IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 2, 2021, accepted July 29, 2021, date of publication July 30, 2021, date of current version August 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3101638

Runtime Randomized Relocation of Crypto Libraries for Mitigating Cache Attacks

YOUNGJOO SHIN[1] AND JOOBEOM YUN[2]
[1]School of Cybersecurity, Korea University, Seoul 02841, Republic of Korea
[2]Department of Computer and Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul 05006, Republic of Korea

Corresponding author: Joobeom Yun (jbyun@sejong.ac.kr)

This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT) (No. 2020R1F1A1065539). This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation), and was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2021-2018-0-01423) supervised by the IITP.

ABSTRACT Crypto libraries such as OpenSSL and Libgcrypt are essential building blocks for implementing secure cloud services. Unfortunately, these libraries are subject to cache side-channel attacks, which are more devastating in cloud environments where inevitable cache contention among different tenants occurs. Previous approaches for mitigating cache side-channel attacks have limitations in terms of the deployability and security; these hinder utilization in cloud services. In this paper, we propose an R2-relocator, a novel library protection technique based on moving target defence. When injected into a running process, the R2-relocator performs randomized relocation of the library during runtime. By doing this, it transforms a vulnerable crypto library into one that randomly changes its memory (cache) location, thereby preventing the delivery of cache side-channel attacks against the library. The proposed technique achieves robust protection against cache side-channel attacks for all crypto libraries, even those containing unpatched critical vulnerabilities, without the need for reconfiguration of the library. Extensive evaluations of security, performance, and deployability of the R2-relocator demonstrate its effectiveness for secure cloud services.

INDEX TERMS Cache side-channel attack, crypto library, moving target defence, attack mitigation, secure cloud computing.

I. INTRODUCTION

Cloud computing brings financial benefits to customers but at the cost of security, as data are outsourced from their on-premise servers to the (untrusted) cloud [1]. Hence, most cloud customers rely on cryptographic algorithms to protect their outsourced data. Owing to the sophisticated structure of cryptographic algorithms, developers commonly use well-crafted crypto libraries such as OpenSSL [2] and Libgcrypt [3] for cloud services, instead of implementing the algorithm by themselves.

However, it is well-known that security flaws that can result in a *cache side-channel attack (CSA)* have been discovered in these libraries [4]–[7]. CSAs are a significant security threat, especially in cloud computing environments where multiple tenants are co-located on a server that shares its computing resources, such as a CPU cache [6], [7].

There have been notable efforts in previous works to mitigate CSAs against crypto libraries. Most mitigation strategies include identifying and fixing security bugs in libraries using the heuristic [8]–[10] or the automatic [4] approach. However, these strategies have limitations that hinder their adoption and deployment in cloud services. One limitation is that these mitigations require reconfiguration of the library, *i.e.,* modification and recompilation of the library source code. After reconfiguration, the newly patched library should be redeployed in the cloud services, which usually have a large number of hosts that need to be patched. Moreover, such a *patching-by-reconfiguration* approach hinders instant responses when new critical security flaws are found in the library. Another limitation is that pinpoint bug-fixing of this approach does not eliminate the possibility of security flaws in the patched and reconfigured library; even a single hidden

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Aljawarneh.

or undiscovered flaw is still highly likely to result in devastating attacks.

In this paper, we propose R2-relocator,[1] a robust and easily deployable method for protecting crypto libraries from CSAs. Instead of adopting the previous approach of patching-by-reconfiguration, our method utilizes moving target defense (MTD), a novel paradigm for defending a system from various security threats [11], [12]. That is, R2-relocator is injected into a running process of a target application that has a vulnerable crypto library and operates with the library inside the process. Specifically, it transforms an executable binary of the crypto library into one that randomly moves its location in the memory. Therefore, the corresponding cache location (*i.e.,* a cache set) is moved as well during runtime.

As the cache location of the library is frequently changed over time in a randomized manner by the R2-relocator, attackers are prevented from identifying a vulnerable target cache set, which is a prerequisite for a successful attack. Therefore, robust protection against CSAs can be achieved regardless of the presence of any CSA-susceptible flaws (including any yet undiscovered ones) in the library. In addition, the R2-relocator can directly be applied to an executable binary. Thus, it does not require modification of the library source code, thereby enabling easy and instant deployment to cloud services. Furthermore, R2-relocator is widely available to various platforms, as it is implemented in software only. This is more beneficial compared to hardware-based CSA countermeasures that leverage hardware features such as transactional memory [13], which limits its applicability.

We describe how the R2-relocator transforms a normal executable binary of a library for the intended purpose. We also present the implementations of this transformation in detail. Our evaluation reveals that the R2-relocator provides crypto libraries with robust protection from CSAs, while introducing negligible performance overhead.

The remainder of this paper is organized as follows. In Section II, we present previous works related to the proposed method. In Section III, we provide background knowledge about CSAs. In Section IV, we explain threat models and the design goals of the proposed technique. In Section V and VI, we describe R2-relocator in detail and present our evaluation results, respectively. Finally, in Section VII, we conclude this paper by summarizing our main findings.

II. RELATED WORK

In this section, we present some related works on CSAs of crypto libraries and mitigation strategies.

A. CACHE SIDE-CHANNEL ATTACKS ON CRYPTO LIBRARIES

Cache contention among co-located processes or virtual machines (VMs) allows CSAs, in which an adversary can infer sensitive information such as cryptographic keys from the victim's applications [14]. CSAs have been demonstrated to be possible on a wide range of victim applications in various environments. In most cases, attacks occur against cryptographic implementations, in particular, crypto libraries such as OpenSSL [2], Libgcrypt [3], and wolfSSL [15]. Vulnerable cryptographic algorithms in these libraries include public key algorithms such as RSA [6], [7], [16], ECDSA [17], ECDH [4], [9], and ElGamal [18] and symmetric key algorithms such as AES [19]–[22].

In this paper, we mainly focus on mitigating CSAs in crypto libraries, although CSAs are not confined to the implementation of cryptographic algorithms. Indeed, several works have demonstrated the existence of CSAs in a variety of (non-cryptographic) applications, such as extracting private information from I/O devices [23], [24] and inferring browsing history from web browsers [25] and firewall policies [26], [27]. We believe that the mitigating method proposed in this paper is applicable not only to crypto applications but also to non-cryptographic applications that have CSA-susceptible security flaws in their libraries.

B. MITIGATION APPROACHES

In the literature, several mitigation approaches have been proposed to protect vulnerable crypto libraries from possible CSAs. Pereida García *et al.* [10] discovered that certain versions of the OpenSSL library are vulnerable to CSAs in the implementation of a DSA algorithm. As a countermeasure, they proposed a security patch that addressed the vulnerability of the library to ensure constant-time execution, regardless of the inputs of the algorithm. Later, they also identified another side-channel vulnerability in the same library but in a different cryptographic algorithm (*i.e.,* ECDSA) [8]. Their result revealed the difficulty in mitigating CSAs using a pinpoint bug patching strategy. Concurrently, Genkin *et al.* [9] discovered a security flaw that allows CSAs in an ECDH implementation of the Libgcrypt library. The proposed mitigating solution involved fixing the flaw in the software, similar to the solution proposed in [8], [10].

The aforementioned works used a heuristic approach to identify the vulnerabilities in libraries. In other words, the flaws were discovered by manually inspecting the source code of the library. Shin *et al.* [4] proposed an automatic method that enables the identification of all vulnerable locations in crypto libraries. Despite the automatic bug identification, it is still necessary to patch the library manually, similar to the previous methods.

The mitigation strategies presented in the literature have several restrictions that limit their deployment and utilization in cloud services. First, these mitigation methods usually require reconfiguration of the vulnerable crypto library; in other words, modification and recompilation of the library source code are necessary. The newly patched (reconfigured) version of the library must then be deployed again to a large number of vulnerable hosts in the cloud; a significant amount of time may be required to complete the

[1] **R2-relocator** is an abbreviation for "runtime and randomized relocator."

deployment. For instance, it takes 2 - 15 hours to deploy security patches to guest VMs in the Azure cloud via the automatic update mechanism [28]. This approach is not useful for enabling instant security responses to newly discovered critical CSA-susceptible vulnerabilities. Moreover, such a pinpoint bug-fixing approach for a vulnerable library cannot eliminate the possibility of security flaws in the patched library. Even a single undiscovered flaw may undermine the security patch applied to the library.

Brumley and Tuveri [29] proposed a countermeasure to CSAs that randomly aligns dynamically allocated memory in a crypto library. Although it seems similar to our method proposed in this paper, their approach has some limitations. First, their method provides one-time randomization; once randomized, the aligned memory remains unchanged during runtime. Hence, an attacker that has successfully identified a target cache set can easily extract secret data from the victim without further intervention. Unlike their method, we attempt to achieve runtime randomized relocation so as to prevent from extracting secret data. Besides, their method does not address the randomization of the library code, which is the main goal that we attempt to achieve in this paper.

III. CACHE SIDE-CHANNEL ATTACKS

Virtualization techniques that are used in cloud computing vary in-depth, from container-based (usually used for PaaS) to hypervisor-based (used for IaaS). However, all the techniques provide a weak level of isolation to the hardware resources, specifically the cache memory. This leads to cache contention between VMs, which is the source of a side-channel through which secret information can be leaked. Many different CSAs have been proposed in the literature. These attacks can be classified into two categories according to the attacker's ability to share the memory pages with a victim.

A. SHARING-DEPENDENT CACHE ATTACK

Flush+Reload is a CSA that exploits a shared memory [6], [17], [20], [24]–[27], [30]–[32]. It exploits memory deduplication, which is a cross-VM page sharing technique that is implemented in hypervisors such as VMware ESXi and KVM. An attacker VM begins the attack by flushing the target memory address (shared with the victim VM) from the cache using `clflush` instruction. Then, the attacker waits for the victim to complete the execution of security operations, during which the target address may be accessed based on a secret value. Finally, the attacker measures the access time while reloading the address. Low latency indicates a cache hit, which implies that the victim accessed the address, whereas high latency implies that the address was not accessed by the victim. By observing the victim's memory access pattern, the attacker can completely recover the secret value.

Flush+Reload attack has several variants, including *Flush+Flush* [33], *Evict+Reload* [24], and *Invalidate+Transfer* [34]. These techniques typically share the precondition that memory deduplication is necessary for a successful attack. Hence, they are easily mitigated by turning off the memory deduplication in the hypervisor.

B. SHARING-INDEPENDENT CACHE ATTACK

Prime+Probe [7], [16], [18], [20], [21], [35]–[37] and its variants [5], [22], [38], [39] are also examples of CSAs. They are effective even when cross-VM memory sharing (*i.e.,* memory deduplication) is not supported by the hypervisor. Instead of exploiting memory sharing, these attacks utilize contention on a cache set of L1 cache [18], [37], [40] or last level cache (LLC) [7], [16], [20], [21], which are shared between an attacker and a victim.

We begin by supposing that a victim has a secret-dependent access pattern at a virtual memory address $\mathcal{A}$, and the cache line corresponding to that address is located on a $\mathcal{S}$-th cache set ($1 \leq \mathcal{S} \leq N$), where N is the number of cache sets: N is determined based on a target in the cache hierarchy (*e.g.,* L1 cache or LLC) at which the attack is delivered. Provided that an attacker has an eviction set e that shares the same location as the victim's target set (*i.e.,* the $\mathcal{S}$-th cache set), he/she mounts a Prime+Probe attack as follows. First, the attacker primes the target set with data on its eviction set e. Then, he/she waits for the victim to perform operations, wherein access to the target set may be obtained based on its secret value. Finally, the attacker probes the set by measuring the access latency for its eviction set. A high latency indicates that the victim has access to the target set.

In this attack, the attacker generally has no prior information regarding the target cache set (*i.e.,* $\mathcal{S}$), as the memory address, $\mathcal{A}$, is obfuscated through address space layout randomization (ASLR) by a guest OS on the victim's VM. Hence, it is necessary for the attacker to determine $\mathcal{S}$ ($1 \leq \mathcal{S} \leq N$) to construct the eviction set successfully. The result of previous work [7] has revealed that identification of target cache set $\mathcal{S}$ is possible using statistical profiling techniques even when ASLR is in operation, provided that address $\mathcal{A}$ remains fixed during the victim's execution.

IV. THREAT MODEL AND DESIGN GOALS

In this section, we present a threat model which is considered in our work and describe some design goals for the proposed method.

A. THREAT MODEL

We assume that an attacker and a victim are co-located on the same physical host, while their own guest VM is running on the host. Co-location of VMs from different security domains (*i.e.,* an attacker and a victim) is common in virtualization-based multi-tenant cloud computing services such as AWS, Microsoft Azure, and Google Compute.

The victim uses cryptographic algorithms in an application to maintain data confidentiality. The algorithms are implemented in crypto libraries such as OpenSSL and Libgcrypt. The attacker aims to extract a secret key or a private key from the victim by launching CSAs against the library.

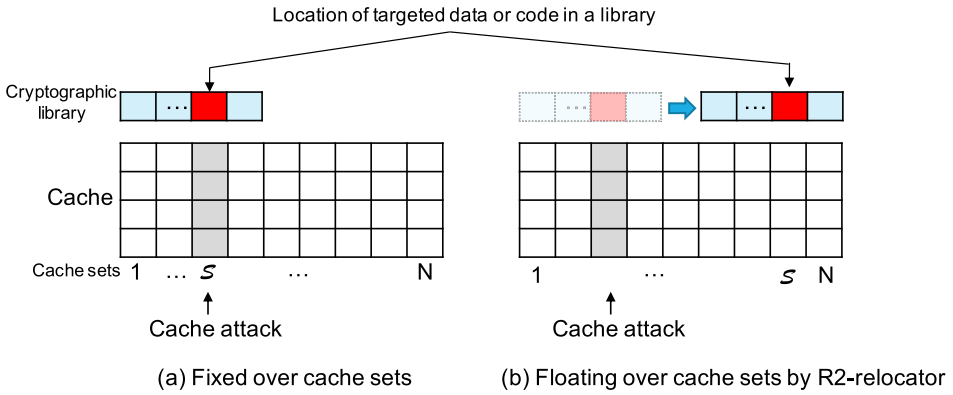


FIGURE 1. Illustration of the operation of the proposed method.

Specifically, he/she exploits implementation flaws of the library that leak secret-dependent memory access patterns through the cache, from which the secret or private key can be recovered. We assume that cross-VM memory sharing is not available on the host. As described in Section III, it is reasonable to assume that most hypervisors turn off memory deduplication as the default setting for security reasons [41]. Thus, the attacker is forced to conduct sharing-independent cache attacks, while assuming that he/she is free to choose the target cache (*e.g.,* the LLC or the L1 cache) for the attack.

We also assume that the details of the executable binary of crypto libraries (*e.g.,* the version information or the binary file itself) in the victim's application are known to the attacker [6], [7].

B. DESIGN GOALS

Our work has three design goals in terms of the security, performance, and deployability as follows.

1) SECURITY

The primary goal of the R2-relocator is to protect the victim's applications that use a (vulnerable) crypto library from attackers trying to extract secret information using sharing-independent cache attacks. More specifically, R2-relocator aims to prevent attackers from *identifying the target cache set* $\mathcal{S}$ of the crypto library, which is necessary for the attacker to deliver the attack successfully. As the attacker is able to target all cache levels, R2-relocator should provide security against sharing-independent cache attacks aiming not only at the LLC but also at the L1 cache.

2) PERFORMANCE

Applying R2-relocator to the vulnerable victim's application may affect the performance in terms of storage (*i.e.,* the size of the executable binary) and computation (*i.e.,* the execution time). Our design goal is to minimize the storage and computational overhead when R2-relocator is in use.

3) DEPLOYABILITY

We attempt to eliminate the need for reconfiguration of crypto libraries while utilizing R2-relocator in a target application. However, modification of the application may be necessary for integration with the R2-relocator. Another goal is to minimize the modification cost. Specifically, we attempt to perform the integration without modifying and recompiling the application's source code.

V. RUNTIME RANDOMIZED RELOCATION

In this section, we present the proposed method for runtime randomized relocation. First, we describe a system overview of the method. Then, we give details on its implementation and operations.

A. SYSTEM OVERVIEW

We briefly provide an overview of the enforcement of security on vulnerable crypto libraries while achieving the desired design goals. The basic idea is to transform an executable binary of the crypto library into one that has its location in the memory, as well as the corresponding cache set, that randomly changes during the runtime. By doing so, an attacker is interfered with mounting CSAs against the library. This approach follows the general idea of MTD to achieve system security [12], [42]. MTD reduces the opportunity of attackers and increases the cost of an attack by dynamically controlling the configuration of executable binaries.

The idea is concisely illustrated in Fig.1. In the case where the R2-relocator is not in use (depicted in Fig.1(a)), the binary of the library occupies specific parts of the cache sets; the occupied sets are determined when the library is loaded to the memory, and they remain fixed once the library is loaded by a dynamic linker. Once the attacker identifies the target set in the library (*i.e.,* $\mathcal{S}$ in Fig.1), he/she is able to extract secret values successfully by mounting a CSA against $\mathcal{S}$, as long as the victim's application continues running.

As illustrated in Fig.1(b), the R2-relocator allows the library to keep floating over cache sets during runtime. In other words, it repeatedly (1) randomly chooses a new base address for a library and (2) relocates the library to the chosen address. This causes the occupied cache sets to change over time, thereby obfuscating the target set $\mathcal{S}$. Unless the attacker can trace a new target set, determining the secret

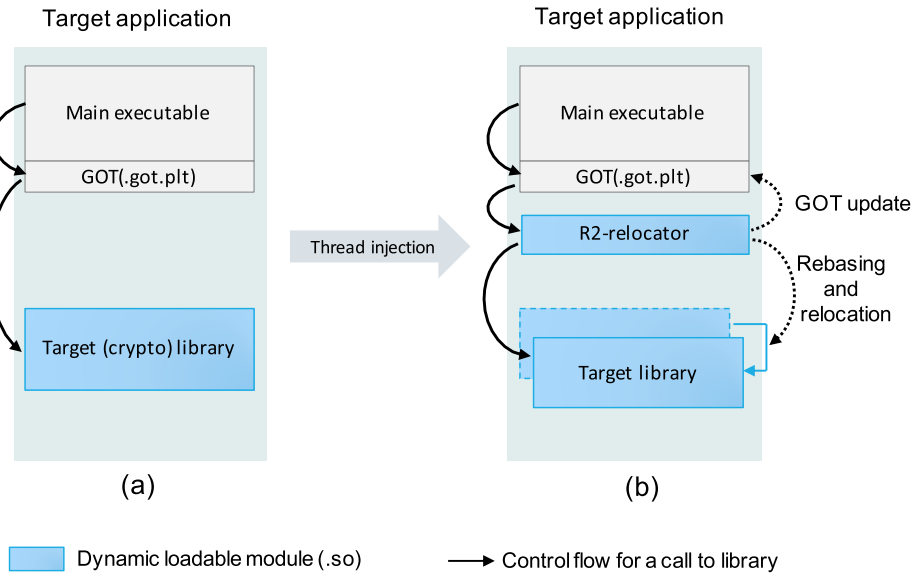


FIGURE 2. Injecting R2-relocator into a target application.

value via cache attacks is infeasible. Frequent operations of randomized runtime relocation significantly reduce the chance for successful identification of $\mathcal{S}$ by the attackers.

B. R2-RELOCATOR

In this section, we describe an implementation of R2-relocator and its operation in detail.

1) IMPLEMENTATION

The general use-case of libraries (including crypto libraries) in applications is to dynamically link them to main executable of the application process in the runtime. In Linux, an executable binary of the library has the form of a dynamically loadable module with the .so file extension, which is built with position-independent code (PIC) (see Fig.2(a)). The PIC allows the library to be dynamically linked and relocated to arbitrary addresses in the process memory. The relocated information is updated accordingly to a global offset table (GOT) in the process so that subsequent library function calls are appropriately directed to the relevant binary of the library.

We implement the R2-relocator in the form of a dynamic loadable module as well (see Fig.2(b)). Using ptrace technology, the R2-relocator can be injected as a thread into arbitrary running processes. ptrace is a set of system calls in Linux that enables process manipulation. As depicted in Fig.2(b), the injected R2-relocator manipulates the GOT in the process, which contains the linkage information of the target library, and begins to perform random rebasing and relocating of the target library. The detailed operations will be presented in the following section.

We would like to emphasize that such an implementation strategy of the R2-relocator facilitates significant benefits in terms of deployability. In other words, without modifying and recompiling either the source code of the target application or the library (thus, redeployment is not necessary), we can transform an application into one that performs runtime randomized relocation to the target crypto library.

2) OPERATION

Once the R2-relocator is injected into a running process of the target application, it is assigned a new thread in the process and begins to perform its operations in the thread context. The R2-relocator runs different operations with respect to its execution phase.

a: SETUP PHASE

After being injected, the R2-relocator starts a setup phase. Specifically, it performs the following operations in this phase.

Step 1 Allocating Memory Regions: First, it allocates two new regions of memory, $region_0$ and $region_1$, and maps them to the address space of the process. These regions are used to conduct re-randomization and relocation of the library. The allocating size of each region is decided based on the size of the target library, S_l, and the degree of randomization, D ($D > 0$). We configure D to be $D = N$, where N is the number of cache sets on the LLC or the L1 cache. Then, the size of each region, S_r, can be determined as follows:

$$S_r = S_l + (D - 1) \cdot b, \tag{1}$$

where b is the size of a cache line ($b = 64$ bytes in most processors) in the cache. It should be noted that S_r is the minimum size of the allocated space for each region to perform an operation.

Allocation of Two Regions: We need to allocate at least two memory regions. This is because performing operations of the R2-relocator (*i.e.,* re-randomization and relocation)

over a memory region while allowing the target application to concurrently execute tasks within a library in the same region that is being relocated is likely to cause consistency problems. The preparation of two memory regions, $region_0$ and $region_1$, can prevent this problem; the R2-relocator performs its operation over one region (*e.g.,* $region_0$) while the main threads of the application simultaneously use a library located in another region (*e.g.,* $region_1$). When the relocation of the library region is completed, the R2-relocator is swapped between these two regions and continues its operation over the other region. In the remainder of this paper, we denote a region in which a library is actively in use by the application as the *active region* and the other region as the *inactive region*.

Step 2 Setting up Hooks in GOT: After allocating memory regions in the process context, the R2-relocator installs hooks for all export functions of the target library at the GOT. As a result, all the library function calls invoked by the application are passed to the R2-relocator, which, in turn, mediates and redirects the function calls to the library at the appropriate location (*i.e.,* the active region).

The function hooks also provide synchronization between the R2-relocator and the target application to ensure consistent executions.

b: RUNNING PHASE

After initialization, the R2-relocator proceeds to a running phase. In this phase, it begins to repeatedly perform an operation over an inactive region via a loop. The operation that the R2-relocator executes for each iteration of the loop consists of three steps, as depicted in Fig.3.

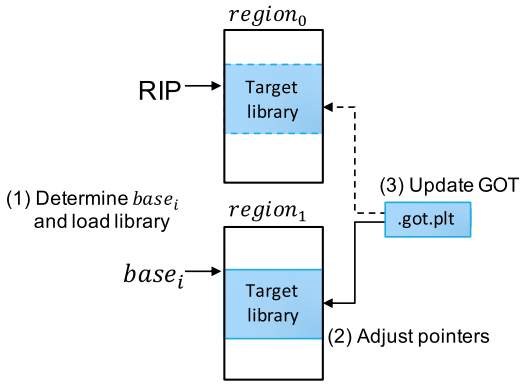


FIGURE 3. Operation of the R2-relocator in a running phase.

Specifically, for the i-th iteration, the R2-relocator proceeds with its operation over inactive region $region_c$, where $c = i \mod 2$ ($i > 0$), as follows.

Step 1 Rebasing and Reloading Library: It determines the new base address of the library from the address space of $region_c$. The base address of the library, $base_i$, is calculated as

$$base_i = base_r + b \cdot H_k(i),$$

where $base_r$ is the base address of $region_c$, b is the line size of the cache, and $H_k : \{0,1\}^* \rightarrow \{0,1,\ldots,D-1\}$ is a cryptographic keyed hash function that uses a secret key k and maps an arbitrary value to an integer in a range from 0 to $D-1$. (D is the randomizing degree.)

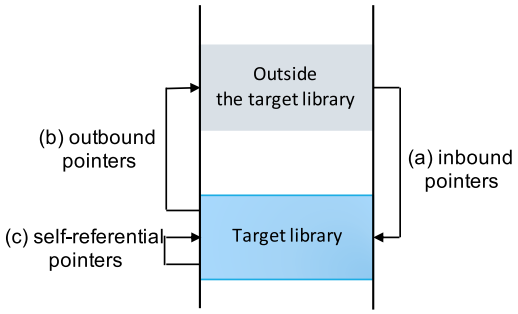


FIGURE 4. Three types of pointers that are relevant to the library.

The R2-relocator then performs a memory copy of the library located in the active region (*e.g.,* $region_0$ in Fig.3) to the newly determined base address $base_i$ in inactive region $region_c$ (*e.g.,* $region_1$ in Fig.3).

Step 2 Pointer Adjustment: Given that the location of the library in the memory changes, we need to adjust relevant pointers accordingly to avoid missing links to objects in the library.

As shown in Fig.4, three types of pointers are relevant to the library: (a) inbound pointers, located outside the library that reference the library objects, (b) outbound pointers, located inside the library that reference objects outside the library (*e.g.,* the main executable or other libraries), and (c) self-referential pointers that are also inside the library but point to its objects in the library.

Among these pointers, we do not consider the outbound and inbound pointers in this step. This is because in the case of outbound pointers, any executables except the target library remain unchanged relative to their locations. For inbound pointers, we need to update the corresponding GOT entries in the .got.plt section of the main executable, which will be conducted in the next step.

Hence, in this step, we only need to consider the self-referential pointers. These pointers are located in a data section of the library, which can be identified via program analysis. We update each pointer in the section by adjusting its value by Δ, where $\Delta = base_i - base_{i-1}$ ($i > 1$).

Step 3 Updating the GOT: In this step, we complete the running phase by updating the GOT. Specifically, we modify entries in the table of .got.plt so that subsequent procedure calls are passed to the newly relocated library.

VI. EVALUATION

In this section, we evaluate the R2-relocator with respect to its performance, security, and deployability.

A. PERFORMANCE

1) EXPERIMENTAL SETUP

We conducted several experiments to evaluate the performance of the R2-relocator. In these experiments,

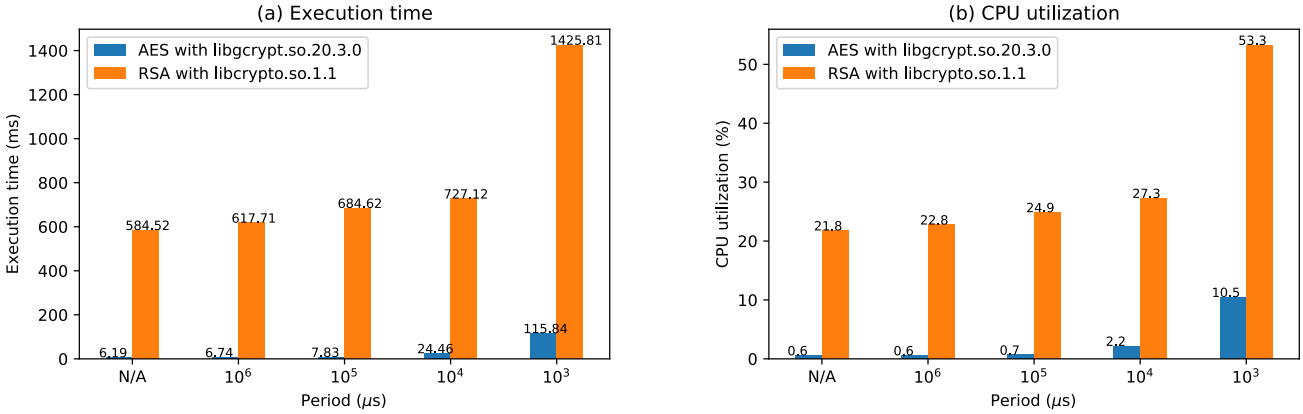


FIGURE 5. Performance evaluation results with respect to the runtime execution of R2-relocator.

we implemented two victim applications that perform several cryptographic operations. In particular, one of those applications performed AES encryptions, wherein the AES algorithm was implemented in the Libgcrypt library (*i.e.,* libgcrypt.so.20.3.0). The other performed RSA encryptions, wherein the algorithm was implemented in the OpenSSL library (*i.e.,* libcrypto.so.1.1). These applications were built using a GNU C compiler for version 9.3.0 and executed in Ubuntu 20.0.4 LTS 64-bit Linux.

The R2-relocator was injected as a running thread into each process of the applications during runtime. For the runtime injection, we used the ptrace APIs provided in Linux.

2) RESULTS

In the experiment, a single execution of each application consisted of one hundred encryptions and decryptions. During the execution, the R2-relocator simultaneously performed runtime randomized relocations of the target library (*i.e.,* libgcrypt.so.20.3.0 or libcrypto.so.1.1) inside the process.

a: COMPUTATIONAL OVERHEAD

We measured the execution time and CPU utilization of the applications as the R2-relocator was employed. Fig.5 presents the results of the experiments. The term 'Period (μs)' in the graph refers to the interval of relocation in microseconds, which represents the frequency at which the R2-relocator is invoked. For comparison, the measurement was also conducted without utilizing the R2-relocator, and the results are shown as 'N/A' in the graph.

We observed that both execution time and CPU utilization gradually increased with the frequency of up to 10^4 μs. In the case of an RSA application, for instance, execution of the R2-relocation with a relocation period of 10^4 μs (*i.e.,* 10 ms) resulted in an execution time of 727.12 ms and 27.3% CPU utilization, which introduced approximately 25% overhead compared to the case where the R2-relocator was not used. For a frequency of 10^6 μs (*i.e.,* 1 s), the R2-relocator introduced the overhead of at most 5% and 9% for the RSA and AES applications, respectively. The performance overhead is also confirmed by the change in the cache hit rate. We monitored the cache hit rate by using a Linux perf tool while executing R2-relocator over the crypto applications. The result is shown in Table 1. For instance, we observe that the cache hit rate of the RSA application slightly decreases from 83.3% where R2-relocator is not in use (*i.e.,* 'N/A') to 77.1% where in use with a frequency of 10^6 μs.

TABLE 1. Cache hit rates with varying the relocating frequency.

Crypto application	Period (μs)				
	N/A	10^6	10^5	10^4	10^3
AES (libgcrypt)	85.4%	79.3%	71.8%	56.4%	43.6%
RSA (libcrypto)	83.3%	77.1%	63.5%	49.6%	38.7%

We can also observe that for a frequency of 10^3 μs, the overhead significantly increases for both applications. We attribute the increase in the overhead to the additional computational burden owing to synchronization between the main process and the R2-relocator.

The running frequency of the R2-relocator may be determined based on the trade-off between security and performance (*i.e.,* the computational overhead). As we will discuss in the next section, running with minimal frequency (*i.e.,* 10^6 μs) is sufficient to provide protection from state-of-the-art CSAs. Hence, the R2-relocator achieves the required security with a minimum computational overhead of only 9%.

b: STORAGE AND MEMORY OVERHEAD

We evaluate the R2-relocator in terms of its storage and memory overhead. The R2-relocator occupies 361 KB in a file system in the case when a compile option "-O2" is used to build the executable binary, which is negligible compared to the size of the target applications.

In terms of the memory overhead, the R2-relocator requires additional memory to allocate two regions (*i.e.,* $region_0$ and $region_1$) during runtime. The required size of each region depends on both the actual binary size of the target library and the degree of randomization, as described in the previous section. In the case of the Libgcrypt library, for instance, the size of the binary file libgcrypt.so.20.3.0 is 4.8 MB. Then, for $D = 16,384$ and $b = 64$, the minimal size

TABLE 2. Experimental result.

Secret length (λ)	Period = $4 \cdot 10^4 \mu$s			Period = $2 \cdot 10^4 \mu$s			Period = $10^4 \mu$s		
	T_{L1} (min)	T_{LLC} (min)	Ratio (T_{L1}/T_{LLC})	T_{L1} (min)	T_{LLC} (min)	Ratio (T_{L1}/T_{LLC})	T_{L1} (min)	T_{LLC} (min)	Ratio (T_{L1}/T_{LLC})
4	105	210	0.50	126	252	0.50	220	423	0.52
8	220	420	0.52	264	520	0.51	450	850	0.53
16	550	850	0.65	660	1050	0.63	1210	1820	0.66
32	1320	1790	0.74	1584	2192	0.72	3420	4010	0.85
64	2190	2780	0.79	2820	3336	0.85	5630	6240	0.90

of a region S_r is determined to be at least 5.8 MB according to Eq.1. Thus, at least 11.6 MB of additional memory is required for the R2-relocator to perform its operation over the Libgcrypt library.

B. SECURITY

To evaluate the security of the R2-relocator, we consider an adversary who mounts a sharing-independent cache attack (*e.g.,* Prime+Probe [7]) against a victim's application that uses a crypto library. The attacker and victim run as a VM on a host, where the number of cache sets of an LLC is N.

As described in Section III, the attacker has to determine the target cache set $\mathcal{S}$ ($1 \leq \mathcal{S} \leq N$) before performing the cache attack. If only the temporal cache access pattern of the victim's application is known, the attacker needs to examine all the LLC sets individually to determine which one is relevant. In most modern processors, the number of LLC sets (N) varies from 2,048 (for desktop processors) to 16,384 (for server processors), none of which are sufficiently small to determine $\mathcal{S}$ within a short period of time.

Even after successfully identifying S, the attacker requires a sufficient amount of time to extract full bits of secret from the target cache set of the victim's application. For instance, Liu *et al.* reported in their work [7] that the recovery of a secret key from GnuPG, which is an executable version of Libgcrypt, required at least 12 min. The reported time included both the identification of $\mathcal{S}$ and the extraction of the secret against $\mathcal{S}$, both of which are referred to as *an online attack* in [7]. It should be noted that the result was obtained under the setting that the location of the victim's executable binary remains static during runtime.

With the R2-relocator enabled, the location of $\mathcal{S}$ changes frequently in a randomized manner during runtime. Hence, it is quite challenging for an attacker to determine target set $\mathcal{S}$ unless he/she can predict the next location. Even if the attacker succeeds in this identification, it is likely that $\mathcal{S}$ will be moved to another set during the cache attack, which will hinder the attacker from extracting secret values from the victim.

The frequency at which the relocation occurs is a configurable parameter of the R2-relocator. In other words, it may be chosen from a range between a few milliseconds to several seconds based on the trade-off between security and performance, as described in the previous section. Regardless of the frequency configured, the period of relocation (*i.e.,* the change in $\mathcal{S}$) is ten orders of magnitude shorter than the required time for the successful attack.

1) PREDICTABILITY OF THE RANDOMIZED RELOCATION

An attacker might be able to bypass the hurdle introduced by the R2-relocator if he/she can predict the next $\mathcal{S}$ chosen by the randomized relocation. The R2-relocator uses a keyed hash function H_k to randomize the next base address of the library. HMAC is an instance of the keyed hash functions. Provided that key k is kept secret, it is infeasible for the attacker to infer the next location chosen by the R2-relocator.

2) SECURITY AGAINST THE L1-BASED SHARING-INDEPENDENT CACHE ATTACK

As sharing-independent cache attacks are available not only to the LLC but also to the L1 cache, we now discuss the security that R2-relocator can achieve against L1-based cache attacks [37], [40], [43], [44]. Modern processors have 32KB L1 instruction (L1-I) cache and 32KB L1 data (L1-D) cache per physical core: each has 64 cache sets. Thus, the randomizing degree with which R2-relocator works over the L1 cache is at most $D = 64$, which is relatively low compared to the maximum degree over the LLC. The low randomizing degree might reduce the security to some extent when applying to the L1-based attacks.

In order to quantitatively measure the gap of security against between L1-based and LLC-based cache attacks, we conducted an additional experiment under the same environmental setting in Section VI-A1. For the experiment, we implemented a victim application that imitates the cache behavior of crypto libraries. It simply makes a specific cache footprint, based on a pre-configured secret value σ of length λ in bits, on a target set $\mathcal{S}$ in the L1-D cache. More specifically, given a bit vector $\sigma = \{b_0, \ldots, b_{\lambda-1}\}$, it iterates memory operation at the address in $\mathcal{S}$: at the i-th iteration, the memory load takes place only if $b_i = 1$. As L1-based attacks are also available to L1-I cache [43], [44], we implemented a variant of the victim that the cache footprint is made over the instruction cache. During the experiment, these victim applications are running with R2-relocator being applied within these processes.

We also constructed a spy program that attempts to extract σ from the victim applications. The spy launches the sharing-independent attack at different cache levels. For the LLC-based attack, the spy employs a technique of [7]. For the attacks aiming at L1-D and L1-I cache, the spy employs techniques of [37] and [44], respectively. In the experiment, we measured the total amount of time spent by the spy on successfully extracting the secret σ from the victim.

Table 2 shows the experimental result. The terms 'T_{L1}' and 'T_{LLC}' refer to the total amount of time spent by the spy for the attack with L1-based and LLC-based techniques, respectively. For the L1-based spy, a smaller one between results of L1-I and L1-D attacks was chosen to be T_{L1}. The term 'Ratio' implies the security gap between L1-based and LLC-based attacks. As shown in the table, the gap is relatively small for the victim with a large length of secret (λ). The security gap will close even further with shorter interval (*i.e.,* Period) of R2-relocator, which will come at the cost of increasing in the computational overhead.

C. DEPLOYABILITY

One of the design goals of the R2-relocator is deployability; it should be easily integrated with applications without modifications to the source code.

As indicated in Section V-B1, the R2-relocator is implemented as a dynamically loadable module. Thus, it can be injected into arbitrary running processes using the ptrace API in Linux. The injected module then transforms the application to a binary level so that it can perform runtime randomized relocation to the target crypto library in the application process. This has a significant advantage with respect to deployability, as it eliminates the need for reconfiguration (*i.e.,* modification and recompilation of the source code) of either the application or the target library.

1) PROCESS STABILITY

R2-relocator can be injected into a running process with help of ptrace system calls. This may affect the process stability. For instance, the process may lose its stability if library calls from the application are made concurrently while the injected R2-relocator is operating relocation over the same library. In order to avoid this problem, R2-relocator does not directly access the target shared library within the injected process. Instead, as described in Section V-B, it manipulates the library over newly allocated memory regions. Besides, it provides synchronization with the target process by setting up function hooks in GOT. Specifically, a mutex-based synchronization is implemented in function hooks so that every library call is checked before being passed to the library. This way, simultaneous accesses to the library from both an application and R2-relocator is prevented. This allows R2-relocator to perform correctly without breaking consistency of the process. Therefore, the process stability can be preserved while R2-relocator is running concurrently with the target process.

VII. CONCLUSION

In this paper, we proposed the application of the R2-relocator to protect crypto libraries from CSAs in a cloud computing environment. Using the novel idea of a MTD, the R2-relocator provides robust protection to a vulnerable crypto library based on random movement over the memory during runtime. Given that the corresponding cache set changes frequently over time, it is difficult for an attacker to identify a target cache set, which is a prerequisite for a successful attack.

The R2-relocator can be easily deployed to cloud services because it does not require reconfiguration (*i.e.,* recompilation of the source code) of the crypto library. All vulnerable libraries that contain unpatched CSA-susceptible flaws can be protected from an attack. We extensively evaluated the proposed method in terms of its performance, security, and deployability. The evaluation results demonstrate its effectiveness for achieving secure cloud services.

REFERENCES

[1] L. Xiao, D. Xu, C. Xie, N. B. Mandayam, and H. V. Poor, "Cloud storage defense against advanced persistent threats: A prospect theoretic study," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 3, pp. 534–544, Mar. 2017.

[2] OpenSSL. (2020). *OpenSSL—Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org/

[3] Libgcrypt. *Libgcrypt-GnuPG*. [Online]. Available: https://gnupg.org/software/libgcrypt/index.html

[4] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 131–145.

[5] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1967–1984.

[6] Y. Yarom and K. Falkner, "Flush + reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23th USENIX Secur. Symp.*, 2014, pp. 719–732.

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.

[8] C. P. García and B. B. Brumley, "Constant-time callees with variable-time callers," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 83–98.

[9] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 845–858.

[10] C. Pereida García, B. B. Brumley, and Y. Yarom, "'Make sure DSA signing exponentiations really are constant-time,'" in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1639–1650.

[11] S. Sengupta, A. Chowdhary, A. Sabur, A. Alshamrani, D. Huang, and S. Kambhampati, "A survey of moving target defenses for network security," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 3, pp. 1909–1941, 3rd Quart., 2020.

[12] M. Torquato and M. Vieira, "Moving target defense in cloud computing: A systematic mapping study," *Comput. Secur.*, vol. 92, pp. 1–11, May 2020.

[13] S. Chen, F. Liu, Z. Mi, Y. Zhang, R. B. Lee, H. Chen, and X. Wang, "Leveraging hardware transactional memory for cache side-channel defenses," in *Proc. Asia Conf. Comput. Commun. Secur.*, May 2018, pp. 601–608.

[14] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptograph. Eng.*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[15] Wolfssl. *WolfSSL, Embeded TLS Library*. [Online]. Available: https://www.wolfssl.com/

[16] B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst. (CHES)*, 2016, pp. 368–388.

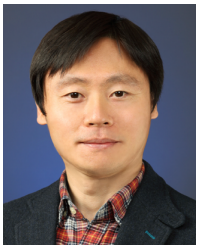
[17] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack," IACR Cryptol. ePrint Arch., Tech. Rep., 2014/140, 2014. [Online]. Available: https://eprint.iacr.org/2014/140

[18] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2012, pp. 305–316.

[19] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *Research in Attacks, Intrusions and Defenses* (Lecture Notes in Computer Science), vol. 8688. 2014, pp. 299–319.

[20] B. Gulmezoglu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross-VM cache attacks on AES," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 2, no. 3, pp. 211–222, Jul. 2016.
[21] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing–And its application to AES," in *Proc. 2015 IEEE Symp. Secur. Privacy*, May 2015, pp. 591–604.
[22] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "PRIME+ABORT: A timer-free high-precision L3 cache attack using Intel TSX," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 51–67.
[23] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 549–564.
[24] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.
[25] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 990–1003.
[26] Y. Shin, "Cross-VM cache timing attacks on virtualized network functions," *IEICE Trans. Inf. Syst.*, vol. E102.D, no. 9, pp. 1874–1877, 2019.
[27] Y. Shin, D. Koo, and J. Hur, "Inferring firewall rules by cache side-channel analysis in network function virtualization," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Jul. 2020, pp. 1798–1807.
[28] M. Azure. (2021). *Azure Automation Update Management Overview*. [Online]. Available: https://docs.microsoft.com/en-us/azure/automation/update-management/ove%rview
[29] B. B. Brumley and N. Tuveri, "Cache-timing attacks and shared contexts," in *Proc. Int. Workshop Constructive Side-Channel Anal. Secure Design (COSADE)*, 2011, pp. 1–10.
[30] T. Hornby, *Side-Channel Attacks on Everyday Applications: Distinguishing Inputs With FLUSH+RELOAD*. BlackHat, 2016.
[31] G. Berk, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush + reload attack on AES," in *Proc. Int. Workshop Constructive Side-Channel Anal. Secure Design (COSADE)*, 2015, pp. 111–126.
[32] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley, "Cache-timing attacks on RSA key generation," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, pp. 213–242, Aug. 2019.
[33] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proc. 13th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, 2016, pp. 279–299.
[34] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, May 2016, pp. 353–364.
[35] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, K. Roemer, and S. Mangard, "Hello from the other side: SSH over robust cache covert channels in the cloud," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 21–24.
[36] L. Zhang, A. A. Ding, Y. Fei, and Z. H. Jiang, "Statistical analysis for access-driven cache attacks against AES," Cryptol. ePrint Arch., Tech. Rep. 2016/970, 2016, pp. 1–31.
[37] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers Track at RSA Conf. (CT-RSA)*, 2006, pp. 1–20.
[38] B. Gras, H. Bos, and C. Giuffrida, "Translation leak-aside buffer : Defeating cache side-channel protections with TLB attacks," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 955–972.
[39] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox—Practical cache attacks in Javascript," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 1406–1418. [Online]. Available: http://arxiv.org/abs/1502.07373
[40] C. Percival, "Cache missing for fun and profit," in *Proc. BSDCan*, 2005, pp. 1–13.
[41] VMware. (2018). *Security Considerations and Disallowing Inter-Virtual Machine Transparent Page Sharing*. [Online]. Available: https://kb.vmware.com/s/article/2080735
[42] C. Lei, H.-Q. Zhang, J.-L. Tan, Y.-C. Zhang, and X.-H. Liu, "Moving target defense techniques: A survey," *Secur. Commun. Netw.*, vol. 2018, pp. 1–25, Jul. 2018.
[43] O. Aciiçmez, "Yet another MicroArchitectural attack: Exploiting I-cache," in *Proc. ACM Workshop Comput. Secur. Archit. (CSAW)*, 2007, pp. 1–13.
[44] O. Aciiçmez, B. Bob Brumley, and P. Grabher, "New results on instruction cache attacks," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*, 2010, pp. 110–124.

YOUNGJOO SHIN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2006, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2008 and 2014, respectively. He was with the National Security Research Institute (NSR), Daejeon, as a Senior Researcher, from 2008 to 2017. He was with Kwangwoon University, Seoul, as an Assistant Professor, from 2017 to 2020. He is currently an Assistant Professor with the School of Cybersecurity, Korea University. His research interests include system and network security, CPU microarchitectural security, cloud computing security, and vulnerability analysis on embedded systems.



JOOBEOM YUN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 1999, and the M.S. degree in computer engineering from Seoul National University, Seoul, in 2001, and the Ph.D. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2012. He is currently an Associate Professor with the Department of Computer and Information Security, Sejong University, Seoul. His research interests include software security, artificial intelligence (AI) security, and network security.