TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit

Siam Hussain, Baiyu Li, Farinaz Koushanfar
University of California San Diego
{siamumar,baiyu,farinaz}@ucsd.edu

Rosario Cammarota
Intel Labs
rosario.cammarota@intel.com

ABSTRACT

We present TinyGarble2 – a C++ framework for privacy-preserving computation through the Yao's Garbled Circuit (GC) protocol in both the honest-but-curious and the malicious security models. TinyGarble2 provides a rich library with arithmetic and logic building blocks for developing GC-based secure applications. The framework offers abstractions among three layers: the C++ program, the GC back-end and the Boolean logic representation of the function being computed. TinyGarble2 thus allowing the most optimized versions of all pertinent components. These abstractions, coupled with secure share transfer among the functions make TinyGarble2 the fastest and most memory-efficient GC framework. In addition, the framework provides a library for Convolutional Neural Networks (CNN). Our evaluations show that TinyGarble2 is the fastest among the current end-to-end GC frameworks while also being scalable in terms of memory footprint. Moreover, it performs 18× faster on the CNN LeNet-5 compared to the existing scalable frameworks.

CCS CONCEPTS

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

1 INTRODUCTION

Multi-Party Computation (MPC) protocols allow two or more parties to jointly compute a function without revealing one party's input to others. In the two-party setting, the most efficient protocol is Yao's Garbled Circuit (GC) [1] executed between a garbler and an evaluator. In GC, the function to be computed is represented as a *netlist* of logic gates. The truth tables of the gates are garbled – encrypted and permuted – to ensure privacy. The Boolean value of each wire in the netlist is split into a pair of *shares* with each party holding one share. The two parties combine their shares to obtain the final output of the computation.

The practicality of a GC frameworks primarily depends on the following properties: (*i*) fast protocol execution, (*ii*) automatic generation of optimized netlists, (*iii*) scalability in terms of memory footprint, (*iv*) reliability, and (*v*) a rich programming paradigm. The first two properties received a lot of research effort over the years. Optimizations both to the protocol [2–4] and the netlists generation [5–8] have resulted in orders of magnitude reduction in the run-time. With the recent surge in the development of practical privacy-preserving systems, scalability, reliability, and a rich programming paradigm are becoming increasingly important. While existing frameworks focus on one or two of the properties, none of them demonstrate the best possible performance on all of them. Our earlier work, TinyGarble [5] demonstrated the best performances in the first four properties. In this work, we present an enhanced version with a rich programming interface as well as state-of-the-art implementations of the underlying security primitives.

1.1 Related Work

A number of the existing GC frameworks [6–10] support C or C++ or their subsets. Among them, Frigate [6], CBMC-GC [7], and ABY [9] generate the entire netlist before protocol execution. This execution style is not scalable in terms of the memory footprint which is one of the primary limiting factors in applying GC to practical size problems such as Convolutional Neural Network (CNN). Modern CNNs require billions of Multiply-Accumulate (MAC) operations per inference. As a result, the secure execution of an entire CNN as a single netlist results in an unmanageably large memory footprint.

The PCF [8] and TinyGarble [5] frameworks partially solve the scalability issue. Through run-time loop unrolling and *sequential* GC, respectively, they ensure that all the garbled gates do not reside in the memory at the same instance. However, the unrolling process slows down the execution by PCF. Moreover, its netlists are 50-80% less optimized compared to the recent frameworks. In the sequential GC by TinyGarble, the same netlist is executed through the protocol for multiple cycles. However, it only supports homogeneous loops, while most practical problems require heterogeneous loops, i.e., loops where possibly different netlists are executed at every cycle. Moreover, this framework only accepts the function in a Hardware Description Language (HDL) which limits flexibility.

To the best of our knowledge, the only existing framework that is scalable while also provides rich programming support is Obliv-C [10] Being an extension of the gcc compiler, Obliv-C inherits its memory management procedures. However, this inheritance comes with limitations. First, Obliv-C does not support abstraction between netlist generation and protocol execution, therefore cannot use the best netlist generation tools. Run-time netlist generation results in an additional slowdown. Second, it supports only a subset of native C data types – 16/32/64-bit integers. Finally, a custom compiler may result in unreliable binaries as exposed by Frigate [6] in Obliv-C as well as other GC frameworks.

The EMP-toolkit [11] also presents a similar framework. While its execution engine is faster than both Obliv-C and ABY, the programming features are not as rich (e.g., lack of support for conditional operation on private variables). An important feature of this framework is the support for the malicious security model while other frameworks target only the honest-but-curious model. However, their maliciously secure framework does not have a programming interface. A user can generate a netlist using the interface from the honest-but-curious framework and use it with the malicious one but will suffer from scalability issues similar to ABY.

PPMLP'20, November 9, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8088-1/20/11.
https://doi.org/10.1145/3411501.3419433

1.2 Our Contributions

In this work, we present TinyGarble2, an end-to-end GC framework that ensures fast and scalable execution while providing a rich C++ library. The framework is developed in two layers. The first layer is the protocol execution back-end that takes a pre-compiled GC-optimized netlist as input and executes the protocol. The second layer, which is a program interface to the GC back-end, supports using common arithmetic and logical building blocks to develop applications. The GC back-end internally manages the secure transfer of shares among consecutive operations (netlists) according to user-defined flags. This ensures that at any instance, only a limited number of garbled gates resides on the memory thus enabling scalable execution. In addition, the abstraction of netlist generation, protocol execution, and program interface allow users to benefit from the enhanced netlists generated with logic synthesis tools, as introduced by our earlier work TinyGarble, without having to use HDLs. Furthermore, the GC back-end has two versions for the two security models: honest-but-curious and malicious. The interfaces to these two versions are identical allowing the same program to be executed in a security model of choice by the user.

To demonstrate the efficiency and practicality of TinyGarble2, we designed a C++ library for privacy-preserving CNN inference. We built LeNet-5 [12] to run inference on the MNIST dataset with the library. Privacy-preserving inference with TinyGarble2 on one image requires 91s with a peak memory usage of 46MB. The same operation with Obliv-C requires 3.01E3s and 127MB, respectively. On evaluation on micro-benchmarks, Obliv-C shows the same level of scalability as TinyGarble2 while being at least 2.5× slower, whereas ABY demonstrates a lack of both speed and scalability.

2 DESCRIPTION OF THE FRAMEWORK

The TinyGarble2 framework is developed in two layers. The first layer is the back-end that executes the GC protocol on any given set of netlists. The second layer provides an interface between the back-end and arithmetic/logical function building blocks. The developers can directly access the back-end from the command line or using a configuration file. However, it is more convenient to use the functionalities from the second layer to develop applications.

2.1 GC Back-end

The GC back-end allows compositions of the functions with the three formats in Eqs 1 - 3 as well as any hybrid combinations. The first format with homogeneous loops represents the sequential GC introduced by TinyGarble. In this version, TinyGarble2, we provide more freedom with support for heterogeneous loops.

$$F \equiv f(f(...f()...)) \tag{1}$$

$$F \equiv f_0(f_1(...f_{Q-1}()...)) \tag{2}$$

$$F \equiv f_P(f_0(), f_1(), ..., f_{Q-1}())) \tag{3}$$

If the number of input wires and gates in the netlist of F is $|I|$ and $|G|$, respectively, the memory footprint in traditional GC execution (i.e., all existing frameworks except [5, 8, 10]) is $O(|I| + |G|)$. In contrast, the memory footprint in TinyGarble2 in the honest-but-curious model is $O(|I| + |g|_M + |i|_M)$, where, $|g|_M$ is the number of gates in the largest among the netlists that constitute the system and $|i|_M$ is the largest among the numbers of inputs to the netlists. It shows that TinyGarble2 is more effective when the number of input/output wires in the netlist of the function is negligible compared to the number of gates, which applies to the majority of the practical functions. For instance, the peak memory footprint for L convolutional layers with $M \times N$ input and C channels of $F \times F$ filters followed by maxpool with $W \times W$ windows is $O(Lb^2MNF^2C)$ in Traditional GC and $O(bMN(1 + F^2C/W^2))$ in TinyGarble2.

In the honest-but-curious model, the reduction at memory footprint comes at no additional cost, However, in the malicious model, there is a trade-off. In the malicious model, one Oblivious Transfer (OT) is performed per gate, as opposed to per input bit in the honest-but-curious model. To benefit from the run-time improvement provided by OT extension, keys of all the gates are generated and transferred together. If F is decomposed into Q parts, Q invocation of OT extension is required which increases the total run-time. However, the support for increasing the number of invocations of OT extension to reduce the memory usage provides flexibility in case the problem at hand is too large to fit into the platform.

2.2 Programming Interface

The second layer of TinyGarble2 architecture – the programming interface provides convenient access to the GC back-end for the developers. It provides functions of common arithmetic and logical building blocks (e.g., =, +, -, ×, / (division), if-else, <, >, &, |, ∧ etc.) along with necessary GC primitives. Currently, it supports integers with any bit-width from 1 to 64. Note that the GC back-end supports any arbitrary length variables. To use variables with more than 64 bits from the programming interface, the developers need to merge multiple integers. In the following, we describe different components of the programming interface.

Protocol Instantiation. The program starts with the instantiation of the desired GC back-end of either of the two security models: honest-but-curious and malicious. The rest of the interface is identical for both models. Therefore, this is the only place where the developers need to specify the security model.

Input Variables. Every input from the parties requires three operations. First, each party registers his/her respective inputs to the GC back-end. Second, they perform the key generation and transfer for the inputs. Third, each party retrieves the keys for all the inputs (from both parties) and associate them with the respective input variables. All the input keys for the evaluator are transferred through OT extension where the total time remains almost constant with an increasing number of input bits beyond a certain threshold. The three-step approach of TinyGarble2 allows the developers to register all the input variables from the evaluator before performing the key generation with only one invocation of OT extension.

Functional Building Blocks. These building blocks are wrappers around the online computation (garbling and evaluation) of the GC protocol for the common arithmetic and logical operations. They take pre-generated shares (from either OT or a previous functional block) as inputs and outputs the shares associated with the output of the function. The parties may choose together to reveal the actual value of any variable to either or both parties.

The wrappers select the netlists according to the operation and the bit-widths of the input. The compiled binary includes pointers

to the pre-compiled netlist files in the installation directory. This abstraction allows the use of the most optimized netlist for a particular operation. According to the study [6] by Mood et. al., the netlists generated by TinyGarble, our earlier work, holds less than or equal non-XOR gates compared to other frameworks. However, its enhanced efficiency comes from using standard logic synthesis tools and therefore needs the function to be written in an HDL as opposed to a programming language like C/C++. In TinyGarble2, netlists of the necessary operations are pre-compiled with TinyGarble and provided with the software distribution. As a result, developers have the convenience of programming in C++ while benefiting from the efficiency of the HDL synthesis tools.

Neural Network Building Blocks. As an optional third layer to the TinyGarble2 framework, we provide the common components for inference with Convolutional Neural Networks (CNN) - Convolution layer, Fully Connected (FC) layer, ReLU, Maxpool, and more. It also includes functionalities to reshape tensors containing secret variables. These building-blocks can be plugged into any CNN developed with C/C++. Even though in general CNNs are designed with Python, automated tools are available to convert any trained Python model to C/C++ [13, 14]. The CNN building blocks in the third layer of TinyGarble2 are mostly based on the functions from its second layer. However, for two of the blocks, namely Convolution and FC, we provide a custom implementation of the matrix multiplication operation which directly access the GC back-end.

3 EVALUATION

We evaluate the run-time and memory usage of TinyGarble2 and compare them with Obliv-C and ABY – two current frameworks with rich program interface. Both ABY and TinyGarble2 (as well as recent GC frameworks [4, 5]) sets the security parameter κ to 128. However, Obliv-C sets it to 80. We show both experimental values and values adjusted for κ for Obliv-C. Evaluation is performed on an Intel Xeon CPU E5-2650 v4 @ 2.20GHz with 128GB memory.

Matrix-multiplication. In this experiment, we compute the product of two $D \times D$ matrices of 64-bit numbers through GC. The run-time and memory usage for different values of D are reported in Table 1. ABY requires both more time and memory compared to TinyGarble2 and Obliv-C. More importantly, its memory usage is $O(D^3)$ as opposed to $O(D^2)$ for TinyGarble2 and Obliv-C. Beyond a certain value of D, it exhausts the entire available memory of the system and is terminated by the OS. TinyGarble2 uses more memory compared to Obliv-C due to the more advanced OT extension protocol which adds a constant overhead to the total memory usage. However, it also makes TinyGarble2 faster than Obliv-C. The higher run-time in Obliv-C is also an effect of run-time netlist generation as opposed to compile-time netlist generation of TinyGarble2. In practical applications involving more computation, the constant overhead of OT extension becomes negligible, and TinyGarble2 becomes more memory-efficient compared to Obliv-C. An example of such applications is presented next.

CNN Inference with LeNet-5. We implemented LeNet-5 [12] using the CNN library of TinyGarble2 to run inference on the MNIST dataset with a pre-trained quantized model. We compare the performance with only Obliv-C since ABY cannot handle such large

Table 1: Matrix multiplication through GC

Dim.	10×10		20×20		30×30		40×40	
	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.
	sec	MB	sec	MB	sec	MB	sec	MB
ABY	11.69	1.03E3	85.59	8.14E3	276.68	2.74E4	–	–
Obliv-C	2.31	3.44	22.75	4.63	67.84	6.23	164.21	8.49
Obliv-C†	3.69	5.51	36.41	7.41	108.55	9.97	262.73	13.58
TinyGarble2	1.26	8.1	12.83	12.23	41.85	14.00	103.11	18.69

†adjustment of security parameter κ
The value used in Obliv-C ($\kappa = 80$) is smaller than the current standard ($\kappa = 128$)

Table 2: Inference on one image with LeNet through GC

	Run-time in sec	Peak-memory in MB
Obliv-C	1.88E+03	79.83
Obliv-C adjusted†	3.01E+03	127.73
TinyGarble2: 64-bits	6.55E+02	110.96
TinyGarble2: 32-bits	2.91E+02	72.75
TinyGarble2: variable	9.11E+01	45.83

†adjustment of security parameter κ from 80 to 128

operations with GC. The required bit-widths of different layers of the model vary from 16 to 24 for accuracy of 96%. Moreover, the intermediate variables for computation of MACs require 64 bits. Obliv-C supports 16, 32, and 64-bit integers. However, it does not support operations involving different bit-widths. Therefore, while implementing the CNN with Obliv-C we used 64-bit integers. With TinyGarble2 we implemented three versions of the CNN: (i) all variables are 64-bits, (ii) all variables are 32-bits except for the MAC outputs which are 64-bits. (ii) the bit-widths of the input variables of different layers set to the minimum requirements and that of the MAC outputs set to 64 bits. Run-times and memory usages for inference on one image are presented in Table 2. The table shows that the three versions are respectively 4.6×, 10.3×, 33.1× faster and 14%, 29%, 64% memory efficient compared to Obliv-C.

REFERENCES

[1] A. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.
[2] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008.
[3] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.
[4] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *S&P*. IEEE, 2013.
[5] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *IEEE S&P*, 2015.
[6] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *EuroS&P)*. IEEE, 2016, pp. 112–127.
[7] N. Büscher, M. Franz, A. Holzer, H. Veith, and S. Katzenbeisser, "On compiling boolean circuits optimized for secure multi-party computation," *Formal Methods in System Design*, vol. 51, no. 2, pp. 308–331, 2017.
[8] B. Kreuter, A. Shelat, B. Mood, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation." in *USENIX Security*, 2013.
[9] D. Demmler, T. Schneider, and M. Zohner, "ABY-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*. The Internet Society, 2015.
[10] S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1153, 2015.
[11] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.
[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.
[13] P. Plonski, "keras2cpp," https://github.com/pplonski/keras2cpp, 2020.
[14] N. Mariella, "From keras to c," https://github.com/aljabr0/from-keras-to-c, 2019.