ZeroShield: Transparently Mitigating Code Page Sharing Attacks With Zero-Cost Stand-By

Mingyu Wang, Fangyu Zheng, Jingqiang Lin, *Senior Member, IEEE*, Fangjie Jiang, and Yuan Ma

***Abstract*— Numerous cache side-channel attack techniques enable attackers to execute a cross-VM cache side-channel attack through the sharing of code pages with the targeted victim. Nonetheless, most prior defense solutions fall short of efficiency and ease of deployment, thus restricting their practicality for real-world implementation. This paper introduces ZeroShield, an adaptive and transparent approach implemented at the hypervisor layer, designed to counteract the code page sharing attack, a subset of cache side-channel attacks, occurring within a single virtual machine (VM) or spanning across multiple VMs. By thoroughly scrutinizing the "by-products" resulting from a code page sharing attack, we meticulously track the attacker's access to security-sensitive code pages. This is achieved through harnessing hardware virtualization features, such as the Intel extended page table, in conjunction with the CR3 register. Utilizing this information, ZeroShield continuously monitors security-sensitive code pages, adeptly navigating complex OS and hypervisor behaviors. The architecture of ZeroShield exhibits an attack-aware design, enabling it to deploy protection measures on demand. Consequently, the system theoretically experiences negligible overhead in the absence of attackers. Empirical evidence confirms the effectiveness of ZeroShield in thwarting code page sharing attacks. It achieves this without imposing any performance penalties in the absence of attackers, and with a minimal overhead of less than 3.8% when attackers are active. Significantly, ZeroShield boasts a cost-free standby state and necessitates no adjustments to upper applications, guest OS, or hardware configurations. This attribute positions ZeroShield as an optimal default solution in real-world cloud environments to effectively counter code page sharing attacks.**

***Index Terms*— Cache side-channel attack, virtualization, EPT, mitigation.**

Manuscript received 11 September 2023; revised 7 April 2024; accepted 8 July 2024. Date of publication 29 July 2024; date of current version 7 August 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB3105802, in part by the National Natural Science Foundation of China under Grant 61902392, and in part by CCF-AFSG Research Fund under Award CCF-AFSG RF20230206. The associate editor coordinating the review of this article and approving it for publication was Dr. Mohammad Ashiqur Rahman. *(Corresponding author: Fangyu Zheng.)*

Mingyu Wang, Fangyu Zheng, and Fangjie Jiang are with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: wangmignyu@163.com; zhengfangyu@ucas.ac.cn; jiangfangjie2016@gmail.com).

Jingqiang Lin is with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei 230026, China (e-mail: linjq@ustc.edu.cn).

Yuan Ma is with the Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: mayuan@iie.ac.cn).

Digital Object Identifier 10.1109/TIFS.2024.3435062

I. Introduction

THE proliferation of virtualization technology gives rise to the large-scale application of cloud computing technology. More and more organizations prefer renting virtual machines (VMs) and deploying their services in VMs rather than purchasing their own physical servers [1]. To isolate resources among tenants, the hypervisor provides various types of logical isolation mechanisms between VMs to deny unauthorized inter-tenant access. However, attackers are able to bypass the inter-tenant isolation mechanisms through shared resources to have unauthorized access to the private information of other tenants in unexpected manners [2], e.g., the well-known cache side-channel attacks.

Cloud providers often employ memory sharing techniques to enhance memory utilization and accommodate a larger number of tenants. This opens a window for the attacker to initiate cache side-channel attacks against shared code libraries to steal tenants' confidential information. In addition to targeting specific cryptographic algorithm implementations, such as scalar multiplication in ECDSA [3], [4], [5] and square-and-multiply exponentiation algorithm in RSA [6], [7], cache side-channel attacks utilizing shared memory have been applied in various practical scenarios [8], [9], [10], [11], [12].

There is a diverse range of libraries designed to handle various hardware modules and software events on devices such as Android devices, personal PCs, etc. These libraries encompass the code required by the system to process events for GPS, Bluetooth, camera, web, and PDF viewers, etc [9]. Consequently, attackers can utilize the shared libraries to monitor the victim's invocation and execution of relevant functions within these libraries, enabling them to track corresponding device events and capture the victim's sensitive information [8], [9], [10].

A. Code Page Sharing Attacks and Their Significant Impact

Evidently, due to the widespread usage of shared libraries, cache side-channel attacks relying on shared code pose a serious threat to user information security. We refer to cache side-channel attacks that target shared code as *code page sharing attacks*, or CPSA for short. In this paper, we primarily focus on this type of attack. Among various cache side-channel attack techniques that have been proposed, attack methods relying on memory sharing [6], [7], [8], [13], [14], [15], [16] can all be employed to initiate CPSA, such as the most representative FLUSH+RELOAD [6].

In practice, CPSA has been used to attack various shared libraries to steal user's secret information, resulting in user information leakage. For instance, the Cache Template Attack [8] and ARMageddon [9] targeted shared libraries on Android devices, aiming to capture user actions like swipe gestures, taps, and text inputs. Dragonblood [11], [12] targeted the code of Dragonfly handshake to extract sensitive information and recover target passwords. Zhang et al. [10] inferred the number of distinct items in a user's e-commerce shopping cart by monitoring how many times a Zend opcode handler was invoked. Moreover, they also targeted the functions used by pseudorandom number generators within programming language runtime to reset passwords and then gain control over the victim's account. Kim et al. [17] utilized FLUSH+RELOAD to attack the MUA (Mail User Agent) program, extracting the RSA private keys when recipients read a single encrypted email and recovering the victim's email contents. Shahverdi et al. [18] presented a FLUSH+RELOAD attack on SQLite that obtains approximate (or noisy) volumes of range queries made to a private database and then reconstructed nearly the exact database through several algorithms. Yan et al. [19] used FLUSH+RELOAD to monitor the execution of the GEMM (Generalized Matrix Multiply) function in OpenBLAS, efficiently obtaining the DNN (Deep Neural Network) architectures.

Therefore, due to the widespread utilization of shared libraries, CPSA can not only be employed to exploit specific cryptographic algorithm implementations or jeopardize internet security protocols but also be extended to disclose different categories of sensitive user data across various situations.

B. Limitations of the Existing Countermeasures

In response to such attacks, numerous protection approaches have been proposed, which we can categorize into two main groups: randomization-based, and isolation-based. Randomization-based solutions disrupt the correlation between memory and cache [20], [21], [22] or introduce additional "noise" [23], [24], [25], [26], [27], [28] to mitigate threats. Isolation-based solutions establish an additional isolation layer to restrict an attacker's access to sensitive data [29], [30], [31], [32], [33], [34].

However, a majority of previous work faces challenges in terms of efficiency, transparency and practicality.

- In reality, the system spends the majority of its time in normal operation, with the attacker's activity being relatively scarce [35]. The previously proposed solutions lack adaptability, as they introduce overhead even when the system is operating normally, i.e., without any attacks occurring. This is particularly true for software-based solutions, where overhead might escalate even further [22] in the presence of attacks.
- The hardware-layer implementations often lack compatibility with contemporary platforms and can be inconvenient to implement [20], [21], [29], [30]. Additionally, some solutions require user participation during operation. This leads to an added burden on users and compromises transparency [23], [24], [28], [31], [33], [34]. Moreover, specific approaches introduce extensive modifications to the system [22], [32]. This addition of significant code could lead to increased complexity in the system and potentially introduce additional vulnerabilities.

C. Contributions and Paper Organization

To address these challenges, this paper introduces ZeroShield (ZS), a solution that offers adaptability, compactness and transparency to effectively counter code page sharing attacks. Particularly, ZeroShield excels when the protection mechanism is in a standby state. The core design philosophy of this approach stems from several key observations obtained through systematic analysis of code page sharing attacks within a single virtual machine (VM) or across multiple VMs:

- Within a CPSA scenario, the victim normally executes sensitive code pages, while the attacker stealthily reads these pages.
- In a typical virtualized environment, mechanisms at the hardware or OS level, such as Intel's Enhanced Page Table (EPT) technology or the Linux Memory Management Unit (MMU), designed to enhance memory efficiency and uphold page mapping, offer an effective and efficient means to detect potential malicious "readers".

Driven by these insights, ZeroShield is realized as a hypervisor-level implementation encompassing two pivotal components: 1) monitor: this core component tracks and identifies potential threats within the system; 2) preloader: this component is responsible for taking proactive actions as needed to enhance system security.

The cornerstone of the monitor is to trace the execution and concurrent reading of target code pages to identify potential attackers. When the monitor detects an instance where a process is executing a target code page and another process is simultaneously reading it, it flags this situation as a potential attack by the "reader". Subsequently, the preloader is alerted to activate the protective mechanism. Specifically, processes that execute target code pages are referred to as "X-processes" (potential victims), while processes that read these pages are referred to as "R-processes" (potential attackers).

In summary, this work makes several significant contributions:

- Firstly, through a comprehensive investigation of code page sharing attacks, we propose an innovative approach, called ZeroShield, to effectively counter these attacks using the well-supported Intel EPT mechanism. Notably, our approach achieves this without necessitating any modifications to the guest OS or applications. This application of the Intel EPT mechanism to mitigate cache side-channel attacks is groundbreaking.
- Secondly, we have developed a prototype of ZeroShield, which is built upon the Linux kernel 4.15 and incorporates the EPT mechanism within the hypervisor layer. ZeroShield boasts seamless integration, requiring no alterations to either the guest OS or applications. Moreover, it offers on-demand activation and meticulous handling

of diverse scenarios. These attributes collectively result in high efficiency and sustained effectiveness in safeguarding targeted code pages.
- Lastly, we conducted a rigorous evaluation of ZeroShield's security and performance through an extensive series of experiments. The empirical results conclusively illustrate that in the absence of potential attackers within the system, ZeroShield imposes negligible overhead. Conversely, when potential threats are present, the system-wide impact remains below 3.8% (as measured by SPEC CPU 2006 scores), with the overhead incurred on individual victims not exceeding 17%.

The advantages of ZeroShield mainly come from an efficient and transparent monitor, providing the following key properties: 1) **adaptive invocation**: ZeroShield operates almost "silently" in the absence of potential attackers, with only modest overhead incurred when such attackers are detected. This adaptability remains effective even amidst complex operating system scenarios or user behaviors, such as when the processes of the attacker and victim are suspended or terminated. 2) **compact implementation**: ZeroShield achieves intricate tasks with a minimal introduction of "trusted code", by utilizing existing mechanisms and hardware components within the operating system, which enhances both security and overall system performance. 3) **transparent deployment**: ZeroShield necessitates no modifications to the guest OS or applications, relying solely on widely supported features such as Intel EPT.

Featuring zero cost when standing by and no modification to the upper applications/guest OS, ZeroShield can act as a default configuration for the real-world cloud environment to mitigate cache side-channel attacks.

The rest of this paper is organized as follows. The background is presented in Section II. In Sections III and IV, we introduce the design and implementation of the solution. The security analysis and performance evaluation are described in Section V. We then compare with related works and make some discussions in Section VI. Finally, we conclude the paper in Section VII.

II. Background

This section provides background on cache side-channel attacks and Intel hardware-assisted virtualization technology.

A. Cache Side-Channel Attacks and CPSA

Cache side-channel attacks monitor the usage of the cache. Specifically, when a victim's cache access patterns correlate with its sensitive information, attackers can deduce the victim's sensitive data by observing cache hit and cache miss events occurring on target cache lines. Currently, there are three categories of cache side-channel attack techniques: EVICT+TIME, FLUSH+RELOAD and PRIME+PROBE (along with their variants). FLUSH+RELOAD and PRIME+PROBE are more prevail due to their high resolution [7]. For further exploration of EVICT+TIME, we delve into detailed discussions in Section VI-B. These two techniques consist of three phases: initializing (where the attacker prepares the cache to a specific state), idling (where the attacker waits for a period while the victim runs), and measuring (where the attacker examines changes in the cache state by measuring delays in certain micro-architectural events, allowing them to infer the victim's sensitive information).

PRIME+PROBE is a more general attack that doesn't rely on memory sharing. However, the PRIME phase takes more time compared to FLUSH, which increases the likelihood that the monitored address might not be evicted, as the monitored address might be accessed between the PROBE and PRIME phases' starting points [8]. Additionally, the time spent in the PROBE phase is also longer than RELOAD, which results in a lower resolution for PRIME+PROBE in terms of distinguishing cache behavior.

FLUSH+RELOAD [6] is a typical technique to launch a cache side-channel attack that relies on memory sharing. It requires the attacker and the victim to share code pages. The attack usually employs flush and reload (memory accessing) to ascertain if the specific cache lines have been accessed. Specifically, the attacker first clears the specific cache line by flushing the target memory line, and then waits for the victim's execution. Finally, the attacker accesses the target memory line and measures latency. According to the length of access latency, the attacker determines whether the victim accessed the target memory line or not. Compared to FLUSH+RELOAD, FLUSH+FLUSH [13] replaces memory accessing with flushing in the last step. EVICT+RELOAD [8] evicts the target cache line by accessing the eviction set instead of using `clflush`. INVALIDATE+TRANSFER [14] takes advantage of the directory protocol of high-efficiency CPU interconnects. It also uses flushing and memory accessing to clear the target cache line and measure the latency, respectively. RELOAD+REFRESH [7] first initializes the target memory line as a candidate for eviction in the cache set by accessing the target memory line and eviction set. Then, the attacker awaits the victim's access. Finally, the attacker accesses the eviction set and then the memory line. According to the latency of accessing the target memory line, the attacker determines whether the victim accessed the target memory line. PREFETCH+REFRESH [15] is a variant of RELOAD+REFRESH, it replaces memory accessing with prefetching in the initialization. PREFETCH+RELOAD and PREFETCH+PREFETCH [16] utilizes prefetching to initialize the initial state of the target cache line. After a certain waiting period, they respectively measure the latency of memory accessing and prefetching. Based on these latency measurements, they deduce whether the victim has accessed the target cache line or not.

CPSA is a subset of cache side-channel attacks that initiates attacks by monitoring the victim's access patterns to shared sensitive code pages. FLUSH+RELOAD and its variants we mentioned before can be used to launch CPSA. In the majority of practical attacks employing cache side-channel techniques, FLUSH+RELOAD stands as the foremost choice and is extensively employed [8], [9], [10], [11], [12].

B. Intel Hardware-Assisted Virtualization

Hardware-assisted virtualization is a technology that adds support for virtualization in hardware such as CPU and I/O devices, thus the system can implement virtualization easily and efficiently. Intel Virtualization Technology (Intel VT) is the general term for hardware virtualization technology on Intel platforms, providing virtualization support for CPU, memory, and I/O devices, such as Intel Virtualization Technology for x86 (Intel VT-x), and the Extended Page Table mechanism (EPT).

Intel VT introduces VMX (Virtual Machine Extension) root operation and VMX non-root operation. In general, a virtual machine manager (VMM) runs in VMX root operation, and guest software runs in VMX non-root operation [36]. In VMX root operation, the processor behavior is identical to that of a traditional IA32 architecture CPU. When the processor is in VMX non-root operation, privileged instructions and other specified instructions could cause a VM exit. Then the CPU switches to the VMX root operation, and the virtualization layer completes the function of the instruction. VMX non-root operation and VMX transitions are controlled by a data structure called virtual machine control structure (VMCS). When a logical processor is in VMX operation, it uses a VMCS which contains six logical groups: Guest-state area, Host-state area, VM-execution control fields, VM-exit control fields, VM-entry control fields, and VM-exit information fields.

In the VM-execution control fields, there are two 32-bit vectors that control the processing of synchronous events. By using specific instructions to modify the corresponding fields, we can specify whether the event can cause a VM exit, which is then handled by the VMM. For example, when the CR3-load exiting flag is set, the operation of updating the CR3 register during context switching triggers a VM exit.

Intel also provides EPT technology to assist memory virtualization. It directly uses hardware to convert Guest Physical Address (GPA) to Host Physical Address (HPA) and synchronize with the host's page table in time. The privilege flags (i.e., read, write, and execute) for the associated GPA are contained in the EPT paging-structure entries. If the memory access violates the privileges maintained in the EPT entries, it will cause a VM exit and the reason code is *EPT violation*. The VMM can obtain the exit reason and the corresponding GPA, vCPU from the VM-exit information fields in the VMCS and use the specific handler for processing.

III. Design

In this section, we begin the description of ZeroShield with the threat model and design goals, then illustrate the design rationale and the overall architecture respectively.

A. Threat Model

Targeting the code page sharing attack, we design ZeroShield under the following threat model:

- The attack is launched in the virtualized environment, where VMs do not trust each other, but the host OS, processor, and hypervisor are trusted. The host OS enables the KSM (Kernel Same-page Merging) service.
- The attacker can utilize CPSA to extract information about the victim.
- The security-sensitive code pages that need to be protected are determined in advance. This can be actually achieved by the Linux distribution package maintainers "automatically" in real-world applications. Specifically, as cryptographic libraries are readily available, the determination of target code can be attributed to the task of Linux distribution package maintainers. It can be completed offline by the Linux distribution package maintainers and requires no additional action from users. Furthermore, our attention is directed towards the vulnerable code within the cryptographic libraries, particularly focusing on specific implementations like the ECDSA algorithm.

More specifically, the following two attack scenarios are considered:

Intra-VM: In this scenario, the victim and the attacker are running in the same VM. The victim invokes the cryptographic service, e.g., running OpenSSL and computing ECDSA signatures continuously, while the attacker `mmaps` the shared library into its own virtual address space and puts probes in the location that it wants to monitor. Then the attacker will launch a code page sharing attack.

Inter-VMs: In this scenario, the victim runs in a VM, and the attacker runs in the other VM, while the two VMs are co-resident. The attacker `mmaps` a copy of the shared library, and then shares with the victim by page deduplication mechanism. Thus, the attacker can launch a code page sharing attack across VMs.

B. Design Goals

In accordance with this threat model, we have devised a defense mechanism tailored for virtualized environments to counter the code page sharing attack. Our design adheres to the following objectives:

- **Adaptiveness:** The approach is capable of dynamically adjusting its defense behavior according to changes within the system. During an attack on the victim, it operates in a busy mode, preventing the attacker from accessing sensitive information by activating the protection mechanism. In all other cases, it functions in a lazy mode, incurring minimal overhead.
- **Compactness:** Instead of completely redesigning a new mechanism, we prefer existing hardware features in the operating system to formulate our mitigation strategy, enhancing compatibility, efficiency, and introducing fewer (trusted) codes to the operating system.
- **Transparency:** Our mitigation can remain hidden from both the guest OS and applications and does not necessitate any modifications to the guest OS, upper-level applications, or shared libraries. This mitigation operates entirely transparently to users, requiring no additional actions from them during runtime. In contrast, alternative solutions like CATalyst [31], Cloak [24], and Ghost Thread [23] necessitate manual requests for protection or application modifications by users.

C. Design Rationale

In this section, we will analyze the behavior of the attacker and identify clear distinctions between attacker and victim behavior. Subsequently, to minimize modifications to the original system, we will analyze the available hardware and OS resources. Finally, we will present the overall architecture and basic workflow of ZeroShield.

1) Analysis of the Attackers' Behavior: The design of ZeroShield starts with the behavior pattern analysis of the code page sharing attackers and tries to exploit the influenced system status to identify potential attackers.

By analyzing and summarizing the various CPSA techniques mentioned in Section II, we have proposed a common behavior pattern for CPSA. CPSA generally consists of the following three steps:

- **INITIALIZE:** The attacker provides a virtual address to be monitored, and initializes the corresponding cache line by flushing [6], [13], [14], evicting [8], [14], accessing [7], or prefetching [15], [16] the monitored memory line from cache.
- **IDLE:** The attacker waits for a specified detection interval, during which the victim may access the monitored memory line.
- **MEASURE:** The attacker measures the latency of a specific micro-architectural event (such as memory accessing [6], [7], [8], [14], [15], [16], flushing [13], prefetching [16]). According to the latency, the attacker determines whether the victim has accessed the sensitive memory line during the IDLE phase, enabling the attacker to infer the victim's sensitive information.

Based on our investigation of existing CPSA techniques, during the INITIALIZE phase, except for the eviction-based method [8], all other attack approaches involve accessing the shared sensitive page (target page). During the MEASURE phase, the attacker revisits the target page and determines whether the victim accessed the target page during the IDLE phase based on the timing delay of the revisit.

Therefore, we conclude that in a code page sharing attack, the attacker reads the target code page at least once. While, normal processes execute the target code page normally. It's worth noting that the solution we ultimately implemented does not rely on the rigid INITIALIZE-IDLE-MEASURE pattern to identify an attacker. This is because different attack methods employ varying techniques during the INITIALIZE and MEASURE phases, which would make the defense mechanisms heavy and inefficient. Given that benign processes in the system rarely read the shared code pages, we use the reading of code pages as a characteristic for identifying attackers. This paper will provide further evidence for this discovery by successfully detecting the CPSA attackers using this characteristic.

Thus we assume an X-process is a potential victim and an R-process is a potential attacker. When we identify a read attempt to the target code pages, we consider that an attacker is identified.

2) Analysis of Available Hardware/OS Resources: Building upon this pattern, the upcoming issue is to ascertain at which level within the system and which available hardware/OS resources to implement the detection mechanism with minimal overhead while achieving the aforementioned design goals.

The foundation of our work lies in the identification of abnormal "readers" behaviors. The impracticality of monitoring every page reading instance due to its performance impact led us to an elaborate approach. We only identify the abnormal "readers" behaviors that occur on the target code pages by utilizing page hashing that we mentioned in our prior work [37]. This method can be seamlessly integrated within the host OS, involving periodic checks privileges of the target code page in corresponding page table entries (PTEs).

However, it's important to note that tracking the privileges in PTEs allows us to detect the presence of an attacker but falls short in determining whether the attacker has ceased their attack. The reason behind this limitation is that PTE does not offer an efficient means of promptly acquiring information about process terminations. Consequently, the protection mechanism would remain active, incurring unnecessary costs even when the attacker is no longer a threat. Therefore, relying solely on PTE within the host OS for our purpose is infeasible.

As a result, we redirect our focus to the hypervisor level. In typical virtualized environments, regular VMs employ EPT to enhance address translation efficiency, translating GPA to HPA while maintaining control over page privileges. A VM exit is triggered if a process in a VM violates the privileges specified by the EPT entry when accessing a code page. This VM exit presents an opportunity for our work.

Nevertheless, like the PTE approach, relying solely on EPT has its limitations. It cannot achieve adaptive termination, as it cannot discern whether the attacker has temporarily suspended their attack or exited completely. Fortunately, the hypervisor level provides an advantage over the host OS: it can capture updates of the CR3 register value through VM exit events. By configuring the relevant fields, we can trigger a VM exit when a process is suspended or terminated. This allows us to promptly detect changes in system status and decide whether it's necessary to deactivate the protection mechanism.

To summarize, the core of our work involves meticulous management of the privileges of target code pages to ensure that **they cannot be read and executed simultaneously** unless the protection mechanism is active. It's important to acknowledge that the primary design may not always be effective due to complex OS scenarios. For instance, page merging can lead to updates of EPT entries, potentially disrupting our ability to accurately track page privileges. We will delve into these issues in the implementation section.

D. Architecture of ZeroShield

ZeroShield consists of two components: a monitor and a preloader. As shown in Figure 1, once the attacker, which is in an inter-VMs scenario or intra-VM scenario, launches an attack, the monitor could be aware of it in time and notify the preloader to start a preloading thread. When the process switches, the monitor could notify the preloader to stop the preloading thread, to reduce the overhead.

ZeroShield relies on a pre-computed hash value of the target code page for monitoring potential victims and attackers.

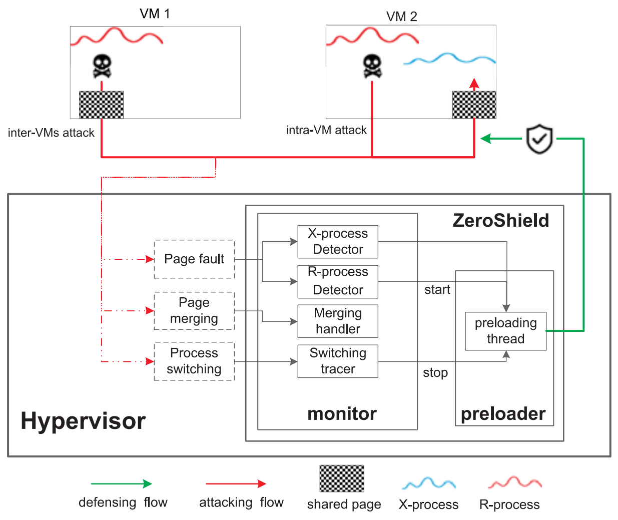


Fig. 1. Architecture of ZeroShield.

So we need to calculate the hash value of each target code page in advance as a reference to identify it. For the shared library, we generate the hash value for the content in the target code page as described in a previous work called TF-BIV [37].

TF-BIV is a transparent and fine-grained binary integrity verification scheme that does not necessitate any modifications or software/driver installations within the VM. Cloud service providers can employ TF-BIV to monitor the integrity of processes running in guest VMs, effectively identifying malicious or compromised processes. To continuously monitor the integrity of target binary files, TF-BIV performs integrity verification immediately when the target binary file is executed. To detect whether a page of the target binary file has been tampered with by malicious processes, TF-BIV calculates a hash value based on the content of the page every time the page is executed, and verifies whether the computed hash value matches the initial hash value of the page. If a page has been tampered with by a malicious process, the computed hash value must inevitably differ from the page's initial hash value. Inspired by TF-BIV, our approach also utilizes content-based hash calculation of pages to identify target code pages.

The hash value is computed offline and subsequently written into the Linux kernel. Since it does not require real-time calculation using an extra thread, it does not consume additional threads. Target code pages are specified in advance. When they need to be updated, we can re-calculate the hash value of the new target code pages.

ZeroShield is running inside the hypervisor, monitoring the system status and discovering potential threats. To achieve the aforementioned design goals, ZeroShield monitors four critical system events (EPT non-readable (NR) exiting, EPT non-executable (NX) exiting, CR3-load exiting, and MMU notifier) transparently and invokes protection mechanisms when necessary.

E. Basic Workflow of ZeroShield

Illustrated in Figure 1, the functioning of ZeroShield is depicted, elucidating its operational mechanisms and workflow.

a) Identifying the X-process: By configuring the privileges of the target code pages as non-executable (NX) in corresponding EPT entries, ZeroShield is able to capture an EPT NX exit event, to identify an X-process (a potential victim). Then ZeroShield is able to keep track of R-processes that share the target code pages with the X-process in the following steps.

b) Identifying the R-process: For the target code pages that are being executed, ZeroShield sets the non-readable bit (NR-bit) of the corresponding EPT entries. Any read attempts will trigger an EPT NR exiting, which we can capture to identify an R-process. In this way, the protection mechanism can be invoked on demand, i.e., when we identify an X-process and an R-process at the same time.

c) Creating a preloading thread: When there is an X-process and an R-process in the system simultaneously, a preloading thread is created to load the target code page continuously, to prevent the attacker from stealing the sensitive data.

d) Tracing process switching: As well as an on-demand invocation feature, ZeroShield also monitors the update to the CR3 register to achieve on-demand suspension. ZeroShield captures the suspension of the R-process and the X-process with the help of CR3-load exiting, to stop the preloading thread in time to reduce overheads.

e) Handling page merging: A dedicated treatment is also required in the inter-VMs scenarios for page merging. ZeroShield utilizes MMU notifier mechanism in Linux to capture any page merging related to the target code page, for discovering new VMs that load the target code page.

The above steps illustrate the basic workflow of ZeroShield. In the practical implementation of ZeroShield, extra treatments are also taken to ensure that ZeroShield works in complicated scenarios, e.g. "reader" may be present before an X-process is identified.

IV. Implementation

We implemented ZeroShield as a kernel extension for Linux kernel 4.15. In detail, by setting the `CR3-load exiting` bit in `VMCS` and configuring EPT entries to enforce that the target code page cannot be read and executed simultaneously (unless the preloading thread is running), we are able to capture the necessary VM exits and handle them in the corresponding handlers. Then, the preloader is notified to create or stop a preloading thread on demand. By leveraging the MMU notifier, we capture the page merging of the target code page and reconfigure the corresponding EPT entries.

In this section, we will further introduce how to identify the X-process/R-process, deal with page merging, and handle the process switching.

A. Identifying the X-Process

In order to monitor the read and execution attempts of a target code page, a naïve approach is to configure the corresponding EPT entries to make it non-executable, non-writable, and non-readable (NX-NW-NR) when loading the target code page for the first time. At this point, if there is an execution or read request to the target code page, it will trigger a VM exit, which we can capture and handle. If we discover that the X-process and the R-process are both running in the system, a preloading thread is created on-demand to keep the victim's secrets from being revealed.

Since read operations are very common in the system, configuring code pages to be unreadable will introduce a lot of VM exits and degrade the performance. By analyzing the characteristics of the attack, we found that we can temporarily ignore the read operations when there is no victim. Therefore, when loading the target code pages, ZeroShield configures the corresponding EPT entries to make them NX-NW-R. Thus, we can capture any execution attempt while avoiding unnecessary VM exits caused by read attempts.

1) Handling Reads Before Identifying the X-Process: Because the privileges in EPT entries are configured as NX-NW-R, when we capture an EPT NX exit event, we cannot determine which VM previously read the target code page. Therefore, once ZeroShield captures an EPT NX exit event, it needs to find the VMs that loaded the target code page previously. We achieve this by leveraging the reverse mapping mechanism. Then ZeroShield updates the corresponding EPT entries, making the read of the target code page trigger a VM exit.

In order to improve the efficiency of page recycling, after Linux 2.6, the reverse mapping mechanism is adopted to store the page table entries (PTEs) associated with a physical page in the page structure. Compared with the mapping from the virtual address to the physical address, the reverse mapping is the mapping from the physical page to the virtual address. We can obtain all Host Virtual Addresses (HVAs) mapped to a known physical page by leveraging the reverse mapping.

In the handler of the EPT NX exiting, with the help of the reverse mapping mechanism, ZeroShield obtains the VMs corresponding to the HVAs that map to the target code page. Then, it configures the corresponding EPT entries to make the target code page NX-NW-NR in these shared VMs. The read attempts in these VMs will trigger a VM exit afterward, allowing ZeroShield to capture and monitor the R-process. If the X-process and the R-process are both running in the system, ZeroShield can be aware of that and create a preloading thread to keep the X-process's secrets from being revealed.

To avoid performing the reverse mapping every time we handle an EPT NX exit event, as well as conveniently manage and trace VMs that share target code pages, we use a list to record the status of these VMs. Specifically, ZeroShield creates a `ZS_VM_state` data structure to record the VM that loads the target code page. Each entry of the `ZS_VM_state` list contains six fields: `VM`, `HVA`, `HPA`, `GPA`, `R_state`, `X_state`, which are related to the VM itself and the target code page. They denote a `kvm` data structure which identifies the current VM, and Host Virtual Address (HVA), HPA, GPA of the target code page, the read state and execution state of the target code page, respectively. With the list, we only need to perform the reverse mapping when processing the EPT NX exit event of the VM for the first time, which improves performance.

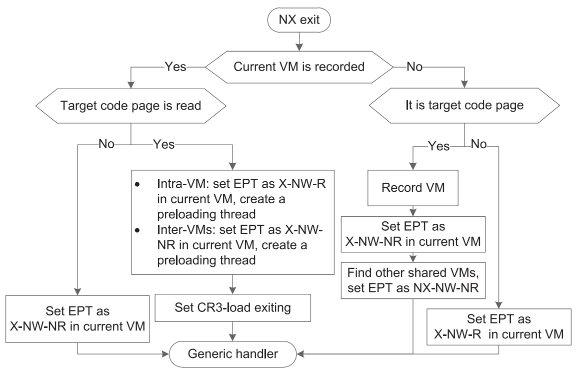

NX exit
Current VM is recorded
Yes
No
Target code page is read
It is target code page
No
Yes
Yes
No
• Intra-VM: set EPT as X-NW-R in current VM, create a preloading thread
• Inter-VMs: set EPT as X-NW-NR in current VM, create a preloading thread
Record VM
Set EPT as X-NW-NR in current VM
Find other shared VMs, set EPT as NX-NW-NR
Set EPT as X-NW-NR in current VM
Set CR3-load exiting
Set EPT as X-NW-R in current VM
Generic handler

Fig. 2. The handler of EPT NX exiting.

2) Reducing Overhead When There Are No Attackers: As we mentioned in Section III-C2, ZeroShield promptly detects the suspension and termination of the X-process by monitoring the updates of CR3 register. To enable such a mechanism, the `CR3-load exiting` bit in the `VMCS` of the current vCPU should be set. Consequently, when the X-process is suspended or terminated, ZeroShield can capture a CR3-load exit event. This will be detailed in Section IV-D.

However, in the absence of attackers, these CR3-load exit events can also lead to considerable performance overheads. To solve this problem, the `CR3-load exiting` bit is cleared by default when there is no R-process in the system. And `CR3-load exiting` bit will be set only when the target code page is read.

We introduce a flag, denoted as `ZS_CR3_Enable`, to trace the status of the `CR3-load exiting` bit associated with the X-process. The decision to disable `ZS_CR3_Enable` occurs when the R-process is not present in the system, and this will be handled in the handler of CR3 exit event (refer to Section IV-D). Based on what we have described, the processing flow of the EPT NX exiting is summarized in Figure 2.

B. Identifying the R-Process

According to our processing of the EPT NX exiting described in Section IV-A, we learn that after identifying the X-process, ZeroShield sets the NR-bit of the corresponding EPT entries, making any read attempts trigger a VM exit (EPT NR exit), thus the R-process can be identified. In the handler of the EPT NR exit event, ZeroShield checks whether there is an X-process in the system. If so, ZeroShield updates the corresponding EPT entries according to different scenarios, and creates a preloading thread. The idea is straightforward, but it faces an issue due to our special design which is described in the Section IV-A2, and thus further treatment has to be considered in the handler of the EPT NR exiting.

1) Handling the Issue Caused by the `ZS_CR3_Enable` Flag Being Cleared: In Section IV-A, to optimize performance, we introduce `ZS_CR3_Enable` to trace whether the CR3-load exiting related to the X-process is enabled. When there is no R-process in the system, the `ZS_CR3_Enable` flag is cleared. Thus, the suspension of X-process does not trigger a VM exit, and we are unable to capture it. Therefore,

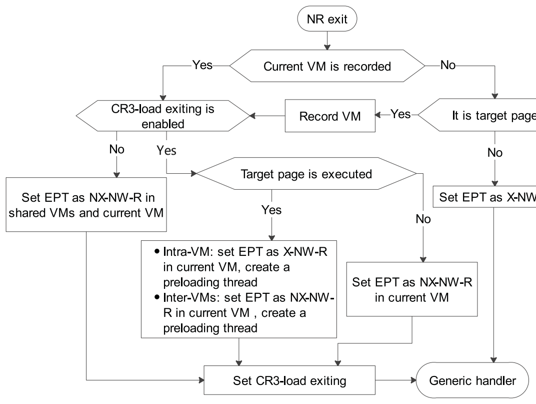

NR exit
Current VM is recorded
Yes
No
CR3-load exiting is enabled
Record VM
Yes
It is target page
No
Yes
No
Set EPT as NX-NW-R in shared VMs and current VM
Target page is executed
Set EPT as X-NW-R
Yes
No
• Intra-VM: set EPT as X-NW-R in current VM, create a preloading thread
• Inter-VMs: set EPT as NX-NW-R in current VM , create a preloading thread
Set EPT as NX-NW-R in current VM
Set CR3-load exiting
Generic handler

Fig. 3. The handler EPT NR exiting.

it is difficult to make sure whether the X-process is running or not.

For this reason, further processing has to be added. ZeroShield needs to check the `ZS_CR3_Enable` flag in the handler of EPT NR exiting.

- If the `ZS_CR3_Enable` flag is cleared, the target code page is configured to be NX in the shared VMs for perceiving the presence of an X-process, thereby preventing potential information leakage. ZeroShield also needs to configure the target code page to be readable in the current VM to continue running. Then ZeroShield sets the `CR3-load exiting` bit in `VMCS` for the current vCPU, to monitor the suspension of the R-process in time.
- If the `ZS_CR3_Enable` flag is set, it means ZeroShield can capture the suspension of the X-process normally. ZeroShield immediately determines whether there is an X-process running in the system. If so, ZeroShield sets the R-bit of the corresponding EPT entries in the intra-VM scenario to enable the application to continue running, or sets the NX-bit and R-bit in the inter-VMs scenario, to enable the application in the current VM to continue running and identify the X-process in the shared VMs. Finally, ZeroShield creates a preloading thread and sets the `CR3-load exiting` bit for the current vCPU.

2) Handling New R-Processes in Unrecorded VMs: In general, when we identify an R-process, the current VM is already recorded in the `ZS_VM_state` list since we use reverse mapping to find the shared VMs when handling the EPT NX exiting. But after the X-process has been identified, there may be a new VM not recorded in the `ZS_VM_state` list that loads the target code page, triggering a page merging related to the new VM. In the handler of the page merging (detailed in Section IV-C.), the new VM's EPT entry is configured to make the target code page NX-NW-NR. Afterward, if there is an R-process in the new VM, an EPT NR exit event will be triggered, which can be captured by ZeroShield. Then, ZeroShield creates a new `ZS_VM_state` data structure to record this new VM in the handler of EPT NR exiting. After that, ZeroShield checks the `ZS_CR3_Enable` flag and processes as we described earlier. Figure 3 shows our processing in detail.

C. Handling Page Merging

In Section IV-A we utilize reverse mapping to handle the situation where the target code page has been read by one or multiple VMs before the X-process is identified for the first time. However, a new VM that is not recorded could also read the target code page afterwards without triggering a VM exit. Because when it loads the target code page for the first time, the privilege of the corresponding EPT entry is configured as NX-NW-R. Once the sensitive page in the new VM is merged with the one that the X-process loads, the secrets could be glimpsed by the attacker if we do not discover and record this new VM in time.

In the Linux OS, when the pages are merged, the MMU notifier mechanism can assist the KVM virtualization subsystem [38], [39] in maintaining proper page mapping. Specifically, the MMU model notifies the KVM model to update the corresponding EPT entry to synchronize the page mapping with the host page table. KVM model registers some MMU notifiers to achieve synchronization.

Among the notifiers involved in page merging, we choose `kvm_mmu_notifier_change_pte()`. ZeroShield can obtain the KSM page in this notifier, then determine whether it is the target code page. If so, ZeroShield updates the EPT entry, that is, it writes the new Page Frame Number (PFN) to the EPT entry to synchronize the mapping with the host, and configures the privileges of the EPT entry to make the target code page as NX-NW-NR. Therefore, any execution and read attempts will trigger a VM exit, which ZeroShield can catch and handle.

It should be noted that we are only dealing with those VMs that have not been previously recorded here. There is a condition where the X-process in the VM has already triggered an EPT NX exit, and subsequently, page merging occurs on this VM. We do not need to perform additional processing on this type of page merging and directly follow the original processing of the host OS. Because as described in Section IV-A, the VM is already recorded and monitored by us, we do not need to specifically update the EPT entries to capture the read and execution attempts in the VM.

D. Handling Process Switching

The preloading thread starts when the X-process and the R-process are both active. It will keep running even if the X-process or the R-process is no longer active, which results in too much performance penalty. As a result, we hope that the preloading thread stops when the X-process or the R-process is suspended to reduce the overhead. This is because when there is no X-process or no R-process in the system, the protection thread does not need to run either. Thus ZeroShield sets the `CR3-load exiting` bit in the handler of the EPT NX exiting and the EPT NR exiting. When the X-process or the R-process is suspended, ZeroShield can capture a VM exit and check whether it needs to stop the preloading thread in the handler of the CR3-load exiting.

For registering a CR3-load exiting event, ZeroShield invokes the function `vmcs_writel()` to set the corresponding bit in `VMCS`. Specifically, ZeroShield sets the `CR3-load`

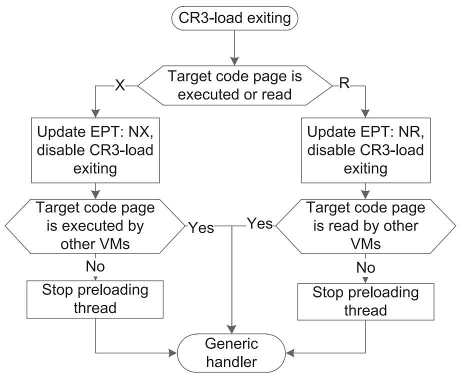

CR3-load exiting
Target code page is executed or read
X
R
Update EPT: NX, disable CR3-load exiting
Update EPT: NR, disable CR3-load exiting
Target code page is executed by other VMs
Target code page is read by other VMs
Yes
Yes
No
No
Stop preloading thread
Stop preloading thread
Generic handler

Fig. 4. The handler of CR3-load exiting.

exiting bit, clears the CR3-target value bit and configures the CR3-target count to 0 in VMCS, then it is able to capture a VM exit once the value of the CR3 register is updated. In the corresponding handler, ZeroShield first checks whether the suspended process is an X-process or an R-process by checking the X_state and the R_state in the current ZS_VM_state. If the X-process is suspended, then ZeroShield updates the corresponding EPT entry to make the target code page non-executable in the current VM again and clears the CR3-load exiting bit. Finally, ZeroShield traverses the ZS_VM_state list to find if there is still an X-process in the system. If there are no X-processes in the system, ZeroShield stops the preloading thread. The handling is similar to the suspension of the R-process. Figure 4 shows how to process the CR3-load exit event in detail.

E. Preloader and Privileges Transition

If the X-process and the R-process are running simultaneously, the monitor notifies the preloader to create a preloading thread. The preloading thread is a kernel thread, therefore it will not be suspended unless ZeroShield notifies it to stop. The preloading thread utilizes the instruction mov in a loop to access the target code pages. We need to clarify that the "target code pages" actually mean "attacked pages" but not all the pages we specify in the kernel. When a target page is read and executed simultaneously, we refer to it as an "attacked page". Although we may specify multiple target pages in the kernel, the preloader only loads the attacked ones. In addition, it should be noted that multiple target pages share a preloader. Specifically, for the target pages (4KB for each page) requiring protection within the system, the preloader sequentially accesses each cache line, and thus loads these pages into the cache. The described operations constitute a loop iteration. The preloader executes this loop iteration continuously during its operation.

As we described in the previous sections, the privileges of an EPT entry are constantly updated based on the running and suspension of the X-process and the R-process. In addition, page merging also affects them. We summarize the transition of the privileges during the whole lifetime of the process, as shown in Figure 5.

V. Evaluation

In this section, we evaluate the security and performance of ZeroShield.

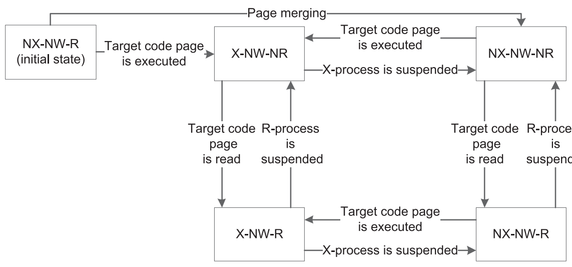
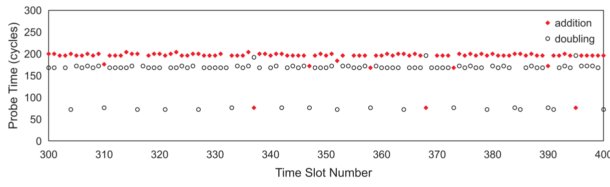
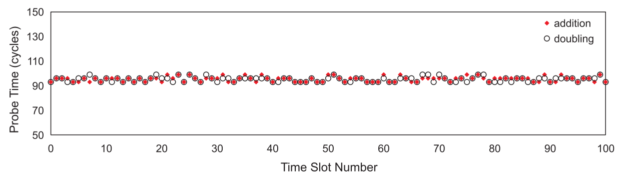


Fig. 5. Privileges transition in an EPT entry.



(a) Result of FLUSH+RELOAD attack in native Linux.



(b) Result of FLUSH+RELOAD attack in Linux with ZeroShield.

Fig. 6. FLUSH+RELOAD attack results.

A. Experiment Setup

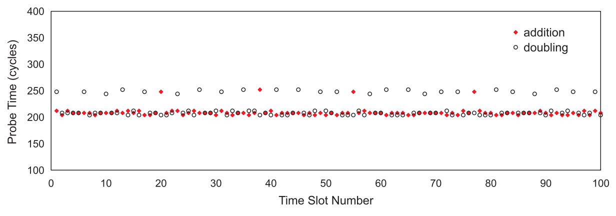
We tested our implementation on a Dell PowerEdge R720 with two Intel Xeon E5-2609 (1.90GHz) processors. The guest OS is Ubuntu 18.04.5 LTS Linux, and each guest VM has two vCPUs. The host uses the Linux kernel v4.15 which we modified to deploy ZeroShield. Qemu-kvm v2.11.1 is installed on the host OS via the apt-get install command. We add less than 2000 lines of code to this Linux kernel.

B. Security Evaluation

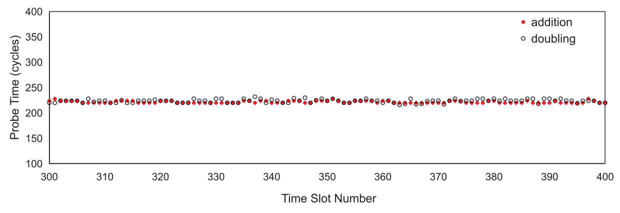
To evaluate the security of our implementation, we use Mastik [40], a toolkit that provides the implementations of side-channel attack techniques, to launch a code page sharing attack, such as FLUSH+RELOAD attack and FLUSH+FLUSH attack, in a virtualized environment. We also consider two scenarios: intra-VM and inter-VMs.

1) FLUSH+RELOAD Attack Resistance: We first demonstrate a FLUSH+RELOAD attack in the native Linux and our modified Linux. The victim computes signatures using the implementation of ECDSA in OpenSSL 1.1.0h. The scalar multiplication in ECDSA is implemented by utilizing the windowed non-adjacent form (wNAF) algorithm [41], [42], [43]. From the previous works [3], [4], [5] we learn that the attacker can launch a FLUSH+RELOAD attack to observe the sequence of additions and doubling used to execute the scalar multiplication, and then recover the victim's private key.

In both intra-VM and inter-VMs scenarios, the attacker monitors and collects the activities of two cache lines that correspond to the point addition and doubling respectively while the victim is computing 100,000 signatures. These two



(a) Result of FLUSH+FLUSH attack in native Linux.



(b) Result of FLUSH+FLUSH attack in Linux with ZeroShield.

Fig. 7. FLUSH+FLUSH attack results.

functions happen to be on the same code page in OpenSSL 1.1.0h. Thus, there is only one target code page in our experiments. The results in the inter-VMs scenario are shown in Figure 6, and the intra-VM scenario is similar. Clearly, in native Linux, if the victim is performing the point addition or the point doubling, the attacker can observe a cache hit on the corresponding cache line. Otherwise, the attacker will observe a cache miss. In our modified Linux, once the victim fetches instructions from the target code page, the probe time observed by the attacker is mostly close to a cache hit until the victim is suspended. Thus, the attacker cannot obtain the specific memory access pattern of the victim through the FLUSH+RELOAD attack, which can mitigate the leakage of sensitive information observably.

2) FLUSH+FLUSH Attack Resistance: We also implement the FLUSH+FLUSH attack in the native Linux and the modified Linux separately, which is similar to the FLUSH+RELOAD attack described before. In the intra-VM and inter-VMs scenarios, we obtain the attack results respectively. Figure 7 shows the results in the inter-VMs scenario, and the intra-VM is similar. Our experiment proves that ZeroShield can indeed mitigate the FLUSH+FLUSH attack.

Besides, in order to evaluate the false positives of ZeroShield, we ran the system for a week. During our evaluation, we do not launch cache side-channel attacks in the system. The system had several long-running services running, such as `init`, `ksmd`, `syslogd`, etc. By analyzing the logs, we found that normal system behavior would not cause ZeroShield to start the preloader.

C. Performance Evaluation

1) Overhead of the Secure-Sensitive Application: We evaluate the impact of ZeroShield on the victim when there is a victim and an attacker in the system simultaneously. Both the inter-VMs scenario and the intra-VM scenario are taken into account. We perform 100,000 signature operations and repeat 10 times, the average time of computing 100,000 signatures

TABLE I

OVERHEAD TO THE VICTIM ("WITHOUT PRELOADER" INDICATES THE OVERHEAD INTRODUCED BY THE MONITOR ITSELF)

Scenario	Native Linux run time (s)	Modified Linux with / without preloader run time (s)	Overhead with / without preloader (%)
Only victim	139.382	139.480 / 139.585	0.07/ 0.15
Intra-VM	172.920	202.949 / 179.782	17.31 / 3.92
Inter-VMs	152.399	178.788 / 162.138	17.32 / 6.39

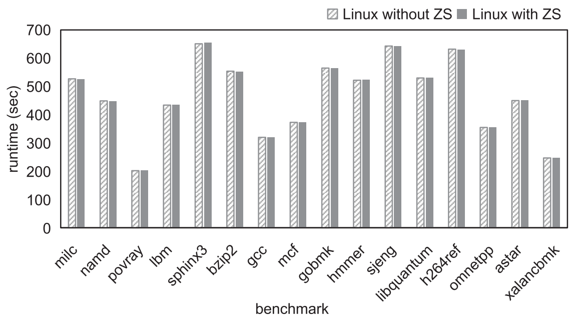
is used to evaluate the performance. Table I illustrates that ZeroShield induces almost no overhead when the system has no R-processes. And it induces a modest performance overhead for the victim due to the additional checks and the preloading thread when the R-process is active.

The overhead introduced by the monitor was also evaluated separately. We conducted this evaluation in the intra-VM scenario and the inter-VMs scenario in the same way as described before. Table I illustrates that the overhead induced by the monitor is 3.92% in the intra-VM scenario and 6.39% in the inter-VMs scenario.

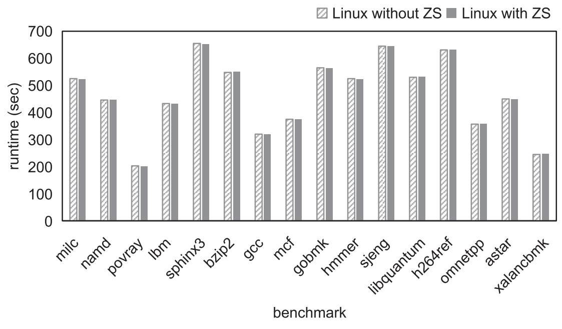
The overhead induced by the monitor includes extra VM exits, hash calculations, and extra code logic added by us. It should be noted that we do not hash code pages when they are first loaded, but only configure the privileges of EPT entry to make the page readable but non-executable. Afterward, the hash value is calculated when an EPT NX exit or an EPT NR exit event is triggered. Therefore, actually, we do not hash every code page at runtime. Hash is very quick and does not induce more overhead. We take the hash as a step in the page fault handler at runtime. Most of the overhead induced by the monitor is caused by VM exits. Since we introduce the `ZS_CR3_Enable` in Section IV-A, there are almost no additional VM exits when there is no attacker.

As we mentioned earlier, a target code page was used in our experiments. Increasing the number of target code pages may induce tiny changes in the overhead of the monitor, but it hardly affects the overhead of the preloader. Moreover, the main overhead is introduced by the preloader, and no matter how many target code pages there are, it keeps accessing memory until it receives a stop notification from the monitor. Therefore, increasing the number of target code pages will not cause a significant decrease in the processor's performance of the entire mitigation. Note that "target code pages" refers to those pages of code containing cache side-channel vulnerabilities, such as the implementation code for ECDSA within the OpenSSL cryptographic library.

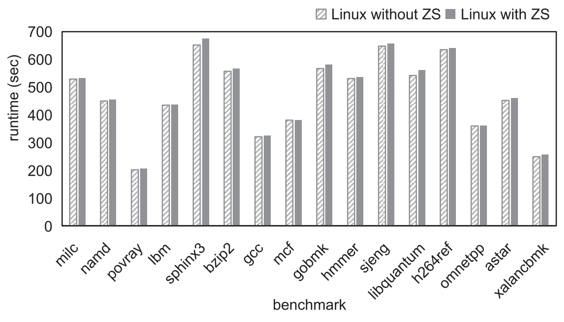
Even with an increase in the number of target pages, the preloader does not impose significant performance overhead. This is because of the following reasons. Firstly, even if there are numerous target pages simultaneously executed and read within the system, ZeroShield still uses only one thread to handle these target code pages, obviating the necessity for creating multiple threads. Secondly, the preloader only loads a target page when it is executed and read simultaneously; it does not load pages that are solely executed or solely read. Therefore, the attacker is highly unlikely to cause the preloader to load a large number of target code pages at a certain



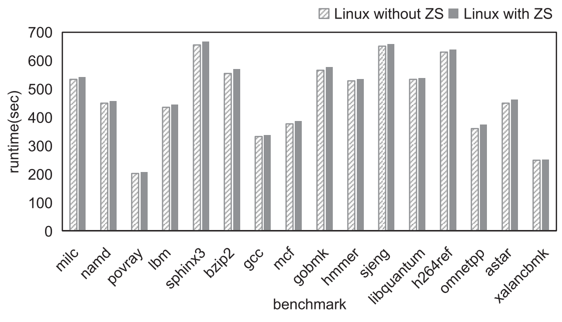
(a) No victim and no attacker in the system.



(b) Only victim in the system.



(c) A victim and an attacker in intra-VM scenario.



(d) A victim and an attacker in inter-VMs scenario.

Fig. 8. SPEC CPU 2006 performance overhead in different scenarios.

moment (we conducted a more detailed analysis on this in Section VI-B). Thus, the overhead induced by the preloader is limited.

2) SPEC CPU 2006 Benchmark: We use SPEC CPU 2006 [44] to evaluate the impact of our implementation on the system. We measure SPEC CPU 2006 scores on the native Linux OS and our modified Linux OS, respectively. Four scenarios are considered: 1) the system has no victims or attackers (idle); 2) there is only a victim; 3) the system has a victim and an attacker in the intra-VM scenario; 4) the system has a victim and an attacker in the inter-VMs scenario.

As shown in Figure 8, our implementation hardly affects the performance of the system in the absence of an attacker, and the overhead is 0.036% on average. Even if there is an attacker, the overhead is 1.8% on average, and the maximum of the overhead does not exceed 3.8%, which comes from *omnetpp* in the inter-VMs scenario.

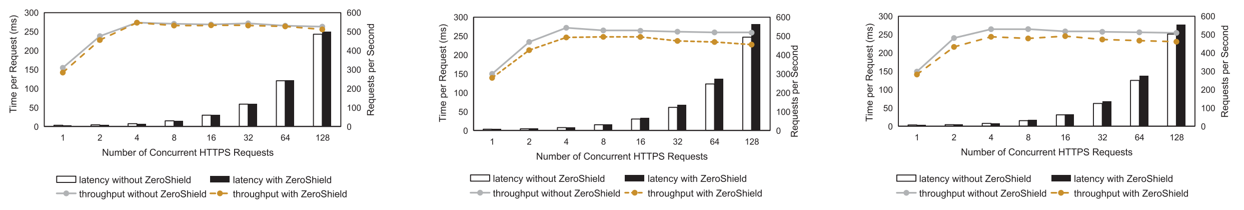
3) HTTPS Throughput and Latency: In order to evaluate the overhead introduced by ZeroShield in actual cryptographic service invocation, we integrate ZeroShield with the HTTPS service, and construct an HTTPS service that requests for an ECDSA signature. Specifically, we run an Apache server in a VM. It uses the ECDSA signature algorithm in OpenSSL 1.1.0h when establishing an HTTPS connection. Therefore, the `httpd` is the victim in our experiment. We consider three scenarios: 1) only the victim in the system; 2,3) the system has a victim and an attacker in the intra-VM and the inter-VMs separately. In each scenario, the client constructs 10,000 HTTPS requests for a 4KB web page at different concurrency levels. As shown in Figure 9, when there is no attacker, the decrease in throughput and latency is about 2.7% and 2.5% on average; when there is an attacker in the intra-VM or inter-VMs scenario, the decrease in throughput is about 8.7% and 8.0% on average separately, and the decrease in latency is about 9.3% and 8.8% on average separately.

VI. Related Works and Discussions

A. Related Work Comparison

As a well-studied problem, various solutions have been proposed to mitigate cache side-channel attacks. We summarize the characteristics of these solutions in Table II. In Table II, we use the average overhead measured by the system benchmark to represent the overhead of the scheme by default. If there is maximum overhead and other special cases (such as extra memory overhead), we mark them in parentheses. For those whose average and maximum overhead were not published in the original article, we use the overhead range published in the article, such as CACHEBAR.

1) Randomization-Based Solutions : Some solutions mitigated the cache side-channel attacks through randomization. Newcache [20] and MIRAGE [21] mitigated the cache side-channel attacks efficiently by redesigning a cache, but the requirement to be deployed in the hardware layer makes them difficult to generalize and apply to actual production environments in a short period. Düppel [22] modified the kernel of the guest OS and cleared the L1 cache at a certain frequency to resist cross-VM side-channel attacks, which limits it to only defending the L1-based attacks, and the performance decreases by up to two orders of magnitude when being attacked. Ghost



(a) No victim and no attacker in the system. (b) A victim and an attacker in intra-VM scenario. (c) A victim and an attacker in inter-VMs scenario.

Fig. 9. HTTPS latency and throughput in different scenarios.

TABLE II

Previous Solutions to Mitigate Cache Side-Channel Attacks

	Solutions	Technical principle	Transparency	Adaptiveness	Overhead with/without attackers
randomization -based	Newcache [20]	random mapping	✓	✗	<3% / <3%
	MIRAGE [21]	random evicting	✓	✗	2% / 2% (<7%, 17%-20% storage overhead)
	Düppel [22]	clearing the L1 cache	✓	✗	up to two orders of magnitude / <7%
	Ghost Thread [23]	reloading data	✗	✗	10%-35% / 10%-35%[1]
	Cloak [24]	preloading data	✗	✗	1567% / 1.2% [1]
	Dynamic software diversity [28]	dynamic control-flow diversity	✗	✗	182% / 182%
	Guard Cache [45]	create false hits and misses	✗	✗	-0.2%-173% / -0.2%-173%
isolation -based	IVcache [29]	cache partition	✗	✗	4.4% / 4.4%
	NoMo cache [30]	cache partition	✓	✗	1.2% / 1.2% (<5%)
	CACHEBAR [32]	copy on access	✓	✗	0.4%-225% / 0.4%-225%
	Chameleon [34]	dynamic page coloring	✗	✗	3% / 3%
	STEALTHMEM [33]	locking cache lines	✗	✗	3%-12% / 3%-12%[1]
	VUsion [46]	copy on access	✓	✗	2.7% / 2.7%
	CATalyst [31]	cache partition	✗	✗	0.2% / 0.2% (<45%)
ZeroShield		**reload data on-demand**	✓	✓	**1.8% (<3.8%) /0.036%** [2]

[1] Performance overhead for cryptographic applications.
[2] The other schemes in the table do not differentiate between defense against code page sharing attacks and date page sharing attacks. In contrast, ZeroShield primarily focuses on defending against code page sharing attacks.

Thread [23] and dynamic software diversity [28] modified the application to introduce random program behavior, therefore, they lack transparency. Cloak [24] modified the application and uses transactional memory to shield the cache miss of the sensitive code and data, which also lacks transparency and incurs significant overhead. Guard Cache [45] proposed an approach to mitigate cache side-channel attacks by creating false cache hits and misses. It proposed using a small fully associative Guard Cache to create false hits and false misses, making it difficult for attackers to observe cache access times accurately. However, Guard Cache requires users to manually configure the working mode, it is still not fully transparent and adaptive. The performance overhead induced by Guard Cache is -0.2%-173%.

2) *Isolation-Based Solutions :* IVcache [29] and NoMo cache [30] thwarted the cache side-channel attacks by cache partition, which is efficient but needs to redesign the cache and requires lots of effort to deploy to the practical production environment. CACHEBAR [32] used the copy-on-access method to resist FLUSH+RELOAD attacks in the virtualized environment and the status of all shared physical memory pages are monitored, which results in about 3.4% to 143% overhead as the number of containers in the system increases. Compared with CACHEBAR, the preloader has the advantage of making as few changes to the hypervisor as possible and can be started and stopped quickly without memory overhead. Chameleon [34], STEALTHMEM [33] and CATalyst [31] were implemented in the hypervisor layer and provide cache isolation in a software-based manner, but they are not transparent completely since victims have to actively request protection of security-sensitive pages. VUsion [46] is a copy-on-access scheme. Although this scheme can defend against many types of attacks and the overhead is moderate. However, a large number of page faults resulted in 4.9% overhead for SPEC CPU 2006. Moreover, forcing delayed page merging also has a certain impact on system performance. Moreover, as the system keeps running and the number of applications increases, these overheads may further increase. The system is always accompanied by these overheads.

Compared to these solutions, our solution is an efficient, transparent, and adaptive approach implemented in the hypervisor layer. It is compatible with most commercial chips and simpler to deploy in practical scenarios with modest performance overhead. Especially in the absence of attackers, which is the most common situation, our method hardly introduces extra overheads. In table II, we compare our solution with the others we mentioned above.

B. Discussions

In this section, we conduct a more in-depth analysis and discussion of ZeroShield, examining it from three distinct perspectives.

1) Scalability on Other Architectures: The prototype of ZeroShield is implemented on Intel processors which provide Intel VT-x and EPT features for virtualization. There are similar mechanisms that support virtualization on other platforms. Firstly, ZeroShield relies on Intel VT-x, which, along with AMD's AMD-V [47] and ARM's VHE [47], can be seen as the same mechanism running on different platforms. These technologies serve as the foundation for hardware virtualization, ensuring the efficient execution of virtualized environments. Secondly, ZeroShield relies on Intel EPT mechanism. While Intel EPT is specific to Intel processors, equivalent functionalities can be achieved on AMD platforms through Nest Paging and on ARM architectures through Stage 2 translation. It is important to emphasize that the core principles underlying ZeroShield remain consistent across these platforms, although some engineering efforts are required for porting and adaptation to ensure compatibility with diverse hardware environments. In summary, while ZeroShield may require minor engineering adjustments to accommodate different platforms, its fundamental principles remain uniform across various hardware architectures. We believe that this scalability highlights the versatility and effectiveness of ZeroShield in providing powerful security solutions across different architectures.

2) Quantity Restriction of the Protected Code Pages : When there are too many target code pages needing to be accessed by the preloading thread in a loop, the time required to reload all the target code pages is longer. This may provide a window during which sensitive data is not reloaded in time.

However, in general, the number of addresses monitored in a single code page sharing attack is limited [8]. There could be several factors contributing to this limitation. Firstly, too many target addresses in a code page sharing attack would result in a longer INITIALIZE and MEASURE phases, then the eviction of some target addresses from the cache because the cache's capacity is finite, thereby impacting the reliability and accuracy of the attack. Secondly, simultaneous monitoring of a large number of addresses by the attacker could result in frequent cache operations, thereby affecting the normal operation of the system and the performance of other applications. Such performance overhead is likely to draw the attention of system administrators or defense mechanisms, thus increasing the risk of detection for the attack. Lastly, the prefetcher also plays a limiting role in FLUSH+RELOAD attacks, as observed by Yarom and Falkner [6]. Therefore, to minimize false positives caused by the prefetcher, the attacker should also not monitor too many addresses [8].

In addition, ZeroShield initiates the preloader and loads the target page only when the target page is executed and read simultaneously. Therefore, if an atatcker attempt to monitor one address and make the preloader busy by reading other fake target pages as well from other cores, the attacker needs to continuously execute victim programs corresponding to these fake target pages on each of the other cores simultaneously. Then, the preloading thread will be triggered to load these fake target pages (while these victim processes indeed require protection). However, for an attacker without privileges, the CPU scheduling policy prevents its processes from monopolizing the CPU continuously. Thus, it is challenging for an attacker to overload the preloader with a large number of target pages at a certain moment. Therefore, it is almost impossible to monitor one address and make the preloader busy by reading other target pages as well from other cores. Due to these technical and performance limitations, attackers in practical scenarios typically monitor several addresses to guarantee the efficiency and reliability of the attack.

In a code page sharing attack, ZeroShield can identify the target code pages and load these code pages into the cache rapidly through the preloader. Thus we believe that the leaked information is too tiny to be exploited for key recovery.

3) Potential Defeating Methods of ZeroShield : By summarizing the CPSA techniques proposed in previous works, such as FLUSH+RELOAD, and FLUSH+FLUSH, we have observed that attackers consistently request read privilege for target code pages rather than execution privilege. However, in order to circumvent our defense approach, attackers might alter their behavior by executing the sensitive code instead of simple reading. As far as we know, such an attack method has not yet been proposed. We briefly analyze potential techniques that attackers might utilize.

One straightforward approach is that the attacker executes the target function, similar to the victim, and measures the time required for execution. However, this approach has low resolution and is susceptible to system noise, leading to plenty of errors. Therefore, its success is challenging and unlikely.

Another unconventional approach which is inspired by return-oriented programming (ROP) [48] could involve the attacker using the `JMP` instruction to jump into the target function for execution and then measuring the time required after the function returns. Nevertheless, this involves inter-segment jumps (far jump and far return). When using a far jump, the attacker has to make the appropriate choice of segment selector. This could be difficult to achieve for the attacker. In practice, only the near jump is exploited to construct the ROP shellcode [49]. Further in-depth exploration and analysis of the far jump are beyond the scope of this paper. However, based on our preliminary analysis, it appears that this approach is also difficult to successfully implement. Thus, at present, we tentatively conclude that attackers face significant challenges in launching a code page sharing attack by executing the sensitive code.

Furthermore, EVICT+TIME is a type of cache side-channel attack that executes the target code page. ZeroShield is unable to resist EVICT+TIME attacks because EVICT+TIME does not require memory sharing between attackers and victims, that is, it does not belong to CPSA. In addition, we consider that the efficacy of EVICT+TIME is currently constrained. The attack relies on the repeated invocation of victim computation by the attacker, then infers secret information according to the latency of the victim's computation, thus resulting in low resolution and susceptibility to noise. In future work, we will continue to delve into the possibilities of attackers executing the sensitive code to initiate a code page sharing attack.

In Page Cache Attacks [50], the attacker observes the low-frequency events of the victim, but the impact of low-frequency events is quite limited. Page Cache Attacks primarily target events with low frequency, such as user-entered passwords and keystrokes. This limitation stems from the fact that the frequency at which an attacker can measure events is constrained to approximately 6.7 times per second on Linux, with a spatial granularity of 4KB. The current high-speed computer systems pose challenges for this coarse spatio-temporal granularity, as discussed in the original article [50]. For instance, due to this limitation, an attacker may overlook some keystrokes from a fast typist. Consequently, the attack efficiency remains notably low, rendering it challenging to target precise and rapid cryptographic calculations, such as computing ECDSA signatures. In addition, once an attacker reads the target page, it transitions into an R-process. The active R-process triggers an EPT NR exit event and is captured by ZeroShield. Subsequently, ZeroShield evaluates the system's status and determines whether to activate the preloader. Finally, In remote scenarios mentioned in the original paper, confirmation of whether the target page is in the page cache is achieved by accessing the target page and measuring the time taken for page-fault handling. However, this measurement exploits access latency between memory and disk accesses rather than those between hardware cache and memory. Actually, the memory pages used by virtual machines are anonymous pages, meaning there are no corresponding pages on the disk. Consequently, such attacks cannot be carried out in virtualized environments.

We also need to declare that the performance overhead caused by ZeroShield is very difficult to use as a DDoS attack vector. There are two main reasons. First, even if the attacker creates a scenario with multiple X-processes and R-processes, thereby increasing the number of target pages requiring protection, the performance overhead of this attack approach is bounded. This is because, even with multiple attacked pages, ZeroShield only utilizes a single preloader. In addition, as described before, due to these technological and performance constraints, attackers are highly unlikely to execute a large number of target pages simultaneously. Second, if there are only attackers (R-processes) in the system, no victims (X-processes), the preloader will not be triggered, thereby inducing no overhead.

VII. Conclusion

We propose an adaptive and transparent approach named ZeroShield to mitigate code page sharing attacks, which can achieve fine-grained control over the monitored physical pages and the activity of the preloading thread without the user's manual intervention. ZeroShield utilizes hardware-assisted virtualization to achieve transparency and reduce the overhead. And it achieves adaptability by tracing the system events constantly. We describe the details of its implementation and evaluation. The evaluation results show that in the absence of an attacker, our solution induces almost no overhead. When the attacker and the victim are both active, the overhead is modest.

References

[1] K. Divya and S. Jeyalatha, "Key technologies in cloud computing," in *Proc. Int. Conf. Cloud Comput. Technol., Appl. Manage. (ICCCTAM)*, Dec. 2012, pp. 196–199.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 199–212.

[3] S. Fan, W. Wang, and Q. Cheng, "Attacking OpenSSL implementation of ECDSA with a few signatures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1505–1515.

[4] J. V. D. Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *Proc. Cryptographers' Track RSA Conf.*, 2015, pp. 3–21.

[5] N. Benger, J. V. D. Pol, N. P. Smart, and Y. Yarom, "'Ooh aah...just a little bit': A small amount of side channel can go a long way," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2014, pp. 75–92.

[6] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. USENIX Secur. Symp.*, K. Fu and J. Jung, Eds., Aug. 2014, pp. 719–732.

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

[8] 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.

[9] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.*, Aug. 2016, pp. 549–564.

[10] 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.

[11] M. Vanhoef and E. Ronen, "Dragonblood: Analyzing the dragonfly handshake of WPA3 and EAP-pwd," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 517–533.

[12] D. De Almeida Braga, P.-A. Fouque, and M. Sabt, "Dragonblood is still leaking: Practical cache-based side-channel in the wild," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 291–303.

[13] 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*, 2016, pp. 279–299.

[14] 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.

[15] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky way: A conflict-based cache covert channel bypassing set associativity," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2022, pp. 646–661.

[16] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 1458–1473.

[17] H. Kim, H. Yoon, Y. Shin, and J. Hur, "Cache side-channel attack on mail user agent," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2020, pp. 236–238.

[18] A. Shahverdi, M. Shirinov, and D. Dachman-Soled, "Database reconstruction from noisy volumes: A cache side-channel attack on SQLite," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1019–1035.

[19] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2003–2020.

[20] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[21] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proc. 30th USENIX Secur. Symp.*, Aug. 2021, pp. 1379–1396.

[22] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 827–838.

[23] R. Brotzman, D. Zhang, M. Kandemir, and G. Tan, "Ghost thread: Effective user-space cache side channel protection," in *Proc. 11th ACM Conf. Data Appl. Secur. Privacy*, Apr. 2021, pp. 233–244.

[24] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proc. USENIX Secur. Symp.*, 2017, pp. 217–233.

[25] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proc. 3rd ACM Workshop Cloud Comput. Secur. Workshop*, C. Cachin and T. Ristenpart, Eds., Oct. 2011, pp. 41–46.

[26] W. Liu, D. Gao, and M. K. Reiter, "On-demand time blurring to support side-channel defense," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2017, pp. 210–228.

[27] Z. Mi, H. Chen, Y. Zhang, S. Peng, X. Wang, and M. K. Reiter, "CPU elasticity to mitigate cross-VM runtime monitoring," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 5, pp. 1094–1108, Sep. 2020.

[28] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[29] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "IVcache: Defending cache side channel attacks via invisible accesses," in *Proc. Great Lakes Symp. VLSI*, Jun. 2021, pp. 403–408.

[30] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–21, Jan. 2012.

[31] F. Liu et al., "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.

[32] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 871–882.

[33] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 189–204.

[34] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw. Workshops*, Oct. 2011, pp. 194–199.

[35] J. Li, C. Liu, X. Wang, and C. Liu, "SCIF-ARF: Container anomaly prediction for container cloud platforms," in *Proc. IEEE 24th Int. Conf. High Perform. Comput. Commun., 8th Int. Conf. Data Sci. Syst., 20th Int. Conf. Smart City, 8th Int. Conf. Dependability Sensor, Cloud Big Data Syst. Appl. (HPCC/DSS/SmartCity/DependSys)*, Dec. 2022, pp. 295–302.

[36] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, USA, 2018.

[37] F. Jiang, Q. Cai, J. Lin, B. Luo, L. Guan, and Z. Ma, "TF-BIV: Transparent and fine-grained binary integrity verification in the cloud," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, Dec. 2019, pp. 57–69.

[38] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments*, 2012, pp. 227–238.

[39] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Linux Symp.*, 2009, pp. 19–28.

[40] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," Univ. Adelaide, Australia, Tech. Rep., 2016. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

[41] K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," in *Proc. Annu. Int. Cryptol. Conf.*, 1992, pp. 345–357.

[42] A. Miyaji, T. Ono, and H. Cohen, "Efficient elliptic curve exponentiation," in *Proc. Int. Conf. Inf. Commun. Secur.*, 1997, pp. 282–290.

[43] J. A. Solinas, "Efficient arithmetic on Koblitz curves," in *Proc. Towards Quarter-Century Public Key Cryptogr.*, 2000, pp. 125–179.

[44] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[45] F. Mosquera, K. Kavi, G. Mehta, and L. K. John, "Guard cache: Creating false cache hits and misses to mitigate side-channel attacks," in *Proc. Silicon Valley Cybersecurity Conf. (SVCC)*, May 2023, pp. 1–8.

[46] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure page fusion with VUsion," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 531–545.

[47] *AMD64 Architecture Programmer's Manual Volumes 1–5*, AMD, Santa Clara, CA, USA, 2020.

[48] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Oct. 2007, pp. 552–561.

[49] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, "Automatic construction of jump-oriented programming shellcode (on the x86)," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, Mar. 2011, pp. 20–29.

[50] D. Gruss et al., "Page cache attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 167–180.

Mingyu Wang received the B.E. degree in computer science and technology from Dalian Maritime University, Dalian, China, in 2018, and the Ph.D. degree in computer application from the University of Chinese Academy of Sciences, Beijing, China, in 2024. Her research interests include applied cryptography and operating system security.

Fangyu Zheng received the B.E. degree in information security from the University of Science and Technology of China, Hefei, China, in 2011, and the Ph.D. degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016. He is currently an Associate Professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high-performance computing.

Jingqiang Lin (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Chinese Academy of Sciences in 2004 and 2009, respectively. He is currently a Full Professor with the School of Cyber Security, University of Science and Technology of China. His research interests include applied cryptography and system security.

Fangjie Jiang received the B.E. degree in network engineering from Hebei University, China, in 2014, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2020. His research interests include system security, network security, and applied cryptography.

Yuan Ma received the B.E. degree from Tsinghua University in 2009 and the Ph.D. degree from the University of Chinese Academy of Sciences in 2014. He is currently an Associate Professor with the Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences. His research interests include secure cryptographic implementation and hardware security modules.