The Fusion of Secure Function Evaluation and Logic Synthesis

Siam U. Hussain | University of California, San Diego
M. Sadegh Riazi | Institute for Global Entrepreneur, University of California, San Diego
Farinaz Koushanfar | University of California, San Diego

Designing custom secure function evaluation compilers has been an active research area. However, intelligent adaptation of the integrated circuit synthesis tools outperforms these compilers. It is time for the custom compilers to embrace this trend.

In the era of big data, ensuring the privacy of sensitive content is a standing challenge. While several heuristic methodologies for privacy-preserving computing have been suggested, due to the large space of possible breaches, it is hard to ensure their resilience. Solutions based on provably secure cryptographic primitives hold promise to provide privacy guarantees within the standard security model.

In 1986, the seminal work[1] by Yao introduced the secure function evaluation (SFE) protocol named *garbled circuit* (*GC*) that allows any polynomial time two-party function to be computed efficiently without revealing the private inputs. The following year, a different approach to SFE was proposed in the Goldreich–Micali–Wigderson (GMW)[2] protocol. Over the years, many subsequent enhancements made Yao's protocol truly practical, while the GMW protocol has also been shown to be effective in some scenarios.

A common property of both of these protocols is that the underlying function is represented as a Boolean circuit, called a *netlist*. The secure computation is performed in a way such that the Boolean value associated with each wire in the netlist is shared among the two parties. Only for the output wires, the parties reveal their respective shares to learn the values associated with them. The key elements of the GC and GMW protocols are outlined in Figure 1.

One of the most crucial parts of executing a function through either GC or GMW is converting the behavioral description of the function to the netlist. On the one hand, several custom compilers supporting (or designing) various programming languages have emerged for addressing this issue. However, such custom compilers have been shown to have reliability issues and limitations in global optimization. On the other hand, techniques for interpreting a behavioral description in a Boolean format are widely researched for designing digital integrated circuits (ICs).

Design automation for the purpose of IC design is a true engineering success story; the tools have enabled us to scale our chips to billions of gates to support complicated tasks. There is a wide gap between the capabilities of conventional IC design automation tools to compile sophisticated functions and what could be achieved by the custom SFE compilers.

Our work in this area, called TinyGarble,[3] bridges this gap by formulating GC netlist generation as an atypical circuit synthesis task, which can be addressed and scaled with standard IC logic synthesis tools. Following the path of TinyGarble, Demmler et al.[4] showed that a similar approach also greatly enhances the performance of circuit generation for the GMW protocol.

In this article, we summarize the recent advances following the paradigm shift introduced by TinyGarble and several novel applications that are enabled by connecting these two seemingly separate but synergistic

Digital Object Identifier 10.1109/MSEC.2020.3049062
Date of current version: 4 February 2021

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

areas. We start with a brief overview of the GC and GMW protocols before delving into the details of circuit construction and synthesis by design automation tools along with exciting new results. We also provide a short history of the development of custom SFE compilers.

What Is GC?

Formally, Yao's GC allows two parties, Alice and Bob, to jointly compute a function $z = \mathcal{F}(x_a, x_b)$ on their private inputs: x_a from Alice and x_b from Bob. The netlist of the function $\mathcal{F}$ consists of two-input, one-output logic gates. Alice assigns each wire in the netlist with two k-b random keys corresponding to the values one and zero. For each gate, a garbled truth table is constructed by encrypting the keys for the output with the corresponding input keys.

Next, Alice sends the garbled tables along with the keys for her input values to Bob. Bob obtains the keys corresponding to his input values through oblivious transfers (OTs), which allows him to retrieve the keys without revealing the values of his inputs. He then uses these input keys to evaluate the encrypted tables gate by gate and decrypt the keys associated with the value of each wire.

> **Yao's protocol drew the interest of researchers around the world upon its appearance in 1986.**

However, the mapping of these keys to the actual values is known only to Alice. Thus, together, they share the secret value of each wire. At the final step, they reveal their respective shares for only the output wires to learn the output z.

The original work by Yao and a majority of the subsequent enhancements adopt a honest-but-curious security model. In this model, both Alice and Bob follow the agreed upon protocol but may want to deduce more from the information at hand. Recent works have shown how to make Yao's protocol secure in the malicious security model, where the parties may deviate from the correct protocol.

A number of optimizations to the GC protocol have been proposed, e.g., free-xor,[5] row-reduction,[6] half-gate,[7] and fixed-key block ciphers.[8] Among these, the most important is free-xor, as it allows the evaluation of xor, xnor, and not gates without costly cryptographic encryption and communication. Therefore, the primary optimization goal while generating the netlist for $\mathcal{F}$ is to minimize the number of non-xor gates (AND, OR, NAND, and so on). The row-reduction and half-gate optimizations reduce the size of the garbled tables for non-xor gates by 25% each.

The GMW and Beaver–Micali–Rogaway Protocols

In the GMW protocol, the value of each wire is split into two shares such that the actual value is the xor of these two. Alice and Bob each receive one share for each wire. Since the xor operation is associative, they can locally xor their respective shares of the input wires to compute the shares of the output wire of the xor gates. Thus, this protocol naturally supports free-xor.

For the non-xor gates, however, Alice computes the output of each gate for all four possible combinations of the shares of the input wires held by Bob, and Bob receives one of them through OT. To minimize the number of communication rounds, all non-xor gates at the same level of the netlist are evaluated in parallel. Thus, the round complexity of the GMW protocol depends on the circuit depth, as opposed to being constant like in the Yao protocol.

However, in settings with low network latency, this protocol has been shown to have superior performance in some cases. More importantly, it scales better to computations involving more than two parties. Even though GC was designed for two-party computations, subsequent enhancements have extended this to multiparty settings, the most notable one being the Beaver–Micali–Rogaway (BMR)[9] protocol.

A Brief History of GC Compilers

Yao's protocol drew the interest of researchers around the world upon its appearance in 1986. However, it was primarily considered a theoretical concept until the emergence of Fairplay,[10] the first realization of GC, in 2004. Fairplay introduced the secure function definition language to write the functions. The compiler converted the

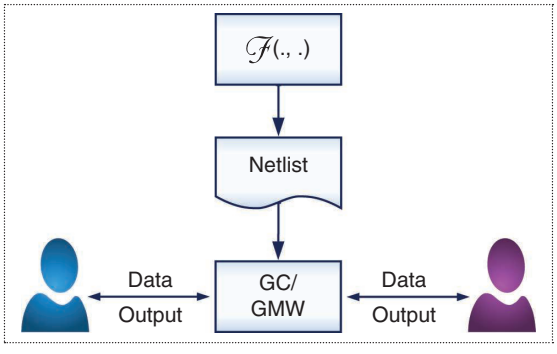

Figure 1. Outline of the GC and GMW protocols.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

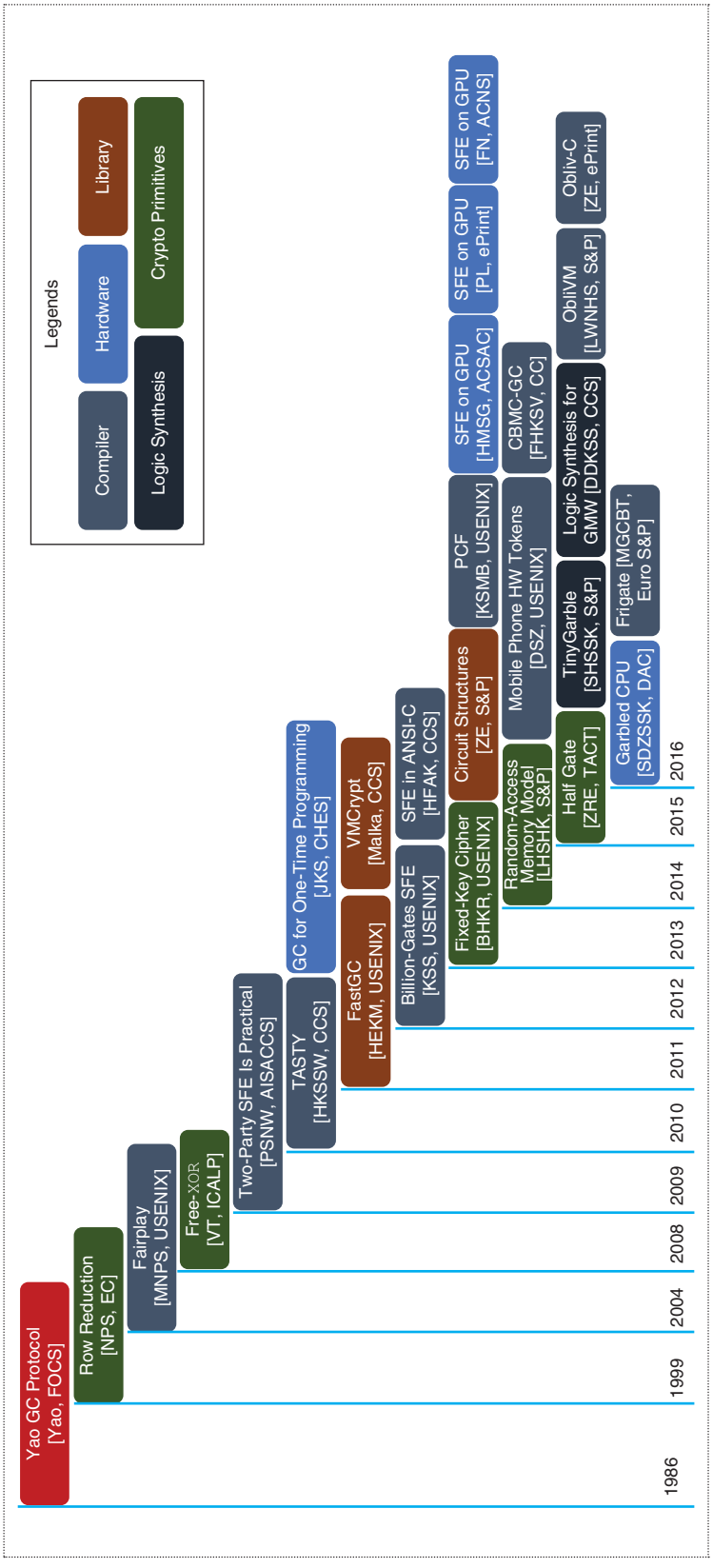

Figure 2. The most impactful works on Yao's GC over the years.

functions to a netlist in secure hardware definition language, which is later executed through the Yao protocol.

Since then, there has been a surge in the research on GC. A selected set of the most impactful works is shown on the timeline presented in Figure 2. In this section, we focus on the different compilers developed to perform SFE through the GC protocol. A brief description of the frameworks is provided in the inset.

The realization of a function through GC entails two major tasks:

1. The behavioral description of the underlying function is compiled to a netlist of Boolean logic.
2. The netlist is executed via a garbling back end.

In the frameworks PCF, KSS, CBMC-GC, TinyGarble, and Frigate, the two tasks are independent of each other. These frameworks provide more flexibility to the user because the netlist compiled by one framework can be executed by another one. Thus, the user can choose the best framework for each task.

For example, the JustGarble framework[8] provided around 20× speed up to the execution task by employing a fixed-key block cipher to garble the gates. This framework does not include the compiler. However, a netlist compiled by any of the mentioned frameworks can be executed by JustGarble to benefit from its speed up.

In contrast to this, in Obliv-C and ObliVM, the two tasks are unified. The outputs of these frameworks are compiled binary of the function to be executed securely. For these frameworks to benefit from the optimization of JustGarble, the source codes of the frameworks need to be modified.

The ABY framework provides a mixed protocol that efficiently combines secure computation based on GC, GMW, and arithmetic sharing. Similar to Obliv-C and ObliVM, this framework treats netlist generation and execution as a unified task. However, recently, Demmler et al. extended[4] the framework to accept externally generated netlists.

Custom-designed compilers enable users to write the program in a high-level language and provide a more user-friendly interface. However, such customized compilers for secure computation introduce their own domain-specific

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

languages, which usually have unfamiliar syntax, thus diminishing their original goal.

One of the most recent frameworks, Frigate, performed extensive research on the reliability of the current frameworks and found that most of them suffer from reliability issues (see "Current GC Compilers"). For example, they reported that PAL, KSS, CMBC, Obliv-C, ObliVM, and PCF crashed on programs that should have been compiled correctly. Moreover, KSS, ObliVM, and PCF generated incorrect netlists. While many of these issues were later taken care of by the respective developers, this research exposed a serious reliability issue regarding the usage of these compilers.

The Fusion of SFE and Logic Synthesis

Conventional IC logic synthesis transforms the behavioral description of a function to a netlist of Boolean gates. This concept can be traced back to as early as 1979, when IBM first started developing the logic synthesis system.[11] Since then, IC design automation and synthesis rose to become one of the most successful engineering ventures, as they have uniquely enabled the modern computing era. The logic synthesis and other automated IC design tools have raised designers' productivity by several orders of magnitude. Indeed, while the computing and information revolution is mostly credited to Moore's law scaling, the complexity hurdle has been addressed by the automated design tools. Contemporary tools and methodologies enable the automation of IC design, verification, and testing of chips across various levels of abstraction and sophistication.

The task of generating an efficient Boolean circuit (netlist) for GC is substantially similar to logic synthesis. For nearly three decades, these two fields did not cross paths until the introduction of TinyGarble[3] in 2015. TinyGarble creates a set of libraries and optimization strategies for industrial logic synthesis tools such that they can efficiently be employed to generate an optimized netlist for GC.

At the time of its publication, TinyGarble demonstrated superiority over the existing custom GC compilers. Recently, the Frigate[12] framework has been shown to outperform all other previous compilers except TinyGarble. In Table 1, the numbers of non-xor gates in selected benchmark functions generated by these two frameworks are compared. Essentially, the key to TinyGarble's efficiency is standing on the shoulders of a giant: IC logic synthesis. The synthesis tools that make TinyGarble a possibility are the same ones that enabled the design of contemporary ICs with billions of gates.

As mentioned in the previous section, the netlist generation and execution of GC can be run independently in the TinyGarble framework. The GC execution of TinyGarble supports secure two-party computation in the honest-but-curious model. However, the capability of its netlist generation tool chain goes well beyond that. For example, it has been shown[13] to be efficient for netlist generation for the BMR protocol that supports secure computation involving more than two parties. The methodology can easily be extended to any GC/GMW/BMR framework, given that they support disintegration of netlist generation and GC execution.

Inspired by TinyGarble, Demmler et al.[4] employed logic synthesis tools to generate netlists for the GMW protocol. Since the round complexity of the GMW protocol depends on the depth of the netlist, they developed a tool chain to optimize the netlist not only for size but also for depth. Their work showed a reduction of depth by up to 14% even over manually optimized netlists.

Both GC and GMW involve logic gates whose functionalities are fixed (e.g., `AND` and `OR`). Therefore, both TinyGarble and the tools of Demmler et al.[4] employ application-specified IC synthesis tools. Dessouky et al.[14] introduce protocols involving lookup tables (LUTs), which can be programmed to realize arbitrary functions. To generate the Boolean circuits, this work employs multi-input LUT-based synthesis tools, which form the core of synthesis for field-programmable gate arrays.

While the logic synthesis tools have been shown to outperform custom compilers in terms of efficiency, the development of practical privacy-preserving systems requires the consideration of several other factors, e.g., language expressibility, richer programming paradigms, and accessibility to developers. To address these issues, we recently published an enhanced version of TinyGarble, namely *TinyGarble2*,[15] in collaboration with Intel Labs. TinyGarble2 is a C framework that provides a program interface to the GC-optimized netlists generated by TinyGarble.

Another significant contribution of TinyGarble[3] is introducing and leveraging the concept of the sequential circuit to GC. We elaborate this concept next.

Sequential GC

TinyGarble's unique ability to garble sequential circuits enables scaling of the designs by leveraging the concept of a memory. As opposed to the combinational circuit, a sequential circuit has internal states stored in registers (flip-flops). At each clock cycle, the state of the circuit depends on the inputs at that cycle and current state of the circuit.

Consider a 32-b adder circuit, for example. The circuit can be represented as

1. a combinational circuit that takes two 32-b numbers and outputs the 32-b sum
2. a sequential circuit, a 1-b full adder, that computes 1 b of the sum at each clock cycle.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

Current GC Compilers

PAL (2012)[S1]: The primary goal of PAL is to generate the netlist at runtime on a mobile platform. This framework takes as the input a program in Secure Function Definition Language (SFDL) and produces a netlist in Secure Hardware Definition Language (SHDL). SFDL and SHDL were developed by the first GC framework, Fairplay[10] which is employed by PAL to execute the SHDL netlist. This framework does not consider the free-XOR optimization while generating the netlist.

KSS (2012)[S2]: The KSS framework provides inherent security against a malicious adversary. The input function is represented using a custom untyped language, and the output is a netlist in binary format that is executed in parallel with generation.

PCF (2013)[S3]: The PCF compiler takes LCC bytecode as input (generated by an LCC compiler from a C program) and generates the netlist in a condensed ASCII format. The netlist is executed through the GC back end, which supports runtime unrolling of the loops inside the program.

CBMC-GC (2014)[S4]: This compiler supports a general-purpose language, a subset of ANSI-C. It employs a bit-precise model checker, CBMC, to translate C programs into the equivalent Boolean netlist in ASCII format. This framework provides only the compiler to generate the netlist and not a GC back end for execution.

Wysteria (2014)[S5]: Wysteria is a high-level programming language and compiler that enables users to write mixed-mode programs, which include a mixture of local and secure computations.

Obliv-C (2015)[S6]: The Obliv-C language is an extension of C designed to write privacy-preserving functions for GC. The compilation and execution tasks are combined in this framework. The output is a compiled binary that executes the function securely.

ObliVM (2015)[S7]: Similar to Obliv- C, this framework also combines the compilation and execution tasks. The input function is written in Java, and the output is a Java-class file. This framework supports oblivious access to arrays when the index depends on the private data.

Frigate (2016)[S8]: The goal of this framework is to generate the netlist reliably and fast. The input function is written in a custom language that resembles C. The function is compiled to a netlist in a custom format, which is executed through its own GC back end.

References

S1. B. Mood, L. Letaw, and K. Butler, "Memory-efficient garbled circuit generation for mobile devices," in *Financial Cryptography and Data Security* (Lecture Notes in Computer Science), 2012, pp. 254–268.

S2. B. Kreuter, A. Shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries," in *Proc. USENIX Security*, 2012, pp. 285–300.

S3. B. Kreuter, A. Shelat, B. Mood, and K. R. Butler, "PCF: A portable circuit format for scalable two-party secure computation," in *Proc. USENIX Security*, 2013, pp. 321–336.

S4. M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ANSI C compiler for secure two-party computations," in *Compiler Construction* (Lecture Notes in Computer Science), Berlin: Springer-Verlag, 2014, pp. 244–249.

S5. A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *Proc. 2014 IEEE Symp. Security Privacy*, pp. 655–670. doi: 10.1109/SP.2014.48.

S6. S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation," *IACR Cryptol*. ePrint Arch, vol. 2015, p. 1153, 2015.

S7. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *Proc. 2015 IEEE Symp. Security Privacy*, pp. 359–376. doi: 10.1109/SP.2015.29.

S8. B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Proc. 2016 IEEE Euro. Symp. Security Privacy (EuroS&P)*, pp. 112–127.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

This circuit is executed for 32 clock cycles to produce a 32-b output. The state of the circuit is the carry of the previous clock cycle. In the general case, an n-b adder can be realized using an l-b adder module, executed for $\lceil \frac{n}{l} \rceil$ clock cycles.

Describing a function as a sequential circuit provides a compact representation that can scale for large input sizes. For example, Figure 3 shows the memory footprint and garbling times for different values of l for realizing a 2^{15}-b adder. As can be seen, the amount of memory required for garbling of the sequential circuit decreases linearly with l.

A major benefit of reducing the memory footprint is that, for several benchmark functions, TinyGarble can synthesize them to compact sizes that can fit within the cache. This compaction results in fewer cache misses and, therefore, a reduction in the garbling time, which is evident in Figure 3. Note that, for a certain value of l, the drop in the execution time reaches a saturation point. At this point, the netlist completely fits into the cache, and the number of cache misses is reduced to its minimum. Further reduction of the netlist size does not further improve the time. This saturation point usually occurs at a higher value of l for more complicated tasks with larger netlists.

The PCF framework also took a similar approach, where it rolls a for-loop and repeatedly garbles it instead of unrolling it in a large circuit. However, sequential circuit description is a well-known and established approach that allows us to use industrial IC compilers to generate more optimized circuits and, at the same time, keep the garbling footprint small. The comparison by Songhori et al.[3] shows that TinyGarble outperforms PCF by up to 85% in terms of the number of non-xor gates for all of the benchmarks.

Moreover, sequential description enables several (previously unreported) functions to be efficiently garbled. For example, the TinyGarble framework has been employed to devise a scalable privacy-preserving solution for stable matching,[16] substring search,[17] and deep learning.[18] Even more interestingly, the sequential format allows garbling a general-purpose processor (CPU). This was not possible by the prior custom compilers, as no sizable CPU can be built as a nonsequential circuit (a circuit without states as well as a combinational circuit). For the sake of brevity, we elaborate one application, stable matching, in this section. In the next section, we provide a brief overview of the garbled processor.

In stable matching, there are two groups of people, where every member has a preference list to be matched to a person in the other group. The stability condition requires that there should be no two persons such that they prefer each other more than their already assigned partners. In secure stable matching, the goal is to ensure the privacy of the preference lists of the members.

Table 1. A comparison of the number of the non-xors of TinyGarble versus Frigate.

Function	Frigate	TinyGarble	Improvement (%)
Sum 1,024	1,025	1,023	0.2
Compare 16,384	16,386	16,384	0.01
Hamming 160	719	159	77.89
Mult 32	995	993	0.2
MatrixMult 5 × 5 32	128,252	127,225	0.8
AES 128	10,383	6,400	38.36

This has been seen as one of the most complicated tasks in privacy-preserving computation. The memory footprint of secure stable matching with combinational circuits cannot scale for real-world group sizes.

The work by Riazi et al.[16] presents a scalable solution to this problem by employing the sequential garbling introduced by TinyGarble. The circuit block diagrams of the combinational and sequential circuits are depicted in Figure 4. The combinational circuit requires $O(n^2)$ submodules, each comprising $O(n^2 \log n)$ number of non-xor gates, where n is the number of people in each group. Since the combinational circuit has $O(n^4 \log n)$ gates, it quickly reaches the limit of circuit synthesis tools. In contrast, the sequential circuit requires only $O(n^2 \log n)$ non-xor gates. Therefore, it provides scalability to much higher set sizes. Moreover, garbling the sequential circuit requires significantly less memory.

Garbling a Processor

Custom-designed compilers usually require the user to write the program in an unfamiliar syntax. Moreover, they suffer from correctness and reliability issues, as studied by Mood et al.[12] An elegant solution to these problems is to have off-the-shelf (unmodified) standard compilers and

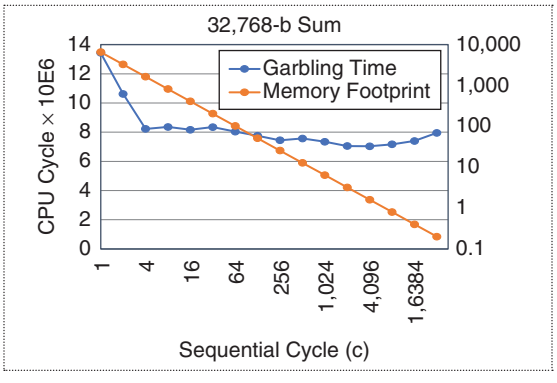

Figure 3. The effect of the sequential representation of the circuit on the garbling time and memory footprint.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

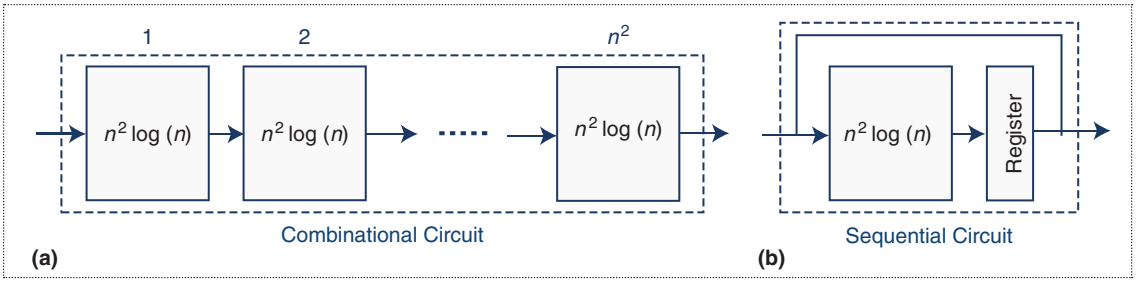

Figure 4. The (a) combinational and (b) sequential circuits for stable matching. n is the number of people in each group.

compile a user's program to the binary code. This approach has the advantage that legacy codes can be used with minimal modification and no special syntax is used.

To securely evaluate the compiled binary code on the user's private data, one has to garble a general-purpose processor. In other words, the Boolean circuit garbled in the GC protocol is a general-purpose processor that includes instruction memory, data memory, and stack along with other components. The compiled binary itself is considered as an input to the processor by being loaded into the instruction memory. The user's private data are also loaded into the data memory. Therefore, by garbling the processor, the program is securely evaluated on users' private input.

The naive adaptation of garbled processor for SFE, however, incurs a large overhead. The reason is that, for each instruction, the entire processor circuit, including instruction fetch, instruction decoding, control paths, and so on, must be garbled. Two solutions have been proposed to reduce this overhead: static[19] and dynamic reduction.[20]

Wang et al.[19] propose creating custom arithmetic logic units (ALUs) for different instructions and evaluate the corresponding ALU for each cycle. However, such coarse-grain optimization (the instruction level as opposed to the gate level) still incurs a significant overhead.

To overcome these limitations, a gate-level reduction approach is proposed in ARM2GC that dynamically identifies gates that can be processed using public data, i.e., the compiled binary. In other words, ARM2GC determines which parts of the processor can be evaluated in plaintext and which need to be garbled. The decision is based on the instruction at each clock cycle and the public/private state of the processor. The latter depends on the previous instructions and how they have affected the public and private status of different parts of the processor.

It has been shown that ARM2GC requires $8.4E+3$ and $49E+3$ times less garbled non-xor gates compared to the proposal by Wang et al.[19] for computing Hamming distances with 32- and 512-b inputs, respectively. In fact, the number of gates garbled in ARM2GC for different benchmarks is comparable or lower compared to high-level GC compilers like CBMC-GC or Frigate. In the ARM2GC framework,[20] users can write the program in a standard language and compile using verified ARM compilers. The compiled program acts as a public input to the ARM processor circuit. In contrast to custom compilers, ARM2GC relies on standard compilers that are rigorously tested and, as a result, is more reliable.

The adaptation of standard logic synthesis techniques to generate the netlist for Yao's GC protocol along with the introduction of sequential GC provide a paradigm shift in the field of privacy-preserving computation. The results of four decades of research in the field of electronic design automation are made available to the security community.

Leveraging the powerful IC synthesis tools for GC compilation uniquely allows the efficient and scalable realization of a number of exciting new privacy-preserving applications. In particular, it enables the execution of a general-purpose processor through the GC protocol. Recent advances in garbling a processor allow the users to write a function in any programming language and compute it efficiently through the GC protocol. The usage of standard and verified compilers for garbling eliminates the unreliability of custom compilers, opening the possibility of the privacy-preserving implementation of services that entail extensive computation and massive data sets, e.g., secure search, neural network, navigation, and many more. ■

References

1. A. C.-C. Yao, "How to generate and exchange secrets," in *Proc. 27th Annu. Symp. Found. Comput. Sci. (SFCS 1986)*, 1986, pp. 162–167. doi: 10.1109/SFCS.1986.25.
2. O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *Proc. Symp. Theory Comput.*, 1987, pp. 218–229. doi: 10.1145/28395.28420.
3. E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 411–428. doi: 10.1109/SP.2015.32.
4. D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.

optimized circuits for secure computation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 1504–1517. doi: 10.1145/2810103.2813678.
5. V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Proc. Int. Colloq. Automata, Languages, Program.*, New York: Springer-Verlag, 2008, pp. 486–498. doi: 10.1007/978-3-540-70583-3_40.
6. M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *Proc. 1st ACM Conf. Electron. Commerce*, 1999, pp. 129–139. doi: 10.1145/336992.337028.
7. S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," 2015.
8. M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 478–492. doi: 10.1109/SP.2013.39.
9. D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proc. 22nd Annu. ACM Symp. Theory Comput.*, 1990, pp. 503–513. doi: 10.1145/100216.100287.
10. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay-secure two-party computation system," in *Proc. USENIX Security*, 2004, p. 9.
11. J. A. Darringer, D. Brand, J. V. Gerbi, W. H. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM J. Res. Develop.*, vol. 28, no. 5, pp. 537–545, 1984. doi: 10.1147/rd.285.0537.
12. B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Proc. IEEE Eur. Symp. Security Privacy (EuroS&P)*, 2016, pp. 112–127. doi: 10.1109/EuroSP.2016.20.
13. M. S. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar, "MPCircuits: Optimized circuit generation for secure multi-party computation," in *Proc. Int. Symp. Hardware Oriented Security Trust (HOST)*, 2019, p. 275.
14. G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the communication barrier in secure computation using lookup tables," in *Proc. NDSS*, 2017, pp. 1–11.
15. S. U. Hussain, B. Li, F. Koushanfar, and, and R. Cammarota, "TinyGarble2: Smart, efficient, and scalable Yao's garble circuit," in *Proc. 2020 Workshop Privacy-Preserving Mach. Learn. Pract.*, pp. 65–67. doi: 10.1145/3411501.3419433.
16. M. S. Riazi, E. M. Songhori, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Toward practical secure stable matching," *Proc. Privacy Enhancing Technol.*, vol. 2017, no. 1, pp. 62–78, 2017. doi: 10.1515/popets-2017-0005.
17. M. S. Riazi, E. M. Songhori, and F. Koushanfar, "PriSearch: Efficient search on private data," in *Proc. 54th Annu. Design Autom. Conf.*, 2017, pp. 1–6. doi: 10.1145/3061639.3062305.
18. B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, pp. 1–6. doi: 10.1145/3195970.3196023.
19. X. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of MIPS machine code," in *Proc. Eur. Symp. Research Comput. Security*, Cham: Springer-Verlag, 2016, pp. 99–117. doi: 10.1007/978-3-319-45741-3_6.
20. E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "ARM2GC: Succinct garbled processor for secure computation," in *Proc. 56th Annu. Design Autom. Conf.*, 2019, pp. 1–6. doi: 10.1145/3316781.3317777.

Siam U. Hussain is a doctoral student in the Department of Electrical and Computer Engineering at the University of California, San Diego, San Diego, California, 92092, USA. His research focus is enabling data-intensive applications with provable privacy guarantee in practical settings. Hussain received an M.S. in computer engineering from Rice University. He is a Student Member of IEEE. Contact him at siamumar@ucsd.edu.

M. Sadegh Riazi is a fellow of the Institute for Global Entrepreneur, University of California, San Diego, La Jolla, California, 92092, USA. His research focus is privacy-preserving machine learning and secure computation. Riazi received a Ph.D. in computer engineering from the University of California, San Diego. He is a Member of IEEE.

Farinaz Koushanfar is a professor and Henry Booker Faculty Scholar in the Electrical and Computer Engineering (ECE) Department at the University of California, San Diego (UCSD), La Jolla, California, 92092, USA, where she directs the Adaptive Computing and Embedded Systems Lab. She is the cofounder and codirector of the UCSD Center for Machine-Integrated Computing & Security which launched in 2018. Before joining UCSD, she was a professor in the ECE Department at William Marsh Rice University, which she joined as an assistant professor nine years earlier. Her research addresses several aspects of efficient computing and embedded systems, with a focus on hardware and system security, real-time/energy-efficient big data analytics under resource constraints, design automation and synthesis for emerging applications, as well as practical privacy-preserving computing. Koushanfar received a Ph.D. in electrical engineering and computer science and an M.A. in statistics from UC Berkeley. She is a Fellow of IEEE. Contact her at farinaz@ucsd.edu.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 28,2024 at 03:50:21 UTC from IEEE Xplore. Restrictions apply.