See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer

Zeyu Zhang[1,2,3*], Xiaoli Zhang[1,2*], Qi Li[1,2], Kun Sun[3], Yinqian Zhang[4], Songsong Liu[3], Yukun Liu[5], Xiaoning Li[5]
[1]Institute for Network Sciences and Cyberspace & Dept. of Computer Science and Technology, Tsinghua University
[2]Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China
[3]Department of Information Sciences and Technology, CSIS, George Mason University
[4]Ohio State University, [5]Alibaba Inc.
{zy-zhang18@mails., zhangxl15@mails., qli01@}tsinghua.edu.cn, ksun3@gmu.edu, yinqian@cse.ohio-state.edu, sliu23@masonlive.gmu.edu, {yidun.lyk, xiaoning.li}@alibaba-inc.com

ABSTRACT
Intel Software Guard Extensions (SGX) offers strong confidentiality and integrity protection to software programs running in untrusted operating systems. Unfortunately, SGX may be abused by attackers to shield suspicious payloads and conceal misbehaviors in SGX enclaves, which cannot be easily detected by existing defense solutions. There is no comprehensive study conducted to characterize malicious enclaves. In this paper, we present the first systematic study that scrutinizes all possible interaction interfaces between enclaves and the outside (i.e., cache-memory hierarchy, host virtual memory, and enclave-mode transitions), and identifies seven attack vectors. Moreover, we propose SGX-Bouncer, a detection framework that can detect these attacks by leveraging multifarious side-channel observations and SGX-specific features. We conduct empirical evaluations with existing malicious SGX applications, which suggests SGX-Bouncer can effectively detect various abnormal behaviors from malicious enclaves.

CCS CONCEPTS
• **Security and privacy** → *Malware and its mitigation*; *Trusted computing*.

KEYWORDS
trusted computing; malware detection; side channel

ACM Reference Format:
Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, Xiaoning Li. 2021. See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*, June 7–11, 2021, Hong Kong, Hong Kong. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3433210.3437531

*Both authors contributed equally to this research.

1 INTRODUCTION
Intel Software Guard Extensions (SGX) is an emerging hardware security feature available on modern Intel processors. It offers hardware protection for user-level applications against attacks from compromised system software. The security of SGX applications is guarded by memory isolation, memory encryption and remote attestation. Since SGX successfully decouples the trust between the applications and the underlying system software, it enables applications to be securely executed on untrusted environments. For example, commercial cloud service providers have offered SGX-enabled infrastructure as a service (IaaS) [10, 11] and function as a service (FaaS) [15, 20]. Meanwhile, software vendors have developed various client-side SGX applications [4, 6]. Particularly, blockchain systems use SGX-protected cryptocurrency wallets [5] and run smart contracts in SGX-enabled platforms for guaranteed computation integrity [18].

Unfortunately, similar to other security mechanisms that have been misused by attackers (e.g., rootkit in SMM [52]), SGX may also be exploited by attackers to protect malicious enclaves [34, 41] or misuse software vulnerabilities in third-party SGX applications [17]. Potential victims include machines running third-party SGX applications [4], worker clients in privacy-preserving Blockchain systems [18], and FaaS clouds [15, 20]. Recently, Schwarz et al. [41] show that enclave malware can recover a full 4096-bit RSA key used in another enclave. SGX-based ransomware [34] has been implemented to encrypt vital information for access restriction, where encryption keys are maintained inside the enclave to defeat key recovery.

One reason for the growing popularity of using SGX as a hotbed for malware is its stealthiness under the state-of-the-art anti-virus software. First, malicious code may stay in the encrypted form before being loaded into enclave, e.g., via Intel Protected Code Loader (PCL). This protects malware against reverse engineering and static code analysis. Second, SGX inherently offers isolated enclaves that cannot be accessed from outside, even with the root privilege. Thereby, existing anti-virus tools cannot access the memory content of malicious enclave. Third, SGX explicitly suppresses x86 hardware debug assistance features when the enclave is running in the release mode [8]. It protects malicious enclave from being analyzed by traditional debuggers. As such, allowing an application to use SGX is equivalent to permitting it to possess a shielded region to conceal malware payload, which unfortunately is out of reach for any state-of-the-art malware detector. Moreover, on Linux servers that support flexible launch control (FLC) [8], as the intended behavior of the SGX driver permits the application to launch any production enclaves (no longer needs a token issued by Launch Enclave), neither the server administrator nor Intel can control the credibility of the enclave code. Therefore, an affected application can be instrumented to run malicious enclave as long as loading an enclave inside the application is expected.

Since Intel SGX has restricted any code running inside enclave from directly calling outside code, making system calls, or executing privileged instructions [8], all enclave interactions with outside are forced to go through their host applications. Therefore, it seems applying advanced malware analysis (e.g., static binary analysis [31] or dynamic syscall tracing [21]) on the applications running outside would be sufficient to detect malicious behaviors of the enclave malware. However, we find this is not the case at least in two scenarios. First, a malicious enclave may exploit the shared micro-architectural components (e.g., cache hierarchy) to launch side channel attacks against other applications [44, 51]. Second, a malicious enclave may hijack the control flow of its host application (e.g., via ROP attacks [43]) and interact with the OS in ways not dictated by the code outside the enclave.

This observation motivates us to seek answers to two questions. First, *what attack methods enable malicious enclave to evade traditional malware detection techniques*. To answer this question, we perform a systematic study on the attack vectors of enclave malware by scrutinizing all interaction interfaces between the enclave and the outside software components. Specifically, there are three types of interaction interfaces. 1) *Cache-memory hierarchy* is shared between enclaves and other co-located processes. This can be exploited by an enclave to launch various side channel attacks and infer sensitive information. 2) *Host virtual memory* can be directly accessed by the enclave to exchange data with the host application. This enables the enclave to conduct abnormal memory read and write and further violate memory safety of the host application. 3) *Enclave-mode transitions* via `EENTER`/`EEXIT` instructions can switch execution flows between the enclave and its host application, which can also be abused by the enclave. Accordingly, we have identified seven concrete attack vectors that fall into these three categories.

Second, we aim to explore *how malicious enclave can be detected with sufficient accuracy*. However, it is not trivial to detect those identified attack vectors. First, traditional detection methods for cache side channel attacks [56] cannot work, since they rely on Performance Monitoring Counters (PMC) to monitor abnormal cache hits/misses, but PMC is disabled when the CPU is in the enclave mode. Second, prior detection methods against abnormal memory access (e.g., ROP attacks) run in emulation modes [23], which is not compatible with SGX; tracing the enclave execution directly is infeasible too. Third, there lacks a generic method to capture enclave-mode transitions, since they can be implemented not only by wrapper functions (e.g., `ECall`/`OCall`) of specific SGX development frameworks but also by enclave developers.

To overcome these challenges, we propose an offline analysis framework for malicious enclave, which we call *SGX-Bouncer*. Following the idea of Google Bouncer [1], our goal is to run SGX-Bouncer as a malicious enclave detection service. It could be leveraged by vendors like Intel who want to build an SGX app store to check if the SGX programs are benign. Also, SGX platform providers can upload enclave programs submitted by third-party software developers to SGX-Bouncer for potential malware detection before running them on production platforms. Similar to many of the malware detection frameworks [45], SGX-Bouncer is designed as a framework, which allows users to customize detection rules and can be extended when new features need to be included. 1 In this paper, we design and implement three detection capabilities. First, to monitor cache behaviors, we put some specific memory lines that use the same cache sets with the enclave into the cache of different levels. By inspecting cache hit/miss events of these memory lines, we can infer whether there exist cache contentions, finally discovering enclave abnormal cache usage behaviors. Second, to detect abnormal memory access behaviors, we devise a monitoring mechanism that is triggered once the CPU switches to the enclave mode. It detects abnormal read and write of host memory via checking side channels and data consistency. Third, to monitor enclave interfaces in a generic manner, instead of focusing on high-level wrapper functions of enclave-mode transitions provided by runtime systems, we directly capture all executions of `EENTER` and `EEXIT` instructions. Particularly, we identify pages that must be accessed before enclave entries and leverage the page-fault exception handler to capture all entries. To capture enclave exits, we leverage the SGX anti-debugging feature and set the debugging flag before enclave entries.

We implement a prototype of SGX-Bouncer by modifying Intel SGX Platform Software (PSW) and Intel SGX driver. Also, we develop a number of proof-of-concept malicious enclaves exploiting various attack vectors, and the experimental results show that SGX-Bouncer can effectively detect all these malicious enclaves.

In summary, the contributions of this paper are three-fold:

• We conduct the first systematic analysis on plausible attack vectors of malicious enclaves. These vectors are comprehensive to represent potential attacks from malicious enclaves.

• We propose a generic SGX malware detection framework SGX-Bouncer that is not limited to specific SGX development frameworks and includes a number of novel techniques to detect malicious behaviors on cache usage, memory access, and enclave-mode transitions. These techniques illustrate the positive use of side channel analysis in defense systems.

• We implement a prototype of SGX-Bouncer and validate its effectiveness with various proof-of-concept malicious enclaves. We also deploy SGX-Bouncer in real world to conduct case study.

2 BACKGROUND

Intel SGX is an extension to the x86 instruction set architecture of Intel processors. It confers hardware protections on user-level applications against hardware attacks and malicious software including compromised OS and hypervisor. SGX builds a shielded execution environment, called *enclave*, that provides confidentiality and integrity protection for inside applications against privileged attacks. All code and data of the enclave are encrypted and stored in isolated memory space, i.e., Processor Reserved Memory (PRM). Particularly, the enclave memory forbids accesses outside the enclave, but the enclave code can access the memory belonging to the host application (denoted as host memory). Besides, an enclave cannot directly make syscalls, since instructions that change privilege levels are illegal inside the enclave.

Enclave entry and exit. Enclave entry and exit are implemented by new instructions. To enter the enclave, the software performs the `EENTER` instruction with the address of one Thread Control Structure (TCS) in the enclave, which enables the processor to find the first instruction for execution. To exit the enclave, the software conducts `EEXIT` that cleans relevant contexts like Translation Lookaside Buffer (TLB) and transfers the execution to a designated location outside the enclave. Both of them will not clear the registers. In addition to the `EEXIT` instruction, Enclave Exiting Events triggered by exceptions or interrupts convert the processor into the non-enclave mode. To prevent potential leakage of secrets, an Asynchronous Enclave eXit (AEX) is performed to securely store states in a State Save Area (SSA) and create synthetic states. The RIP register will be replaced with Asynchronous Exit Pointer (AEP) that indicates the `ERESUME` instruction to re-enter the enclave.

SGX development framework. In both industry and academia, a wide spectrum of SGX development frameworks have been proposed to facilitate development of SGX applications. Some of them (e.g., Intel SGX SDK and Rust SGX SDK) demand the developers to divide the application into trusted and untrusted parts, and install the trusted part into the enclave. Others like Graphene-SGX [47] support to load unmodified application into an enclave to simplify the development complexity. To provide runtime libraries for applications building upon SGX, these frameworks generate the runtime systems including the SGX Kernel driver in the kernel space, the untrusted runtime system (uRTS) in the user space, and the trusted runtime system (tRTS) in the enclave space. uRTS and tRTS wrap low-level instructions as high-level enclave interfaces, e.g., `ECall`/`OCall`, that can be directly invoked by SGX developers. Also, SGX developers can customize runtime systems to provide enclave interfaces.

3 THREAT ANALYSIS

In this paper, we consider a scenario where SGX platform owners and developers of enclave programs are mutually distrusted. SGX platforms are victims that could be PCs, cloud virtual machines [10], bare-metal machines [11], or machines supporting function as a service (FaaS) [15]. Enclave developers are attackers who leverage the SGX technology as a new obfuscation or analysis-evasion tool to develop malware that is capable of evading existing static/dynamic anti-virus software. The enclave developers provide to the SGX platform hostile SGX programs including enclaves and their host applications [4, 34, 44] or only malicious enclaves that can be integrated as third-party libraries in SGX programs [43]. These SGX programs are executed on SGX platforms of benign users or clouds, targeting other co-located applications [41, 44, 51] or system software [34]. We do not differentiate these scenarios in this paper.

We assume the enclave code is hidden from the platform owner. The rest of the application code, however, is visible to the provider and has been manually reviewed or inspected by automated code analysis tools like [31]. We assume the goal of the malicious enclave is to breach the confidentiality and/or integrity of the hosting platform. We do not consider denial-of-service (DoS) attacks from malicious enclaves, such as SGX-Bomb [30] and memory bus locking [57]. DoS attacks slow down or even hang the entire system, which are by definition easy to spot. We also do not consider the TSX-based attacks inside the enclave, as Intel has recently disabled TSX in the SGX by a microcode patch [12], which has been validated by us on multiple SGX CPUs (i5-8500, i7-7700, and i7-8700). This paper does not focus on detecting attacks compromising host systems via issuing syscalls, which can be captured by existing dynamic syscall based detection tools [21] since these syscalls can only be issued by the host applications of the enclaves.

We present a systematical analysis of all interaction interfaces between the enclave and the outside (see Table 1), including *cache-memory hierarchy*, *host virtual memory*, and *enclave-mode transition*.

3.1 Attacks via Cache-Memory Hierarchy

Although there are lots of shared hardware components in the cache-memory hierarchy that have been exploited to launch side channel attacks in the literature, not all of them can be successfully mounted in our context. Hence, we first clarify what needs to be addressed and then present attack vectors of malicious enclaves.

What attacks need to address. Side channel attacks on the cache-memory hierarchy can be classified into single-core attacks and cross-core attacks. Single-core attacks can be further classified into HyperThreading (HT) based and Enclave Exit (EX) based. In the HT-based single-core attacks, the attacker enclave shares the same CPU core with the victim via HyperThreading [14, 46]. We do not consider HT-based attacks because Foreshadow [48] and MDS [42] attacks against SGX can only be prevented without HT and it has become a common practice to run SGX applications with HT disabled (which can be verified through remote attestation).

In the EX-based single-core attacks, malicious enclaves exploit the shared cache-memory hierarchy after transitioning to the non-enclave mode via Enclave Exits (e.g., AEX or `EEXIT`). Most of such EX-based attacks do not need to be considered, either.

First, the microcode patch [9] that mitigates Foreshadow attacks [48] automatically flushes the L1 data cache at enclave exit and, therefore, side channel attacks against L1 data cache from the malicious enclave cannot succeed. Second, an enclave cannot attack the outside software via shared TLB [28] as the TLB is cleared in enclave entries and exits. Third, as microcode patches for Spectre attacks, such as IBRS, prevent branch poisoning from the outside, side channel attacks on branch prediction units are infeasible from malicious enclaves. Thus, branch shadowing attacks [32] from malicious enclaves are also out of scope. As such, among all the known single-core attacks, we only consider *EX-based attacks via shared L2 caches.* Note attacks on L1 instruction cache attacks can work, but these attacks also affect the inclusive L2 cache and thus do not need to be considered separately.

For cross-core attacks, we consider attacks exploiting L3 caches via Prime+Probe [33] and Flush+Reload/Flush [26, 55]. We also consider cache-DRAM attacks [39, 51] that target fine-grained information leakage and require frequently cleansing L3 cache. We do not, however, consider Prime+Abort [24] side channel attacks, since they rely on TSX, which is not available in enclaves [12]. Also, since malicious enclaves cannot directly access page tables or manipulate APIC, page fault based side channel attacks [50] or APIC-based side channel attacks [32, 49] are out of scope.

Table 1: Summary of attack vectors

Interaction Interface	Attack Vector	Monitoring Module (MM)	Detection Module (DM)
Cache-memory hierarchy	❶ L2 cache Prime+Probe attack	L2Cache-MM	L2Cache-DM
	❷ L3 cache Prime+Probe attack	L3Cache-MM	L3Cache-DM
	❸ Flush+Reload/Flush attack	MemoryR-MM	FRF-DM
	❹ Cache-DRAM attack	L3Cache-MM	L3Cache-DM
Host virtual memory	❺ Memory disclosure attack	MemoryR-MM	MemoryR-DM
	❻ Host control-flow manipulation	MemoryW-MM	MemoryW-DM
Enclave-mode transition	❼ EEXIT abuse	EnclaveT-MM	EnclaveT-DM

We summarize the considered attack vectors as follows.

Vector ❶. *L2 cache Prime+Probe attack.*

To infer victims' activities via L2 cache, attackers rely on process scheduling (or context switching). Specifically, they evict all cache lines from some cache sets of L2 cache (prime operation), enable executions of victims for a while on the same core, and then reschedule the attack program and re-access the corresponding memory lines (probe operation). Although malicious enclaves running at the user-level cannot actively schedule processes (for improving attack accuracy), they can still obtain patterns from long-term data collection via offline analysis removing noises [19, 33].

Key observation: It is inevitable to probe the whole or a large portion of the L2 cache by Prime+Probing in such attacks. The reasons are as follows: (i) the attacker does not know the virtual-physical address mapping of the victim to precisely pinpoint the targeted cache regions; and (ii) the attacker needs to filter out noise generated by external activities during the side channel analysis.

Vector ❷. *L3 cache Prime+Probe attack.*

To steal sensitive information (like RSA keys) via L3 cache, attackers can execute simultaneously with victims, since L3 cache is shared among all cores of a chip. Meanwhile, to achieve fine-resolution inference via L3 cache of several MiB, instead of directly priming and probing the whole L3 cache, attackers scan L3 cache and monitor one cache set at a time, until pinpointing the cache set associated with victims' activities [33]. Recently, Schwarz et al. [44] have demonstrated a malicious enclave can extract RSA keys from victim enclaves via L3 cache based side channel attacks, regardless of whether victims run inside other enclaves or are isolated in Docker containers.

Key observation: Scanning (by Prime+Probing) the whole or a large portion of the L3 cache in a set-by-set manner is inevitable in such attacks, as the attacker needs to identify the cache sets of interest and to eliminate noise due to background processes.

Vector ❸. *Flush+Reload/Flush attack.* Malicious enclaves monitor specific memory lines belonging to its host application, as it may share the same physical pages with other processes. The root cause is that existing OSes and hypervisors implement content-aware page sharing mechanisms (e.g., using shared libraries or page deduplication) to reduce memory footprint. Basically, a malicious enclave first flushes the target memory line of its host application from the cache, waits for a time interval, then reloads the memory line and measures the access time. A fast access means that the victim has accessed the target memory line. Instead of re-accessing the memory, Flush+Flush attacks re-flush the memory line and observe the timing differences to infer victims' memory access patterns. Similar user-level attacks have been launched to infer secrets of other programs residing in the same platform via Flush+Reload [55] or Flush+Flush [26].

Key observation: During memory flushes or reloads, some executable pages in the host memory (which contain specific memory lines in shared pages) are accessed frequently by malicious enclaves.

Vector ❹. *Cache-DRAM attack.*

This attack exploits the shared row buffer of the same DRAM bank to detect contention on DRAM banks and thereby to infer the memory access patterns of victims [51]. Specifically, a malicious enclave allocates two memory blocks (denoted as p and q) that are mapped to the same DRAM bank but different rows. p and the sensitive data (denoted as v) are on the same row. Then, the enclave accesses the memory block q, waits a while for victim operations, accesses the block p and measures the access time. A faster access indicates that the victim may have accessed the data v. To guarantee that the victim fetches v from DRAM, the attacker will prime some cache sets of L3 cache that would store v. This attack can only target other enclaves as the rows are not shared between PRM and non-PRM regions.

Key observation: Several L3 cache sets are repeatedly and frequently primed by a malicious enclave during the attack to guarantee subsequent DRAM accesses.

3.2 Attacks via Host Virtual Memory

An enclave and its host application reside in the same virtual memory space. The enclave is allowed to access the entire user-space memory space of the host application. It enables enclaves to use the following two attack vectors to attack the host application.

Vector ❺. *Memory disclosure attack.*

A malicious enclave conducts abnormal memory read operations, e.g., to find usable code gadgets outside the enclave. Generally, it needs to scan the host memory. This is because the enclave does not know what the outside code is and where it is. Even when the outside code is developed by the same developer (due to ASLR).

Key observation: During enclave execution, an enclave performs abnormal memory read, e.g., scanning executable memory pages.

Vector ❻. *Host control-flow manipulation.*

With identified code gadgets, the enclave can divert the execution to them by deliberately writing host memory. SGX-ROP [43] practically demonstrated that malicious enclaves can construct control flow hijacking attacks against the memory safety of the host application. Also, it can modify other values saved in stack, e.g., *OCall_Table* passed by `ECall` functions and can look up addresses of `OCalls` according to index. By constructing a fake *OCall_Table*,

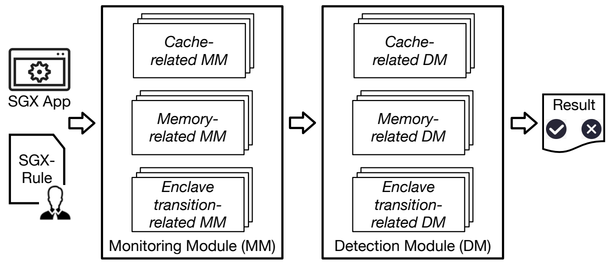

Figure 1: The overview of the SGX-Bouncer framework

the host application would execute arbitrary code once an `OCall` is called by the enclave.
Key observation: The key unusual behavior is that some contents of the host memory (e.g., saved RBPs and RIPs of stack frames) are manipulated by hostile enclaves.

3.3 Attacks via Enclave-mode Transition

SGX provides new instructions (i.e., `EENTER`, `ERESUME` and `EEXIT`) for CPUs to enter and exit the enclave mode. When entering the enclave and exiting the enclave, the register values are set to achieve enclave-mode transitions.

Vector ❼. `EEXIT` *abuse.*

A malicious enclave may manipulate register values related to control flows when executing `EEXIT`. For example, it can exit to a specific yet non-standard point (where the execution of the host application should start) by setting the RBX register [8].
Key observation: Enclave malware abuse `EEXIT` with abnormally crafted register values.

4 SGX-BOUNCER: DETECTING ENCLAVE MALWARE

SGX-Bouncer detects malicious enclaves by monitoring their run-time behaviors observed from the outside of enclaves. As shown in Fig. 1, it consists of two types of modules. The monitoring modules (MMs) are responsible for inspecting enclave runtime behaviors. There are five monitoring modules: two cache related MMs, two memory related MMs, and one enclave-transition related MM (see Table 1). The detection modules (DMs) are designed to detect each attack vector by analyzing data collected from the corresponding MM and uncovering abnormal behaviors incurred by the attack vector. As a framework, SGX-Bouncer allows new monitoring/detection modules to be added and specific modules to be enabled. It also allows users to provide detection rules (dubbed *SGX-Rules*) to capture new features of attack vectors.

4.1 Monitoring Modules

There are five monitoring modules: L2Cache-MM and L3Cache-MM that monitor L2 and L3 cache access behaviors of an enclave, respectively, MemoryR-MM and MemoryW-MM that capture enclave read/write to specific area of host memory, respectively, and EnclaveT-MM that monitors enclave entries/exits and triggers the other four MMs.

4.1.1 L2Cache-MM. To monitor L2 cache access behaviors of enclaves, we devise a side channel based method, called *Probe+Check*.

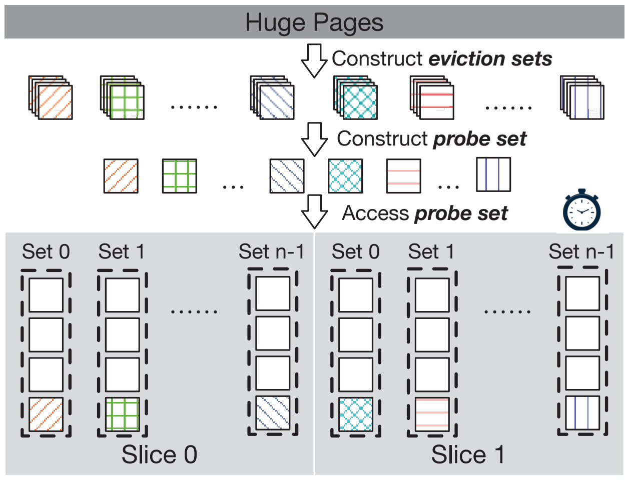

Figure 2: The construction of probe set for L3 cache.

The insight behind this design is that, attackers need to constantly fill the cache sets to perform L2 cache Prime+Probe attacks. Hence, we construct a *probe set* where each memory line is exactly indexed to one cache set of L2/L3 caches. We monitor cache access behaviors by consecutively probing these memory lines and checking cache hits/misses. A cache miss illustrates that the corresponding cache set has been filled and possibly abused by attackers.

To construct a *probe set*, we use 2MB huge pages that use the lower 21 bits of the virtual address to express the page offset. In this way, we can directly get cache set indices of L2 caches without the need to figure out the virtual-physical address mapping, as it is indexed by physical address bits 6-15. Meanwhile, we use several huge pages rather than only one page (4 pages in our experiment). Because, though one huge page is large enough to construct a probe set for L2 cache (256KB in our experiment), using multiple huge pages guarantees that memory lines in the probe set would not be prefetched and prevents introduced noises.

To monitor L2 cache, we access all memory lines in the probe set before enclave entries (i.e., the *probe* step) and check whether these data are evicted after enclave exits (i.e., the *check* step). Particularly, since L2 cache is small and can be filled by normal operations of the enclave, we have to probe/check the set frequently, so as to reduce false positives. Thus, we also check these memory lines when AEXs occur, which are frequently triggered by I/O events or timer interrupts. Note that we use PMC to examine cache hits/misses, as measuring timing difference would be less reliable. Finally, we obtain an *L2 cache monitoring array* where each element (denoted as N_{l2m}) represents the number of L2 cache misses in one check step. We demonstrate in Section 5 that our fine-grained L2 cache monitoring incurs low false positive even for memory-intensive applications (e.g., neural network training).

4.1.2 L3Cache-MM. The *Probe+Check* is also used to monitor L3 cache access behaviors. Unlike L2 cache which is exclusively used by each core, L3 cache is shared among all cores of one CPU processor and divided into per-core slices. That is, a cache set in L3 cache is determined by both a slice id and a set index. The set index is retrieved from the virtual address of the huge page (i.e., bits $6-16$)

directly, but the slice id is hard to be obtained as it is calculated by CPU using an undocumented hash function over physical addresses. To construct a probe set, we first follow the methodology proposed by Liu et al. [33], i.e., we create eviction sets that exactly fill up all cache sets, which is applied to create eviction sets for different CPU models. Then, we pick one arbitrary memory line from each eviction set to constitute the probe set, as shown in Fig. 2. The probe set will not be changed once it is constructed.

To monitor the entire L3 cache, we continuously access all memory lines in the probe set and record cache hits/misses by analyzing whether the access time is larger than a pre-defined threshold. A slow access means this memory line has been evicted from the cache set and the cache set is possibly filled by the malicious enclave. For efficiency, we can access memory lines in parallel using multi-threads and each thread only monitors part of the L3 cache sets. The outcome of the monitoring process is an *L3 cache monitoring matrix* where each row represents cache usage of all cache sets in one *check* step. Each element is 1 or 0, where 1 means the corresponding cache set is filled and 0 is the opposite.

4.1.3 MemoryR-MM. We monitor host memory read of enclaves via side channels. Our key observation is that both flushing and accessing memory lines update *Accessed* [50, 51] flags of corresponding pages' Page Table Entries (PTEs). Thus, we inspect host memory read behaviors by checking PTEs' *Accessed* flags of specific pages of host memory. In SGX-Bouncer, we enable users to define which type of pages enclaves are allowed to access, as described in Sec. 4.2.1. Here, we focus on host executable pages by default, which should not be accessed during enclave executions.

Before enclave entries, we clear *Accessed* flags of specific host pages (e.g., host executable pages) and remove related records in the TLB. Once the enclave exits, we check whether these flags are updated. To get fine-grained memory access behaviors (e.g., attacker's access frequency), we repeatedly perform three operations before enclave exits: 1) checking *Accessed* flags of these pages, 2) setting these flags as 0, 3) flushing corresponding records in the TLB. Finally, we build a memory monitoring matrix where each column denotes one page and each row records whether the corresponding host memory page has been read in one scanning step.

To analyze multi-threaded SGX programs, we need to perform additional operations. When there are two threads, one running inside the enclave and the other running outside, simply starting memory read monitor from enclave entries suffers from high false positive, since the outside thread also updates *Accessed* flags of host executable pages (which is not a malicious behavior) . To address this problem, we can limit multi-threaded SGX applications running on a single CPU core and only allow one thread working at any moment by restricting the OS scheduling mechanism.

4.1.4 MemoryW-MM. In order to monitor if enclaves perform write operations on specific contents of host memory, we copy the specified data to *tracer* before enclave entries and then check the data consistency after enclave exits. Compared with the method of monitoring Dirty bit of PTE, it can obtain the malicious payload for analysis. Similar to MemoryR-MM, users are allowed to define which type of pages enclaves can write to. In the paper, we focus on checking if enclaves manipulate the control flow of the host application.

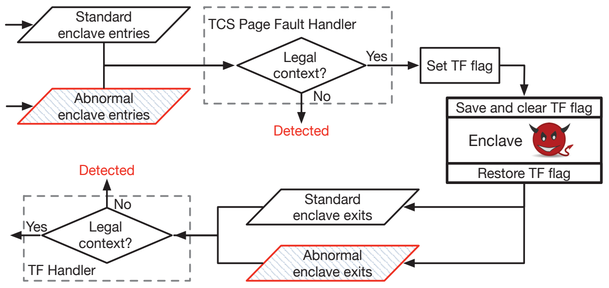

Figure 3: The workflow of EnclaveT-MM.

We monitor the RBP/RIP pairs in the stack. Any modification to them during the execution of the enclave is considered as malicious [43]. Specifically, we use a chain of RBP and RIP pairs in stack frames to represent the control flow, where RBP is the stack pointer and RIP is the instruction pointer of the calling instruction in the previous stack frame. Specifically, we get the value of current RBP register. It is used to construct RBP and RIP chain of the whole control flow, since the value of RIP is stored in the address adjacent to the RBP and the frame pointer of the previous frame can be traced by accessing the content pointed by the current RBP. Finally, we produce a pair of RBP-RIP chains for each pair of enclave entries and exits.

4.1.5 EnclaveT-MM. EnclaveT-MM monitors enclave entry/exit behaviors and their contexts. Here, we directly monitor all occurrences of `EENTER`/`EEXIT` instructions, since enclave entries and exits can be implemented via attacker-crafted enclave transition functions, which will be missed by hooking the standard uRTS of the SGX development framework.

Unfortunately, it is difficult to monitor `EENTER`/`EEXIT` instructions. Since `EENTER` and `EEXIT` are user-level instructions and their execution will not incur the CPU privilege switching from ring3 to ring0, we cannot trace them in the kernel. Besides, if we track them by single-stepping every instruction of host applications, it incurs significant performance slowdown and increases detection time due to frequent exceptions. To address this issue, we design a page-fault based method to efficiently identify enclave entries. The key insight is that the Thread Control Structure (TCS) pages will be indirectly and inevitably accessed by `EENTER`. To capture all enclave exits, we leverage a key feature of SGX where enclave logic cannot be debugged. Thus, after handling page-fault exceptions, we single-step SGX applications and utilize debugging exceptions to capture all enclave exits. We describe details of how to monitor all enclave entries and exits below. We omit asynchronous enclave entries and exits (i.e., AEX and ERESUME), since their correctness is guaranteed by the benign OS.

Enclave Entry Monitoring. We capture enclave entry behaviors by monitoring whether enclave TCS pages are accessed. There are two key observations: 1) The TCS is the first memory page that would be accessed when executing `EENTER`. 2) One TCS identifying one executing threads in the enclave is stored as a 4KB enclave page. TCS pages are added to the enclave memory via the privileged instruction `EADD` during the enclave initialization. According to these insights, we monitor enclave entries in a side channel manner.

Table 2: Summary of rule options

Keyword	Parameter format	Description
C-PPL2	0/1, [Threshold] (Default: 1, [0.05])	Detect malicious L2 cache access behaviors (vector ❶) using the threshold.
C-PPL3	0/1, [Threshold] (Default: 1, [0.04])	Detect malicious L3 cache access behaviors (vectors ❷ and ❹) using the threshold.
M-FRF	0/1, [Threshold] (Default: 1, [1])	Detect Flush+Reload/Flush attacks (vector ❸) using declared threshold.
M-Read	0/1, [(ExePage/StackP/(Content), R/NR /W/NW)] (Default: 1, [ExePage, R])	Detect abnormal memory read (vector ❺). It allows users to define pages with a certain page property (or specific area) and limit their access permissions. *ExePage* is executable pages, *StackP* is saved pointers (i.e., RBPs and RIPs) of the stack, *(Content)* specifies pages containing the contents. Access permissions are Readable (R), Non-Readable (NR), Writable (W), and Non-Writable (NW).
M-Write	0/1, [(ExePage/StackP/(Content), R/W/NW)] (Default: 1, [StackP, NW])	Detect abnormal memory write (vector ❻). Parameter definitions are described above.
E-EntryEx	0/1, [(Entry/Exit)##(RegName), (=/!=/</>), (Immediate/Entry/Exit)## (RegName)] (Default: 1)	Detect `EEXIT` abuse and enable users to input conditional statements (vector ❼). We detect whether enclave exit addresses are those specified in the standard uRTS (e.g., Intel SGX SDK's) by default.

More precisely, we first get addresses of all TCS pages when the SGX driver adds them into the enclave. Once the enclave has been initialized (i.e., via `EINIT`), we clear the *Present* flags of PTEs of the TCS pages and remove the corresponding TLB entries. Meanwhile, we hook the page-fault handler. If a page-fault exception on the TCS page arises, an enclave entry occurs. We also record the context of the exception (the values of registers) which are used to identify abnormal enclave entries later.

Enclave Exit Monitoring. We capture events of enclave exits by leveraging the SGX anti-debugging feature in the release mode. Concretely, when switching to the enclave mode, the Trap Flag (TF) of the FLAGS register is cleared. Before that, the TF value would be saved in a register called `CR_SAVE_TF` (which is invisible to enclave software [8]). When the CPU exits the enclave mode, the TF flag can be restored and CPU will raise a TF exception when fetching the next instruction. Thus, we set the debugging flags before each enclave entry and capture TF exceptions to discover enclave exits.

In detail, we launch one process called *tracer* to track the execution of the SGX application, e.g., using the syscall `ptrace`. We set the TF flag before the CPU switch to enclave mode so as to capture enclave exits. However, the flag cannot be set when handling the TCS page fault exception (for capturing enclave entries). Therefore, we produce a software interrupt via the `INT 3` instruction (placed in the front of `EENTER` in the standard uRTS). It enables the tracer to set the TF flag for the SGX application. After executing the enclave code which looks like a giant instruction, the host application is stopped and the enclave exit is captured.

4.2 Detection Modules

We first define a rule description language that allows users to customize SGX-Rules in SGX-Bouncer. Then, we present how the detection modules detect malicious enclaves according to the rules. As shown in Table 1, there are 6 detection modules: L2Cache-DM uncovering L2 cache Prime+Probe attacks (attack vector ❶); L3Cache-DM detecting L3 cache Prime+Probe attacks and cache-DRAM attacks (attack vectors ❷ and ❹); FRF-DM detecting Flush+Reload/Flush attacks (attack vector ❸); MemoryR-DM identifying host memory disclosure attacks (attack vector ❺); MemoryW-DM uncovering host control-flow manipulation (attack vector ❻); and EnclaveT-DM defending against `EEXIT` abuse (attack vector ❼).

4.2.1 Rule Description Language. One SGX-Rule consists of two parts: a rule action and rule options. The rule actions indicate SGX-Bouncer how to act when it discovers that the SGX application's behaviors match the rule criteria specified by rule options. A rule option contains one keyword and multiple arguments which are separated by a colon. Multiple rule options in one *SGX-Rule*, separated by &, form a logical `AND` statement. To detect malicious enclaves, the client can submit a rule file containing multiple SGX-Rules. These rules, separated by semicolons, form a logical `OR` statement. In addition, the client can specify detection time in the first line of the rule file using the keyword *DetectionTime.*

Rule action. SGX-Bouncer supports two rule actions, *Alert* and *Terminate. Alert* means if the rule criteria is matched, SGX-Bouncer produces an alert with a log specifying which item in rule options is violated (without terminating the application). *Terminate* denotes that SGX-Bouncer immediately terminates the SGX application and send a log to the client.

Rule option. Table 2 lists 6 rule options and name them in the format "*C/M/E-Name*", where *C/M/E* denotes if cache/memory/enclave-mode-transition based monitoring module's data is used in the detection and *Name* states the detection content. In addition, if there are many arguments for one rule option, they should be enclosed in square brackets and separated by commas. `##` is a concatenation operator in rule options. Consider *I-EntryEx-S* as an example. "*E-EntryEx*: [EntryRDX, !=, ExitRDX], [ExitRBX, !=, 0x410000]" finds abnormal enclave exits, if RDX values of enclave entries do not equal to corresponding RDX values of enclave exits and RBX values of enclave exits are not 0x410000.

4.2.2 L2Cache-DM. To detect L2 cache Prime+Probe attack (vector ❶), we analyze the L2 cache monitoring array of the size of X generated by L2Cache-MM. We use a counter N_{l2c} to denote the number of abnormal L2 cache access behaviors in X check steps. That is, if the element N_{l2m} of the array is greater than a pre-defined threshold T_{l2m}, we add N_{l2c} by one. Finally, if the ratio of unusual

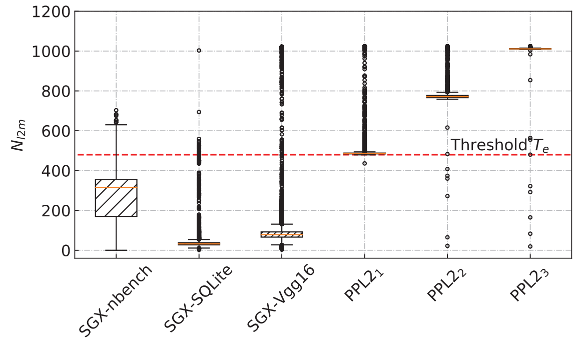

Figure 4: The number of L2 cache misses N_{l2m} in one check step for benign and malicious PPL2 applications.

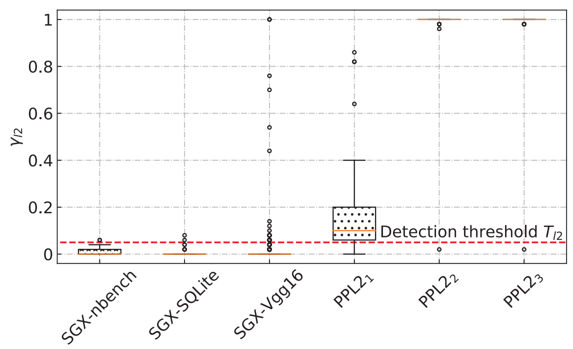

Figure 5: The ratio of unusual elements γ_{l2} for benign and malicious PPL2 applications.

elements γ_{l2} (i.e., $\gamma_{l2} = \frac{N_{l2c}}{X}$) is larger than a pre-defined threshold T_{l2}, it is considered as launching an L2 cache Prime+Probe attack.

4.2.3 L3Cache-DM. To detect L3 cache Prime+Probe attacks (attack vector ❷) and Cache-DRAM attack (attack vector ❹), we use a window of the size of $m \times n$ to scan the L3 cache monitoring matrix produced by L3Cache-MM. Note that m denotes the number of rows and n is the total number of cache sets. We also maintain a counter (denoted as N_w) to record the number of unusual windows. Concretely, in each window, if the number of cache misses corresponding to some cache sets is larger than $\gamma_{l3m} \times m$ (where γ_{l3m} denotes the ratio of permitted cache misses in the window), we regard the window as an unusual window and add 1 to counter. Finally, if the ratio of unusual windows γ_{l3} (i.e., $\gamma_{l3} = \frac{N_w m}{M}$) exceeds the detection threshold T_{l3}, we identify there exist abnormal L3 cache access behaviors caused by Prime+Probes. We validate the effectiveness and efficiency of this detection method in Section 5.

4.2.4 FRF-DM. To detect Flush+Reload/Flush attacks (attack vector ❸), we analyze the memory monitoring matrix generated by MemoryR-MM checking executable pages of the host memory. If some elements in the matrix are 1s, it means that some executable pages are accessed by the enclave. We also count the number of accesses to each executable page. Since Flush+Reload/Flush attacks exhibit frequent accesses to several pages, we use the maximum number of page accesses among all pages in the detection. If the maximum value exceeds a threshold T_{frf} (which can be defined by users), it indicates a Flush+Reload/Flush attack occurs. We demonstrate the detection performance later.

4.2.5 MemoryR-DM. To uncover the memory disclosure attacks (attack vector ❺), we analyze the matrix generated by MemoryR-MM. In SGX-Bouncer, we check whether executable pages of the host memory are scanned by an enclave by default. Thus, we calculate the ratio of the number of executable pages that are accessed by the enclave among the total number of executable pages. The attacks are identified by a threshold γ_{memr}.

4.2.6 MemoryW-DM. To detect host control-flow manipulations (attack vector ❻), we verify that specific contents of the host memory (which can be defined by users as Table 2 shows) are same before enclave entries and after enclave exits. In the default setting, we inspect whether RBPs and RIPs of stack frames are manipulated by an enclave (vector ❻). Any inconsistency says the enclave has hijacked the control flow of the host application.

4.2.7 EnclaveT-DM. To detect abused `EEXIT` (vector ❼), we compare contexts of enclave exits with those specified in the rule. The standard exit points are different in various SGX development frameworks. Take the Intel SGX SDK as an example, it has only one exit point following `EENTER`, and it will not change after *libsgx_urts.so* is loaded into memory. Here, we forbid malicious enclaves to modify *libsgx_urts.so* after we load it.

5 EVALUATION

We implement a prototype of SGX-Bouncer in C/C++. It is built atop Intel SGX Platform Software (PSW) and Intel SGX driver. Our testbed is built on Dell OptiPlex 3060 equipped with 8GB RAM and six-core, 3.00Ghz Intel i5-8500 Coffee Lake CPU. The processor has 256KB L2 cache with 1024 cache sets, as well as 9MB L3 cache with 6 slices and 12288 cache sets. The OS of the testbed is Ubuntu 16.04 (Linux Kernel 4.8.0) LTS and SGX applications are developed with Intel SGX SDK (commit 34421657). Also, we disable the C-States, Intel SpeedStep and TurboBoost to make the timer stable for accurate identification of cache hits and misses.

5.1 Detection Effectiveness

We evaluate detection effectiveness of six DMs as follows.

5.1.1 L2Cache-DM. We evaluate the detection effectiveness of L2Cache-DM against L2 cache Prime+Probe attack (attack vector ❶, dubbed PPL2). We first determine parameters in L2 cache misbehavior detection, which are the array size X, the threshold T_{l2m} for determining abnormal elements in the array, and the threshold T_{l2} for identifying L2 cache based attacks. With well-determined parameters, we consider the cache misbehavior detection as a binary classification problem and estimate the true positive rate and false positive rate. To evaluate its detection effectiveness, we select three benign and three malicious applications, i.e., *SGX-nbench*[1], SGX-SQLite[2], SGX-Vgg16 (an convolutional neural network model inside enclave), and *PPL2*$_1$, *PPL2*$_2$, *PPL2*$_3$. The latter three are L2 cache Prime+Probe attacks that scan the half, three quarters, and entire of L2 cache, respectively.

We set the size of L2 cache monitoring array X as 50. We collect 600 arrays for each application, each runs for about one minute.

[1]https://github.com/utds3lab/sgx-nbench.git
[2]https://github.com/yerzhan7/SGX_SQLite.git

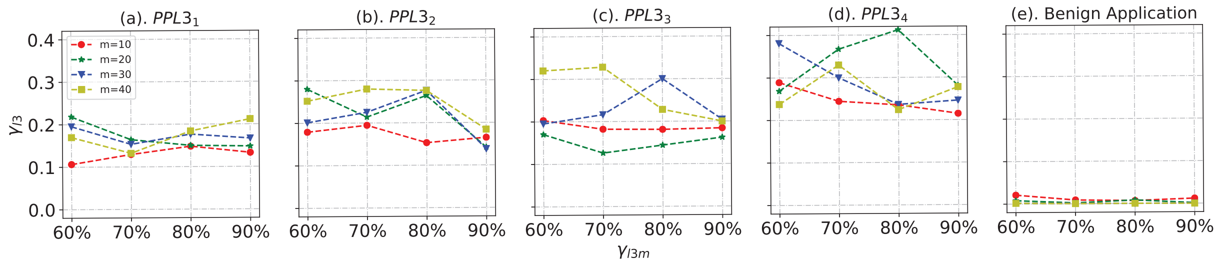

Figure 6: The ratio of unusual windows γ_w under different PPL3 applications and different window settings.

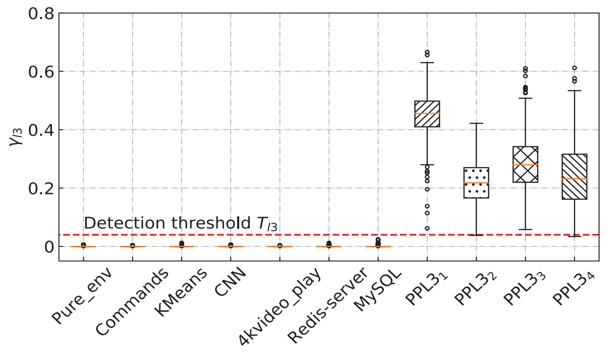

Figure 7: The ratio of unusual windows γ_{l3} for benign and malicious (PPL3) applications.

Figure 4 shows the distribution of N_{l2m} for different SGX applications. We can see that N_{l2m} for benign applications and attacks are differentiated. According to the results, we set the threshold T_{l2m} as 480. Then, we evaluate the ratio of unusual elements γ_{l2} for different SGX applications. From Fig. 5, we can see that when T_{l2} is 0.05, the true positive rate is much greater than 99.9% and false positive rate is less than 0.01%. The number of false positives/false negatives also depends on the number of real-world SGX programs, which is expected to increase along with the wide deployment of SGX.

5.1.2 L3Cache-DM. We evaluate the detection effectiveness of L3Cache-DM against L3 cache Prime+Probe attacks (dubbed PPL3, attack vector ❷) and Cache-DRAM attacks (attack vector ❹). Similar to the detection of L2 cache misbehaviors, we require to determine parameters for detecting L3 cache Prime+Probe attacks. They are the scanning window size ($m \times n$), the ratio of permitted cache misses γ_{l3m}, the size of one cache monitoring matrix ($M \times n$), and the threshold T_{l3} for detection, where n means the number of cache sets (i.e., 12288). Then, we estimate whether L3Cache-DM can accurately detect PPL3 under different attack settings without incurring high false positive. In particular, we compare its effectiveness with one existing PMC-based detection method (HexPADS [38]), using public L3 cache Prime+Probe attack applications [2].

We clarify two attack parameters: 1) *attack_slot* is the CPU cycles taken for one attack round; 2) *attack_num* is how many attack rounds that would be taken for one cache set. Here, we implement four attacks with different attack_slots and attack_nums in the evaluation: *PPL3*$_1$ with *attack_slot* = 5000 and *attack_num* = 1000; *PPL3*$_2$ with *attack_slot* = 7500 and *attack_num* = 1000; *PPL3*$_3$ with *attack_slot* = 5000 and *attack_num* = 3000; *PPL3*$_4$ with *attack_slot*

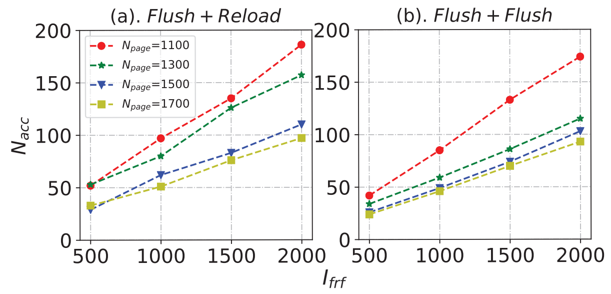

Figure 8: The number of memory accesses captured by MemoryR-MM during 2500 Flush+Reload/Flush attacks.

= 7500 and *attack_num* = 3000. Also, we select various benign applications some with large memory footprints for comparison. They are *Pure_env* only running L3Cache-MM on the testbed; *Commands* executing 40 common Linux commands and utilities [36]; *KMeans* performing K-Means algorithm with 2.4G memory; *CNN* running ResNet-50 with 1.2G memory; *4kvideo_play* playing 478.4MB 2160p video with Totem v3.18.1; *Redis-server* fetching random values with redis-benchmark; *MySQL* running random SQL statements with *mysqlslap*. Note that different from the evaluation of PPL2 detection that uses benign SGX applications, we use benign applications without SGX to assess detection effectiveness of L3Cache-DM. This is because that L3 cache is shared among all cores on one chip and our detection design can be applied to any Prime+Probe attacks regardless of whether they are mounted inside the enclave or not.

We study the effect of the window size m and the ratio γ_c. For simplicity, we set the size of the L3 cache monitoring matrix as $10,000 \times n$ ($M = 10,000$), which is large enough to cover one PPL3 attack even if it scans all L3 cache sets. Figure 6 shows the ratio of unusual windows γ_{l3} for different SGX applications under different window settings. All results are averaged over 200 measurements. Note that γ_{l3} for the benign application is the maximum value among all results of benign applications. We can find that all attacks can be obliviously differentiated from benign applications under different window sizes m. We set m as 20 and γ_{l3m} as 90%.

We explore the true positive and false positive of L3Cache-DM. We run each program for 45 minutes and produce about 900 γ_{l3} per application, all shown in Fig. 7. With the detection threshold T_{l3} of 0.04, we can differentiate benign and malicious applications. The true positive rate is near 100% and false positive rate is 0. In addition, we also test 109 benign programs generated by Graphene-SGX [13]. The results show that the ratio of unusual windows for these benign applications is consistently low as the 7 benign apps in Fig. 7.

Table 3: Runtime overhead of SGX-Bouncer for SGX-Vgg16

Runtime overhead	L2Cache-MM	L3Cache-MM	MemoryR-MM	MemoryW-MM	EnclaveT-MM	SGX-Bouncer
Forward propagation	×3.27	×1.01	×1.17	×1	×1.03	×3.89
Back propagation	×2.66	×1.02	×1.22	×1	×1.01	×3.26

Since our detection method is not limited to discover Prime+Probe attacks inside the enclave, we compare it with one of the existing PMC-based detection methods called HexPADS [38], using the above benign applications and public L3 cache-based attacks (*L3-capture*, *L3-capturecount* and *L3-scan*) in Mastik suite [2]. We find that even if we run these attacks only once, SGX-Bouncer can effectively discover these attacks but not HexPADS. The ratios of unusual windows γ_{l3} of them are 0.082, 0.092 and 0.052, which all exceed the threshold T_{l3} (i.e., 0.04). Meanwhile, HexPADS misidentifies *KMeans* and *CNN* as attacks. Therefore, L3Cache-DM outperforms HexPADS. The root cause is that, it can obtain cache access behaviors of each cache set, rather than cache misses of the entire L3 cache.

5.1.3 FRF-DM. We evaluate the detection effectiveness of FRF-DM against Flush+Reload/Flush attacks (dubbed FRF, attack vector ❸). We implement these attacks targeting an RSA-based cryptographic algorithm using the *square-and-multiply* algorithm in GnuPG (v1.4.12). We locate the addresses of the algorithm at runtime and pass the location to the enclave. The enclave launches Flush+Reload/Flush attacks by repeatedly flushing memory lines of the Square function, idle looping, and reloading/flushing them.

We measure how many memory accesses to some specific pages (denoted as N_{acc}) are captured by MemoryR-MM, using two parameters, i.e., the time interval (CPU cycles) between Flush and Reload/Flush operations I_{frf} and the number of executable pages of host memory as N_{page}. We perform 2500 Flush+Reload and Flush+Flush attacks, respectively.

The results show only one page is accessed and its access number captured by MemoryR-MM increases linearly as the attack interval becomes larger (see Fig. 8). In an extreme case where FRF has finished before MemoryR-MM first checks the corresponding pages, we can only detect one memory access. In SGX-Bouncer, we set the detection threshold T_{frf} as 1 to tolerate strong FRF attacks that succeed with a small number of Flush+Reload/Flush operations. In addition, we also assess the false positive rate with three benign SGX applications used above, and no host memory access is captured.

5.1.4 MemoryR-DM. We evaluate the detection effectiveness of MemoryR-DM against memory disclosure attacks (attack vector ❺). To detect abnormal memory read, we measure how many accessed executable pages are captured by MemoryR-MM. Here, we build an enclave that scans the host memory. We set N_{page} as 1700 and enable the enclave to linearly scans 300 executable pages of the host memory. The experimental result shows that MemoryR-MM captures all 300 accessed executable pages when the enclave exits. We set the default threshold γ_{memr} as 10%, which means if 10% executable pages are accessed by an enclave, it is scanning the host memory, e.g., for the purpose of finding usable code gadgets.

5.1.5 MemoryW-DM. We evaluate the detection effectiveness of MemoryW-DM against host control-flow manipulation (attack vector ❻). We implement an enclave that modifies the stack pointers and injects a fake stack frame as described in [43]. When running the enclave, MemoryW-DM can discover the RBP-RIP chain since the enclave entry is not the same as that after the enclave exit.

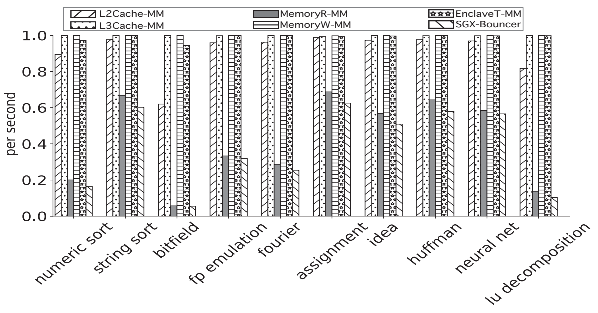

Figure 9: Runtime overhead for micro-benchmark suite

5.1.6 EnclaveT-DM. We evaluate the detection effectiveness of EnclaveT-DM against `EEXIT` abuse (attack vector ❼). We construct an SGX application that performs `EENTER` and `EEXIT` without relying on implementations of the uRTS. These behaviors are all captured by the page-fault handler and TF handler. We inspect the register values that represent the contexts of enclave entry/exit events, and the entry and exit positions different from the standard positions of uRTS in Intel SGX SDK are uncovered.

5.2 Efficiency Evaluation

We assess the efficiency of SGX-Bouncer monitoring modules on benchmark suites. The micro-benchmark and macro-benchmark suite are *SGX-nbench* and *SGX-Vgg16* respectively.

Micro-benchmark suite. SGX-nbench consists of 10 algorithms that call `ECall`/`OCall`s from 1,000 to 300,000 times respectively. The normalized overhead is shown in Fig. 9. We can see that the overhead of MemoryR-MM dominates the overall overhead and L2Cache-MM also leads to slowdown. The reason is that MemoryR-MM has to clear the *A* bit before entering the enclave and check it before exiting the enclave, thus delaying the execution of the program. L2Cache-MM requires to finish accessing all memory lines in the probe sets, once the enclave exits via `EEXIT` and AEX.

Macro-benchmark suite. Different from SGX-nbench, SGX-Vgg16 produces a large memory footprint (1.2G) with sparse `ECall`s and `OCall`s (about one call per 3 seconds). The runtime overhead of SGX-Bouncer is shown in Table 3. Of all MMs, L2Cache-MM introduces the most overhead, since a large number of AEXs occur and trigger L2Cache-MM accordingly.

Note that runtime overhead of SGX-Bouncer is positively related to the number of L2 cache sets, the number of executable pages of the host memory, and the number of enclave exits and AEXs. In addition, we discuss in more details how the slowdown of execution may be used by the malware to evade detection in Sec. 7.

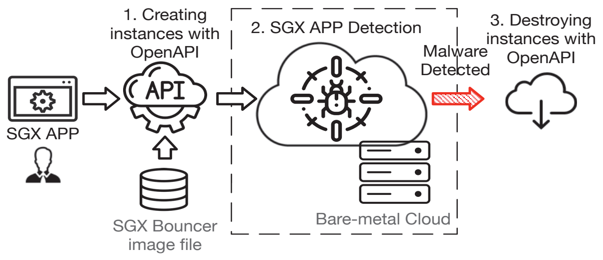

Figure 10: Real-world deployment on an Iaas cloud.

6 REAL-WORLD DEPLOYMENT

We present the real-world deployment of SGX-Bouncer in a commercial IaaS Cloud and an example configuration rule that can be used to detect real SGX malware in the cloud.

6.1 Deployment in Real Cloud

We have deployed SGX-Bouncer in one of the largest public IaaS clouds as a security service (see Figure 10), which is similar to Google-Bouncer that detects malicious Android apps on the official Android app store (i.e., *Google Play*). Particularly, instead of running it in VMs, we run it on a bare-metal cloud, since the patches to support SGX virtualization for both KVM and Xen are not mature for production use [7]. Note the cloud provides extensible multi-tenant bare-metal cloud services and hosts up to 16 bare-metal instances in a single physical server, which allows the cloud to serve tens of thousands of users each day.

To flexibly start up an SGX-Bouncer, we craft an SGX-Bouncer image by installing the relevant SGX run-times and SGX-Bouncer program. Meanwhile, we also install automatic artifact generation systems, e.g., UBER [25], in the platform, which makes the state and configuration of the platform realistic in the view of SGX programs and prevents enclave malware from evading detection. When an enclave detection request arrives, the service launches an SGX-Bouncer instance created from the SGX-Bouncer image, which takes as input an SGX program and a detection rule file provided by the user. Similar to Google-Bouncer, SGX-Bouncer examines a submitted SGX program for five minutes (the default value) and then outputs the detection result.

6.2 Case Study

We give an example of using a rule file to detect a real SGX malware called SGX-ROP [43], which constructs a control flow hijacking attack. The rule file consists of three SGX-Rules, as shown in Fig. 11.

1: **Alert** *M-Read*: 1;
2: **Terminate** *M-Write*: 1;
3: **Terminate** *E-EntryEx*: 1;

Figure 11: A sample file of detection rules.

A malicious enclave launches SGX-ROP [43] to hijack the host's control flow by finding code gadgets from the host memory and exiting enclave to them. Since the application available online[3] cannot run in our testbed (due to disabled TSX), we reimplement it without using TSX. Moreover, to help the enclave code to find the required gadgets without crashing the application, the code outside the enclave passes the application memory layout to enclave. Note such changes do not contribute to the detection of the malware.

```
1: mov aep, %rcx
2: mov abnormal_exit_point, %rbx
3: mov $4, %rax
4: enclu
```

Figure 12: Manipulating the target execution address outside the enclave via RBX when invoking `EEXIT`.

To divert the execution to the crafted ROP chain, we develop two types of implementations: 1) *SGX-ROP*$_1$ injects a fake stack frame and modifies the RBP/RIP values in the stack as described in [43]; 2) *SGX-ROP*$_2$ directly executes the code gadget by specifying the exit address (i.e., RBX) via `EEXIT`, as shown in Fig. 12.

Detection results. From the detection logs generated by SGX-Bouncer, both *SGX-ROP*$_1$ and *SGX-ROP*$_2$ are terminated. SGX-Bouncer terminated *SGX-ROP*$_1$ after the enclave exit, as the pair of RBP-RIP chains does not match. SGX-Bouncer terminated *SGX-ROP*$_2$, because the enclave exit address is not the standard exit location as implemented by the uRTS of Intel SGX SDK.

7 MALWARE EVASION TECHNIQUES AND COUNTERMEASURES

Enclave malware that is aware of the SGX-Bouncer may alter its runtime behavior to evade detection. There are two methods that may be leveraged by enclave malware to determine the presence of SGX-Bouncer: First, the enclave malware can heuristically analyze whether the state and configuration of a platform are realistic, like observing an abnormally small number of process [3] or inspecting the platform's wear-and-tear characteristics [35]. Second, the enclave malware may exploit the timing difference between running on a normal platform and on our detection platform, since SGX-Bouncer leads to performance slowdown for SGX applications, for example, via cache contention, frequent interruption, PTE manipulation, and code instrumentation.

Countermeasures to traditional malware evasion techniques have been studied in the literature [35], which can be integrated into SGX-Bouncer to address the first evasion method. To address the second, SGX-Bouncer can leverage the lack of trusted clocks inside enclaves to obscure their true execution time.

Specifically, existing methods of getting a trusted clock inside enclave so far are either unreliable or coarse-grained. First, `RDTSC/RDTSCP` instructions are not allowed in the enclave mode in SGXv1. Even if they are supported in SGXv2, the return values can be modified by the untrusted software. Second, software timer created by a counting thread [44] can be manipulated by changing the CPU frequency or cleansing cache [27], which is difficult to achieve high accuracy. Third, the Converged Security and Management Engine (CSME) of Intel SGX SDK only provides trusted time service in the order of seconds [22], which is too coarse-grained. Moreover, such a service is unreliable, as requests to the service and responses from it can be arbitrarily delayed and dropped by the untrusted software. Similarly, remote timing sources (e.g., wall clock of another server) are not reliable as the network communication is subject to manipulation, too. We anticipate the method with which the

[3]https://github.com/IAIK/sgxrop

enclave malware may detect the presence of SGX-Bouncer is by itself a research topic that we hope our study can inspire.

8 RELATED WORK

8.1 Enclave Malware

SGX has been exploited by attackers to conceal malware inside enclaves [34, 41, 43, 44, 51]. For example, enclave malware have been developed to break the restriction of enclave and hijack control flows of its host application [43], steal sensitive data from a co-located process [41, 44, 51], or restrict access to vital information (i.e., SGX-based ransomware) [34].

In particular, SGX allows attackers to generate stealthy malware that can evade state-of-the-art anti-virus software. For example, enclave malware can defeat static code analysis [29, 31], since enclave code may stay encrypted before being loaded into the enclave. Moreover, memory isolation and the suppressed debugging feature inside the enclave make traditional memory forensics and debugger tools [23] invalid. Although enclave malware infecting the host system via issuing syscalls can be detected by existing dynamic syscall-based detection tools like [21], in this paper we still identify seven serious attack vectors of enclave malware that can evade detection. Thus, it is critically important to detect such enclave malware since they more stealthy and incur more significant damages, particularly when the enclave cannot be controlled by Intel and the platform owners.

8.2 SGX Defense and Analysis Tools

Prio arts inspect properties of SGX applications developed with Intel SGX SDK. However, all of these tools only focus on part of properties and cannot systematically detect malicious enclaves. For example, SGX-Perf [53] analyzes the performance of SGX programs by capturing `ECall`/`OCall`, yet does not define malicious enclave interface behaviors. SGX-Step [49] enables single-stepping enclaves by raising interrupt to CPU with Advanced Programmable Interrupt Controller (APIC) timer, yet lacks deep analysis for malware detection. SGX-Fun [16] only shows how to extract enclave metadata from SGX binaries. EnGarde [37] introduces a static analysis framework in the tRTS, to examine both static enclave code and the dynamically loaded code. However, it only checks some simple violations like whether the program links old-version libraries and does not handle sophisticated attacks like side channel attacks. SGX-Jail [54] intends to defeat SGX-based third-party libraries that may subvert control flow of host applications. It isolates host and enclave memory by executing the suspicious enclave as a separate sandbox process. The sandbox process is confined by Seccomp and communicates with host application via Inter-Process Communication (IPC). However, it requires developers to make big changes to eliminate pointers marked by *user_check* which is very common in some SGX programs and cannot defeat side channel attacks.

8.3 Defenses against Side channel Attacks

Traditional cache side channel detection can be classified into two categories, i.e., code analysis and performance monitoring counters based detection. The first type of methods analyze static code with reverse engineering tools, e.g., investigating the number of specific instructions such as CLFLUSH and RDTSC [29]. Also, dynamic binary instrumentation can be leveraged to detect attacks [40]. However, these methods cannot be applied to detect encrypted enclave code or dynamically loaded code. The second type of methods detect attacks by collecting performance events and adopting signature based or anomaly based [56] methods. Unfortunately, production enclaves set the Anti Side Channel Interference (ASCI) bit to suppress the performance monitoring activities [8], and thus PMCs cannot monitor cache misses/hits events of malicious enclaves. In this paper, SGX-Bouncer detects malicious cache behaviors by capturing side channel information directly which achieves high detection accuracy.

9 CONCLUSION

In this paper, we perform the first systematic study on malicious enclaves and summarize seven concrete attack vectors by analyzing all interaction interfaces between enclaves and the outside software. We develop SGX-Bouncer, a detection framework that inspects side channel information and utilizes SGX-specific features to detect malicious enclaves exploiting the attack vectors above. We prototype SGX-Bouncer and demonstrate its effectiveness.

ACKNOWLEDGMENTS

The work was supported in part by the National Key R&D Program of China under Grant 2018YFB1800304, NSFC under Grant 61572278, NSF under Grant CNS-1815650, CNS-1750809 and CNS-1718084, and ARL under Grant W56KGU-20-C-0008. Qi Li is the corresponding author.

REFERENCES

[1] 2012. Android and Security. https://googlemobile.blogspot.com/2012/02/android-and-security.html.
[2] 2016. Mastik: A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik/.
[3] 2016. Ursnif Banking Trojan Campaign Ups the Ante with New Sandbox Evasion Techniques. https://www.proofpoint.com/us/threat-insight/post/ursnif-banking-trojan-campaign-sandbox-evasion-techniques.
[4] 2017. Intel and NeuLion Bring Secure, 4K UHD Sports Streaming to Computers. https://newsroom.intel.com/news/intel-neulion-bring-secure-4k-uhd-sports-streaming-computers/.
[5] 2017. Ledger Bitcoin Wallet Partners With Tech Giant Intel. https://news.bitcoin.com/ledger-wallet-partners-tech-giant-intel/.
[6] 2017. Password manager Dashlane now integrates with Intel SGX for hardware security. https://www.digitaltrends.com/computing/dashlane-intel-sgx/.
[7] 2017. SGX Virtualization. https://01.org/zh/node/4486?langredirect=1.
[8] 2018. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
[9] 2018. Q3 2018 Intel Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html.
[10] 2019. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.
[11] 2019. ECS Bare Metal Instance. https://www.alibabacloud.com/product/ebm.
[12] 2019. TSX broken again. https://erik.science/intel/tsx/2019/05/26/new-tsx-bugs.html.
[13] 2020. Graphene-SGX. https://github.com/oscarlab/graphene/.
[14] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *Proc. of CHES*. 110–124.
[15] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In *Proc. of ACM CCS Workshop*. 185–199.
[16] J Aumasson and Luis Merino. 2016. SGX Secure Enclaves in Practice: Security and Crypto Review. *Black Hat* (2016).
[17] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *Proc. of USENIX Security*. 1213–1227.
[18] Marcus Brandenburger and Christian Cachin. 2018. Challenges for Combining Smart Contracts with Trusted Computing. In *Proc. of Workshop on System Software*

for Trusted Execution. ACM.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies.*

[20] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. 2017. Secure cloud micro services using Intel SGX. In *Proc. of Springer DAIS.* 177–191.

[21] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A Quantitative Study of Accuracy in System Call-based Malware Detection. In *Proc. of ACM ISSTA.* 122–132.

[22] Shanwei Cen and Bo Zhang. 2017. Trusted Time and Monotonic Counters with Intel® Software Guard Extensions Platform Services. https://software.intel.com /sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf.

[23] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *Proc. of IEEE S&P.* 161–175.

[24] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Proc. of USENIX Security.* 51–67.

[25] Pengbin Feng, Jianhua Sun, Songsong Liu, and Kun Sun. 2019. UBER: Combating Sandbox Evasion via User Behavior Emulators. In *Proc. of Springer ICICS.*

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *Proc. of Springer DIMVA.* 279–299.

[27] Wei Huang and Yueqiang Cheng. 2019. Aion Attacks: Exposing SGX Software Timers. *Blue Hat* (2019).

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proc. of IEEE S&P.*

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. MASCAT: Stopping Microarchitectural Attacks Before Execution. *IACR Cryptology ePrint Archive* 2016 (2016), 1196.

[30] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proc. of ACM SysTEX.* 5.

[31] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host.. In *Proc. of USENIX Security*, Vol. 4. 351–366.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. of USENIX Security.* 16–18.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Proc. of IEEE S&P.* IEEE, 605–622.

[34] M Marschalek. 2018. The Wolf In SGX Clothing. *Blue Hat* (2018).

[35] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *Proc. of IEEE S&P.*

[36] R. Natarajan. 2010. 50 most frequently used unix/linux commands (with examples). https://www.thegeekstuff.com/2010/11/50-linux-commands/?utm_%2520s ource=feedburner.

[37] Hai Nguyen and Vinod Ganapathy. 2017. EnGarde: Mutually-Trusted Inspection of SGX Enclaves. In *Proc. of IEEE ICDCS.*

[38] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *Proc. of Springer ESSoS.* 138–154.

[39] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *Proc. of USENIX Security.* 565–581.

[40] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. 2018. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime-Probe. In *Proc. of IEEE/ACM ICCAD.* 1–8.

[41] Michael Schwarz and Moritz Lipp. 2018. When Good Turns Evil: Using Intel SGX to Stealthily Steal Bitcoins. *Black Hat Asia* (2018).

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proc. of ACM CCS.*

[43] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *Proc. of Springer DIMVA.* 177–196.

[44] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proc. of Springer DIMVA.* 3–24.

[45] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190.

[46] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

[47] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proc. of USENIX ATC.*

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. of USENIX Security.* 991–1008.

[49] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proc. of ACM SysTEX.*

[50] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proc. of USENIX Security.* 1041–1056.

[51] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proc. of ACM CCS.* 2421–2434.

[52] Filip Wecherowski. 2009. A real smm rootkit: Reversing and hooking bios smi handlers. *Phrack Magazine* 13, 66 (2009).

[53] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proc. of ACM MIDDLEWARE.* 201–213.

[54] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *Proc. of RAID.* 353–366.

[55] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proc. of USENIX Security.* 719–732.

[56] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *Proc. of RAID.* 118–140.

[57] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2017. Dos attacks on your memory in cloud. In *Proc. of ACM AsiaCCS.* 253–265.