice sends her initial labels to Bob. (3) If it is the first cycle, Alice sends Bob’s initial labels to him through OT. (4) Alice sends her input labels for this cycle to Bob. (5) Alice sends Bob’s input labels for this cycle to him through OT. (6) Alice sends garbled tables for this cycle to Bob. (7) Bob evaluates the circuit for one cycles. (8) Go to step (9) if cycle reaches \( cc \), if not go to (1) (9) Alice and Bob exchange LSB of output labels to learn the output.
Note that in this mode of communication, the number of invocation of OT increases from 1 to \( cc \). Since we are using the OT extension, the cost of one invocation of OT is almost constant when the number of transferred labels is large enough (in our case larger than 128)\(^\dagger\). This means that when the number of Bob’s inputs (bit widths of e_input times \( cc \)) is larger than 128, the cost of transferring labels through OT increases using pipelining. Therefore, there is a trade-off between the cost of multiple invocations of OT and the memory footprint when the number of Bob’s input or sequential cycles is large.

B.5 Terminate Signal

As explained in Chapter 4, sequential circuits are required to be garbled/evaluated for a number of sequential cycles (\( cc \) cycles). \( cc \) has to be determined regardless of the inputs to ensure the privacy. To do so, it is set such that the circuit functionality is ensured to be finished during \( cc \) cycles, i.e., the worst case scenario. There are sequential circuits that, given specific inputs, can complete the computation faster than their predetermined number of cycles. For example, the sequential circuit presented in [4] for Stable Match algorithm of \( n \) pairs can finish the computation in \( O(n) \) while in the worst case scenario, the number of cycles is \( O(n^2) \).

To reduce the unnecessary cost after the computation is finished, TinyGarble enables users to add a one-bit output to a sequential circuit that indicates in which cycle the output of the circuit is ready. We call this output the terminate signal. The use of the terminate signal was first proposed in [4] for a particular application of stable

\(^\dagger\)The cost of OT is dominated by the computation of modular exponentiation. In OT extension, the number of modular exponentiation is \( O\left(\min(l, p)\right) \) where \( l \) is the number of messages (labels) and \( p \) is the security parameter.
matching using sequential GC. We generalized this approach for all sequential circuits. The terminate signal enables users to stop garbling/evaluation of the circuit when no further computation is needed, avoiding futile communication and computation. In TinyGarble, the signal is revealed every $K$ clock cycles where $1 \leq K \leq cc$. The parties determine $k$ before starting the computation. Depending on the structure of the sequential circuit and its functionality, revealing the terminate signal leaks some information about the inputs. Thus, it has to be used with careful consideration. $K = 1$ shows the exact number of cycles for the given inputs to finish the function. Larger $K$ exposes less information and $K = cc$ reveals no information about the inputs. Therefore, revealing the terminate signal offers a trade-off between the privacy and the cost of garbling a sequential circuit.

**B.6 SkipGate Implementation**

TinyGarble GC engine supports the SkipGate algorithm on top of its implementation of the GC protocol. We change the code of the engine according to the SkipGate algorithm described in Chapter 5. Two new ports are allowed in the circuit for public input variables: (i) Public initial value (p_init) that is connected to I input of FFs and is read only at the first cycle. (ii) Public input (p_input) that is directly connected to the gates of the circuit and is read every sequential cycle. The first line of the SSCD format also includes the size of these two new ports: p_init_size, g_init_size, e_init_size, p_input_size, g_input_size, e_input_size, diff_size, output_size, terminate_id, and gate_size.
Appendix C

Garbled Circuit Library

In this section, we introduce TinyGarble library that consists of Garbled Circuit (GC) optimized circuits for complex mathematical/logical operations that can be used as building blocks practical applications. The Simple Sequential Circuit Description (SSCD) and netlist files of the operations in the library can be found in the TinyGarble repository∗.

C.1 Division and Remainder

The TinyGarble library includes the circuit for integer division which takes two numbers as inputs and outputs quotient and remainder (modulus). The library includes the circuits for 16-, 32-, and 64-bit integers. We also add two additional circuits in the library that output only quotient and remainder respectively and have a fewer number of gates compared to the original circuit.

C.2 Floating Point Operations

The TinyGarble library includes an extensive set of operations for both IEEE-754 single- and double precision floating-point numbers. It supports addition (fp_add), subtraction (fp_sub), division (fp_div), and multiplication (fp_mult) of two inputs. We also include a comparison circuit (fp_cmp) that outputs three bits: less than,

∗https://github.com/esonghori/gls{tinygarble}/blob/master/scd/netlists/v.tar.gz
greater than, equal signals. The natural exponentiation circuit (fp_exp) receives a floating-point input $a$ and computes $e^a$. The logarithm circuit (fp_log2) calculates the logarithm of the input in base 2. Other floating-point operations include square (fp_square) and square-root (fp_sqrt).

### C.3 Encoder

Encoder circuit converts a one-hot representation into a binary one. One-hot representation has only one active bit (usually equal one). Number $i$ is represented in one-hot by activating the $i^{th}$ bit. Thus, the encoder circuit outputs the index of the active input bit of the one-hot representation. Therefore, for $n$-bit one-hot input, the output $o$ is $\log(n)$ bit wide. For example, if the $n = 16$ and the $5^{th}$ bit is one, the output should be $o = 0101$ (4-bit wide). If none of the inputs is one, the encoder outputs zero.

We implement the encoder operation using a recursive structure. An encoder for $n$-bit input is implemented using two smaller encoders for $n/2$-bit input. The first half of the input (0 to $n/2 - 1$) is given to the first small encoder, and the rest ($n/2$ to $n - 1$) is given to the second one. One of these two small encoders that receives the inactive part of input outputs zero and the other one outputs the binary representation in its half. Depending on which half was active, a bit is attached to the final output as the Most Significant Bit (MSB). In other words, the MSB of $o$ is set to one if the output of the second encoder is nonzero and otherwise zero.

### C.4 Argmax

Given an array of numbers, Argmax circuit outputs the largest number along with its index in the array. The current implementation supports integers, but it can easily
be extended to support any type of inputs with appropriate comparison block. The size of the array \( n \) and the number of bits \( b \) for each number are parameters of the circuit and can be set before compilation. The combinational circuit of Argmax has \( n - 1 \) comparison blocks, \( n - 1 \) two-to-one \( b \)-bit-wide Multiplexer (MUX)s, and \( n - 1 \) two-to-one \( \log_2 n \)-bit-wide MUXs. Its sequential circuit has one comparison blocks, and two MUXs for selecting the index and the largest number and two registers to store them. The sequential circuit has to be evaluated for \( n - 1 \) sequential cycles.

### C.5 CORDIC

Coordinate Rotation Digital Computer (CORDIC) circuit computes hyperbolic and trigonometric functions. It is an iterative algorithm that improves the accuracy of the result by (typically) one bit at each iteration. We have implemented CORDIC as a sequential circuit which performs one iteration at each sequential cycle. It includes a lookup table, shift, addition, and subtraction operations. CORDIC circuit takes three inputs \( (x_0, y_0, \text{ and } z_0) \) and outputs three values \( (x_n, y_n, \text{ and } z_n) \). The subscript \( n \) denotes the final outputs after running \( n \) iterations CORDIC. All of the inputs, outputs, and intermediate results are in fixed-point representation. The number of integer bits and fractional bits are parameters that can be set before compilation.

CORDIC has three operation modes: (i) Circular, (ii) hyperbolic, and (iii) linear. The circular mode can rotate an arbitrary vector by a given angle. With specifically chosen inputs \( (x_0 = 1, y_0 = 0, \text{ and } z_0 = \theta) \) in circular mode, the circuit computes the trigonometric functions \( (x_n = \cos(\theta), y_n = \sin(\theta), \text{ and } z_n = 0) \). The circuit computes two different functions without facing any more overhead compared to computing only one of them. Similarly, the hyperbolic mode can compute hyperbolic functions. Please note that in the hyperbolic mode iterations \( 3 \times i + 1 \) need to be computed twice. The
linear mode can compute the multiplication of the inputs \((x_n = x_0, y_n = y_0z_0, z_n = 0)\). Using CORDIC’s outputs, one can create a few other non-linear functions as well, for example, \(\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}\), \(\tanh(\theta) = \frac{\sinh(\theta)}{\cosh(\theta)}\), \(\exp(x) = \sinh(x) + \cosh(x)\), and \(\text{Sigmoid}(\theta) = \frac{1}{1 + \cosh(\theta) - \sinh(\theta)}\).

### C.6 Evaluation

Table C.1 reports the result of integer division and remainder functions for various bit-widths. \texttt{Div}\_\texttt{rem} function calculates both the remainder and division in the same circuit. A few of the previous work reported the number of non-XOR gates for 32-bit integer division only: 1,437 \cite{27}, 1,210 \cite{18}, and 2,236 \cite{19}. As shown in the table, the number of non-XOR gates for our 32-bit division is 599. It means 2 to 3.7 times improvement compared to the previous custom compilers.

Table C.2 illustrates the number of non-XOR and total gates for the floating point operations. The table reports the results of both single precision (\texttt{float}) and double precision (\texttt{double}). Pullonen et al. (2015) are the only previous work that reported the result for the similar floating-point operations \cite{111}. They used the custom compiler of CBMC-GC \cite{26} to compile software implementations of these floating-point operations. As can be seen in the table, the circuits in TinyGarble library outperforms the ones in \cite{111}.
Table C.1: The result of integer division and remainder functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Bit-width</th>
<th>Non-XOR</th>
<th>Total gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reminder</td>
<td>16</td>
<td>186</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>631</td>
<td>1,777</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2,290</td>
<td>6,636</td>
</tr>
<tr>
<td>Division</td>
<td>16</td>
<td>170</td>
<td>452</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>599</td>
<td>1,681</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2,226</td>
<td>6,444</td>
</tr>
<tr>
<td>Div_rem</td>
<td>16</td>
<td>186</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>631</td>
<td>1,778</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2,290</td>
<td>6,637</td>
</tr>
</tbody>
</table>
Table C.2: The result of floating-point functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Precision</th>
<th>Previous work</th>
<th>TinyGarble Library</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[111]</td>
<td></td>
<td>GTD</td>
</tr>
<tr>
<td></td>
<td>Non-XOR</td>
<td>Total gates</td>
<td>Non-XOR</td>
<td>Total gates</td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td>Double</td>
<td>Single</td>
<td>Double</td>
</tr>
<tr>
<td>fp_add</td>
<td>Single</td>
<td>5,671</td>
<td>7,052</td>
<td>936</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>13,129</td>
<td>15,882</td>
<td>2,032</td>
</tr>
<tr>
<td>fp_sub</td>
<td>Single</td>
<td>5,671</td>
<td>7,052</td>
<td>936</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>13,129</td>
<td>15,882</td>
<td>2,032</td>
</tr>
<tr>
<td>fp_mult</td>
<td>Single</td>
<td>5,138</td>
<td>7,701</td>
<td>3,554</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>13,104</td>
<td>25,276</td>
<td>17,019</td>
</tr>
<tr>
<td>fp_div</td>
<td>Single</td>
<td>12,851</td>
<td>21,384</td>
<td>3,810</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>36,133</td>
<td>73,684</td>
<td>17,585</td>
</tr>
<tr>
<td>fp_cmp</td>
<td>Single</td>
<td>-</td>
<td>-</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>-</td>
<td>-</td>
<td>435</td>
</tr>
<tr>
<td>fp_exp</td>
<td>Single</td>
<td>-</td>
<td>-</td>
<td>12,596</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>393,807</td>
<td>579,281</td>
<td>-</td>
</tr>
<tr>
<td>fp_log2</td>
<td>Single</td>
<td>-</td>
<td>-</td>
<td>13,072</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>-</td>
<td>-</td>
<td>16,355</td>
</tr>
<tr>
<td>fp_square</td>
<td>Single</td>
<td>-</td>
<td>-</td>
<td>1,763</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>-</td>
<td>-</td>
<td>8,461</td>
</tr>
<tr>
<td>fp_sqrt</td>
<td>Single</td>
<td>35,987</td>
<td>66,003</td>
<td>1,842</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>85,975</td>
<td>169,932</td>
<td>8,636</td>
</tr>
</tbody>
</table>
Appendix D

Open Source Logic Synthesis Tools

TinyGarble offers a generic methodology for generating Garbled Circuit (GC) that is transparent to the underlying logic synthesis tool. To show this point, we demonstrate an implementation of TinyGarble using the Yosys [112] and ABC [50] logic synthesis toolchain for circuit generation. Both of these tools are open-source and available online. We compare the performance of the commercial logic synthesis tool, i.e., Synopsys DC, with this open-source synthesis toolchain. ABC is an academic package developed at the University of California Berkeley. Yosys is a logic synthesis tool which calls ABC for its technology mapping. The Hardware Description Language (HDL) inputs for describing both sequential and combinational circuits are written in the Verilog programming language.

We compare the performance of these open-source tools to the commercially available Synopsys DC. Table D.1 reports the results. For comparison purposes, we compute $GTD$ and $MFE$ using the netlists generated by Synopsys DC as the reference. For most of the benchmark functions, $GTD$s are either minuscule or zero which implies that the number of non-XOR gates in circuits generated by Yosys and by Synopsys DC is almost similar. Regarding memory footprint, different tools perform better for different benchmark functions. These results show that TinyGarble is transparent to the underlying logic synthesis tool as long as the tool is up to date with respect to the known methods for logic minimization and mapping. Since the logic synthesis tools perform a series of optimizations, they may use different (heuristic) algorithms
for some of their internal steps which could lead to slightly different results. A user can choose between the various synthesis tools based on their performance and availability.
Table D.1: Comparison of circuit generation performance between the commercial Synopsys DC and Yosys+ABC open source logic synthesizer.

<table>
<thead>
<tr>
<th>Function</th>
<th>cc</th>
<th>Synopsys DC Non-XOR gates</th>
<th>Synopsys DC Total gates</th>
<th>Yosys Non-XOR gates</th>
<th>Yosys Total gates</th>
<th>Comparison</th>
<th>GTD</th>
<th>MFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare16284</td>
<td>1</td>
<td>16,384</td>
<td>65,536</td>
<td>16,383</td>
<td>32,767</td>
<td>0%</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16,384</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0%</td>
<td>1.33</td>
<td></td>
</tr>
<tr>
<td>Hamming160</td>
<td>1</td>
<td>158</td>
<td>1,039</td>
<td>158</td>
<td>1,158</td>
<td>0%</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>10</td>
<td>10</td>
<td>48</td>
<td>0%</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>Hamming1600</td>
<td>1</td>
<td>1,597</td>
<td>10,679</td>
<td>1,597</td>
<td>11,364</td>
<td>0%</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>320</td>
<td>13</td>
<td>13</td>
<td>57</td>
<td>0%</td>
<td>0.82</td>
<td></td>
</tr>
<tr>
<td>Hamming16000</td>
<td>1</td>
<td>15,994</td>
<td>107,226</td>
<td>15,994</td>
<td>112,421</td>
<td>0%</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3,200</td>
<td>16</td>
<td>16</td>
<td>66</td>
<td>0%</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>Sum 128</td>
<td>1</td>
<td>127</td>
<td>634</td>
<td>127</td>
<td>763</td>
<td>0%</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>0%</td>
<td>0.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>128</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>0%</td>
<td>0.71</td>
<td></td>
</tr>
<tr>
<td>Sum 256</td>
<td>1</td>
<td>255</td>
<td>1,274</td>
<td>255</td>
<td>1,531</td>
<td>0%</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>0%</td>
<td>0.71</td>
<td></td>
</tr>
<tr>
<td>Sum 1024</td>
<td>1</td>
<td>1,023</td>
<td>5,114</td>
<td>1,023</td>
<td>6,139</td>
<td>0%</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,024</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>0%</td>
<td>0.71</td>
<td></td>
</tr>
</tbody>
</table>
Appendix E

Glossary

#GT  Number of Garbled Tables.  81, 82

AES  Advanced Encryption Standard.  21, 22, 68, 69, 83, 85, 87, 88, 95–97, 101, 
     105–111, 122, 140, 143, 144

AES-NI  AES New Instructions.  89, 100, 121, 122, 133, 143, 144

ALU  Arithmetic Logic Unit.  59, 60, 64, 65, 75, 94, 98, 119


API  Application Program Interface.  70, 71

ARM  short for Advanced RISC Machine; is a family of RISC architectures for processors.  iii, 8, 10, 11, 13–15, 58, 70–76, 103, 104, 108, 111, 112, 118, 120, 
     125–127

ARM2GC  is a framework introduced in this thesis for high-level two-party secure function evaluation using Yao’s garbled circuit protocol. iii, 8, 10, 11, 13–15, 

ASIC  Application-Specific Integrated Circuits.  25, 26, 30, 37, 121

BMR  Beaver-Micali-Rogoway.  5, 22

BRAM  Block Random-Access Memory.  99, 100
C is a high-level general-purpose programming language. iii, 4, 9–11, 15, 25–28, 64, 71, 80, 91, 95, 104, 105, 107–110, 116–118

CISC Complex Instruction Set Computing. 63

CL Combinational Logic. 23

CORDIC Coordinate Rotation Digital Computer. 112, 113, 152, 153

CPU Central Processing Uni. 80, 89, 90, 100, 103, 121, 133

D-FF Data Flip-Flop. 23, 24

DAG Directed Acyclic Graph. 6

DC Design Compiler. 140

DM Data Memory. 63–65, 93, 94, 97, 98

FA Full Adder. 34, 35

FF Flip-Flop. 23, 24, 30, 35, 37–39, 60, 61, 63, 64, 72, 75, 99, 100, 145, 149

FPGA Field-Programmable Gate Array. 25, 26, 30, 95, 99–103, 121, 122, 133

Frigate is a framework introduced in [27] for high-level two-party secure function evaluation using Yao’s garbled circuit protocol. 108, 117, 118


GMW Goldreich-Micali-Wigderson. 5, 22, 120, 121
GPU  Graphics Processing Unit. 121, 133

GT  Garbled Tables. 68

GTD  Garbled Tables Difference. 81, 82, 84, 85, 87–89, 91

HA  Half Adder. 34, 35

HBC  Honest-But-Curious. 19, 122–124

HDL  Hardware Description Language. 10, 11, 25–28, 86, 91, 92, 105, 133, 156

HLS  High-Level Synthesis. 10, 26–28, 80, 91, 108, 109

IM  Instruction Memory. 63–65, 94, 97, 98

ISA  Instruction Set Architecture. 9, 10, 14, 62–67, 93, 95–99, 101, 102, 104

LSB  Least Significant Bit. 146, 147

LUT  Lookup table. 30, 100, 113

MFE  Memory Footprint Efficiency. 81, 84–88, 91

MIPS  short for Microprocessor without Interlocked Pipeline Stages; is a family of RISC architectures for processors. 7–10, 14, 58, 63–65, 67, 68, 73, 74, 79, 92–95, 99, 101–103, 108, 110, 119, 126

MSB  Most Significant Bit. 151

MUX  Multiplexer. 30, 38–40, 42, 43, 54, 59, 61, 72, 75–78, 105, 135, 136, 138, 139, 152
**netlist** is a description of a Boolean circuit through listing its Boolean gates and the dependencies between them. 10, 20, 27–30, 66, 86, 92, 104, 134, 143, 146, 150, 156

**ORAM** Oblivious Random-Access Machine. 61, 70, 75, 76, 78, 117

**OT** Oblivious Transfer. 19, 20, 45, 46, 62, 68, 119–121, 144, 146–148

**PAL** Programmable Array Logic. 25

**PC** Program Counter. 64, 65, 94

**PCF** short for Portable Circuit Format, is a framework introduced in [25] for high-level two-party secure function evaluation using Yao’s garbled circuit protocol. 5, 6, 79, 82, 84–88, 116, 126, 133


**PSI** Private Set Intersection. 95–97

**PSI-CA** PSI Cardinality. 95

**RAM** Random Access Memory. 63

**RDTSC** is an assembly instruction for x86 processors that is used for measuring CPU time. 89

**RISC** Reduced Instruction Set Computing. 63

**ROM** Read-Only Memory. 63

**SCD** Simple Circuit Description. 145
SFNL  Secure Function Definition Language. 115, 116


SHA  Secure Hash Algorithm. 83, 85, 88, 89, 106, 107, 109–111, 126

SHDL  Secure Hardware Description Language. 115


SoC  System on a Chip. 103

SSCD  Simple Sequential Circuit Description. 10, 11, 68, 143, 145, 146, 149, 150

Synopsys DC short for Synopsys Design Compiler, is a commercial synthesis tool targeting Application-Specific Integrated Circuits (ASIC). 30, 80, 84, 95, 104, 142, 156, 158

TG  TinyGarble. 106, 109

TinyGarble is a framework introduced in this thesis for two-party secure function evaluation using Yao’s garbled circuit protocol. i–iii, 10–13, 24, 27–29, 31, 37, 43, 60, 61, 72, 73, 79–82, 84–89, 92, 94, 103–109, 116, 119–121, 124–128, 130, 140, 143, 144, 146–150, 153, 155, 156

UC  Universal Circuit. 59, 60

UTM  Universal Turing Machine. 59
**Verilog** is a Hardware Description Language (HDL) standardized as IEEE 1364 and used for designing digital hardwares. 25, 27–29, 80, 91, 105, 107, 109, 143, 146, 156

**VHDL** short for VHSIC Hardware Description Language, is a Hardware Description Language (HDL) used for designing digital hardwares. 25, 27, 28
Bibliography


