Time Is Money, Friend! Timing Side-channel Attack against Garbled Circuit Constructions

Mohammad Hashemi[1], Domenic Forte[2], and Fatemeh Ganji[1]

[1] Worcester Polytechnic Institute, Worcester, MA 01609, USA
[2] University of Florida, Gainesville FL 32611, USA
dforte@ece.ufl.edu {mhashemi,fgangi}@wpi.edu

Abstract. With the advent of secure function evaluation (SFE), distrustful parties can jointly compute on their private inputs without disclosing anything besides the results. Yao's garbled circuit protocol has become an integral part of secure computation thanks to considerable efforts made to make it feasible, practical, and more efficient. For decades, the security of protocols offered in general-purpose compilers has been assured with regard to sound proofs and the promise that during the computation, no information on parties' input would be leaking.
In a parallel effort, timing side-channel attacks have proven themselves effective in retrieving secrets from implementations, even through remote access to them. Nevertheless, the vulnerability of garbled circuit frameworks to timing attacks has, surprisingly, never been discussed in the literature. This paper introduces Goblin, the first timing attack against commonly employed garbled circuit frameworks. Goblin is a machine learning-assisted, non-profiling, single-trace timing SCA, which successfully recovers the garbler's input during the computation under different scenarios, including various GC frameworks, benchmark functions, and the number of garbler's input bits. In doing so, Goblin hopefully paves the way for further research in this matter.

Keywords: Secure Function Evaluation; Timing Side-channel Analysis; Clustering; Non-profiling Attack; Single-trace Attack.

1 Introduction

Secure function evaluation (SFE) has had an immense impact on the field of cryptography. Practical implementations of general SFE have been proposed and flourished after the introduction of garbled circuits (GCs) by Yao [93]. It has found several applications including secure multi-party computation [6,22,23,57], functional encryption [28,27,80], key-dependent message security [3,2], homomorphic encryption [76,26], and recently, quantum circuits [9]. The key premise of GCs is that it allows two parties to evaluate any (known) function on their respective inputs x and y without violating their privacy. Besides real-world applications foreseen for GCs traditionally (e.g., credit evaluation function, background-

and medical history checking, privacy-preserving database querying, etc. [53,82]), nowadays GCs have found applications in privacy-preserving genome analysis [43], email spam filtering [36], image processing [12] and machine learning and statistical analysis [71,24,14,75], just to name a few. To face obstacles preventing further adoption of GCs in real-world systems, optimization techniques have been developed, aiming to reduce communication and computation costs. Here we focus on two of the most acknowledged methods, namely free-XOR [53] and half-gates [97]. Similar to other optimization mechanisms, the main argument put forward by these techniques is that security is not compromised for the sake of being efficient. However, the question is whether this holds true when implementing these protocols. This becomes even more critical since today's applications of GCs (or potential ones) encompass services run on distributed computing systems, cloud services, connected devices, etc.

Timing side-channel analysis. Irrespective of what cryptographic functions are embedded in programmable instruction set processors, such systems can exhibit observable features and data-dependent behavior that leak information about users' data/keys from the implementation. As a prime example, timing side channels can be observed when the time taken to execute a piece of code depends on the secret variables [52,63,72,90]. In this regard, two broad categories of timing side channels can be identified: instruction-related and cache-related cf. [92]. The former refers to the number or type of instructions executed along a path that can differ depending on the values of secret variables. On the other hand, cache-related timing side channels correspond to the case, where the memory subsystem may behave differently based on the values of secret variables. In both categories, CPU instruction execution, specifically the branch prediction, memory access, and data caches, have been exploited to launch successful SCA on the cryptographic systems cf. [78,1,7,29,20]. Recently, the security of open-source cryptographic libraries and implementations of protocols (excluding GC) has just been evaluated in an extensive study [45], where the vulnerability of some of those libraries to timing SCA has been demonstrated. More interesting and inspiring from the perspective of this work is the gap between academic research and cryptographic engineering when it comes to timing SCA.

SCA against GC constructions. Despite the achievements made to prove the security of GC schemes, there is a gap between what theoretical findings have suggested and what observations can be made by parties involved in executing a GC protocol. The only example of studies addressing this gap is a recent attack proposed by Levi et al., which leverages the side-channel leakage as a result of using a secret, global value for free-XOR, correlated with the power consumption of the garbler's device [55]. Although multiple assumptions have been made to launch the attack, their attack has successfully disclosed the global value used to perform free-XOR optimization. Now the question is whether one can go even beyond this attack and perform timing SCA and whether some of the assumptions made in [55] can be relaxed in that case.

Generally speaking, timing attacks feature outstanding properties that make them more interesting [45]: first, timing attacks can be launched remotely, in-

cluding cases of running code in parallel to the victim code without the need for local access to the target computer; second, timing attacks can be carried out covertly. In light of this state of affairs, this work attempts to answer the following question: *Is it possible to reveal parties' input by observing the timing information leaking when executing a GC protocol?* More specifically, we answer this question positively for free-XOR- and half-gates-optimized constructions. The contribution of our work is as follows.

Our **Contributions** are summarized as follows.

1. We introduce *Goblin*, the first non-profiling, single-trace timing SCA that successfully extract the user's input, which by definition, should have been kept secret. To better demonstrate the power of our attack, we compare it with the recent attack in [55]. The power SCA in [55] has successfully extracted the global secret used in free-XOR optimization, whereas Goblin focuses entirely on the recovery of the garbler's input. Needless to say that even with the help of the disclosed secret, the garbler's input could not be fully recovered. Moreover, in contrast to [55], Goblin's effectiveness is limited to neither circuits with a minimum number of input gates nor gate types (XOR or AND).
2. Goblin is machine-learning assisted in disclosing the garbler's input, regardless of its size. For this purpose, k-means clustering is applied, where no manual tuning or heuristic leakage models are needed. It is, of course, advantageous to the attacker and allows for scalable and efficient attacks.
3. Last but not least, our paper highlights the vulnerabilities of multiple available garbling tools to timing SCA. We believe that this constitutes a basis for studying the SCA with respect to GC.

2 Background and Adversary Model

Notations. We follow a standard notation typically used in SFE-related literature. $\in_R$ denotes uniform sampling, $\|$ is used to show concatenation of bit strings. $\langle a, b\rangle$ represents a vector with two components a and b, whereas $a \| b$ is its bit string representation. A *gate* is denoted by $W_c = g(W_a, W_b)$ with input wires W_a and W_b, output wire W_c and $g: \{0,1\}^2 \rightarrow \{0,1\}$.

2.1 Yao's Garbled Circuit (GC)

One of the most widely studied SFE approaches, designed to meet the needs of Boolean circuits, is garbling [56,58]. The first protocol within the context of GC is Oblivious transfer (OT). We consider 1-out-of-2 OT, which is a two-party protocol with the following definition. The sender P_1 possesses two secret messages m_0, and m_1, and the receiver P_2 has a selection bit $i \in \{0,1\}$. By executing the protocol, P_2 learns m_i, but not m_{1-i}, while the sender P_1 does not learn anything about i.

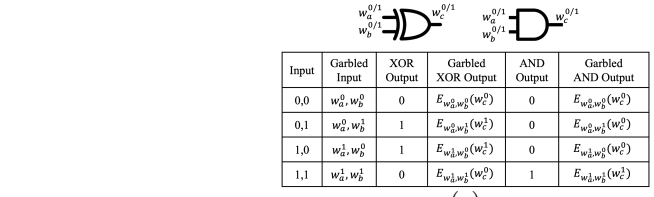
Garbling. The protocol execution begins with garbling the circuit C, where the garbler (P_1) randomly chooses secrets w_i^j with the garbled value of $j \in \{0,1\}$ on each wire W_i. Needless to say that it is expected that w_i^j does *not* reveal any information about j. Practical implementations of Yao's GC, e.g., [86] considered in this paper, represent each of the logical "0" and "1" values with n-bit values, where n is often referred to as the security parameter. In this sense, w_i^j (so-called token) is the encryption of the concatenation of j and $(n-1)$-bit values drawn uniformly. After generating the tokens, the garbler creates a garbled table T_i for each gate G_i, where each row of the gate truth table is encrypted output with regard to the tokens, and the output of the gate is called a "ciphertext," illustrated in Figure 1.(a) as the output of the operand $E(\cdot)$, i.e., the encryption operation (symmetric key operations, e.g., fixed-key block cipher). Since the table rows can reveal information about the internal wire values, they are permuted. The main property of T_i is that its output can be recovered given a set of garbled inputs, while this process does not leak any information about the garbler's and evaluator's (P_2) inputs. For this, along with T_i's, the token corresponding to the garbler's input value is obliviously transferred to P_2 through OT. P_2 is then able to obtain the garbled output by evaluating the garbled circuit gate by gate using the tables T_i and receiving j for the output wire from P_1 cf. [87]. Garbling of the output wires of the circuit can be skipped so that two parties learn (only) the output of the circuit [53].

Optimizations of Yao's GC. Reducing the computation and communication costs of SFE protocols has been an objective of numerous studies. Among optimization techniques introduced in the literature, **free-XOR** has attracted considerable attention since it reduces the cost on the garbler side effectively, namely by 25%. To reduce garbler's cost, the wire values are garbled as presented in Figure 1.(b). For any gate G_i, $w_i^1 = w_i^0 \oplus R$ for some secret, global $R \in_R \{0,1\}^{n}$[3]. Here, for the sake of simplicity, let $(A, A \oplus R)$ and $(B, B \oplus R)$ denote the wire labels. **half-gates** protocol complements the free-XOR protocol in the sense that not only are XOR gates evaluated for free, but also AND gates are garbled using only two ciphertexts (see Figure 1.(c)). Since Goblin is interested in recovering the garbler's input, in Figure 1.(c), we show how the half-gates are generated on the garbler's side, where garbler knows which inputs she wants to garble (for more information about the whole process, see [97]).

2.2 *k*-means Algorithm

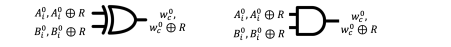
The main goal of clustering algorithms, like k-means, is to group samples of a set with some common features into subsets, i.e., *clusters*. With regard to the pairwise distances, clusters are made around the mean vectors, which are called *centroids* [91,37]. k-means aims to partition N members of a set into k clusters in a way that each member of a cluster has a close value to the centroid of the cluster [91]. To be more specific, k-means finds partitions (clusters) $p =$

[3] For specifics of the encryption function in the free-XOR protocol, see [13,34].



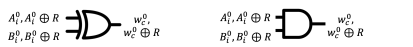
Input	Garbled Input	XOR Output	Garbled XOR Output	AND Output	Garbled AND Output
0,0	w_a^0, w_b^0	0	$E_{w_a^0,w_b^0}(w_c^0)$	0	$E_{w_a^0,w_b^0}(w_c^0)$
0,1	w_a^0, w_b^1	1	$E_{w_a^0,w_b^1}(w_c^1)$	0	$E_{w_a^0,w_b^1}(w_c^0)$
1,0	w_a^1, w_b^0	1	$E_{w_a^1,w_b^0}(w_c^1)$	0	$E_{w_a^1,w_b^0}(w_c^0)$
1,1	w_a^1, w_b^1	0	$E_{w_a^1,w_b^1}(w_c^0)$	1	$E_{w_a^1,w_b^1}(w_c^1)$

(a)



Input	Garbled Input	XOR Output	Garbled XOR Output	AND Output	Garbled AND Output
0,0	A_t^0, B_t^0	0	$w_c^0 = A_t^0 \oplus B_t^0$	0	$E_{A_t^0,B_t^0}(w_c^0)$
0,1	$A_t^0, B_t^0 \oplus R$	1	$w_c^0 \oplus R$	0	$E_{A_t^0,B_t^0 \oplus R}(w_c^0)$
1,0	$A_t^0 \oplus R, B_t^0$	1	$w_c^0 \oplus R$	0	$E_{A_t^0 \oplus R,B_t^0}(w_c^0)$
1,1	$A_t^0 \oplus R, B_t^0 \oplus R$	0	w_c^0	1	$E_{A_t^0 \oplus R,B_t^0 \oplus R}(w_c^0 \oplus R)$

(b)



Known input	Other input	Garbled Input	XOR output	Garbled XOR output	AND Output	Garbled AND output
0	0	A_t^0, B_t^0	0	$w_c^0 = A_t^0 \oplus B_t^0$	0	$E_{B_t^0}(w_c^0)$
	1	$A_t^0, B_t^0 \oplus R$	1	$w_c^0 \oplus R$		
1	0	$A_t^0 \oplus R, B_t^0$	1	$w_c^0 \oplus R$	0	
	1	$A_t^0 \oplus R, B_t^0 \oplus R$	0	w_c^0	1	$E_{B_t^0 \oplus R}(w_c^0 \oplus R)$

(c)

Fig. 1: Garbled gates look-up table with (a) no optimization, (b) free-XOR optimization, and (c) half-gate optimization.

$\{p_1, p_2, \cdots, p_k\}$ for the dataset $c = \{c_i\}_{i=1}^n$ to minimize

$$\min_{p,\{\mu_j\}_1^k} \sum_{j=1}^{k} \sum_{c_i \in p_j} ||c_i - \mu_j||^2,$$

where μ_j is the mean of all examples assigned to j^{th} centroid [39]:. Here the squared Euclidean distance is one of the commonly applied distance measures applied to minimize the total cluster variance [85].

2.3 Cache Architecture

Modern x86 processors comprise three cache layers: L1, L2, and L3, with data inclusively across all levels [74,25]. Figure 2 presents the Intel core-i7 cache architecture. Each CPU core has a dedicated L1 and L2 cache, with the former divided into data and instruction caches of 32KB each [74]. L2 cache is shared across CPU threads and has a larger capacity (256KB [74]). The largest cache, L3, is shared across all CPU cores with an 8MB capacity [74].

Processor instructions fall into three categories: memory read/write, control flow (data processing), and arithmetic/logic operations [25,15]. The execution time of the latter is determined by the type of operation and the number of arithmetic-logic unit (ALU) calls [15]. Memory access time, however, depends on whether the instruction is accessing RAM or cache [94,64,81,32,73].

For efficient memory access management, the CPU stores operation results in the cache hierarchy (L1, L2, L3) and an instantiation in RAM [54,98]. On data request, the CPU checks the data availability in this order: L1 cache, L2 cache, L3 cache, and finally RAM [54].

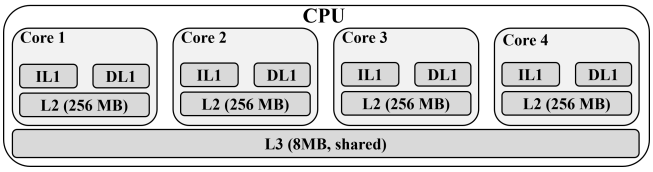


Fig. 2: Intel core-i7 cache architecture [74].

Cache eviction strategies. Four eviction strategies can be considered as highlighted in [31]. The first and second use static and dynamic eviction sets, respectively, with static access patterns. The third uses both dynamic eviction set and access pattern, enabling fully automated attacks. The fourth uses a static eviction set with a dynamic access pattern, but is less efficient [31].

2.4 Adversary Model

The security of GCs has been considered in two main paradigms, namely honest-but-curious and malicious adversary models. The latter reflects the situation, where a party potentially adopts an arbitrary attack strategy. On the other hand, honest-but-curious parties follow the protocol honestly, although they may attempt to learn additional information from the execution, similar to the one launching SCA. This has also been well-formulated in [5], where it is suggested that Yao's GC reveals no side-information beyond the function being computed, i.e., no information about parties' inputs leaks. One closely relevant adversary model is devised for server-aided or cloud-assisted, where the standard SFE protocol is run with the help of a server (or a small set of them), which does not contribute to running the protocol by giving inputs, but by making their computational resources available to the parties cf. [50,16,8,17,49,48,11,10]. In the proposed setting, the server is instantiated by a public cloud service provider, where parties who need more computational power (e.g., the garbler) can outsource their computations. In such scenarios, the server can be honest-but-curious [51]. Our model goes one step further and take into account any -even unprivileged- access to the CPU during the execution of the protocol.

Our adversary model assumes that the parties and the server are independent, i.e., none of them collude [50,21,48]. In practice, given the consequences in terms of losing the reputation and legal actions, it is reasonable to assume that the server will not collude with the parties. The adversary is capable of performing local code execution, potentially even on the same core. Additionally, the adversary must possess the capability to evict data from the cache to the main memory. Note that although throughout the paper, we refer to the server as the entity collecting the timing information, this does not rule out the fact that any entity with the capabilities mentioned above can launch the attack.

3 Timing Side-channel Leakage in Garbling Tools: An Observation

Broadly speaking, timing side-channels leak due to the dependency of the time taken to execute a piece of software code on the values of secret variables. Here, two types of timing side-channels are of interest, namely instruction-related and cache-related ones. The former indicates that the number or type of instructions executed along a path depends on the values of secret variables. In contrast, cache-related timing side channels refer to the difference due to the memory subsystem behavior depending on the values of secret variables, e.g., a cache hit takes a few CPU cycles. Still, a miss takes hundreds of cycles cf. [92]. By analyzing the code line-by-line, the adversary can find and further exploit such vulnerabilities. Nevertheless, manual analysis of the timing characteristics of a code is challenging as it requires thorough knowledge of the code and the platform on which it is executed. The broad range of existing tools for automatically checking timing side-channel leakage can help pinpoint such vulnerabilities. In doing so, we select a recent tool recommended in the literature [44], namely SC-Eliminator [92]. Among the most important features of SC-Eliminator is the fact that, in view of available garbling protocols, it can analyze codes written in C/C++. To this end, using an LLVM compiler performs static analyses to identify the sensitive variables and timing leakage associated with them, given a program and a list of secret inputs.

GC tools. To explore whether GC frameworks would be vulnerable to timing SCA, we selected 5 open-source tools written in C/C++, which mostly support AES-NI (Advanced Encryption Standard New Instruction) instruction set (for more features of these tools cf. [38]). As a result, they have made computing AES encryptions on modern processors efficient, and consequently, the computation cost of GC is reduced drastically. **JustGarble** [4] is a library for garbling and evaluating circuits licensed under GNU GPL v3 license; however, JustGarble does not support communication or circuit generation and is, therefore, not a general-purpose framework. Nevertheless, it has become a cornerstone of various frameworks, e.g., [87,70,47,35,33,30]. The reason behind JustGarble's efficiency is its ability to make only one AES call per garbled-gate evaluation which makes it far faster than any prior reported results [4]. JustGarble exploits the cryptographic permutations realizable by fixed-key AES acting like a public random permutation [4]. Although this might be a strong assumption cf. [33,35], thanks to its efficiency and the theoretical foundation laid for JustGarble, it has been used in a wide variety of MPC and GC frameworks cf. [70,30].

Songhori et al. [86,87] extended JustGarble in **TinyGarble**, a highly compressed and scalable sequential GC, which is a self-contained framework that can directly be used in MPC applications [38]. Three steps are taken in TinyGarble, namely converting a function defined in Verilog to a netlist format, converting that netlist to a custom circuit description (SCD), and finally, securely evaluating the resulting Boolean circuit using a garbled circuit protocol. This flow has been considered a strict improvement over JustGarble as TinyGarble further in-

Table 1: The number of leaky IF conditions (IF) in various frameworks. (for a detailed report, refer to Appendix A)

Framework	IF
TinyGarble [86] (half-gate)	4
TinyGarble [86] (free-XOR)	7
JustGarble [42]	11
EMP-toolkit [67]	0
Obliv-C [95]	4
ABY [18]	0

cludes recent protocol and circuit optimizations. Nevertheless, and irrespective of the flexibility of TinyGarble for producing hardware circuits, changes made to JustGarble have introduced timing side-channel leakage, as will be discussed in Sections 5-6.

In contrast to TinyGarble, which is an extension of Verilog, **Obliv-C** is an extension of C that executes a GC protocol in a two-party setting [96]. The C language is extended by adding an obliv qualifier that is applied to C types and constructs. By enforcing typing rules, obliv types remain secret unless explicitly revealed. In doing so, it is suggested that oblivious functions and conditionals could modify public data, if they are executed within a qualified obliv block, where the code is always executed cf. [96,95]. In addition to the data security achieved by means of these rules, modular libraries can be easily developed when using Obliv-C. Thanks to this property, Obliv-C has found application in, e.g., linear regression [24], decentralized certificate authorities [46], aggregated private machine-learning models [89], classification of encrypted emails [36] and stable matching [19].

Besides the frameworks mentioned above, we also took EMP-toolkit [67] and ABY [18], libraries developed in C++, into account. EMP-toolkit is composed of multiple MPC frameworks and allows for executing circuit-based protocols due to the available circuit generation and cryptographic libraries. ABY library offers a mechanism for mixing protocols, including optimized versions of Yao's garbled circuit protocol.

Our observations. As mentioned earlier, as a first, we examined the possibility of mounting timing SCA against GC frameworks enumerated above. In such an attack scenario, the adversary attempt to take advantage of possible unbalance if-else statements (branches). The adversary can assume that different operations performed to generate garbled inputs in free-XOR and half-gate optimized Yao's GC protocols (see Figure 1) can result in leakage if neither a constant-time implementation nor branch-less assignments are used for sensitive branches. To examine this, SC-Eliminator [92] is applied against TinyGarble [86], JustGarble [42], EMP-toolkit [67], Obliv-C [95], and ABY [18]. Table 1 contains the number of leaky IFs for this experiment. When taking a close look at the list of leaky IFs among the set of leaky IFs, we observed unbalanced IF statements in the garbled-input generation, i.e., garbled inputs were generated in a

secret-dependent manner. The existence of these unbalanced IFs demonstrates the likelihood of timing attacks to be successfully mounted against them. According to the results in Table 1, EMP-toolkit [67] and ABY [18] do not have any leaky IFs. Nevertheless, we should stress that although SC-Eliminator does not find any vulnerability in terms of leaky IFs in these frameworks, this does not rule out the possibility of other attacks. Next, we introduce our attack, Goblin, to leverage the timing side-channel leaking from existing unbalanced IF statements.

4 Goblin and Its Building Blocks

The main steps in Goblin's flow are: (1) filling the cache with junk by using junk generator (JG) to evict the garbler secret from the cache. This step aims to maximize the CPU core's access time to the global secret (R) from the cache and capture the CPU cycles corresponding to each gate connected to input wires (i.e., gates in the input layer); (2) measuring the time on the CPU, including the time taken to generate garbler token, linked to the input size; (3) recovering the garbler's secret (i.e., garbler's input) after pre-processing the acquired CPU cycles and running a clustering algorithm.

4.1 Our Eviction Method: Junk Generator

We presume that the server and parties are independent (see Section 2.4), i.e., the adversary lacks knowledge of the cache slice function or the victim's physical addresses; hence, static eviction set and static access pattern strategies are impossible to employ [31]. As implementing a dynamic eviction set and static access pattern strategy requires informing the adversary about the target's replacement policy, it is not feasible [31]. Hence, our JG adopts the dynamic eviction set and dynamic access pattern strategy [31]. Our JG is, in fact, an enhancement of the dynamic eviction set and access pattern method in [31]. Our attack shares similarities with Evict+Time attacks presented in the literature [65]. Specifically, in our attack, JG accesses the memory frequently in the form of reading and writing from/to it similar to [77]; although in their attack, the adversary should first determine which part of the critical information is accessed during the encryption. In contrast, Goblin does not require this as the time difference between garbling "1" and "0" reveals the input bit ("0/1") directly. To maximize this time difference, the JG algorithm recursively generates eviction sets and performs memory accesses randomly. Despite requiring many eviction tests, this approach needs minimal system information, enabling automated attacks on unknown systems. It is also considered more efficient than the static eviction set and dynamic access pattern strategy [31]. Cache eviction can also be achieved by reading the cache line [77]. Yet, we opted to generate junk on the fly to bypass CPU memory management [31]. Despite the simplicity of iterative `For` loops used in our JG (see Appendix D), we chose the recursive function for JG to generate junk indefinitely, considering the unknown duration of a circuit garbling process.

4.2 Measuring Time on CPUs

After the JG boosts the difference between the input bit-dependent execution times, the time can be measured. According to Martin et al. [68], to measure the time without breaking the software, there are three main sources to take advantage of cf. [66]: (1) internal, hardware time sources, e.g., timestamp counters; (2) external time sources, e.g., external interrupts; and (3) creating a virtual clock, for instance, the virtual clock implementation on multi-processor systems with shared memory [79]. Without loss of generality, we focus on how timing information can be retrieved using the first option, namely `rdtsc`. The Read Timestamp Counter `rdtsc` is an x86 instruction that returns the value of the CPU timestamp counter (TSC) register. In general, the `TSC` register is shared with every user with any level of privileged access [66]; therefore, it can be accessed by: (1) a privileged/non-privileged user who has complete control over the CPU; (2) a service provider who shares the processor with the victim, such as cloud servers [68]; (3) a virtual-machine user with a privileged/non-privileged access level, who runs a process on a shared processor with the victim (e.g., cross-virtual machine attacks) [66]. Hence, the adversary can have either privileged/non-privileged access to (1) the CPU on which the garbling scheme is running, (2) the CPU of the service provider's system, or (3) a cross-virtual machine to share the processor with the victim running the garbling scheme. What could make a difference is that an unprivileged attacker cannot precisely control the garbler's execution and interrupt it, unlike a privileged attacker. Nevertheless, if the attacker can figure out when the garbling process begins, or use a trigger signal such as a cache-based side channel [83], then the collected traces can be aligned based on that timing information [62]. Therefore, without loss of generality, we consider aligned timing measurements to mount the attack, similar to [69,41]. For the sake of demonstration, we have inserted the `rdtsc` before and after the garble gate function in the frameworks source code, which are all publicly available, and achieved the time stamps based on their difference.

Resolution of timing measurements. The timestamps provided by `rdtsc` often have a resolution between 1 and 3 cycles on modern CPUs cf. [61]. For example, on AMD CPUs until the Zen microarchitecture, a cycle-accurate resolution can be obtained; however, more recent generations come with a significantly lower resolution as the register is only updated every 20 to 35 cycles. Another example is Intel Core $i7-7700$ Processors, i.e., what has been used in this study, where the `rdtsc` register is updated every cycle [40]. Nevertheless, although it might be thought that lower resolutions might make performing attacks more challenging, Goblin is not affected since it requires mainly the difference between two readings with the same resolution (see Section 6 for more details). Therefore, in contrast to attacks requiring repetition when relying on `rdtsc`, it is not needed for Goblin to do so and use the average timing differences over all executions. We stress that Goblin is a single-trace attack, i.e., thanks to the gate-by-gate operation in GC frameworks, the time difference directly driven from `rdtsc` is a collection of time stamps associated with gates. We should also add that our

attack is an example of a timing attack, meaning that we believe other methods for acquiring the timing information can definitely be applied.

4.3 Recovering Garbler's Input

Counting the gates in the input layer. According to our adversary model, we assume that the adversary is neither the garbler nor the evaluator. Therefore, there is no information about the circuit, input size, and gate types in the input layer. Here we describe how this information is retrieved by Goblin when the garbler uses JustGarble, as an example of GC tools. This example is selected due to its broad applications (see Section 3) and its role as the core of other garbling frameworks, e.g., ones considered in our study [87,96]. Listing 1 illustrates a high-level description of JustGarble primary functions. In Listing 1, `NF`, `LF`, `GT`, `IF`, `INL`, `WL`, `GC`, and `OL`, denoted in Lines 1–9, refer to the number of fan-outs, location of fan-outs, gates' types, the value of filled input fan-out, initial input values, wire labels, Garbled circuit, and output labels, respectively.

According to the protocol flow of JustGarble (see, Listing 1), in the first step, the garbler's tokens for zero and one logical values (`IL`) are constructed through `createNewWire` (Listing 1 line 5). Then, the parser function (the label corresponding function `createInputLabels` Listing 1 line 3) starts parsing the simple circuit description (`SCD`) file and `g_init` files, which contain information about the circuit and the garbler's input values. The parser function learns about the circuit (`GT`) and locates the fan-in and fan-out of the input layer gates (`LF` and `NF`) that are connected to the garbler input based on `g_init` file information. For every input, the `createInputLabels` is called once for garbler label and once for the evaluator label of the input, twice per input in total. At this point, Goblin starts counting the number of `createInputLabels` calls and calculating the number of input layer gates as half of the total number of `createInputLabels` function calls. Afterward, the gates are garbled one by one by calling the `garbleCircuit` function (Listing 1 line 9), starting from the input layer gates, where the garbler's and evaluator's inputs are fed, before proceeding to the following layer gates. This fact allows Goblin to count the CPU cycle associated with each gate in the input layer by knowing the number of input gates.

Goblin against free-XOR optimization. When the framework starts garbling the gates, output labels (`OL`) and garbled tables (`GT`) are generated in the order provided in the `SCD` file. As JustGarble, similar to various modern garbling frameworks, utilizes the free-XOR optimization to generate garbler tokens for input value 1, the garbler must access the R frequently. When free-XOR optimization is enabled, `GarbleCircuit` function (Listing 1 line 9) skips line 11 to line 14 of the Listing 1. Therefore, regardless of whether the input is known or secret, it checks the type of the input gate (`GT`) and treats all inputs as a secret. If the gate type is `XOR`, including all gates categories that are considered `XOR` in GC protocols (`INV`, `XOR` and `XNOR` gates), it generates the `OL` as the XOR results of labels 0 and 1 (Listing 1 line 16); otherwise, the `OL` is constructed through a

series of encryptions, see, Listing 1, line 18 to 22. It is clearly observable that in the last part of the encryption, Listing 1 line 14 and between lines 25 and 28, if the garbler input value is "1", one more encryption, one memory access, and one XORing take place, which can result in the input dependency observable in the execution time of garbling process.

In other words, when garbling `AND` (non-XOR) gates (including (`AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN`), there is an unbalanced `if` condition, which means a longer execution time for input value one. This is the point that Goblin takes advantage of differences in execution time of the garbling process for each gate due to their input value. If R is available in the L1 level of the cache, this difference is subtle and, in most cases, negligible to the time of the encryption process. Hence, to maximize the difference between the time taken to generate tokens for input 0 and 1, the JG (see Section 4.1) starts filling the cache with junks parallel to the execution of the `createNewWire` function (Listing 1 line 5) to enforce CPU to fetch R into L1 cache from RAM, which increases the execution time difference between 0 and 1 token generation. To boost the effect of JG, Goblin first finds the CPU core and thread on which the garbling process is happening by calling the `LSCPU` instruction; then asks the server to assign the JG task to the same thread, or if not possible, at least to the same core on which the garbling process is happening. It should be indicated that neither any privilege is needed nor any restriction on assigning the JG to the same core is posed as it fills the shared L3 cache level; nevertheless, assigning JG to the same core as the garbling process core will result in faster cache filling and fewer errors as JG first fills L1 and L2 level cache.

Goblin against half-gate optimization. Though JustGarble doesn't support half-gate optimization, subsequent frameworks like TinyGarble and Obliv-C do. Despite this, Goblin remains effective against these frameworks. When half-gate optimization is enabled, `HalfGarbleGate` (see Listing 2) is called by `GarbleGate`. When the input value (`IF`) is zero and the gate type (`GT`) is `ANDGATE`, the function bypasses the garbling process, assigning a constant to `OL`, thus reducing the execution time compared to the garbling process for input value one or other gate types. If the input value is one, encryption occurs (Listing 2 line 11), introducing an unbalanced `if` path and creating a dependency between the garbling process execution time and the input value. Just like with free-XOR optimization, Goblin capitalizes on these differences in execution times due to the unbalanced `if` conditions in Listing 2, lines 3 and 8. The rest of the steps are not interesting for Goblin because they do not hold any information about the secret (garbler's input), and the above-mentioned information is adequate to launch the Goblin; therefore, from now on, Goblin can continue the attack from an offline phase.

Pre-processing the acquired CPU cycles. As explained before, when employing free-XOR optimization, the attacker expects to see a significant difference between the CPU cycle of `INV`, `XOR`, and `XNOR` gates and other gate types, including `AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN` gates (refer to Section 5 for more information). This significant difference is because in the free-XOR optimization, as its name implies, an XOR-type gate is garbled by simply

```
def JustGarble(g_init, SCD):
    NF, LF, GT = createNewWire(g_init, SCD)   #Pasrses the circuit, locate the fan-outs, and generates wire labels.
    IF, INL = createInputLabels(NF, LF) #Fills tokens to input fan-outs (called twice per garbler input).
    GC, OL, TT = garbleCircuit(IF, IFS, WL, GT) #Generates garbled tables and Garbled output tokens.
def createNewWire(g_init, SCD):
    for i in SCD[0]: #first line of SCD, which contains the information about input layer gates
        IF[i][0] = randomBlock();
        IF[i][1] = xorBlocks(R, IF[i][0]);
def garbleCircuit(IFS, WL, GT):
    R = AESEcbEncryptBlks(AES_Key)
    if(IFS == known):
        GC, OL = HalfGarbleGate(GT, IF)
        return GC, OL
    else: #(IFS == secret):
        if(GT == XORGATE):
            OL = XorBlock (IFS, R) #free-XOR optimization
        else: #if(GT == ANDGATE)
            mask1, mask2, mask3, mask4=AESEcbEncryptBlks(AES_Key,4)
            #AND encryptions
            OL = XorBlock(mask1 , mask2)
            if (IFS == 1):
                OL = XorBlock(OL , R);
        GC = [XorBlock(OL, mask3), XorBlock(OL, mask4)]
    if(gate_location is in input_layer): #Generates associate garbler tokens to be transferred to Evaluator.
        if(g_init == 0):
            TT = IF;
        else:
            TT = xorBlocks(R, IF);
    return GC, OL, TT
```

Listing 1: Protocol flow of primary functions of JustGarble.

using the XORing operation that takes a few CPU cycles. On the other hand, garbling other types of gates, such as an AND gate, requires reading/writing from/to memory and cipher generation, which results in extra memory reads; hence, accumulating these leads to a drastic increase in CPU cycles. This is evident thanks to the definition of this optimization technique and the number of operands included in the computation of those gates, see Figure 1.(b). When employing clustering to discover the garbler's input in a non-profiled manner, this difference causes the gate types to be dominant centroids of the clustering algorithm over the input values. To overcome this challenge, Goblin first divides the CPU cycle into the number of subgroups equal to the number of available gate types, i.e., `AND` (`AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN`) and `XOR` (`INV`, `XOR` and `XNOR` gates, hereafter called `XOR` gates) with regard to the median of the CPU cycles. Afterward, it normalizes each subgroup of CPU cycles by employing *z-score* normalization, and finally, concatenates the normalized data to form the CPU cycle array while maintaining the order of captured CPU cycles. Normalization minimizes the difference between the CPU cycle requirements of XOR and AND gate types, consequently improving the SR.

The first step is more complicated in a case where the half-gates optimization is enabled. Specifically, according to our observation, not only garbling the XOR gates exhibits a significantly larger number of CPU cycles compared to other

```
def HalfGarbleGate(GT, IF):
    R = AESEcbEncryptBlks(AES_Key)
    mask1, mask2 = AESEcbEncryptBlks(AES_Key,2)
    if(IF[0] == 0):
        if(GT == ANDGATE):
            OL = mask1 #XorBlock(mask1, 0)
        else: #if(GT == XORGATE):
            OL = XorBlock(mask1, IF[1])
    if(IF[0] == 1):
        if(GT == XORGATE):
            OL = mask1 #XorBlock(mask1, 0)
        else: #if(GT == ANDGATE):
            OL = XorBlock(mask1, R)
    GC = XorBlock(OL, mask2)
    if(gate_location is in input_layer): #Generates associate garbler
    tokens to be transferred to Evaluator.
            if(g_init == 0):
                TT = IF;
            else:
                TT = xorBlocks(R, IF);
    return GC, OL, TT
```

Listing 2: HalfGarbleGate function flow.

gate types, but also there is a dramatic difference in the number of CPU cycles in the OR/NOR gates garbling process. There is, of course, a reason behind this, namely how gates with truth tables containing an odd number of ones (e.g., `AND`, `NAND`, `OR`, `NOR`, etc.) can be expressed and constructed. Generally speaking, these gate can be defined as $G : (v_a, v_b) \rightarrow (\alpha_a \oplus v_a) \wedge (\alpha_b \oplus v_b) \oplus \alpha_c$, where v_a and v_b are logical values and α_a, α_b, and α_c are constant values cf. [97]. For `AND` gate, α values are set to 0, whereas for OR gate, they are set to 1. Therefore, it is unsurprising that the CPU cycles collected when garbling OR/NOR gates compose a cluster different from the others. In the same vain, one can also observe that it takes more time for the garbler to generate the garbled `OR/NOR` gate with input "0", as opposed to `AND/NAND` gates with input "1". Therefore, contrary to the case of free-XOR optimization, where AND/NAND and OR/NOR can be considered as belonging to the same type, it is challenging to make a distinction between AND/NAND gates with input "0" and OR/NOR gates with input "1". This overlap results in inaccurate clustering since the algorithm puts both into one cluster, although they should be put into two different clusters due to their inputs.

To counter this challenge, Goblin applies the following additional data scaling technique before the normalization to force the pattern to match other gate types (i.e., a larger number of CPU cycles for input 1). First, similar to the free-XOR case, the CPU cycle collected from the input gates $\{c_i\}_{i=1}^{n}$ should be partitioned into subsets corresponding to different gate types: XOR/XNOR, AND/NAND, and OR/NOR. For this, Goblin calculates 66^{th} percentiles of elements in $\{c_i\}_{i=1}^{n}$ and assign the elements larger than that to the subset c_{OR}. The remaining elements of $\{c_i\}_{i=1}^{n}$ are assigned to AND and XOR subsets similarly as done in the free-XOR case: the larger elements are assigned to c_{AND} by considering the median of the $\{c_i\}_{i=1}^{n} \setminus c_{AND}$. The remaining elements are then assigned to the subset corresponding to the XOR/XNOR gates. Afterward, Goblin applies the

transformation $t_i = ac_i + b$ for $c_i \in c_{OR}$, where a and b are calculated as

$$a = \frac{\text{Max}(c_{AND}) - \bar{c}_{AND}}{\text{Max}(c_{OR}) - \bar{c}_{AND}}, \quad b = \bar{c}_{AND} - a \cdot \bar{c}_{OR},$$

where $\text{Max}(\cdot)$ and $\bar{c}$'s denote the maximum and the average of the subsets, respectively. After this step, normalization is applied, similar to the free-XOR case.

Extracting garbler's input through clustering. After obtaining the pre-processed data, Goblin launches the clustering algorithm to determine each garbler's input bit. As Goblin applies normalization to the CPU cycle data, the gate types' dominance in the centroids has vanished; therefore, Goblin clusters CPU cycles into only two clusters corresponding to input zero and input one, regardless of the gate types. To disclose the input bits, Goblin keeps track of the $\text{Max}(\{c_i\}_{i=1}^n)$ before normalization. When the clustering process is over, all cluster members that include the maximum element are labeled as "1", meaning that the garbler input bit is "1"; consequently, other cluster includes c_i's corresponding to garbler's input bit "0".

4.4 Performance Metric

Let $\mathbf{c}_i$ be a leakage measurement, i.e., the number of CPU cycles, for a garbler input $x = x_1 \cdots x_n$ with n-bits corresponding to n wires giving the garbler's input to the circuit. For instance, for a garbled 128-bit AES design, $n = 128$. To evaluate the effectiveness of our attack, we calculate its success rate of recovering the garbler's input given a *single* trace $\{c\}_i^n$. Note that Goblin is a non-profiling attack; hence, as opposed to profiled attacks, no leakage profile is made and used during the attack. k-means clustering algorithm is used as a distinguisher so that any observation c_i is assigned to either cluster p_0 or p_1 associated with input bit x_i being "0" or "1". Precisely, the success rate is defined as follows.

$$\text{SR} := \sum_{j \in \{0,1\}} \sum_{i=1}^{n} \Pr(c_i \in p_j \mid x_i = j).$$

To put this simply, SR indicates how many bits are correctly disclosed out of n bits in the garbler's input. Note that this definition aligns with the general case considered in SCA-related literature [88]. In this context, we consider the success rate of order 1, i.e., the probability that the correct key is ranked first.

5 Experimental Results

We ran the JustGarble, TinyGarble and Obliv-C frameworks, publicly available via GitHub repositories [42,86,95]. Garbler and evaluator codes ran on two systems with Linux Ubuntu 20, 16 GB of memory, and an Intel Core $i7-7700$ CPU 3.60GHz CPU. Two systems were connected through a local area network (LAN)

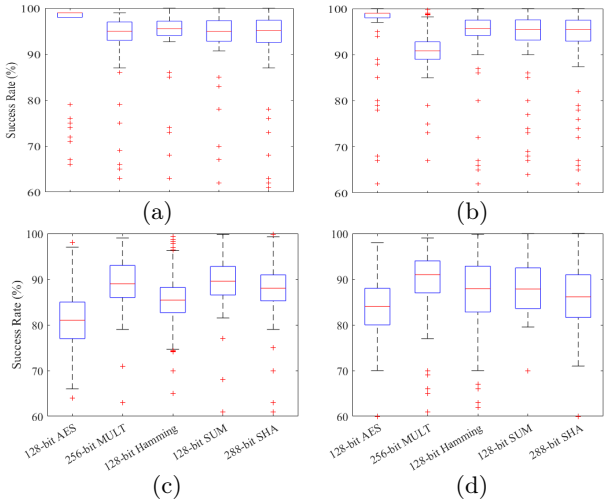


Fig. 3: SR of Goblin for 1000 randomly chosen inputs given to GC garbled by TinyGarble [87] with (a) free-XOR, (b) half-gate optimizations, (c) JustGarble [42], and (d) Obliv-C [95].

cable. As garbling process might access R anytime during garbling process, to force CPU to fetch R from RAM to L1 level cache in maximum possible cases, we started JG as soon as the garbling process begins. This can be easily determined by calling non-privileged CPU instructions showing which applications run on each core. Moreover, we assigned the JG to the same core that generates garbled circuits on the garbler system. To capture each trace, i.e., multiple time stamps, we used `rdtsc` as discussed before in Section 4.2. We have also used the k-means clustering algorithm implemented in Matlab 2021.

5.1 Results for Benchmark Functions

To evaluate the efficacy of Goblin, we have targeted the commonly-used benchmark functions, including 128-AES, 288-SHA3, 256-bit Multiplier, 128-bit Summation, and 128-bit Hamming garbled by JustGarble [42], TinyGarble [87], and Obliv-C [95] (results for the benchmark functions with various input sizes can be found in Section 5.3). For this purpose, to calculate the success rate (SR), we have applied various garbler's inputs and provided the statistics in this section. Launching Goblin against all combinations of inputs is impractical due to the massive number of input combinations (i.e., for a 256-bit Multiplier, the attack had to be launched 2^{256} times); therefore, we have chosen 1000 random inputs to run Goblin. For each of these inputs, a single trace is captured that has multiple time stamps. In the k-means algorithm setting, the centroids are chosen at 100 different starting values, and the algorithm returns the result for the least within-cluster sums of point-to-centroid distances.

Figure 3 shows the SR when free-XOR or half-gate optimization was enabled. The red lines in the boxes indicate the average SR of the attack against these

benchmark functions. It is observable in Figure 3.(a) that the attack achieved a better SR when launched against the AES benchmark compared to, e.g., the 256-bit Multiplier. The reason is three-fold. First, only 1000 inputs are tested; therefore, the results might vary. Second, the input layer of the 256-bit Multiplier contains more XOR gates than the AES, which are more challenging because of the subtle difference between the number of clock cycles taken for "1" and "0". Third, per input, notice that Goblin is a non-profiling, single-trace attack, meaning that it receives one timing measurement per gate (and per input bit, consequently); hence, the more input bits, the better Goblin determines them. This is further studied in Section 5.2.

Compared to Figure 3.(a), Figure 3.(b) corresponding to the half-gates optimization shows an overall reduced SR for the same benchmark functions. This is because of the increase in the number of gate types to be identified for the same number of input bits and observations, consequently. Needless to say, even for circuits with various gate types, such as AES, Goblin achieved an average SR of more than 90%, which means the effect of variation in the gate types does not affect Goblin's SR drastically. Imperfect process of filling the L3 level cache with junk accounts for the outliers in Figure 3. The implication of this is that the availability of R in the L1 cache level of the garbler core decreases the execution time difference between garbler 0 and 1 token generation. However, these outliers happen barely, i.e., in 11 out of 1000 experiments, which means the JG has a small error. Note that even for the outliers, Goblin still revealed the garbler's input with a range of 60% to 100% SR.

5.2 Scalability of Goblin

To test Goblin's scalability, we have launched Goblin against three benchmark functions, including MULT, SUM, and Hamming, with a range of input sizes between 128 and 1024. Figure 4 illustrates the results, where Figure 4.(a) and Figure 4.(b) depict the free-XOR and half-gate optimization results. As shown in Figure 4.(a), increasing the input size increases the minimum and average SR for virtually all cases. This SR increment is because Goblin has a broader range of data to cluster, which means it has more observations to compare with one another. Similar to previous experiments, outliers can be observed in Figure 4. To reduce the number of outliers, the natural question to ask is whether it is possible to launch Goblin without JG. We conducted experiments to answer this questions and found out that for JustGarble [42] and Obliv-C [95], the SR could decrease dramatically (close to 50%) due to the small difference between the execution times for garbler's input "0" and "1." Nonetheless, for TinyGarble [86], it is indeed possible to mount the attack with high SR without using JG (see Appendix C).

5.3 Impact of the Number of Traces

In previous expriments in this section, to evaluate the effectiveness of our attack, we selected 1000 random inputs since capturing CPU cycles for all inputs is

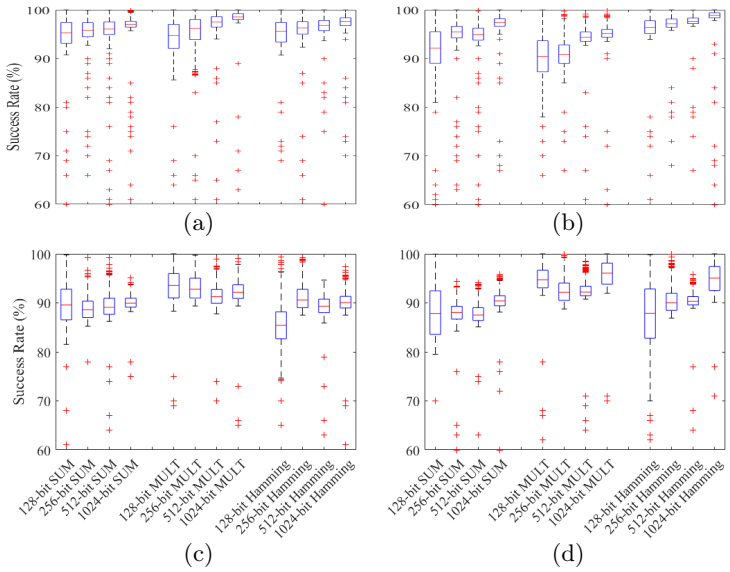


Fig. 4: SR of Goblin against benchmark functions for a range of input bits garbled by TinyGarble [86] with (a) only free-XOR optimization, (b) half-gate protocol, (c) JustGarble [42], and (d) Obliv-C [95] for 1000 randomly chosen inputs.

impractical and infeasible. This can directly impact the variance in our results. To investigate this, we collected CPU cycles after feeding powers of tens (from 10-100, 000) random inputs into the 128-bit SUM, Hamming, and MULT benchmark functions, i.e., the ones demonstrating a fairly high variance (see, Figure 3). Figure 5 illustrates the SR of Goblin when being launched against a range of CPU cycle traces. As can be seen, increasing the number of CPU cycle traces results in increasing the SR of Goblin. We have observed that for a higher number of traces, SR exhibits less variance, and the average settles around 97% in all cases, except for 128-MULT. The reason behind this is the variation in the gate types as discussed before. Note that since Goblin is a single trace attack, each trace is processed by Goblin individually. In other words, the increase in the number of traces does not impact each attack but reduces the variance of the overall results. Therefore, to judge the effectiveness of Goblin, it is recommended to use more traces. We could not do this in the first place due to the time-consuming process of collecting traces for all benchmark functions. Nonetheless, comparing the results for 1000 and 100, 000 traces, the change in the average SR is subtle.

6 Discussion

Relative accuracy of `rdtsc`. For applications using `rdtsc`, successive calls must have a difference that accurately reflects the number of cycles between two calls. This is referred to as "relative accuracy" cf. [68], meaning that any measurement through `rdtsc` is accurate with regard to the previous call/measurement.

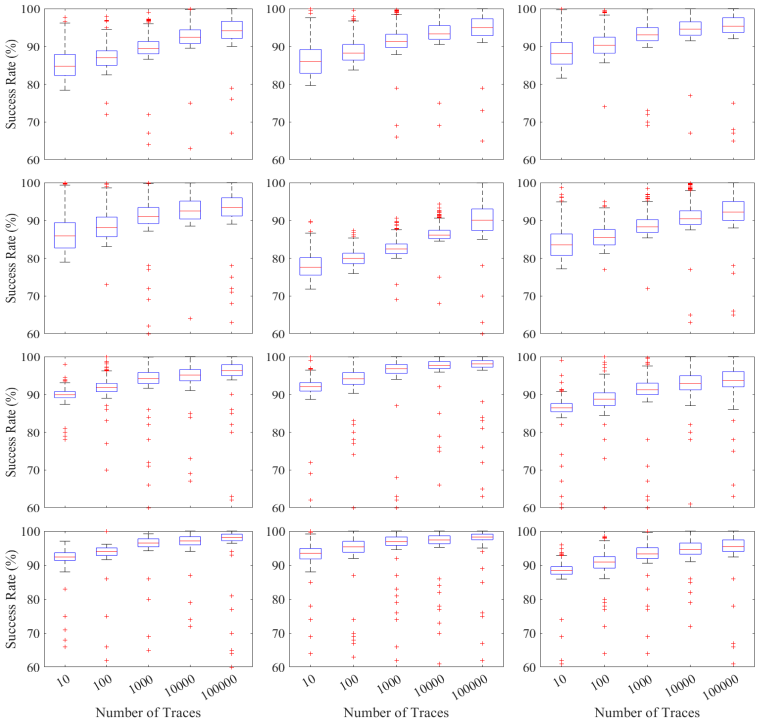


Fig. 5: SR of Goblin against (a) 128-bit SUM, (b) 128-bit Hamming, and (b) 128-bit MULT for a range of 10-100,000 randomly chosen inputs (first to last row: JustGarble [42], Obliv-C [95], TinyGarble [86] with free-XOR, and with half-gate optimizations).

The relative accuracy does not pose any constraint to the application since they must tolerate some variations as `rdtsc` instruction's number of cycles can vary due to the state of caches, DVFS, scheduling, etc. [68]. Similarly, Goblin is resilient against variations as long as the variation is smaller than the difference between the number of cycles spent on garbling the XOR and non-XOR gates (in order of tens of thousands of cycles).

Limited resolution of `rdtsc` on some platforms. As introduced in Section 4.2, `rdtsc` can have various resolutions depending on the platform. In the same vein, as explained about the relative accuracy of the time read using `rdtsc`, the resolution cannot impact the effectiveness of Goblin. The point is that as long as the XOR gates can be distinguished from non-XOR ones, Goblin can successfully extract the garbler's input. For this purpose, it is necessary to have at least a resolution comparable to the number of cycles taken to garble the XOR gates (couples of tens cycles, e.g., 80 cycles as observed in our experiments).

6.1 Potential Countermeasures

To come up with a countermeasure against Goblin, one should first determine factors contributing to Goblin's success. Here we describe these factors and emphasize that if they are considered and encountered when proposing a framework, the likelihood of Goblin's success can decrease.

The coding style of the framework. Frameworks like EMP-toolkit [67], Obliv-C [95], and ABY [18] securely tackle the vulnerability in unbalanced `IF` statements by generating both 0 and 1 garbler's tokens, although it's less optimized than one-token-per-input methods in TinyGarble [87] and JustGarble [42].

Memory management. Assigning R to a fixed memory address reduces memory access time. Usage of registers can lead to overwrites, forcing the CPU to fetch R from RAM and causing time variation in token generation. Most frameworks like EMP-toolkit [67], Obliv-C [95], JustGarble [42], and ABY [18] fixed R's address, but TinyGarble [87] used registers in token generations, leading to possible overwrites when using JG.

Can restricting access stop Goblin? Restricting high-resolution timer access can deter the Goblin attack, but also negatively impact certain unprivileged applications like adb, cargo, Docker [59]. It's noted that an attacker could still use a counting thread to establish a timestamp [60,84,61], which could even have higher resolution than the `rdtsc` instruction on Intel CPUs [84].

7 Conclusion

Nowadays, several applications, including multi-party computation, rely on the efficient implementations of GC.To achieve this efficiency, many optimizations, such as free-XOR and half-gates, have been presented to reduce the cost of garbling progress. This paper has introduced Goblin, the first machine learning-assisted, non-profiling, single-trace timing SCA against GC frameworks. Specifically, Goblin targets frameworks using free-XOR and half-gate by collecting and analyzing the time stamps of the garbling process by reading the time stamp counter, i.e., calling `rdtsc`. In doing so, the garbler's inputs that should have been kept secure can be disclosed without prior knowledge about the circuit being garbled. In this regard, Goblin can be run in parallel to the garbling framework without requiring any privileged access. Goblin has also been proven to be scalable when targeting large circuits. We have studied several cases, including various GC frameworks, benchmark functions, and the number of garbler's input bits. Under different scenarios, Goblin disclosed the garbler's input with high probability. Further, we have discussed Goblin's success factors and countermeasures against that.

8 Responsible Disclosure

Corresponding authors and/or owners of GitHub repositories of the affected frameworks [86,95,42] were contacted about their GC framework vulnerabilities presented in this paper.

9 Acknowledgments

This work has been supported partially by Semiconductor Research Corporation (SRC) under Task IDs 2991.001 and 2992.001 and NSF under award number 2138420.

References

1. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: International Conference on Information and Communications Security. pp. 112–121. Springer (2006)
2. Applebaum, B.: Key-dependent message security: Generic amplification and completeness. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 527–546. Springer (2011)
3. Barak, B., Haitner, I., Hofheinz, D., Ishai, Y.: Bounded key-dependent message security. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 423–444. Springer (2010)
4. Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: 2013 IEEE Symp. on Security and Privacy. pp. 478–492. IEEE (2013)
5. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Proc. of the 2012 ACM Conf. on Computer and Comm. security. pp. 784–796 (2012)
6. Benhamouda, F., Lin, H.: K-round multiparty computation from k-round oblivious transfer via garbled interactive circuits. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 500–532. Springer (2018)
7. Bernstein, D.J.: Cache-timing attacks on AES (2005)
8. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., et al.: Secure multiparty computation goes live. In: International Conference on Financial Cryptography and Data Security. pp. 325–343. Springer (2009)
9. Brakerski, Z., Yuen, H.: Quantum garbled circuits. In: Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing. pp. 804–817 (2022)
10. Carter, H., Lever, C., Traynor, P.: Whitewash: Outsourcing garbled circuit generation for mobile devices. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 266–275 (2014)
11. Carter, H., Mood, B., Traynor, P., Butler, K.: Outsourcing secure two-party computation as a black box. Security and Communication Networks **9**(14), 2261–2275 (2016)
12. Chen, D., Chen, W., Chen, J., Zheng, P., Huang, J.: Edge detection and image segmentation on encrypted image with homomorphic encryption and garbled circuit. In: 2018 IEEE International Conference on Multimedia and Expo (ICME). pp. 1–6. IEEE (2018)

13. Choi, S.G., Katz, J., Kumaresan, R., Zhou, H.S.: On the security of the "free-xor" technique. In: Theory of Cryptography Conference. pp. 39–53. Springer (2012)
14. Cock, M.d., Dowsley, R., Nascimento, A.C., Newman, S.C.: Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data. In: Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security. pp. 3–14 (2015)
15. Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., Sadeghi, A.R.: Losing control: On the effectiveness of control-flow integrity under stack attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 952–963 (2015)
16. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Annual International Cryptology Conference. pp. 378–394. Springer (2005)
17. Damgård, I., Ishai, Y., Krøigaard, M., Nielsen, J.B., Smith, A.: Scalable multiparty computation with nearly optimal work and resilience. In: Annual International Cryptology Conference. pp. 241–261. Springer (2008)
18. Demmler, D., Schneider, T., Zohner, M.: Aby-a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
19. Doerner, J., Evans, D., Shelat, A.: Secure stable matching at scale. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1602–1613 (2016)
20. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid prototyping for microarchitectural attacks. In: USENIX Security Symposium (2022)
21. Feige, U., Killian, J., Naor, M.: A minimal model for secure computation. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing. pp. 554–563 (1994)
22. Garg, S., Srinivasan, A.: Garbled protocols and two-round mpc from bilinear maps. In: 2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS). pp. 588–599. IEEE (2017)
23. Garg, S., Srinivasan, A.: Two-round multiparty secure computation from minimal assumptions. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 468–499. Springer (2018)
24. Gascón, A., Schoppmann, P., Balle, B., Raykova, M., Doerner, J., Zahur, S., Evans, D.: Privacy-preserving distributed linear regression on high-dimensional data. Proc. Priv. Enhancing Technol. **2017**(4), 345–364 (2017)
25. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**, 1–27 (2018)
26. Gentry, C., Halevi, S., Vaikuntanathan, V.: i-hop homomorphic encryption and rerandomizable yao circuits. In: Annual Cryptology Conference. pp. 155–172. Springer (2010)
27. Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Proceedings of the forty-fifth annual ACM symposium on Theory of computing. pp. 555–564 (2013)
28. Gorbunov, S., Vaikuntanathan, V., Wee, H.: Functional encryption with bounded collusions via multi-party computation. In: Annual Cryptology Conference. pp. 162–179. Springer (2012)
29. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 955–972 (2018)

30. Groce, A., Ledger, A., Malozemoff, A.J., Yerukhimovich, A.: Compgc: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive (2016)
31. Gruss, D., Maurice, C., Mangard, S.: Rowhammer. js: A remote software-induced fault attack in javascript. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13. pp. 300–321. Springer (2016)
32. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+ flush: a fast and stealthy cache attack. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13. pp. 279–299. Springer (2016)
33. Gueron, S., Lindell, Y., Nof, A., Pinkas, B.: Fast garbling of circuits under standard assumptions. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 567–578 (2015)
34. Guo, C., Katz, J., Wang, X., Weng, C., Yu, Y.: Better concrete security for half-gates garbling (in the multi-instance setting). In: Annual International Cryptology Conference. pp. 793–822. Springer (2020)
35. Guo, C., Katz, J., Wang, X., Yu, Y.: Efficient and secure multiparty computation from fixed-key block ciphers. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 825–841. IEEE (2020)
36. Gupta, T., Fingler, H., Alvisi, L., Walfish, M.: Pretzel: Email encryption and provider-supplied functions are compatible. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 169–182 (2017)
37. Hastie, T., Tibshirani, R., Friedman, J.H., Friedman, J.H.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, vol. 2. Springer (2009)
38. Hastings, M., Hemenway, B., Noble, D., Zdancewic, S.: Sok: General purpose compilers for secure multi-party computation. In: 2019 IEEE symposium on security and privacy (SP). pp. 1220–1237. IEEE (2019)
39. Hettwer, B., Gehrer, S., Güneysu, T.: Applications of machine learning techniques in side-channel attacks: a survey. J. of Cryptographic Engineering **10**(2), 135–162 (2020)
40. Intel Corporation: Intel Core i7 Processors. `[Online]https://www.intel.com/content/www/us/en/products/details/processors/core/i7.html` [Accessed: Jan.30, 2023] (2017)
41. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! a fast, cross-vm attack on aes. In: Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17. pp. 299–319. Springer (2014)
42. irdan: Justgarble framework. `[Online]https://github.com/irdan/justGarble` [Accessed Jan.30, 2023] (2014)
43. Jagadeesh, K.A., Wu, D.J., Birgmeier, J.A., Boneh, D., Bejerano, G.: Deriving genomic diagnoses without revealing patient genomes. Science **357**(6352), 692–695 (2017)
44. Jancar, J.: The state of tooling for verifying constant-timeness of cryptographic implementations. `[Online]https://neuromancer.sk/article/26` [Accessed: Feb.7, 2023] (2021)
45. Jancar, J., Fourné, M., Braga, D.D.A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.A., Acar, Y.: "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 632–649. IEEE (2022)
46. Jayaraman, B., Li, H., Evans, D.: Decentralized certificate authorities. arXiv preprint arXiv:1706.03370 (2017)

47. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: {GAZELLE}: A low latency framework for secure neural network inference. In: 27th USENIX Security Symp. (USENIX Security 18). pp. 1651–1669 (2018)
48. Kamara, S., Mohassel, P., Raykova, M.: Outsourcing multi-party computation. Cryptology ePrint Archive (2011)
49. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18. pp. 195–215. Springer (2014)
50. Kamara, S., Mohassel, P., Riva, B.: Salus: a system for server-aided secure function evaluation. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 797–808 (2012)
51. Kamara, S., Mohassel, P., Riva, B.: Salus: A system for server-aided secure function evaluation. Cryptology ePrint Archive (2012)
52. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Annual International Cryptology Conference. pp. 104–113. Springer (1996)
53. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: Intrl. Colloquium on Automata, Languages, and Programming. pp. 486–498. Springer (2008)
54. Lai, C.H., Zhao, J., Yang, C.L.: Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In: Proceedings of the 54th Annual Design Automation Conference 2017. pp. 1–6 (2017)
55. Levi, I., Hazay, C.: Garbled-circuits from an sca perspective: Free xor can be quite expensive... Cryptology ePrint Archive (2022)
56. Lindell, Y., Pinkas, B.: A proof of yao's protocol for secure two-party computation. eccc report tr04-063. In: Electronic Colloquium on Computational Complexity (ECCC) (2004)
57. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Annual Intrl. Conf. on the theory and applications of cryptographic techniques. pp. 52–78. Springer (2007)
58. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation. J. of cryptology **22**(2), 161–188 (2009)
59. Lipp, M., Gruss, D., Schwarz, M.: Amd prefetch attacks through power and time. In: USENIX Security Symposium (2022)
60. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: {ARMageddon}: Cache attacks on mobile devices. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 549–564 (2016)
61. Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D.: Take a way: Exploring the security implications of amd's cache way predictors. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. pp. 813–825 (2020)
62. Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D.: Platypus: Software-based power side-channel attacks on x86. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 355–371. IEEE (2021)
63. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: 2016 IEEE international symposium on high performance computer architecture (HPCA). pp. 406–418. IEEE (2016)

64. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE symposium on security and privacy. pp. 605–622. IEEE (2015)
65. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. ACM Computing Surveys (CSUR) **54**(6), 1–37 (2021)
66. Lyu, Y., Mishra, P.: A survey of side-channel attacks on caches and countermeasures. Journal of Hardware and Systems Security **2**(1), 33–50 (2018)
67. Malozemoff, A., Wang, X., Katz, J.: Emp-toolkit framework. `[Online]https://github.com/emp-toolkit` [Accessed Jan.30, 2023] (2022)
68. Martin, R., Demme, J., Sethumadhavan, S.: Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: 2012 39th Annual International Symposium on Computer Architecture (ISCA). pp. 118–129. IEEE (2012)
69. Moghimi, A., Irazoqui, G., Eisenbarth, T.: Cachezoom: How sgx amplifies the power of cache attacks. In: Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. pp. 69–90. Springer (2017)
70. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 591–602 (2015)
71. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE symposium on security and privacy (SP). pp. 19–38. IEEE (2017)
72. Mowery, K., Keelveedhi, S., Shacham, H.: Are AES x86 cache timing attacks still feasible? In: Proceedings of the 2012 ACM Workshop on Cloud computing security workshop. pp. 19–24 (2012)
73. Mushtaq, M., Mukhtar, M.A., Lapotre, V., Bhatti, M.K., Gogniat, G.: Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for RSA. Information Systems **92**, 101524 (2020)
74. Nakamoto, A.: W-shield: Protection against cryptocurrency wallet credential stealing. In: Workshop on Security and Privacy in E-Commerce 2018. pp. 71–107 (2018)
75. Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., Taft, N.: Privacy-preserving ridge regression on hundreds of millions of records. In: 2013 IEEE symposium on security and privacy. pp. 334–348. IEEE (2013)
76. Ostrovsky, R., Paskin-Cherniavsky, A., Paskin-Cherniavsky, B.: Maliciously circuit-private FHE. In: Annual Cryptology Conference. pp. 536–553. Springer (2014)
77. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Cryptographers' track at the RSA conference. pp. 1–20. Springer (2006)
78. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive (2002)
79. Percival, C.: Cache missing for fun and profit (2005)
80. Sahai, A., Seyalioglu, H.: Worry-free encryption: Functional encryption with public keys. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 463–472 (2010)
81. Saxena, A., Panda, B.: Dabangg: A case for noise resilient flush-based cache attacks. In: 2022 IEEE Security and Privacy Workshops (SPW). pp. 323–334. IEEE (2022)
82. Schneider, T.: Practical secure function evaluation. In: Informatiktage. pp. 37–40 (2008)

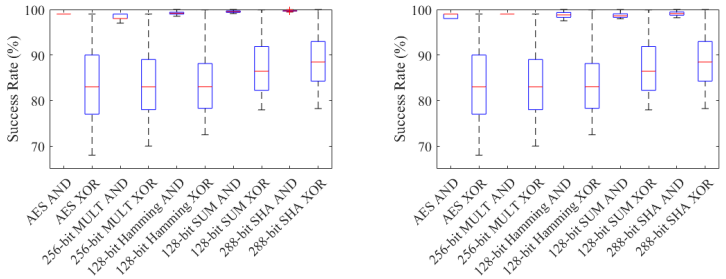
83. Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., Mangard, S.: Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 587–600 (2018)
84. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: Using SGX to conceal cache attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer (2017)
85. Sherali, H.D., Tuncbilek, C.H.: A squared-euclidean distance location-allocation problem. Naval Research Logistics (NRL) **39**(4), 447–469 (1992)
86. Songhori, E., Siam, H., Riazi, S.: Tinygarble framework. `[Online]https://github.com/esonghori/TinyGarble` [Accessed Jan.30, 2023] (2019)
87. Songhori, E.M., Hussain, S.U., Sadeghi, A.R., Schneider, T., Koushanfar, F.: Tinygarble: Highly compressed and scalable sequential garbled circuits. In: 2015 IEEE Symp. on Security and Privacy. pp. 411–428. IEEE (2015)
88. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Annual Intrl. Conf. on the Theory and Applications of Cryptographic Techniques. pp. 443–461. Springer (2009)
89. Tian, L., Jayaraman, B., Gu, Q., Evans, D.: Aggregating private sparse learning models using multi-party computation. In: NIPS Workshop on Private Multi-Party Machine Learning (2016)
90. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating fine grained timers in xen. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 41–46 (2011)
91. Whitnall, C., Oswald, E.: Robust profiling for DPA-style attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 3–21. Springer (2015)
92. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 15–26 (2018)
93. Yao, A.C.C.: How to generate and exchange secrets. In: 27th Annual Symp. on Foundations of Computer Science (sfcs 1986). pp. 162–167. IEEE (1986)
94. Yarom, Y., Falkner, K.: Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14). pp. 719–732 (2014)
95. Zahur, S., Kerneis, G., Necula, G.: Obliv-C secure computation compiler. `[Online]https://github.com/samee/obliv-c` [Accessed Feb.2, 2023] (2018)
96. Zahur, S., Evans, D.: Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive (2015)
97. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 220–250. Springer (2015)
98. Zhao, L., Iyer, R., Makineni, S., Newell, D., Cheng, L.: Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In: Proceedings of the 7th ACM international conference on Computing frontiers. pp. 121–130 (2010)

Appendix A

Table 3 contains details of leaky IF conditions in each function of TinyGarble [86], EMP-toolkit [67], Obliv-C [96], and ABY [18].

Table 2: Type of the gates in the input layer of the AES and 256-bit MULT modules.

	AES		256-bit MULT	
	Percentage (%)	Count	Percentage (%)	Count
AND gates in input layer	75	96	50	256
XOR gates in input layer	25	32	50	256



(a) (b)

Fig. 6: SR of Goblin computed separately for AND and XOR input gates of 128-AES, 256-bit MULT, 128-bit Hamming, 128-bit SUM, and 288-bit SHA modules with (a) free-XOR and (b) half-gate optimization.

Appendix B

To investigate the effects of the gate types in the input layer on the SR, we counted the number of XOR and AND gates in the input layer of the AES and 256-bit MULT since the results for these two benchmark functions vary largely as shown in Figure 3. Table 2 contains the detail about the type of the gates in the AES and 256-bit MULT benchmark functions. Moreover, the category of AND gate contains AND/NAND, OR/NOR, ANDN, ORN, NANDN, and NORN gates, and the category of XOR gate includes NV, XOR, and XNOR gates as described in 4.3. It is observable that the AND gates are dominant in the AES input layer (75% input layer gates) while the portions of XOR and AND gates are equal in the input layer of 256-bit MULT. This can explain why the results for these two benchmark functions are different. In fact, it is because of the fact that it is more challenging to determine the inputs given to XOR gates. To further analyze the reason behind this, we have separately calculated the SR of Goblin against applied against AND and XOR gates. Figure 6 illustrates the results for launching Goblin against 128-AES, 256-bit MULT, 128-bit Hamming, 128-bit SUM, and 288-bit SHA modules, similar to Figure 3, where the results for AND and XOR gates are combined. As observable in Figure 6, Goblin's average SR when launching against AND gates are always close to 100% while its average SR has a range between 100% and 65% when launching against XOR gates for the benchmark functions. This is aligned with the results presented in

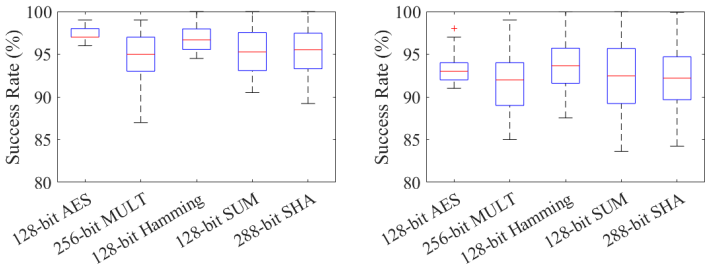


Fig. 7: SR of Goblin for 1000 randomly chosen inputs given to GC garbled by TinyGarble [87] when (a) only free-XOR or (b) half-gate optimization is enabled and JG is disabled.

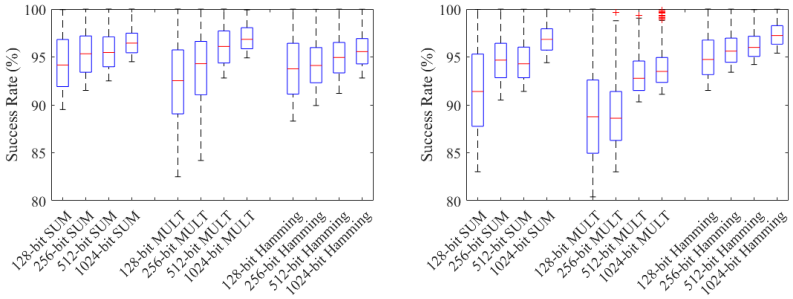


Fig. 8: SR of Goblin against MULT, SUM, and Hamming benchmark functions for a range of inputs garbled by TinyGarble [86] when (a) only free-XOR optimization, (b) half-gate protocol is enabled, and JG is disabled.

Figure 3. In that figure, the difference between the mean values of CPU cycles collected for inputs "0" and "1" is larger for AND gates in comparison to XOR gates.

Appendix C

To study the impact of an implementation in which not all timing side-channel vulnerability is not considered, we have launched Goblin against TinyGarble when the JG has been disabled. Figure 7 illustrates the results of Goblin against TinyGarble when JG is disabled. It is observable in Figure 7 that even without JG, Goblin can reveal the garbler's input with an average SR average of 95% or higher, slightly lower than the case when JG is enabled. To further investigate this, we launched Goblin against MULT, SUM, and Hamming benchmarks with input ranges between 128 and 1024 bits when JG was disabled. Figure 8 shows the results of launching Goblin against MULT, SUM, and Ham-

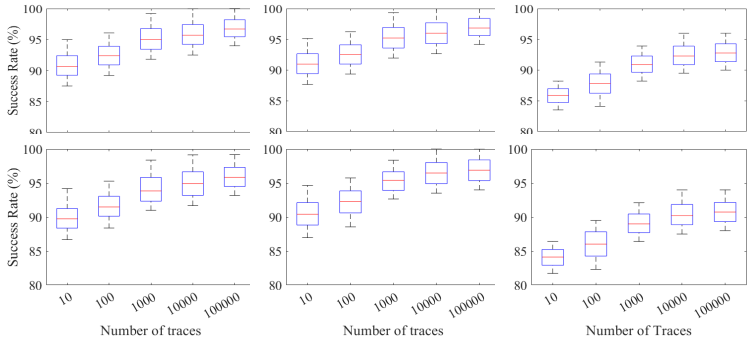


Fig. 9: SR of Goblin against 128-bit (a) SUM, (b) Hamming, and (c) MULT. CPU cycle traces captured from 10-100,000 randomly chosen inputs when JG is disabled. (Top: TinyGarble [86] with only free-XOR, Bottom: with half-gate optimization).

ming benchmark functions for a range of inputs garbled by TinyGarble when (a) only free-XOR optimization, (b) half-gate protocol is enabled, and JG is disabled. Same as results in Sec 5.2, one can observe a similar pattern of increasing SR of Goblin according to the increased size of benchmarks input. As another part of our investigations, we have launched Goblin against MULT, SUM, and Hamming modules without JG. Figure 9 illustrates SR of Goblin against 128-bit (a) SUM, (b) Hamming, and (b) MULT benchmarks for a range of CPU cycle traces captured from $10 - 100,000$ randomly chosen inputs when JG is disabled. These results prove that Goblin can reveal garbler information from an insecurely implemented framework even without the help of JG.

Appendix D

The JG, as in Algorithm 1, works as follows. The iteration's parameter n determines how many cell indexes in the array are summed and updates another array cell. This procedure repeats until it reaches the index of (Size-1). At this point, JG produces new random numbers and repeats the process indefinitely, resulting in cache disruption and potentially evicting critical data, like the global parameter R used for free-XOR [53]/Half-gates [97] optimizations.

Algorithm 1 Junk Generator pseudo code

Require: $Size = size\ of\ cache/64$
Ensure: $Junk \leftarrow Array[size]$ and $n \leftarrow 1$
  function JG(n)
    while User Interrupt **do**
      if $n == 1$ **then**
        $Seed \leftarrow t\_time$
        $Junk[0...\overline{3}] \leftarrow rand(Seed)$
        $n \leftarrow n + 1$ ▷ Initiate recursive algorithm.
        return $JG(2)$
      else if $n == (Size - 1)$ **then**
        return $JG(1)$
      else if $n \neq (Size - 1)$ and $n \neq 1$ **then**
        $i \leftarrow n$
        Loop over $i \leq (Size - n - 1)$:
          $Junk[i + n + 1] \leftarrow Junk[i] + Junk[n]$
        $n \leftarrow n + 1$
        return $JG(2)$
      end if
    end while
  end function

Table 3: A detailed report of leaky IF conditions (IF) of every function call in JustGarble [4], TinyGarble [86] with half-gate and free-XOR optimization, EMP-toolkit [67], Obliv-C [96], and ABY [18].

Framework	Function	IF	Framework	Function	IF
TinyGarble (half-gate) [86]	GarbledLowMem	0	JustGarble [42]	createNewWire	0
	GarbledGate	2		TRUNCATE	0
	ParseInitInputStr	0		TRUNC_COPY	0
	RemoveGarbledCircuit	0		getNextId	0
	HalfGarbleGateKnownValue	0		getFreshId	0
	NumOfNonXor	0		getNextWire	0
	HalfGarbleGate	2		createEmptyGarbledCircuit	0
	InvertSecretValue	0		removeGarbledCircuit	0
	XorSecret	0		startBuilding	0
	OutputBN2StrLowMem	0		finishBuilding	2
	RandomBlock	0		extractLabels	0
	Total	**4**		garbleCircuit	8
TinyGarble (free-XOR) [86]	GarbledLowMem	2		blockEqual	0
	GarbledGate	5		mapOutputs	0
	ParseInitInputStr	0		createInputLabels	0
	RemoveGarbledCircuit	0		randomBlock	0
	NumOfNonXor	0		xorBlocks	0
	XorSecret	0		findGatesWithMatchingInputs	1
	OutputBN2StrLowMem	0		**Total**	**11**
	RandomBlock	0	EMP-toolkit [67]	HalfGateGen	0
	Total	**7**		parse_party_and_port	0
Obliv-C [96]	yaoGenerateGate	3		NetIO	0
	yaoGenrRevealOblivBits	0		**Total**	**0**
	yaoGenrFeedOblivInputs	1	ABY [18]	YaoSharingInit	0
	yaoKeyNewPair	0		BooleanCircuit	0
	yaoSetBitAnd	0		init_aes_key	0
	yaoSetBitOr	0		ceil_divide	0
	yaoSetBitXor	0		clean_aes_key	0
	yaoFlipBit	0		EncryptWire	0
	yaoSetHashMask	0		EncryptWireGRR3	0
	yaoSetHalfMask	0		PrintKey	0
	yaoSetHalfMask2	0		PrintPerformanceStatistics	0
	yaoKeyDouble	0		XOR_DOUBLE_B	0
	Total	**4**		**Total**	**0**