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, (2) University of Florida

Abstract

With the advent of secure function evaluation, distrustful parties can jointly compute on their private inputs without disclosing anything besides the results. Yao's garbled circuit protocols have become an integral part of secure computation thanks to considerable efforts made to make it feasible, practical, and more efficient. These efforts have resulted in multiple optimizations on this primitive to enhance its performance by orders of magnitude over the last years. Such improvement targets have been defined to primarily reduce the cost of garbling in terms of computation and communication required for the creation, transfer, and evaluation of the garbled tables. The advancement in protocols has also led to the development of general-purpose compilers and tools made available to academia and industry. For decades, the security of protocols offered in those tools 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, however, side-channel analysis has gained momentum in connection with the real-world implementation of cryptographic primitives. 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 constructions, in particular, the optimized one to timing attacks, has, surprisingly, never been discussed in the literature. This paper introduces Goblin, the first timing attack against two commonly employed optimized garbled circuit constructions, namely free-XOR and half-gates. Goblin is a machine learning-assisted, non-profiling, single-trace timing attack, which successfully recovers the garbler's input during the computation. As the first step, Goblin targets the TinyGarble family and its core garbling tool, JustGarble. In this regard, Goblin hopefully paves the way for further research.

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 [89]. It has found several applications including secure multi-party computation [12, 28, 29, 56, 89], functional encryption [32, 33, 76], key-dependent message security [6, 7], homomorphic encryption [31, 72], and recently, quantum circuits [15]. 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. In addition to real-world applications being foreseen for GCs traditionally (e.g., credit evaluation function, background- and medical history checking, privacy-preserving database querying, etc. [53, 77]), nowadays GCs have found applications in privacy-preserving genome analysis [46], email spam filtering [40], image processing [17] and machine learning and statistical analysis [20, 30, 66, 70], just to name a few. To become practical and feasible solutions, GCs have undergone dramatic improvements and optimization to reduce the computation and communication costs (proportional to the size of the circuit). Thanks to the increase in hardware performance and the improvement in GC algorithms themselves, one of the main bottlenecks for these protocols has become the network bandwidth needed to transmit the garbled gates [11, 90].

To face these obstacles preventing further adoption of GCs in real-world systems, multiple optimization techniques have been developed, which aim to reduce communication and computation costs. Here we focus on two of the most acknowledged methods, namely free-XOR [53] and half-gates [90]. 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. 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. The security of open-source cryptographic libraries and implementations of protocols has just been evaluated in an extensive study [47], where the vulnerability of some of those libraries to *timing side-channel analysis* has been demonstrated. In this regard, more interesting and inspiring from the perspective of this work is the gap between academic research and cryptographic engineering when it comes to timing side-channel analysis.

Timing side-channel analysis. Irrespective of what cryptographic functions are embedded in programmable instruction set processors, such systems exhibit observable features and data-dependent behavior that can leak information about users' data/keys from the implementation. Side-channel attacks leverage this information through analyzing execution time [16, 24], power consumption [51], instruction or data cache behavior [1, 13, 73], branch predictor behavior [4], pipeline instruction and execution behavior as well as pipeline speculation behavior [86].

Timing side-channel analysis (SCA) has been launched to deduce information on the secret, e.g., keys, security tokens, and passwords [52, 62, 67, 85]. In general, timing side channels can be observed when the time taken to execute a piece of code depends on the secret variables. The temporal behavior of a code may depend on the control flow of a program, on its data flow, and on its contention over resources the program has to share with other running programs [21]; therefore, any of these depends on the secret, timing SCA can be launched. In this regard, two broad categories of timing side channels can be identified: instruction-related and cache-related cf. [88]. The former refers to the number or type of instructions executed along a path that can differ depending on the values of secret variables, leading to differences in the number of CPU cycles. 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 exhibited to leak adequate timing side-channel information to launch successful side-channel attacks on the cryptographic systems cf. [3, 13, 26, 34, 74]. As prime examples, the branch predictions [4] and memory accesses [1] are dependent on the inputs of the job/process, meaning that there is an input-dependent deviation in the execution time of the same job/process working on different inputs [64].

Privileged vs. unprivileged access for timing SCA. To access fine-grained side-channel observations, e.g., timing, both privileged and unprivileged adversaries have been considered in the literature; see, e.g., [18]. In the former case, one possible scenario constitutes privileged attackers in control of the operating system. In fact, such an attacker can be capable of launching more direct attacks than side-channel attacks [64]; however, platforms supporting shielded execution, e.g., Intel Software Guard eXtension (SGX), may not come under attack by even most privileged adversaries. Hence, SCA in the presence of privileged attacker, for instance, through intercepting the control flows, inferring page table and last-level cache, has become an important research direction [18, 25, 71]. Nevertheless, unprivileged attackers still attract attention as they can mount attacks in various settings, e.g., an unprivileged process in a native environment, an unprivileged process in a virtual machine, and a sandboxed process cf. [18, 36]. Examples of such attacks include [1, 2, 69, 75, 84, 91]. Moreover, `rdtsc` exemplifies the instructions used to obtain fine-grained timing information by providing unprivileged access to a model-specific register, which stores the current cycle count, commonly used for cache attacks on, e.g., Intel CPUs cf. [60].

Adversary models in the context of SFE. After reviewing how SCA, in particular, timing SCA, has been studied in other domains, we shift our focus to how this topic is relevant in the context of SFE. The security of GCs has been considered in two main paradigms, namely honest-but-curious and malicious adversary models. While the latter reflects the situation, where a party potentially cheats by corrupting the function to be jointly computed or, generally, adopting an arbitrary attack strategy, honest-but-curious parties follow the protocol honestly, although they may attempt to learn additional information from the execution. From this definition and what has been discussed about SCA, it is clear that an adversary capable of performing SCA can be classified as an honest-but-curious one. This has also been well-formulated in [10], 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. Nevertheless, it should be noted that these adversary models were developed to reflect the conventional setting assumed for SFE, namely, parties' symmetric computation power.

To encompass all aspects of real-world applications of SFE, the notion of server-aided or cloud-assisted SFE has been introduced. In this setting, 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. [14, 22, 23, 49]. In spite of all difficulties in deploying even *single*-server-aided protocols relying on Yao's GC, it has been demonstrated that it is indeed possible to construct practical ones [49]. 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. An intrinsic part of the proposed methodology is the application of optimization techniques, namely free-XOR. [49] has indeed well explained how any two-party SFE can be converted to a server-aided protocol if the server and the garbler are not simultaneously *malicious*. The proposed protocols have been proven secure under various circumstances, where parties and the server can be of different adversary types, including honest-but-curious server [50]. Nonetheless, as explained in Section 3.1, although the adversary model assumed in studies devoted to server-aided protocols might seem the most relevant to ours, any -even unprivileged- access to the CPU running the SFE protocol can be enjoyed by Golbin.

SCA against SFE constructions. Despite the achievements made to define the adversary models, prove the security, and construct numerous SFE schemes, considering SCA, there is a gap between what theoretical findings have suggested and what observations can be made by parties involved in executing an SFE 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 three main shortcomings as enumerated in [54]: (1) a secret, the global value is used to perform Free-XOR; (2) power consumption of the garbler's device can be linked to this secret value. Although multiple assumptions have been made to launch the attack (see Section 3), their attack has successfully disclosed the secret value by taking advantage of the leakage from the garbler's transmitted labels along with the leakage from non-linear gates. Now that the possibility of SCA against free-XOR-optimized SFE implementation has been indicated, the question is whether one can go even beyond that attack and mount timing attack and whether some of the assumptions made in [54] can be relaxed in that case.

Generally speaking, timing attacks feature outstanding properties that make them more interesting compared to other types of SCA, e.g., power and electromagnetic (EM) attack cf. [47]. First and foremost, timing attacks can be launched remotely, including cases of running code in parallel to the victim code without the need for local access to the target computer; hence, restricting physical access to the target machine cannot prevent timing attacks. Second, timing attacks can be carried out covertly. In light of this state of affairs and of the fact that timing attacks against SFE construction have never been discussed in the literature, this work attempts to answer the following question:

Is it possible to reveal parties' input by observing the timing information leaking when executing an SFE protocol?

More specifically, we answer this question positively for free-XOR- and half-gates-optimized constructions. The contribution of our work is as follows.

Contributions. We introduce *Goblin*, the first non-profiling, single-trace timing SCA that successfully targets the implementations of Free-XOR and half-gates garbling techniques. Notice that, to demonstrate the power of our attack, we compare it with the recent relevant attack presented in [54].

1. In contrast to [54], Goblin's effectiveness is not limited to circuits with a minimum number of input gates being garbled.

2. The attack in [54] has successfully extracted the global secret used in free-XOR optimization. Needless to say that even with the help of the disclosed secret, the garbler's input cannot be fully recovered. This is in contrast to our attack scenario, which focuses entirely on the recovery of the garbler's input. In the same vein, Goblin reveals not only the garbler's input fed into the non-linear gates (e.g., AND gates), but also linear ones, namely XOR ones.

3. 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.

2 Background

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 Oblivious Transfer (OT)

We consider 1-out-of-2 OT, which is a two-party protocol with the following definition. The sender P_1 posses 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.

2.2 Yao's Garbled Circuit (GC)

One of the most widely studied SFE approaches, designed to meet the needs of Boolean circuits, is garbling [55, 57]. One of the main building blocks of GC is the primitive associated with the cryptographic operation, often referred to as "encryption," namely hashing or symmetric key operations, e.g., pseudorandom functions (PRFs), dual-key ciphers, fixed-key block cipher. The protocol execution begins with garbling the circuit C, where the garbler (P_1) randomly chooses secrets w_i^0 and w_i^1, i.e., w_i^j is the garbled value of j 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., [44, 83] 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." This is illustrated in Figure 1(a), where the operand $E(\cdot)$ denotes the encryption operation. Since the table rows can reveal information about the internal wire values, they are permuted so that the recovery of the output labels does not result in uncovering the garbler input. 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. [83]. It is also possible to skip garbling the output wires of the circuit; therefore, two parties learn (only) the output of the circuit [53].

2.2.1 Optimizations of Yao's GC and Tools

Reducing the computation and communication costs of SFE protocols has been an objective of numerous studies. The legacy construction of GCs requires four garbled values per gate, corresponding to the combinations of values on the input wires. To reduce these cost on the evaluator and/or garbler side, various optimization methods have been introduced in the literature. For instance, the point-and-permute optimization, introduced by Beaver, Micali, and Rogaway [8] aims to do so on the evaluator side, where instead of trying all four ciphertexts, the evaluator can simply select the appropriate one based on the select bits of visible wire labels. For this, a random select bit is appended to each wire label; hence, the two labels on each wire have opposite select bits. This does not change the garbling cost, which is still four encryption per gate. To address this, garbled row-reduction has been introduced as an effective way to reduce the number of ciphertexts per gate [68]. The basic idea underlying the row-reduction technique is to set one of the four ciphertexts in each gate (e.g., the first one) to all-zeroes string, which is not needed to be sent; therefore, only three ciphertexts per gate need to be sent.

Among optimization techniques, *free-XOR* has attracted considerable attention since it reduces the cost on the garbler side effectively, namely by 25%.

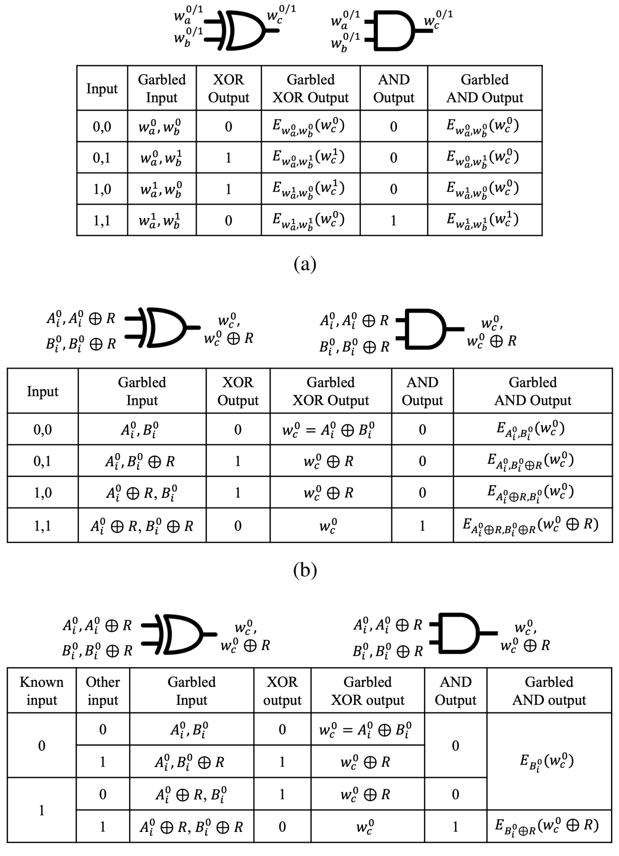
Free-XOR protocol. In the protocols reviewed above, XOR gates cost as much as other gates, e.g., AND or OR gates, with regard to the garbler's computation and communication costs. To reduce garbler's cost, the wire values are garbled as follows. For any gate G_i, $w_i^1 = w_i^0 \oplus R$ for some secret, global $R \in_R \{0,1\}^n$[1]. For the sake of simplicity, let $(A, A \oplus R)$ and $(B, B \oplus R)$ denote the wire labels. The evaluator possessing one of $(A, A \oplus R)$ receives one of $(B, B \oplus R)$ and by XORing them obtains one of $(C, C \oplus R)$, which is the correct result (see Figure 1(b)).

Half-gates protocol. This 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. The key idea behind half-gates optimization is to guarantee that the output wire of an AND gate is of the form $(C, C \oplus R)$, compatible with free-XOR optimization. Since Goblin is interested in recovering the garbler's input, we discuss how the half-gates in the input layer is generated on the garbler's side (for a more general case, we refer to [90]). For this purpose, the garbler enjoys the fact that one of the inputs is known to her. Therefore, when $a = 0$, she garbles a unary gate that always outputs false; otherwise, she garbles a unary identity gate. During the evaluation phase, the evaluator obtains C for both values of b if $a = 0$; otherwise, one of $(C, C \oplus R)$ is output. It is further possible to reduce the number of ciphertexts to be sent by applying the *row-reduction* technique to set the first of the two ciphertexts to all-zeroes.

GC tools: JustGarble and TinyGarble family. JustGarble [9], an open-source library licensed under GNU GPL v3 license, is an efficient circuit-garbling framework that has been widely used in many garble circuit (GC) and multi-party computation (MPC) approaches. JustGarble also offers multiple optimization techniques, as explained before. Another interesting aspect of JustGarble is the application of the AES-NI instruction set with encryption pipelining, which significantly reduces the cost of the AES computations. 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 [9]. JustGarble exploits the cryptographic permutations realizable by fixed-key AES acting like a public random permutation [9]. Although this might be a strong assumption cf. [37, 39], thanks to its efficiency and the theoretical foundation laid for JustGarble [9], it has been used in a wide variety of MPC and GC frameworks. However, JustGarble has not supported sequential circuits [82] and has not included communication or circuit generation [42]; therefore, it has not been considered a general-purpose framework.

Songhori et al. [81] proceeded JustGarble and proposed TinyGarble, a highly compressed and scalable sequential GC, which is a self-contained framework and can directly be used in MPC applications. In another approach, Mohassel et al. [65] proposed a fast and secure three-party computation for GC, which is an implementation based on the JustGarble framework. Groce et al. [35] also introduced the CompGC library, an efficient offline/online semi-honest two-party computation library based on the JustGarble implementation. We should add that we have chosen the TinyGarble family and their core garbling scheme, JustGarble, as a starting point

[1] The encryption function needed by the free-XOR protocol should have more specific properties. Since this is beyond the scope of this study and the vulnerability observed by us is not relevant to this, we refer to [19, 38] for more details.



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_i^0, B_i^0	0	$w_c^0 = A_i^0 \oplus B_i^0$	0	$E_{A_i^0,B_i^0}(w_c^0)$
0,1	$A_i^0, B_i^0 \oplus R$	1	$w_c^0 \oplus R$	0	$E_{A_i^0,B_i^0 \oplus R}(w_c^0)$
1,0	$A_i^0 \oplus R, B_i^0$	1	$w_c^0 \oplus R$	0	$E_{A_i^0 \oplus R,B_i^0}(w_c^0)$
1,1	$A_i^0 \oplus R, B_i^0 \oplus R$	0	w_c^0	1	$E_{A_i^0 \oplus R,B_i^0 \oplus R}(w_c^0 \oplus R)$

(b)

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

(c)

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

to point out how timing attacks could be applicable in the context of GCs. We also stress that the timing attack is not the result of how (in)correctly the fixed-key AES is used in GC schemes cf. [39], but the shortcomings caused by implementations of optimized garbling methods. Further investigation into other tools is left as future work.

2.3 *k*-means Algorithm

In general, clustering algorithms are mostly in the category of unsupervised machine learning. The main goal of clustering algorithms is to group samples of a set with some common features into subsets, i.e., *clusters*. The similarity of all members of each cluster is measured based on the pairwise distances of values (so-called features) [41]. With regard to the pairwise distances, clusters are then made around the mean vectors, which are called *centroids* [87].

k-means is a clustering algorithm that 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 [87]. The process of clustering data seems to be simple, but in fact, it is an NP-hard problem [87]. Hence, there exist heuristic algorithms that find a local optimum centroid over a series of finding iterations [87]. To be more specific, k-means finds partitions (clusters) $p = \{p_1, p_2, \cdots, p_k\}$ for the dataset $c = \{c_i\}_{i=1}^n$ that minimizes the total cluster variance [43]:

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

where μ_j is the mean of all examples assigned to j^{th} centroid. Here the squared Euclidean distance is one of the commonly applied distance measures applied to minimize the total cluster variance [80].

3 Attack Overview and Building Blocks

3.1 Adversary model

The attack model taken into consideration in this paper is the semi-honest model, where parties follow the protocol, but there is an attempt to learn information from the execution of the protocol. In doing so, even though the (single) server-aided SFE seems to be the most relevant adversary model from the SFE protocols' perspective, we consider the challenging setting where the server itself is honest-but-curious. Moreover, similar to models taken into account in [27,48,49], we assume that the parties and the server are independent, meaning that none of them collude. In practice, given the consequences in terms of losing the reputation and legal action, it is reasonable to assume that the server will not collude with the parties. 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 having access to the timing information can launch the attack.

The main requirement for Goblin to be launched successfully is the ability to acquire fine-grained timing information when the garbling process is run. The attacker's goal is to take advantage of this information to extract the garbler's input. Next we explain how this can be possible in real-world implementations.

3.1.1 Measuring Time on CPUs

According to Martin et al. [64], to measure the time without breaking the software, there are three main sources to take advantage of cf. [63]: (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 [75]. Here without loss of generality, we focus on how timing information can be retrieved using the first option, namely through using `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 [63]; 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 [64];

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) [63].

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, in contrast to 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 [78], then the collected traces can be aligned based on that timing information [61]. Therefore, without loss of generality, we assume that the attacker can align the timing measurements in order to mount the attack.

Resolution of timing measurements. The timestamps provided by `rdtsc` often has a resolution between 1 and 3 cycles on modern CPUs cf. [60]. 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 [45]. 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 5 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 although Goblin is a single-trace attack since multiple gates are being garbled one after another, the time difference can be directly driven from `rdtsc`. We should also add that our attack is an example of timing attacks, meaning that we believe other methods for acquiring the timing information can definitely be applied.

3.2 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 $\{c\}_i^n = 1$. Note that Goblin is a non-profiling attack; hence, in contrast 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.

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

To put this simply, SR indicates how many of the bits are disclosed correctly out of n bits in the garbler's input.

3.3 Goblin Flow

According to our adversary model, we assume that the adversary is neither the garbler nor the evaluator, and therefore, does not have any information about the circuit, size of the input, and input layer gates types. Hence, the Goblin flow contains three main steps:

1. Finding the size of the input and the number of gates in the input layer, then capturing the CPU cycles corresponding to each gate connected to input wires (i.e., gates in the input layer).

2. Pre-processing the acquired CPU cycle and making them ready for the clustering algorithm.

3. Running clustering algorithm over pre-processed CPU cycle to predict the Garbler secret, which is the Garbler inputs.

Here we provide an example of GC protocol flow as used in the TinyGarble family to provide insight into Goblin's flow.

3.3.1 Counting the Gates in the Input Layer

According to the protocol flow of TinyGarble/JustGarble, in the first step, the garbler's tokens for zero and one logical values are constructed. Then, the parser function 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 and locates the fan-in and fan-out of the input layer gates that are connected to the Garbler input based on g_init file information. After that, the `InputinitAloc` function is called, which allocates the tokens to the input layer gates fan-ins. We have observed that the number of `InputinitAloc` function calls equals two times the number of input layer gates. To have a systematic observation, we first applied the garbling process to a single XOR and a single AND gate (hereafter called the one-gate circuits). We have seen that the `InputinitAloc` function was called twice during the garbling process. Based on our observation, the parser first found two fan-ins from the one-gate circuit SCD file and used the number of fan-ins when it was calling the `InputinitAloc` function. To examine if this holds true for other circuits with different sizes, we implemented toy examples, for instance, a circuit with four gates, including an AND and an XOR gate in its input layer followed by an AND and an XOR gate in the next layer (hereafter called the four-gate circuit). Note that in all observations, similar to any GC, one of the fan-ins (input wires) is devoted to the garbler's inputs. We have seen similar behavior from the parser and the `InputinitAloc` function, where the parser finds four fan-ins and the `InputinitAloc` function was called four times. The reason behind this behavior is due to the GC protocol rules, which mandate all the gates in the circuit to have the exact number of two inputs and one output [10]. At this point, Goblin starts counting the number of `InputinitAloc` calls and calculating the number of input layer gates as half of the total number of `InputinitAloc` function calls. Then, the TinyGarble framework starts garbling the gates in the order provided in the SCD file, which mandates the framework to garble the gates one by one by calling the GarbleGate function, starting from the input layer gates, whose inputs are connected to the garbler and evaluator, before proceeding to the next layer gates. This fact allows Goblin to store the CPU cycle of each input layer gate garbling process by knowing the number of input gates based. The rests of the steps are not interesting for Goblin because they do not hold any information about the secret (Garbler input), and the above-mentioned information is adequate to launch the Goblin; therefore, from this point on, Goblin can continue the attack from an offline phase.

3.3.2 Pre-processing the Acquired CPU Cycles

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 4 for more information). The reason behind this significant difference is that in the free-XOR optimization, as its name implies, an XOR-type gate is garbled by simply 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 as well as cipher generation, which results in extra memory reads; hence, the accumulation of 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). 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). Afterward, it applies normalization to each subgroup 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. The normalization approach minimizes the difference between the garbling process CPU cycle requirements of XOR gates and all other gate types.

However, the first step is more complicated in a case, where the half-gates protocol is enabled. Specifically, according to our observation, not only garbling the XOR gates exhibits a significantly larger number of CPU cycles compared to other gate types, but also there is a dramatic difference in the number of CPU cycle 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. [90]. For AND gate, α values are set to 0, whereas for OR gate, they are set to 1. Therefore, it is not surprising that the CPU cycle traces collected when garbling OR/NOR gates compose a cluster different from the other gates. In the same vain, one can also observe that it takes more time for the garbler to generate the garbled OR/NOR gate when its input is zero, as opposed to AND/NAND gates, where for input one, it takes more time from the garbler to construct the gate. Moreover, 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 clustered data since the clustering algorithm puts both into one cluster, although they should be put into two different clusters due to their inputs. To counter this challenge, before applying normalization, Goblin applies the following additional data scaling technique 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 66th 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

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

where $\mathrm{Max}(\cdot)$ 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.

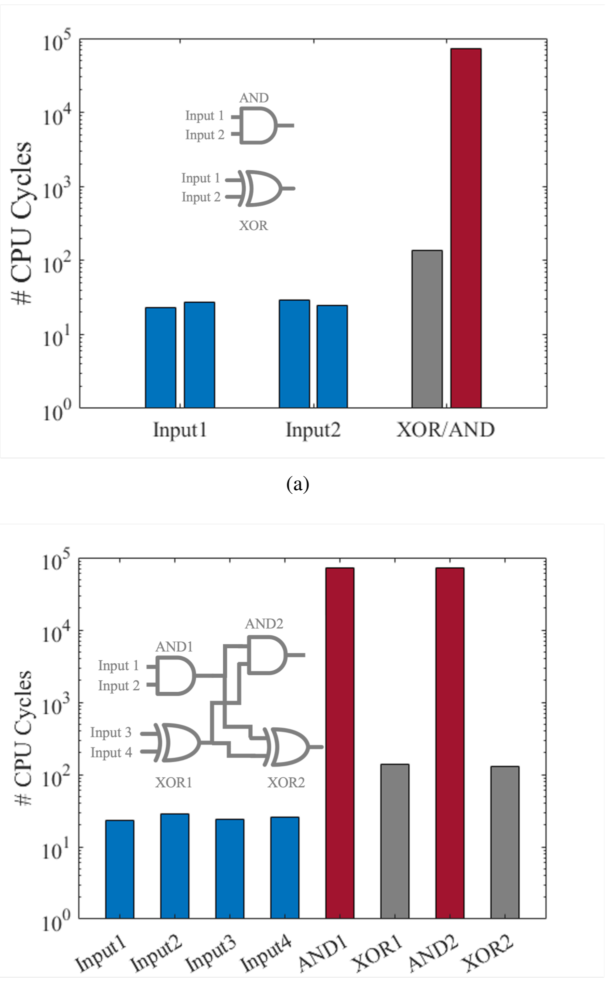
3.3.3 Recovering Garbler's Input

Up to this point, Goblin has obtained the pre-processed data and is ready to launch the clustering algorithm. As Goblin applies normalization to the CPU cycle data, the gate types' dominance in the centroids has vanished; therefore, the clustering algorithm of Goblin clusters CPU cycle 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 $\mathrm{Max}(\{c_i\}_{i=1}^{n}$ before normalization. When the clustering process is over, all members of the cluster that includes the maximum element are labeled as "1", meaning that the garbler input bit is "1". The other cluster then includes c_i's corresponding to garbler's input being equal to "0".

4 Experimental Results

As the first step, we have considered TinyGarble (see Section 2.2.1), publicly available via GitHub [81] To prepare our experimental setup, we have run the TinyGarble framework on two systems, one acting as the garbler and the other as the evaluator, connecting through local area network (LAN) cable. Each TinyGarble framework garbler and evaluator code ran on a system with Linux Ubuntu 20, 16 GB of memory, and an Intel Core i7-7700 CPU 3.60GHz computer processor unit (CPU). To employ the clustering method on the collected CPU cycle, we have used the k-means algorithm implemented in Matlab 2021. Moreover, the main code of the TinyGarble framework is based on Justgarble [9], and in the most updated version, TinyGarble is upgraded to be compatible with the half-gate [90] approach.

To capture the CPU clock cycles, we used `rdtsc` command to capture the clock cycles associated with the garbling of each gate, excluding the oblivious transfer (OT) module, initialization, scheduler, parsing, and fan-out assignment of the TinyGarble source code. Moreover, for the attack against free-XOR optimization, we have disabled half-gate and row-reduction optimizations in the source code and OT from the bash file. This helps us to analyze the susceptibility of these optimization techniques one by one. However, for the attack against half-gate, we have only disabled row-reduction and kept free-XOR optimization because it is at the heart of the half-gate approach.



(a)

(b)

Figure 2: CPU clock cycles captured for `InputinitAloc` (blue bars) and `GarbleGate` functions calls on (a) an XOR/AND gate, (b) the four-gate circuit and its corresponding CC. In both (a) and (b), Garbler's inputs (per gate, one of the fan-ins) are set to one. (Note: The y-axis has a logarithmic scale.)

4.1 Results for a single gate

The first question that we have attempted to answer is: if only one gate is garbled, is it possible to determine the garbler's input? For this purpose, we have captured the CCs from two circuits, including only one of the XOR or AND gates. Figure 2 illustrate the CPU clock cycles of the two steps of the TinyGarble framework garbling process. Note that, in Figure 2, a logarithm scale is used due to the significant gap between the number of clock cycles taken for `InputinitAloc` and `GarbleGate` when garbling the AND gate. It is observable in Figure 2.a that in the garbling process of both one-gate circuits (AND and XOR gates), the `InputinitAloc` function (the Blue bars) is called twice (see Section 3.3.1 for the discussion). This is a proof of concept that the number of the input layer gates is equal to half the number of the InputinitAloc function calls. To examine a more complex circuit, we have repeated the same experiment on a circuit with four gates (so-called four-gate circuit), one XOR and AND gate in the input layer, and one XOR and AND gate in the second layer.

As observable in Figure 2.b, the `InputinitAloc` (see the blue bars) is called four times, meaning there are two gates in the input layer of the circuit. Moreover, we have also observed that the TinyGarble framework garbles the input layer gates at the beginning of the garbling process before proceeding to the other layers.

Up until this point, our main focus of interest has been XOR and AND gates. In the next experiments, we have shifted that to the main other types of gates, namely, XOR, XNOR, AND, NAND, OR, and NOR. The question is: can the observation made about the XOR and AND gates be generalized to other gates as well? Moreover, by repeating the experiments 30 times (corresponding to a 95% confidence level), we have tried to understand how noisy the timing traces could be.

Figure 3 shows the CPU cycles for each type of gates. Note that, due to the free-XOR optimization, the number of CPU cycles of XNOR and XOR gates was drastically smaller than other gates due to their free garbling process [53]. Therefore, for the sake of readability, we have included results for AND, NAND, OR, and NOR separately in Figure 3.a, and those for XOR and XNOR in Figure 3.b. The reason behind the differences in the number of CPU cycles between the 30 experiments is other programs with root access to the CPU. These mandatory programs are run by the TinyGarble framework and introduce a small amount of noise when capturing the clock cycles. To the best of our knowledge, these root programs cannot be disabled nor be set to have less priority than the Linux Ubuntu task, which runs the TinyGarble framework code; hence, the CPU cycle noise is inevitable. Nevertheless, we have observed that this noise does not interfere with Goblin because this noise is far less than the difference in the number of CPU cycles of the gates for different inputs. We repeated the same experiments when the half-gate protocol was enabled and captured the CPU cycles of the gates mentioned above.

Figure 3.c and Figure 3.d illustrate the CPU cycles in this case. Similar to the free-XOR optimization case, we have depicted the results for XOR and XNOR in Figure 3.c, whereas ones for other gates are shown in Figure 3.d.

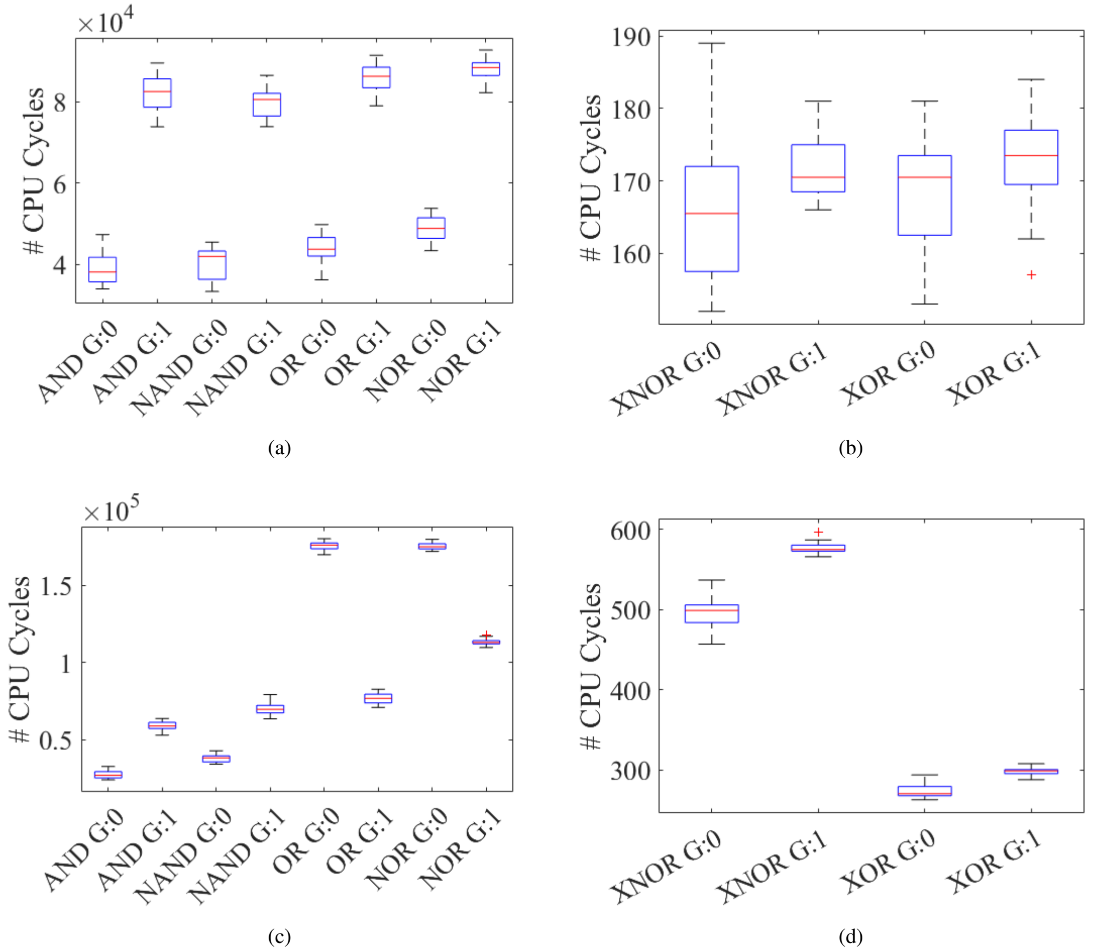
Summary. The main messages conveyed here are as follows. First and foremost, even in the presence of noise, it is possible to distinguish the garbler's inputs thanks to the considerable difference between the number of clock cycles taken in each case. Second, as explained in Section 3.3.2, it is definitely possible to categorize the gates into 2 and 3 types for the free-XOR and the half-gates optimizations, respectively. Last but not least, even if only one gate is garbled, if Goblin has a *reference*, it can distinguish between gates and the garbler's inputs. Here reference means that Goblin learns, for instance, how many cycles takes for garbler's input "0" or "1". This places absolutely no burden on Goblin, when more than one gate is garbled, which happens in a real-world scenario. In that case, just by comparing the clock cycles collected per gate, the difference becomes evident to Goblin, and it can determine the inputs. This is what has been studied through garbling benchmark functions in the next section.

4.2 Results for Benchmark Functions

On order to evaluate the efficacy of Goblin, we have targeted the most commonly-used benchmark functions, also available in the TinyGarble toolbox, including AES-128, SHA3, 256-bit Multiplier, 128-bit Summation, and 128-bit Hamming. 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 are impractical task to do 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 and launched Goblin against this set. In the setting of the k-means algorithm, 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 4.a and Figure 4.b show the SR when only free-XOR 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 4.a that the attack achieved a better SR when launched against the AES benchmark compared to 256-bit Multiplier. The reason is three-fold. First, only 1000 inputs are tested; therefore, the results might vary. Second, although the size of the inputs is almost the same (in the range 128-288), 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, Goblin can observe a few examples. Notice that Goblin is a non-profiling, single traces 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 4.3.

Figure 4.a and Figure 4.b show the SR of Goblin launched against benchmark functions when the half-gate protocol was enabled. Compared to Figure 4.a, Figure 4.b 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 observation, 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.



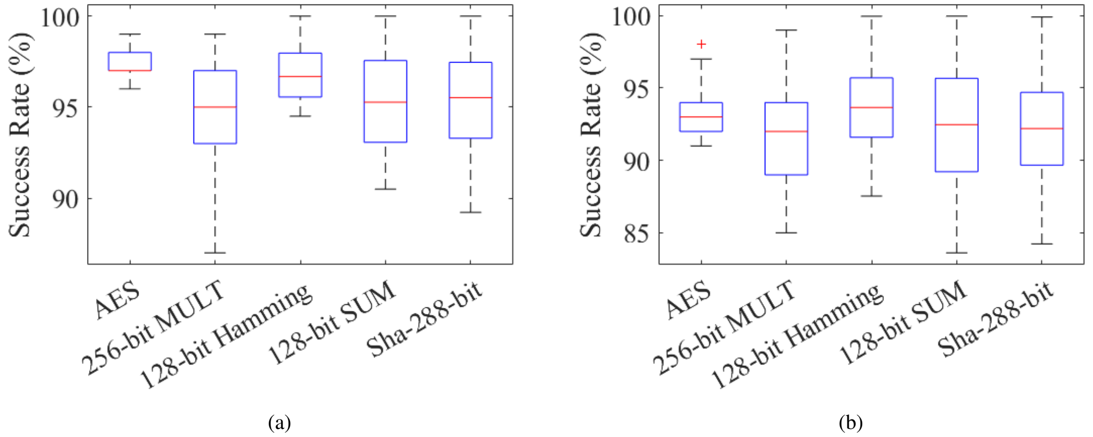
(a)

(b)

(c)

(d)

Figure 3: The number of clock cycles for (a) AND, NAND, OR, and NOR (b) XOR and XNOR gates when only free-XOR optimization is enabled (c) AND, NAND, OR, and NOR (d) XOR and XNOR gates when the half-gate protocol is enabled.



(a)

(b)

Figure 4: SR of Goblin against benchmark functions for 1000 randomly chosen inputs when (a) only free-XOR optimization (b) half-gate protocol are enabled.

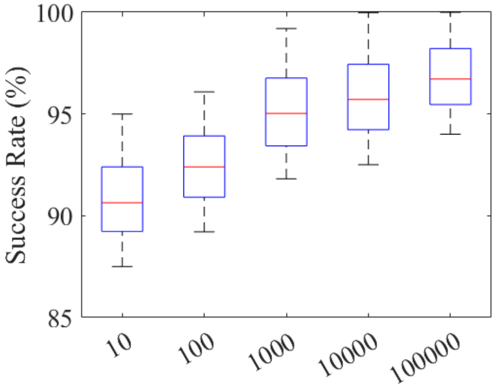


Figure 5: SR of Goblin against 128-bit SUM for a range of CPU cycle traces captured from 10, 100, 1000, 10000, and 100000 randomly chosen inputs when only free-XOR optimization is enabled.

Impact of the number of traces. As mentioned earlier, to evaluate the effectiveness of our attack, we selected 1000 random inputs since capturing CPU cycles for all inputs is impractical and infeasible. This can directly impact the variance in our results. To investigate this, we have applied Goblin on a range of CPU cycle traces captured from 10, 100, 1000, 10,000, and 100,000 random inputs of a 128-bit SUM module, i.e., the one demonstrating a fairly high variance (see, Figure 4). Figure 5 illustrates the SR of Goblin when being launched against a range of CPU cycle traces. As observable in Figure 5, 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%. 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 just 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.

4.3 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 6 illustrates the results for launching Goblin against these benchmark functions where Figure 6.a containes the results for free-XOR optimization, whereas Figure 6.b is for half-gate one. As shown in Figure 6.a, not only does increasing the input size decrease the SR but also increases the minimum and the median of Goblin SR against these benchmark functions. This SR increment is due to the fact that the k-means has a broader range of data to cluster, which means it has more examples to compare with one another.

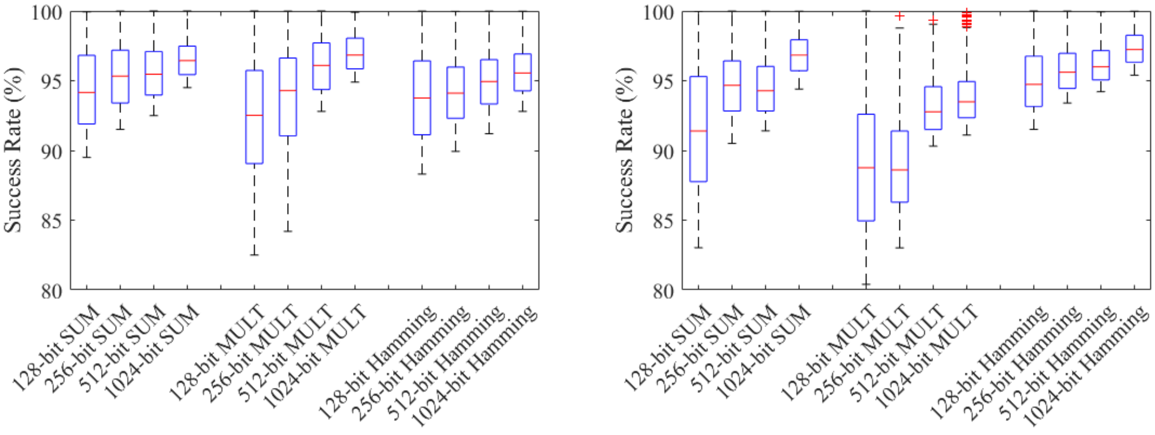
5 Discussion

After introducing Goblin and its effectiveness through experimentation, a natural question to ask would be whether countermeasures could be devised to stop Goblin. Here we discuss different aspects of the attack that can be considered in this regard.

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. [64], meaning that any measurement through `rdtsc` is accurate with regard to the previous call/measurement. 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. [64]. 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 before in Section 3.1.1, `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).

Restricting access. As it has been nicely put forward in [58], unprivileged usage of the high-resolution timers can be prevented by setting control registers, e.g., `CR4.TSD` bit in AMD [5] (volume 2, Section 3.2.5). This results in `rdtsc`, `rdtscp`, and `rdpru` being unavailable to the attacker. Although this might be tempting, such a restriction can have a negative impact on unprivileged applications depending on `rdtsc`, e.g., adb, cargo, and Docker [58]. Moreover, in doing so, not all timing primitives can be disabled. It is possible for the attacker to come up with a *counting thread* that constantly increments a global variable that serves as a timestamp without relying on platform specifics [59, 60, 79]. It is further shown that such a counting thread can have even a higher resolution than the `rdtsc` instruction on Intel CPUs [79].



(a)

(b)

Figure 6: SR of Goblin against MULT, SUM, and Hamming benchmark functions for a range of inputs when (a) only free-XOR optimization (b) half-gate protocol is enabled for 1000 randomly chosen inputs.

6 Conclusion

Several applications, including multi-party computation, are taking secure function evaluation (SFE) into account thanks to the efficient implementations of GC presented by Yao. To achieve this efficiency, many optimizations, such as free-XOR, row-reduction, half-gate, etc., have been presented to reduce the cost of garbling progress. However, it has been recently shown that these optimizations are vulnerable to side-channel attacks in a way that the global secret used in the free-XOR setting can be revealed by launching power side-channel attacks. Goblin, on the other hand, has demonstrated that the frameworks which are using free-XOR and half-gate are vulnerable to timing-side channel attacks. For this, Goblin can be launched by collecting the CPU cycles of the garbling process by reading the time stamp counter, i.e., calling `rdtsc`. In this regard, Goblin can be run in parallel to the garbling framework without requiring any privileged access. Moreover, Goblin can extract Garbler's input directly from each garbling process (clock cycles per gate) without any prior knowledge about the circuit being garbled; therefore, Goblin can be considered a non-profiling attack that can be launched against one timing trace. Utilizing clustering algorithm k-means, Goblin has achieved up to 98% success rate when launching against the most commonly used benchmarks such as Sum, MULT, Hamming, AES, and SHA to extract the Garbler's input directly from the garbling process. Goblin has also been proven to be a scalable attack against more extensive circuits.

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] Aciiçmez, O. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture* (2007), pp. 11–18.

[2] Aciiçmez, O., Brumley, B. B., and Grabher, P. New results on instruction cache attacks. In *International workshop on cryptographic hardware and embedded systems* (2010), Springer, pp. 110–124.

[3] Aciiçmez, O., and Koç, Ç. K. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security* (2006), Springer, pp. 112–121.

[4] Aciiçmez, O., Koç, Ç. K., and Seifert, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), pp. 312–320.

[5] Advanced Micro Devices Inc. Amd64 architecture programmer's manual. [Online]https://www.amd.com/system/files/TechDocs/24593.pdf [Accessed: Dec.19, 2022], 2017.

[6] Applebaum, B. Key-dependent message security: Generic amplification and completeness. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2011), Springer, pp. 527–546.

[7] Barak, B., Haitner, I., Hofheinz, D., and Ishai, Y. Bounded key-dependent message security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2010), Springer, pp. 423–444.

[8] Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), pp. 503–513.

[9] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symp. on Security and Privacy* (2013), IEEE, pp. 478–492.

[10] Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. In *Proc. of the 2012 ACM Conf. on Computer and Comm. security* (2012), pp. 784–796.

[11] Ben-Efraim, A., Lindell, Y., and Omri, E. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 578–590.

[12] Benhamouda, F., and 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* (2018), Springer, pp. 500–532.

[13] Bernstein, D. J. Cache-timing attacks on aes.

[14] 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* (2009), Springer, pp. 325–343.

[15] Brakerski, Z., and Yuen, H. Quantum garbled circuits. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing* (2022), pp. 804–817.

[16] Brumley, D., and Boneh, D. Remote timing attacks are practical. *Computer Networks 48*, 5 (2005), 701–716.

[17] Chen, D., Chen, W., Chen, J., Zheng, P., and 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)* (2018), IEEE, pp. 1–6.

[18] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), pp. 7–18.

[19] Choi, S. G., Katz, J., Kumaresan, R., and Zhou, H.-S. On the security of the "free-xor" technique. In *Theory of Cryptography Conference* (2012), Springer, pp. 39–53.

[20] Cock, M. d., Dowsley, R., Nascimento, A. C., and 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* (2015), pp. 3–14.

[21] Coppens, B., Verbauwhede, I., De Bosschere, K., and De Sutter, B. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 45–60.

[22] Damgård, I., and Ishai, Y. Constant-round multiparty computation using a black-box pseudorandom generator. In *Annual International Cryptology Conference* (2005), Springer, pp. 378–394.

[23] Damgård, I., Ishai, Y., Krøigaard, M., Nielsen, J. B., and Smith, A. Scalable multiparty computation with nearly optimal work and resilience. In *Annual International Cryptology Conference* (2008), Springer, pp. 241–261.

[24] Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., and Willems, J.-L. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Advanced Applications* (1998), Springer, pp. 167–182.

[25] Dong, X., Shen, Z., Criswell, J., Cox, A. L., and Dwarkadas, S. Shielding software from privileged {Side-Channel} attacks. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 1441–1458.

[26] Easdon, C., Schwarz, M., Schwarzl, M., and Gruss, D. Rapid prototyping for microarchitectural attacks. In *USENIX Security Symposium* (2022).

[27] Feige, U., Killian, J., and Naor, M. A minimal model for secure computation. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing* (1994), pp. 554–563.

[28] Garg, S., and Srinivasan, A. Garbled protocols and two-round mpc from bilinear maps. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)* (2017), IEEE, pp. 588–599.

[29] Garg, S., and Srinivasan, A. Two-round multiparty secure computation from minimal assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2018), Springer, pp. 468–499.

[30] Gascón, A., Schoppmann, P., Balle, B., Raykova, M., Doerner, J., Zahur, S., and Evans, D. Privacy-preserving distributed linear regression on high-dimensional data. *Proc. Priv. Enhancing Technol. 2017*, 4 (2017), 345–364.

[31] Gentry, C., Halevi, S., and Vaikuntanathan, V. i-hop homomorphic encryption and rerandomizable yao circuits. In *Annual Cryptology Conference* (2010), Springer, pp. 155–172.

[32] Goldwasser, S., Kalai, Y., Popa, R. A., Vaikuntanathan, V., and Zeldovich, N. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing* (2013), pp. 555–564.

[33] Gorbunov, S., Vaikuntanathan, V., and Wee, H. Functional encryption with bounded collusions via multi-party computation. In *Annual Cryptology Conference* (2012), Springer, pp. 162–179.

[34] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 955–972.

[35] Groce, A., Ledger, A., Malozemoff, A. J., and Yerukhimovich, A. Compgc: Efficient offline/online semi-honest two-party computation. *Cryptology ePrint Archive* (2016).

[36] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), pp. 368–379.

[37] Gueron, S., Lindell, Y., Nof, A., and Pinkas, B. Fast garbling of circuits under standard assumptions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 567–578.

[38] Guo, C., Katz, J., Wang, X., Weng, C., and Yu, Y. Better concrete security for half-gates garbling (in the multi-instance setting). In *Annual International Cryptology Conference* (2020), Springer, pp. 793–822.

[39] Guo, C., Katz, J., Wang, X., and Yu, Y. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 825–841.

[40] Gupta, T., Fingler, H., Alvisi, L., and 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* (2017), pp. 169–182.

[41] Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. *The elements of statistical learning: data mining, inference, and prediction*, vol. 2. Springer, 2009.

[42] Hastings, M., Hemenway, B., Noble, D., and Zdancewic, S. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP)* (2019), IEEE, pp. 1220–1237.

[43] Hettwer, B., Gehrer, S., and Güneysu, T. Applications of machine learning techniques in side-channel attacks: a survey. *J. of Cryptographic Engineering 10*, 2 (2020), 135–162.

[44] Hussain, S., Li, B., Koushanfar, F., and Cammarota, R. Tinygarble2: Smart, efficient, and scalable yao's garble circuit. In *Proc. of the 2020 WKSP on Privacy-Preserving Machine Learning in Practice* (2020), pp. 65–67.

[45] Intel Corporation. Intel® Core™ i7 Processors. [Online]https://www.intel.com/content/www/us/en/products/details/processors/core/i7.html [Accessed: Dec.19, 2022], 2017.

[46] Jagadeesh, K. A., Wu, D. J., Birgmeier, J. A., Boneh, D., and Bejerano, G. Deriving genomic diagnoses without revealing patient genomes. *Science 357*, 6352 (2017), 692–695.

[47] Jancar, J., Fourné, M., Braga, D. D. A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.-A., and 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)* (2022), IEEE, pp. 632–649.

[48] Kamara, S., Mohassel, P., and Raykova, M. Outsourcing multi-party computation. *Cryptology ePrint Archive* (2011).

[49] Kamara, S., Mohassel, P., and Riva, B. Salus: a system for server-aided secure function evaluation. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 797–808.

[50] KAMARA, S., MOHASSEL, P., AND RIVA, B. Salus: A system for server-aided secure function evaluation. *Cryptology ePrint Archive* (2012).

[51] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Annual international cryptology conference* (1999), Springer, pp. 388–397.

[52] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[53] KOLESNIKOV, V., AND SCHNEIDER, T. Improved garbled circuit: Free xor gates and applications. In *Intrl. Colloquium on Automata, Languages, and Programming* (2008), Springer, pp. 486–498.

[54] LEVI, I., AND HAZAY, C. Garbled-circuits from an sca perspective: Free xor can be quite expensive... *Cryptology ePrint Archive* (2022).

[55] LINDELL, Y., AND 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).

[56] LINDELL, Y., AND 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* (2007), Springer, pp. 52–78.

[57] LINDELL, Y., AND PINKAS, B. A proof of security of yao's protocol for two-party computation. *J. of cryptology 22*, 2 (2009), 161–188.

[58] LIPP, M., GRUSS, D., AND SCHWARZ, M. Amd prefetch attacks through power and time. In *USENIX Security Symposium* (2022).

[59] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 549–564.

[60] LIPP, M., HADŽIĆ, V., SCHWARZ, M., PERAIS, A., MAURICE, C., AND 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* (2020), pp. 813–825.

[61] LIPP, M., KOGLER, A., OSWALD, D., SCHWARZ, M., EASDON, C., CANELLA, C., AND GRUSS, D. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 355–371.

[62] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND 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)* (2016), IEEE, pp. 406–418.

[63] LYU, Y., AND MISHRA, P. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security 2*, 1 (2018), 33–50.

[64] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)* (2012), IEEE, pp. 118–129.

[65] MOHASSEL, P., ROSULEK, M., AND 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* (2015), pp. 591–602.

[66] MOHASSEL, P., AND ZHANG, Y. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)* (2017), IEEE, pp. 19–38.

[67] MOWERY, K., KEELVEEDHI, S., AND SHACHAM, H. Are aes x86 cache timing attacks still feasible? In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop* (2012), pp. 19–24.

[68] NAOR, M., PINKAS, B., AND SUMNER, R. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce* (1999), pp. 129–139.

[69] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on aes. In *International Workshop on Selected Areas in Cryptography* (2006), Springer, pp. 147–162.

[70] NIKOLAENKO, V., WEINSBERG, U., IOANNIDIS, S., JOYE, M., BONEH, D., AND TAFT, N. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE symposium on security and privacy* (2013), IEEE, pp. 334–348.

[71] OLEKSENKO, O., TRACH, B., KRAHN, R., SILBERSTEIN, M., AND FETZER, C. Varys: Protecting {SGX} enclaves from practical {Side-Channel} attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 227–240.

[72] OSTROVSKY, R., PASKIN-CHERNIAVSKY, A., AND PASKIN-CHERNIAVSKY, B. Maliciously circuit-private fhe. In *Annual Cryptology Conference* (2014), Springer, pp. 536–553.

[73] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference* (2006), Springer, pp. 1–20.

[74] Page, D. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive* (2002).

[75] Percival, C. Cache missing for fun and profit, 2005.

[76] Sahai, A., and Seyalioglu, H. Worry-free encryption: functional encryption with public keys. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), pp. 463–472.

[77] Schneider, T. Practical secure function evaluation. In *Informatiktage* (2008), pp. 37–40.

[78] Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., and 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* (2018), pp. 587–600.

[79] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer, pp. 3–24.

[80] Sherali, H. D., and Tuncbilek, C. H. A squared-euclidean distance location-allocation problem. *Naval Research Logistics (NRL) 39*, 4 (1992), 447–469.

[81] Songhori, E., Siam, u, H., and Riazi, S. Tinygarble framework. [Online]https://github.com/esonghori/TinyGarble [Accessed Dec.20, 2022], 2019.

[82] Songhori, E. M., Hussain, S. U., Sadeghi, A.-R., and Koushanfar, F. Compacting privacy-preserving k-nearest neighbor search using logic synthesis. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), IEEE, pp. 1–6.

[83] Songhori, E. M., Hussain, S. U., Sadeghi, A.-R., Schneider, T., and Koushanfar, F. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symp. on Security and Privacy* (2015), IEEE, pp. 411–428.

[84] Tromer, E., Osvik, D. A., and Shamir, A. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[85] Vattikonda, B. C., Das, S., and Shacham, H. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), pp. 41–46.

[86] Wang, Z., and Lee, R. B. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (2006), IEEE, pp. 473–482.

[87] Whitnall, C., and Oswald, E. Robust profiling for dpa-style attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2015), Springer, pp. 3–21.

[88] Wu, M., Guo, S., Schaumont, P., and Wang, C. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), pp. 15–26.

[89] Yao, A. C.-C. How to generate and exchange secrets. In *27th Annual Symp. on Foundations of Computer Science (sfcs 1986)* (1986), IEEE, pp. 162–167.

[90] Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), Springer, pp. 220–250.

[91] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 990–1003.