EasyChair Preprint

№ 13841

Cabin: Confining Untrusted Programs within Confidential VMs

Benshan Mei, Saisai Xia, Wenhao Wang and Dongdai Lin

September 21, 2024

Cabin: Confining Untrusted Programs within Confidential VMs

Benshan Mei[1,2], Saisai Xia[1,2], Wenhao Wang[1,2(✉)], and Dongdai Lin[1,2]

[1] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract. Confidential computing safeguards sensitive computations from untrusted clouds, with Confidential Virtual Machines (CVMs) providing a secure environment for guest OS. However, CVMs often come with large and vulnerable operating system kernels, making them susceptible to attacks exploiting kernel weaknesses. The imprecise control over the read/write access in the page table has allowed attackers to exploit vulnerabilities. The lack of security hierarchy leads to insufficient separation between untrusted applications and guest OS, making the kernel susceptible to direct threats from untrusted programs. This study proposes Cabin, an isolated execution framework within guest VM utilizing the latest AMD SEV-SNP technology. Cabin shields untrusted processes to the user space of a lower virtual machine privilege level (VMPL) by introducing a proxy-kernel between the confined processes and the guest OS. Furthermore, we propose execution protection mechanisms based on fine-gained control of VMPL privilege for vulnerable programs and the proxy-kernel to minimize the attack surface. We introduce asynchronous forwarding mechanism and anonymous memory management to reduce the performance impact. The evaluation results show that the Cabin framework incurs a modest overhead (5% on average) on Nbench and WolfSSL benchmarks.

Keywords: Confidential Computing · Trusted Execution Environment · Encrypted Virtualization · Execute-Only Memory · Intra-process isolation · Syscall Filtering

1 Introduction

Privilege separation involves dividing privileges among different entities or processes within a system to limit potential damage caused by a compromised component. In traditional computing systems, privilege separation is achieved by separating the kernel code and userspace code. The kernel, trusted with access to all resources, is segregated from userspace programs, which are confined to their own address spaces. This separation is enforced by the CPU's execution mode and security checks performed by the memory management unit (MMU).

However, traditional privilege separation has certain drawbacks. Firstly, the kernel-user interface, represented by system calls, can allow untrusted processes

to bypass kernel protections due to the large code base of the kernel. While measures like sandboxing and system call filtering can restrict attackers' ability to abuse the interface, they also increase the kernel's attack surface since these countermeasures are often implemented as part of the kernel itself. Secondly, the MMU lacks fine-grained protection for applications. The access permissions defined in the page table entries (PTEs) can only be configured as either writable or non-writable, invariably remaining readable. This limitation hinders the efficient implementation of execute-only memory (XOM), which is known to be effective in thwarting code-reuse attacks by making it challenging for attackers to identify usable gadgets.

In recent years, hardware-based trusted execution environment technologies, such as Intel SGX [25], AMD SEV [42], Intel TDX [13], and ARM CCA [1], have paved the way for the emergence of confidential computing. This new computing paradigm focuses on safeguarding the guest or enclave from attacks originating from potentially untrusted hosts. In the context of confidential computing, protecting the guest kernel assumes even greater significance, as it is responsible for securing users' most sensitive data. If the guest kernel is compromised, the entire CVM is at risk of compromise, potentially resulting in the leakage of any associated sensitive data.

To address the concerns mentioned above, particularly the risks associated with CVMs, we have introduced Cabin, a novel secure execution framework tailored to confine vulnerable processes running within a CVM. Our framework leverages hardware-based isolation mechanisms, i.e., VMPL within AMD SEV-SNP, to establish a secure environment for executing vulnerable processes. Notably, with VMPL, one can assign read, write and execute permissions independently, allowing XOM to work efficiently. Specifically, in our framework, untrusted programs are placed at a lower VMPL, ensuring the protection of the guest OS from vulnerable or malicious applications. A trusted proxy kernel within the lower VMPL acts as an intermediary, facilitating communication between confined processes and the trusted guest OS. To minimize the overhead of VMPL switches, we have designed an asynchronous method for handling events triggered by the application, such as system calls, page faults, interrupts, and exceptions. This approach reduces the number of required VMPL switches and improves overall efficiency. Additionally, our framework allows for flexible monitoring and tracing of processes running within the user-space of the lower VMPL without requiring intervention from the guest OS. This enables the CVM owner to define custom policies for monitoring confined processes. Lastly, our framework incorporates monitoring and logging capabilities to detect any suspicious activities and provide valuable insights into potential threats. This additional layer of security enables proactive threat detection and response.

We have implemented a prototype of the Cabin framework on commodity AMD SEV-SNP servers, utilizing the system to provide execution protection and syscall filtering. Through evaluations on various benchmarks, including syscall routing, page fault handling, Nbench, and WolfSSL, we observed that despite that the VMPL switch is costly (in particular, syscall routing is about sev-

eral times slower than the baseline), Cabin introduces acceptable overhead in real world applications – approximately 5% and 10% for the Nbench and WolfSSL benchmarks respectively. Overall, our confined secure execution framework provides a pratical solution for enhancing the security of CVMs, ensuring the protection of sensitive data from unauthorized access.

Contributions. The contributions of this paper are as follows.

- Designing and implementing a secure execution framework for processes within CVMs based on the fine-grained control of VMPL privilege, protecting the guest OS from direct threats posed by vulnerable or malicious programs.
- We propose VMPL-enhanced cross-layer execute-only protection for vulnerable programs and proxy-kernel running in lower VMPL, making it harder to find exploitable gadgets.
- We introduce asynchronous forwarding mechanism to minimize the performance impact on confined processes. Self-managed memory provided by the proxy-kernel further reduces the performance impact.
- We evaluate the performance impact of the Cabin framework on the Nbench and WolfSSL benchmarks. The evaluation results demonstrate modest overhead of the proposed framework.

2 Background

The emergence of new hardware-based privilege separation mechanisms within CVM presents new opportunities to enhance system and application security. With advancements in research on execute-only protection, intra-process isolation, and syscall filtering, we strive to leverage these technologies to further strengthen the system security. Therefore, we adhere to the traditional paradigm of software security, which emphasizes protecting the guest OS from potential threats posed by untrusted programs.

2.1 SEV-SNP and VMPL

It is crucial to protect the guest VM from malicious host in confidential computing. AMD SEV (Secure Encrypted Virtualization) is the first generation of hardware-assisted virtualization technology that solves the problem with memory encryption and isolation enhanced security [30]. To defend against malicious hypervisors, the SEV and SEV-ES (Encrypted State) are proposed in succession by AMD to encrypt the memory pages and the private register contents of VMs with different keys [31]. However, the nested paging is still in the control of the hypervisor, so the SEV VM's pages could be mapped to another VM or the hypervisor [38]. Although the private status and pages of VM is encrypted under different keys, SEV/SEV-ES lacks integrity protection, e.g., the hypervisor can perform memory replay attacks.

In 2020, AMD introduced SEV-SNP (Secure Nested Paging), further enhancing the protection for CVM from malicious hypervisor [42]. In SEV-SNP, an encrypted physical page can not be mapped to multiple owners by a malicious

hypervisor. This mechanism is realized by the introduction of a Reverse Mapping Table (RMP). The RMP is a metadata table managed by the AMD Platform Security Processor (AMD PSP). It records the ownership of each system physical page and dictates read, write and execute permissions for each VMPL. On every nested-page table walking, the RMP is consulted for the permission and ownership of each system physical memory page. A nested page fault (#NPF) will be raised on illegal access to physical pages. It is captured and handled by the hypervisor. The hypervisor manages VM Saved Areas (VMSAs) corresponding to four VMPLs. The access permission to the physical memory pages is restricted by configuring the VMPL of each page in the RMP. A vCPU can run in different VMPL contexts by switching the corresponding VMSAs with the help of the hypervisor.

Compared to page table protection, the RMP managed VMPL privilege is more flexible. Traditionally, we have NX, R/W, and U/S bits to denote non-executable, read-only, and user pages. However, the read and write permissions are not orthogonal in the page table. The RMP therefore separates the read and write access to guest physical pages, allowing one-way information flow between different VMPLs. Moreover, it separates the user and supervisor execution privilege for guest physical pages, preventing the code regions from being executed by unauthorized supervisor or user applications running in the lower VMPL. It is complementary to traditional SMEP (Supervisor Memory Execution Prevention) mechanism on x86 platform, combined with U/S and NX bits. The fine-grained privilege separation allows for strong execution protection.

2.2 Execute-only memory

Over the last thirty years, there has been substantial advancement in software attack and defense technology. The memory safety issue has been a long standing unsolved problem. Strategies like address space layout randomization (ASLR), stack canaries, and data execution prevention (DEP) have been used to address memory safety weaknesses. Despite these improvements, attackers persist in discovering new methods to exploit software vulnerabilities, underscoring the ongoing competition between attackers and defenders in the cyber-security realm.

The absence of code confidentiality enables attacker to gain arbitrary access to a running process by analyzing the code region for exploitable gadgets resides in the vulnerable software [49]. Various software and hardware mitigation have been proposed to enhance the code confidentiality through eXecute-Only Memory (XOM) [27]. XOM stands out as a straightforward and effective method that minimizes the attack surface and significantly raises the bar for attackers seeking to exploit software vulnerabilities. Through restricting access to code regions during runtime, XOM offers an additional security layer that prevents unauthorized access and manipulation of critical processes.

Previous researches have demonstrated the effectiveness XOM in strengthening software security [47]. By preventing access to code pages, attackers are hard to find gadgets for subsequent attacks. Numerous Protection Key Registers User-space (PKRU)-based sandbox frameworks have emerged recently [23,

41, 45]. However, due to the unprivileged nature of these hardware-based intra-process isolation mechanisms, they can be easily circumvented by exploiting the confused-deputy of the virtual-memory related syscalls [40]. Despite efforts to bolster the isolation between trusted and untrusted components, it is still considered to be weak in security-sensitive environments. Essentially, this mechanism offers safety rather than security.

Traditional page table-based memory protection is inadequate due to the absence of read/write access separation. The R/W bit on the x86 platform cannot be used to enable execute-only memory for vulnerable programs, allowing attackers to easily locate gadgets and compromise the software system in either the kernel or user-space. Even with PKRU-based execute-only protection, where read and write permissions are separated for each memory domain, it remains coarse-grained and can be circumvented in user-space [34].

2.3 Syscall filtering

Syscall filtering plays a vital role in safeguarding OS from vulnerable and malicious software [18,19,39]. Existing syscall filtering mechanisms often reside in the kernel space. Once bypassed, the entire system is in danger. The PKRU-based in-process sandboxes is lightweight and efficient in ensuring the security of software [26,47]. However, the non-privileged hardware intra-process isolation primitives can be easily bypassed through the confused deputy of the syscall [15,34]. In recent years, syscall filtering is widely used to ensure the security of such hardware-based intra-process isolation mechanisms [23,40,41]. However, most of these syscall filtering mechanism are within the kernel space. Once compromised, the entire system is in dangers. The lack of layered defence poses a great threat to the kernel. We argue that the user and supervisor separation is insufficient and exploitable. To reduce the attack surface, the untrusted programs should be isolated from direct interact to the guest OS within CVM.

2.4 Threat model

Our threat model aligns with that of confidential computing, where everything outside the virtual machine is considered untrusted. This includes the host OS. Our system relies on critical services from the guest OS, which is trusted. The proxy-kernel acts as a bridge between the confined processes and the guest OS, and it is also trusted. We assume that the applications are untrusted and may contain memory safety errors. Additionally, side channels and hardware attacks are outside the scope of our considerations. We operate under the assumption that the hardware functions as described in the official documentation. Furthermore, memory encryption and integrity protection measures are in place to provide an extra layer of security.

3 Design

We observe that the precise control over VMPL privilege on each guest physical page enables the execution of programs under a lower VMPL. However, merely possessing this control is insufficient to propose a secure isolated execution framework. The introduction of four permission bits in the VMPL mechanism addresses issues associated with traditional page table protection flags and is specifically tailored to ensuring the security of code running in the user and kernel space of the lower VMPLs. Therefore, to safeguard the guest OS, we introduce the Cabin framework, which confines untrusted programs to the user space of lower VMPL through fine-grained VMPL privilege management. To accomplish this, the architectural design is detailed as follows.

3.1 Overview

Fig. 1 presents an overview of the Cabin framework. We introduce a proxy-kernel within the lower VMPL to facilitate the scheduling of processes at lower VMPLs. The proxy-kernel directly monitors confined processes and mediates the communication between the guest OS and these processes. This mediation enables the application of flexible security policies before forwarding syscalls and exceptions to the guest OS. Consequently, it establishes a layer of defense against untrusted processes. The owner of the CVM is allowed to customize policies to monitor these processes without requiring intervention from the guest OS. This design ensures the flexibility in process monitoring and tracing.

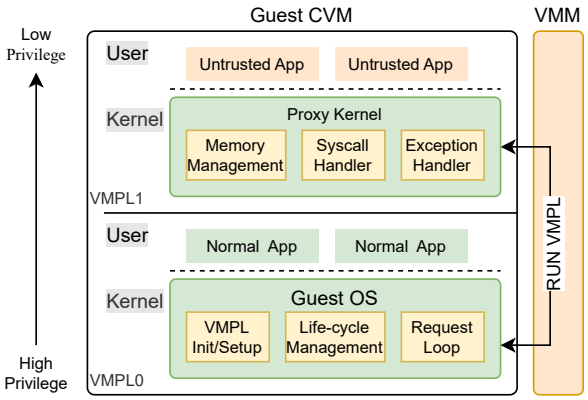

Fig. 1. An overview of the Cabin framework.

3.2 System design

The Cabin shields untrusted programs in the user-space of lower VMPL. We should ensure a secure and reliable environment for untrusted applications run-

ning at lower VMPLs. Managing runtime state of confined processes is crucial. To address this, we introduce a proxy kernel to serve these confined processes. The proxy kernel functions as an intermediary between the restricted processes and the underlying guest OS, managing syscalls, and interrupts on their behalf.

The system design of the Cabin framework consists of four main components: the life-cycle management of confined threads, the context switch, syscall routing, and exception model. Below, we elaborate on each aspect of the design.

Life-cycle management. The Cabin framework supports scheduling each thread independently to the user-space of lower VMPL. The life-cycle of each thread comprises three stages: creation, entry, and exit. The guest OS manages the life-cycle of the untrusted processes as illustrated in Fig. 1. During the initialization, the guest OS prepares the runtime environment for all lower VMPLs. Before entering the lower VMPL, the guest OS assigns a specific VMPL to each thread and synchronize the hardware state of the thread to the corresponding VMSA. Then, by requesting the hypervisor to execute in the specified VMPL, the current CPU directly switches to the corresponding VMPL and resumes the execution. Initially, Cabin enters the kernel mode of the lower VMPL, performing a series of initialization tasks for syscall and interrupt handling. Then it directly switches to the user-space, and continues the execution of the user thread. The proxy-kernel waits for syscall and interrupt events from the user-space, and forwards these events to the guest OS or handles event by itself. Upon receiving a request from the lower VMPL, the guest OS decides whether it is an interrupt or syscall event, and calls the corresponding handler in the guest OS. The request loop continues until receiving the exit and exit_group syscalls from the lower VMPL, the guest OS no longer schedules the thread to the lower VMPL. Finally, the guest OS releases the resource for confined processes.

Context switch. The guest OS manages the context switch of confined process as usual. Compared to normal context switch in the guest OS, the hardware state of the confined process is saved in the VMSA of the lower VMPL, which is allocated by the guest OS during initialization. Because the guest OS has direct access to the hardware state of all lower VMPLs, Cabin synchronizes these state from VMSA with guest OS managed Task Control Block (TCB) on context switch. Therefore, the guest OS just loads and restores the hardware task state for confined processes at a different place. To optimize resource utilization, Cabin supports all lower VMPLs to minimize contention for limited VMSA. The confined processes are assigned to different lower VMPLs, eliminating the need to restore context when the lower VMPL is not preempted by other processes.

Syscall routing. The syscall routing logic is outlined in Fig. 1. For confined processes running in the user space of lower VMPLs, syscalls are handled by the proxy-kernel before forwarded to the guest OS. By switching VMPL, the syscall arguments are automatically saved in the VMSA of the lower VMPL. The guest OS can directly access this hardware state. The result is returned to the proxy-kernel by modifying the VMSA of the lower VMPL. Meanwhile, certain syscalls can be directly handled by the proxy-kernel. To this end, we

just simulate the syscall and sysret semantics with VMPL switching, allowing syscalls to be handled by the guest OS as usual.

Exception model. The exception in the lower VMPL should be forwarded to the guest OS in principle. All necessary information is stored in the trap frame during a trap event, which will then be forwarded to the guest OS. Exceptions are managed in a standard manner. After handling the trap event, the guest OS requests the hypervisor to schedule the confined process. To reduce context switch, the proxy-kernel handles certain exceptions by itself. Exceptions are redirected to the guest OS as regular syscalls but are managed in a different setting. Handling exceptions involves changing the preempt mode and interrupt status of VMPL0 to ensure that the handler is invoked in a correct environment.

With the above design, we enable untrusted processes to be scheduled to the user-space of a lower VMPL, isolated from the guest OS with the VMPL hardware mechanism. The proxy-kernel mediates the communication between the untrusted programs and the guest OS. Unlike existing works, the guest OS in the Cabin framework manages all resources needed by the lower VMPLs. This innovative design brings numerous opportunities in the security aspect, which are detailed in the following sections.

3.3 Performance optimization

Asynchronous forwarding. Most kernel operations execute quickly, rendering it costly to forward syscalls and exceptions synchronously via VMPL switching. To improve the performance, Cabin incorporates an asynchronous forwarding mechanism into the proxy-kernel. With no barriers between threads in different VMPLs, this mechanism relies on shared-memory and spinlock-based cross-thread communication. During the initialization stage, Cabin initiates a service thread that waits for requests using a spinlock. Upon entering the lower VMPL, the proxy-kernel of the lower VMPL can utilize this interface to forward syscalls and interrupts to the service thread. Once the request is completed, the proxy-kernel returns the result to the confined process, which then resumes execution until the next syscall or interrupt occurs. Compared to other asynchronous forwarding mechanisms [33,48], Cabin directly intercepts the syscall and exception in the proxy-kernel, requiring no modification to the confined programs. The untrusted programs are not allowed to directly utilize this mechanism to bypass the proxy-kernel, reducing the attack surface at the user-space.

Self-managed memory. To mitigate the performance impact of expensive VMPL switching, Cabin further incorporates anonymous memory management into the proxy-kernel, allowing direct handling of virtual memory related syscalls on anonymous pages requirement. The physical pages are granted by the guest OS, and managed by the proxy-kernel directly. When needed, the proxy-kernel requests additional memory pages from the guest OS. These pages are allocated on demand for confined processes, with any page faults on these anonymous pages being handled by the proxy-kernel, bypassing the guest OS.

4 Case studies

With the proxy-kernel, Cabin framework enables a series of optimization and security mechanisms for confined processes. The case studies on the Cabin framework cover three main points: execute-only protection for untrusted processes and proxy-kernel, syscalls filtering for untrusted processes, and exceptions intercepting for flexible process monitoring and tracing. These studies showcase potential applications of the Cabin framework.

4.1 Execute-only protection

According to the official document [42], there are four distinct permission bits for each guest physical page: read, write, user, and super execution permissions. This approach is orthogonal and distinct from traditional page table flags, where read and write access are not independent. It adds an extra layer of protection against guest physical pages. Here we present two security enhancement mechanisms based on fine-grained management of VMPL privilege.

Firstly, we propose VMPL-enhanced XOM. The guest OS revokes the read access to the code regions and then assigns execution privilege to user or super-level based on security needs. By restricting execute-only VMPL privilege to the code pages, we prevent attackers from exploiting vulnerabilities both in the kernel and user-space of lower VMPL. Due to the privileged nature of VMPL mechanism, it overcomes the short comings that arises in most non-privileged hardware-based intro-process isolation mechanism, i.e., PKRU-based XOM.

Secondly, we introduce VMPL-enhanced cross-layer execute-only protection, serving as an enhanced SMEP mechanism. This is achieved through fine-grained separation of user and super execute privilege. As the VMPL further separates the execution privilege for user and kernel space, we can not only prevent the execution of untrusted user code in the kernel space, but also forbid the privileged code from being executed in user space even in the absence of U/S bit protection in the page table.

By utilizing the VMPL hardware mechanism, Cabin establishes a strict boundary between the kernel and user space at lower VMPLs. It enforces both intra-process isolation and cross-layer protection. It makes the attacker more difficult to exploit vulnerabilities at the lower VMPL. Overall, we utilizes VMPL to enable "one-way visibility" of a reference monitor, ensuring that code regions cannot be inspected and altered at the lower VMPL.

4.2 Process monitoring

Since the proxy-kernel mediates the communication between the guest OS and untrusted programs. It can directly handles the syscall and exception from user-space before forwarding to the guest OS. This mechanism can be leveraged to enhance the performance or track the execution of confined user programs.

Syscall filtering. Cabin introduces VMPL-enforced execute-only protection to reduce the attack surface for vulnerable programs. However, it is not sufficient

for malware. The syscall filtering can be leveraged as a layer of defense in the lower VMPL without intervention from the guest OS.

Process tracing. By intercepting the breakpoint exception, Cabin enable dynamic monitoring of untrusted programs without guest OS intervention, offering a flexible tracing mechanism. This allows us to utilize the hardware breakpoint based dynamic intercepting mechanism without relying on the guest OS. Similar to the kprobes mechanism in the Linux kernel, we enable automatic process tracing running in the lower VMPL. Additionally, dynamic instrumentation can be readily supported on Cabin for closed-source binaries.

Malware analysis. For malware where no source code can be accessed, the exception intercepting mechanism allow flexible security policies to be applied to each confined process without requiring intervention from guest OS. It is especially useful in analyzing the behaviour of malware. Since the policies are outside of the guest OS, modifying the security policy is made simple.

5 Implementation

The current implementation of the Cabin framework supports Linux running on AMD SEV-SNP enabled CPUs. It is based on the lasted infrastructure from AMD SEV[3]. To streamline the management of confined processes, Cabin consists of a kernel module and a proxy-kernel. The kernel module manages the life-cycle of the confined processes, while the proxy-kernel serves these processes in the lower VMPL. The kernel module comprises approximately 6600 lines of code (LoCs), the proxy-kernel has 11000 LoCs, and the musl-libc [4] contributes around 500 LoCs for GHCB protocol-based syscall forwarding mechanism.

Application interface. We offer two interfaces for applications that need confinement. The vmpl_init is utilized to setup the environment at the process level. The vmpl_enter_user is used to prepare thread-level resources and enter the lower VMPL. The thread can be scheduled independently to the lower VMPL. Besides, we introduce a preload library for unmodified binary programs. There is no need to modify or statically instrument the source code, greatly reducing the deployment effort.

5.1 Syscalls and interrupts handling

In the Cabin framework, the proxy-kernel directly handles the syscalls and the interrupts from the confined process. The forwarding mechanism follows standard GHCB protocol [5]. The MSR (Model-Specific Registers) protocol serves as a bootstrapping mechanism for GHCB protocol before GHCB registration. Once the GHCB is registered at the lower VMPL, Cabin directly shifts to the GHCB-based forwarding mechanism. To ensure the functionality and efficiency of syscalls and interrupts handling, the implementation of the Cabin framework

[3] https://github.com/AMDESE
[4] https://musl.libc.org

includes the following features: the vDSO support, asynchronous forwarding, and transparent debugging.

Syscall routing. Cabin supports anonymous memory management in the proxy-kernel. Certain virtual memory related syscalls, such as mmap, munmap, mprotect, and mremap can be handled by the proxy-kernel without VMPL switching. Unsupported syscalls are still forwarded to the guest OS. A simple filtering mechanism is also implemented in the forwarding logic, allowing intercept each syscalls independently with priority. To enforce syscall security, security policies can be enforced prior to entering the lower VMPL.

vDSO support. The vDSO (virtual Dynamic Shared Object) is a conventional mechanism that allows programs to make syscalls directly without transition to kernel mode [7]. It is a memory area used by the kernel to provide optimized versions of commonly used syscalls (i.e., clock_gettime). This improves performance by reducing the overhead of context switch. The vDSO is mapped into the address space of every user-space process, allowing programs to access it easily when making these syscalls. Cabin naturally supports such mechanism by allowing access to those memory pages at lower VMPL.

Asynchronous forwarding. To reduce the costly VMPL switching, the asynchronous forwarding mechanism is derived from SGX-HotCalls [48]. By removing the Intel SGX-related components, it seamlessly integrates with the Cabin framework. Unlike the original version, this mechanism is integrated into the syscall and interrupt handlers of the proxy-kernel. Currently, the Cabin framework supports asynchronous forwarding for syscalls, while exceptions and interrupts remain GHCB protocol-based synchronous forwarding mechanism.

Transparent debugging. Transparent debugging is essential in the Cabin framework for confined processes. It ensures seamless debugging capabilities for the lower VMPL. The hardware state of the lower VMPL is synchronized with the guest OS-managed TCB, encompassing debug registers, during context switches. The trap frame from the user-space of lower VMPLs is delivered to the guest OS to facilitate the handling of breakpoints and debug exceptions triggered at the lower VMPL, allowing transparent debugging for confined processes.

5.2 Dynamic VMPL management

It is crucial to adjust the VMPL permission of each physical memory pages for a confined process to run in the user space of the lower VMPL on AMD SEV-SNP platform. This process mainly includes intercepting syscalls and exceptions, as outlined below.

Table 1. System call categories.

Category	syscalls
Virtual Memory	mmap, mremap, munmap, brk, mprotect, pkey_mprotect, madvise, shmat, shmdt, remap_file_paegs, mlock, mlock2, mlockall

Syscall interposition. To update VMPL permission on time, we adjust the permission of the relevant physical pages after each system call and page fault, so that the process can be running at a lower VMPL. We identified several virtual memory-related syscalls (e.g., brk, mmap) in Table. 1. However, these syscalls do not always populate the page table due to lazy allocation. To streamline the process, we still traverse the page table and grant access to corresponding memory area. Despite being imprecise and inefficient, the evaluation shows a modest overhead through other optimizations.

The syscalls mentioned above typically accept memory address and length as arguments and can be easily monitored for VMPL management. However, certain other syscalls (e.g., read) implicitly alter the page tables by synchronizing memory contents between hardware storage and memory. In these cases, the kernel will inform subscribers before and after modifying the page table. We utilize such notification mechanism to adjust the VMPL permission.

Page fault interception. Due to the lazy-allocation and demand paging mechanism, we update the corresponding VMPL permission after the guest OS successfully handles the page fault on non-present and copy-on-write (COW) pages. However, it is not sufficient to run the process at lower VMPL. The kernel preallocates physical pages before actually accessing those pages due to the prefault mechanism. Therefore, we promptly adjust the VMPL permission for prefault pages. Otherwise, it may cause RMP permission violations caused by being unable to access these physical pages at a lower VMPL.

Notably, a better way to improve the performance is to grant the entire memory access rights to all lower VMPLs according to the firmware specification [6]. In this way, all guest physical pages are allowed to access at lower VMPL. However, it is still necessary to conditionally adjust the VMPL permissions for security. It is a complex task to track all updates to the page table of a process. Our prototype focuses on demonstrating the viability and security of running untrusted programs within the user-space of a lower VMPL. Therefore, we do not focus on a precise tracking mechanism in this work. However, a precise page table tracking mechanism can be realized with further efforts.

6 Performance Evaluation

In this section, we evaluate the performance of the Cabin framework. The evaluation is performed in a single-threaded environment. This includes using GHCB protocol and HotCalls to forward syscalls and page faults to the guest OS. Afterwards, we measure the performance on Nbench and WolfSSL benchmarks. The evaluation is performed on a dual-socket 3rd Gen AMD EPYC processor (codenamed Milan) with 128 logical cores and 64GB RAM, supporting the SEV-SNP technology. The host system operates QEMU 6.1.50 on Ubuntu 22.04 (kernel version 6.5.0-rc2-snp-host), while the VM is allocated with 64 vCPUs and 16GB RAM, running Ubuntu 22.04 (kernel version 6.5.0-snp-guest).

Syscall. Fig 2 depicts the time taken to execute each syscall 10,000 times under various conditions. The GHCB protocol-based forwarding mechanism incurs

more time consumption than the original syscall instruction. Employing the HotCalls mechanism for syscall forwarding shows a noticeable reduction in execution time compared to the GHCB protocol. However, HotCalls still lags behind in speed compared to the original syscall method due to its asynchronous nature, resulting in varying latency across syscalls. Notably, the dynamic VMPL management mechanism introduces significant overhead on read and mmap syscalls. Importantly, with Cabin supporting the vDSO mechanism, there is no impact on the clock_gettime syscall.

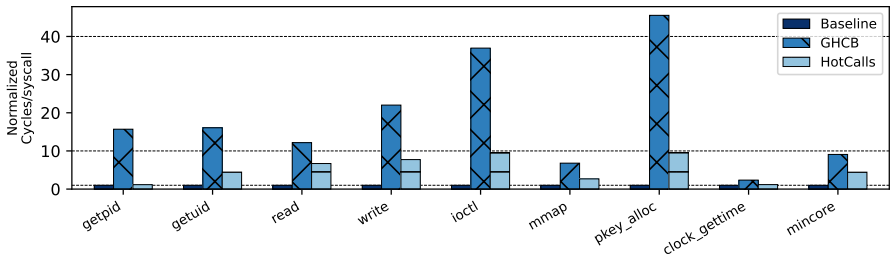

Fig. 2. The evaluation of syscall overhead in different scenarios.

Page fault. Table 2 presents the duration for handling page faults in different scenarios. When assigning 10000 private memory pages via mmap syscall, each memory page access prompts a page fault without preloading. Remarkably, the time taken to manage page faults is considerable, almost matching the overhead from GHCB protocol. In the lower VMPL, the page fault forwarding mechanism operates approximately three times as slowly as in the original user-space. Compared to the forwarding syscall, the page fault has a greater performance impact because it involves synchronizing the trap frame to guest OS. Forwarding certain page faults with HotCalls is possible, but the current implementation hasn't adopted a HotCalls-based forwarding mechanism.

Table 2. Delay in handling page exception in different scenarios.

	Baseline	VMPL-CPL0	VMPL-CPL3
page fault	13026	29627	29936

In the following, we evaluate the impact on classical performance benchmarks, showcasing the advantages of the Cabin framework.

Nbench [3]. Fig. 3 shows the evaluation of the Cabin framework on Nbench. This benchmark includes ten calculation-intensive tasks. We utilize proxy-kernel provided mmap and munmap syscalls for small-scale anonymous memory requirement. The GHCB-512 and 1024 indicate that the proxy-kernel manages 512 and 1024 memory pages continuously without guest OS intervention. It is evident that despite implementing the self-managed memory mechanism, there

is still an overall performance overhead. This is due to the necessity of forwarding all other syscalls and interrupts. Although Cabin supports vDSO based clock_gettime, it is still forwarded to the guest OS in Nbench. Nevertheless, as the proxy-kernel manages more physical pages, the performance impact notably decreases across most benchmarks. Additionally, there is a substantial performance enhancement observed in FP EMULATION and ASSIGNMENT when the proxy-kernel manages more memory pages.

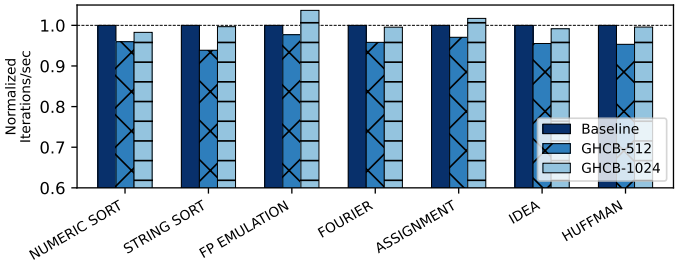

Fig. 3. The performance evaluation on Nbench.

WolfSSL [8]. We evaluate the Cabin framework on WolfSSL benchmark. This benchmark consists of evaluation on cryptography algorithms, such as encryption, decryption, digests, and signature verification. Here, the anonymous memory allocation is also handled by the proxy-kernel rather than the guest OS. As illustrated in Figure 4, over a half of tasks perform significant better than baseline, while the other remains an overall performance overhead of about 1% to 10%. This indicates that in certain cases, using autonomous management of anonymous page memory allocation can bring performance improvements.

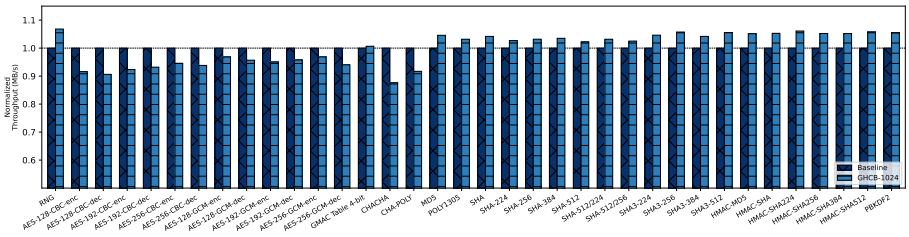

Fig. 4. The performance evaluation on WolfSSL benchmark.

In above evaluations, the Cabin incurs significant overhead on each syscall due to costly VMPL switching. Both syscall and exception forwarding require more cycles when the process is scheduled to the lower VMPL. However, Cabin incurs modest overhead in most cases on Nbench and WolfSSL benchmarks. The performance impact can be reduced significantly with asynchronous Hot-

Calls mechanism and self-managed memory mechanisms, thereby outstanding the advantage of confined execution of Cabin.

7 Discussion

7.1 Advantages

Defense in-depth. Compared to traditional sandbox frameworks, Cabin shields untrusted processes to the lower VMPL within the same CVM, preventing vulnerabilities from malicious exploits with VMPL-enhanced execute-only memory and cross-layer execution prevention. By isolating processes at the user space of lower VMPL, Cabin provides layered protection for the guest OS within the CVM. This framework allows for flexible process monitoring and tracking of untrusted legacy applications without requiring intervention from the guest OS.

Compatibility. One advantage of the Cabin is the compatibility with other frameworks. In the Secure Virtual Machine Service Module (SVSM) [4], the guest OS operates in lower VMPL other than VMPL0. Our schema naturally aligns with this framework. In this case, the process is scheduled to at most two VMPLs. To accommodate other frameworks like Veil [9], Cabin require at least one VMPL lower than the guest OS. The trusted services and enclaves are positioned at higher VMPLs, while the untrusted processes are scheduled to a lower VMPL. However, Veil positions the guest OS at the lowest VMPL, rendering it challenging to integrate the Cabin framework. Cabin is also naturally supports PKRU-based sandbox frameworks [26,40,46,47], which can still be used to enhance intra-process isolation for confined processes at lower VMPLs.

Alternative design. Compared to safeguarding the proxy-kernel with VMPL, an alternative design to the proposed execution protection mechanism is based on the SVSM framework. This approach restricts read access to the code regions of guest OS. However, there exist numerous code regions necessitating read access and even modification rights. Modifying these code regions allows the kernel to dynamically change behavior during runtime. Consequently, it is less practical than protecting a minimal proxy-kernel in lower VMPL.

7.2 Limitations

One drawback of the Cabin framework is the performance impact. The VMPL switching leads to delays in syscall and exception handling. The imprecise page tracking for dynamic VMPL management results in extra overhead. Currently, Cabin does not well support thread migration across CPUs. Because the GHCB is not shared among CPU cores, Cabin binds the thread to one CPU, limiting task scheduling flexibility. Other constraints involve multi-threading and multi-processing. Although Cabin supports preemptive scheduling, the incomplete support for fork and clone syscalls limits the application to single-thread environment. Nevertheless, it is possible to schedule child threads to the user space of lower VMPLs while keeping the main thread in the original user space.

Most issues can be solved with further effort, but the delays from VMPL switching remain a challenge to efficiently address.

7.3 Extending to other CVM platforms

Although the Cabin framework is based on the latest feature from AMD SEV-SNP, it can be extended to other CVM platforms such as Intel TDX and ARM CCA. By introducing a proxy-kernel within an isolated CVM, we shield the guest OS from potential threats posed by untrusted processes. The communication between confined processes and the guest OS is managed by the proxy-kernel and the trusted hypervisor located outside the CVM. As for the Intel TDX, the TDX Module facilitates communication between the proxy kernel and the guest OS across different CVMs. Meanwhile, in ARM CCA, the Realm Management Monitor (RMM) oversees the interaction between the proxy-kernel and the guest OS. In both scenarios, trusted hypervisors like Intel TDX Module and ARM RMM play a crucial role in establishing a secure channel between different CVMs.

Recently, ARM CCA introduced support for different planes within a CVM [10]. Each plane is essentially a separate VM, with a shared guest physical address space. Plane 0 holds more privilege and can host a paravisor to control switches between planes and restrict other planes' memory access. Similarly, less privileged planes can be used to shield untrusted applications from guest OS.

8 Related Work

AMD SEV-SNP and VMPL. Various researches are underway to enhance the security of AMD SEV [11,28,38]. The SVSM [4] framework leverages VMPL0 to protect secure service from untrusted guest OS. Hecate [21] uses VMPL0 as a trusted L1-hypervisor to facilitate communication between the guest OS and untrusted hypervisor. SVSM-vTPM [32] is a security-enhanced vTPM based on the SVSM framework, leveraging VMPL0 to isolate the virtual TPM (vTPM) from the guest OS, ensuring the integrity of vTPM's functions. CoCoTPM [36] reduces the trust needed towards the host and hypervisor by running a vTPM in an encrypted VM using AMD SEV. Honeycomb [29] is a secure GPU computation framework that runs a validator within VMPL0, which inspects the binary code of a GPU kernel to ensure that every memory instruction in the kernel can solely reach designated virtual address space, utilizing static analysis. The mushroom [2] framework runs integrity protected workloads based on AMD's SEV-SNP technology, which could be the basis of a secure remote build system. Veil [9] is a service framework providing secure enclave and services for process and the guest OS respectively. In general, these works follow traditional threat model of confidential computing, and do not focus on untrusted applications in CVM, while the Cabin framework protects the guest OS by confining untrusted programs to the user space of lower VMPLs.

Execute-only memory. Execute-only memory (XOM) [12,27] is an effective method in software security. PicoXOM [43] is an efficient XOM mechanism based

on ARM's Data Watchpoint and Tracing unit for embedded systems. Nojitsu [35] leverages XOM-Switch to enforce execute-only permission for static code regions in JIT. SECRET [49] protects COTS binaries from disclosure-guided code reuse attacks, while MonGuard [47] applies PKRU-based XOM protection to the multi-variant execution (MVX) monitor. IskiOS applies XOM to safeguard code pages of a unikernel [22]. Cerberus [46] is a notable sandbox framework that protect the reference monitor with PKRU-based XOM. To the best of our knowledge, the fine-grained control over VMPL permissions has not been utilized to enhance execute-only protection for untrusted programs in previous studies.

Intro-process isolation. The lightweight PKRU-based intra-process isolation mechanism is also a hot research topic in recent years [23, 26, 45]. Various research efforts have been made to enhance the security of PKRU-based isolation mechanisms [40, 46]. However, its unprivileged nature makes it susceptible to bypassing in user-space through side-effects or confused-deputy issues from syscalls [34]. Attackers can exploit this vulnerability by constructing unsafe instruction sequences to gain unauthorized access to sensitive data and code [15]. Such systems require complex syscall filtering policy to prevent WRPKRU exploitation and enforce the security of their sandbox [40].

Syscall filtering. Securely confining untrusted legacy applications has been a long-standing challenge for the past decades [20, 24, 37]. The syscall filtering plays a crucial role in traditional software system security [18, 19, 39], including container security [44]. The syscall filtering is also widely applied in PKRU-based intro-process isolation mechanism [14, 40]. PHMon [17] and FlexFilt [16] introduces new hardware design for efficient syscall filtering and process monitoring on RISC-V platform. Nevertheless, due to limited privilege separation, these mechanisms still confine to conventional user and kernel separation.

9 Conclusion

Cabin is an isolated execution framework that effectively shields untrusted programs from guest OS within CVM. By introducing a trusted proxy-kernel for untrusted applications, Cabin enables efficient and flexible process monitoring and tracing, enhancing a layered security defense outside of the guest OS. By utilizing VMPL-enforced execute-only protection, Cabin making it harder for vulnerabilities to be exploited at lower VMPL. With fine-grained control over VMPL execution privilege, Cabin further isolates the proxy-kernel and confined processes, strengthening the cross-layer isolation between the user and kernel space of lower VMPLs. To reduce the performance impact, Cabin integrates asynchronous forwarding mechanism and self-managed memory allocation in the proxy-kernel. In essence, the Cabin framework can be generalized to other commercial CVM platforms as well. The evaluation results on Nbench and WolfSSL benchmarks demonstrate modest performance overhead for confined processes.

Acknowledgment. This work was supported by National Natural Science Foundation of China (Grant No.62272452). Corresponding author: Wenhao Wang (wangwenhao@iie.ac.cn).

References

1. Arm confidential compute architecture. https://developer.arm.com/documentation/den0125/0300/. Referenced December 2024 (2024)
2. Freax13/mushroom: Run integrity protected workloads in a hardware based trusted execution environment. https://github.com/Freax13/mushroom (2024)
3. Linux/unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html (2024)
4. Secure vm service module for sev-snp guests. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf (July 2024)
5. Sev-es guest-hypervisor communication block standardization. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf (2024)
6. Sev secure nested paging firmware abi specification. https://www.amd.com/system/files/TechDocs/56860.pdf. Referenced December 2024 (2024)
7. vdso - wikipedia. https://en.wikipedia.org/wiki/VDSO (2024)
8. Wolfssl and wolfcrypt benchmarks — embedded ssl/tls library. https://github.com/wolfSSL/wolfssl (2024)
9. Ahmad, A., Ou, B., Liu, C., Zhang, X., Fonseca, P.: Veil: A protected services framework for confidential virtual machines. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4. pp. 378–393 (2024)
10. Arm: Evolution of the arm confidential compute architecture. https://www.youtube.com/watch?v=1AsvIt7bSLY (2024)
11. Buhren, R.: Resource control attacks against encrypted virtual machines. Ph.D. thesis, Dissertation, Berlin, Technische Universität Berlin, 2022 (2022)
12. Chen, Y., Zhang, D., Wang, R., Qiao, R., Azab, A.M., Lu, L., Vijayakumar, H., Shen, W.: Norax: Enabling execute-only memory for cots binaries on aarch64. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 304–319. IEEE (2017)
13. Cheng, P.C., Ozga, W., Valdez, E., Ahmed, S., Gu, Z., Jamjoom, H., Franke, H., Bottomley, J.: Intel tdx demystified: A top-down approach. arXiv preprint arXiv:2303.15540 (2023)
14. Christou, G., Ntousakis, G., Lahtinen, E., Ioannidis, S., Kemerlis, V.P., Vasilakis, N.: Binwrap: Hybrid protection against native node. js add-ons. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security. pp. 429–442 (2023)
15. Connor, R.J., McDaniel, T., Smith, J.M., Schuchard, M.: {PKU} pitfalls: Attacks on {PKU-based} memory isolation systems. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1409–1426 (2020)
16. Delshadtehrani, L., Canakci, S., Blair, W., Egele, M., Joshi, A.: Flexfilt: towards flexible instruction filtering for security. In: Proceedings of the 37th Annual Computer Security Applications Conference. pp. 646–659 (2021)
17. Delshadtehrani, L., Canakci, S., Zhou, B., Eldridge, S., Joshi, A., Egele, M.: {PHMon}: A programmable hardware monitor and its security use cases. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 807–824 (2020)
18. DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., Kemerlis, V.P.: Sysfilter: Automated system call filtering for commodity software. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). pp. 459–474 (2020)

19. Gaidis, A.J., Atlidakis, V., Kemerlis, V.P.: Sysxchg: Refining privilege with adaptive system call filters. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1964–1978 (2023)
20. Garfinkel, T., Pfaff, B., Rosenblum, M., et al.: Ostia: A delegating architecture for secure system call interposition. In: NDSS (2004)
21. Ge, X., Kuo, H.C., Cui, W.: Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 1231–1242 (2022)
22. Gravani, S., Hedayati, M., Criswell, J., Scott, M.L.: Fast intra-kernel isolation and security with iskios. In: Proceedings of the 24th international symposium on research in attacks, intrusions and defenses. pp. 119–134 (2021)
23. Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M.L., Shen, K., Marty, M.: Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 489–504 (2019)
24. Ibrahim, K.A.: Secure isolation and migration of untrusted legacy applications (2021)
25. Intel: Intel software guard extensions developer guide. https://www.intel.com/content/www/us/en/content-details/671334/intel-software-guard-extensions-intel-sgx-developer-guide.html. Referenced December 2022.
26. Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Franz, M.: Pkru-safe: automatically locking down the heap between safe and unsafe languages. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 132–148 (2022)
27. Kwon, D., Shin, J., Kim, G., Lee, B., Cho, Y., Paek, Y.: {uXOM}: Efficient {eXecute-Only} memory on {ARM}{Cortex-M}. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 231–247 (2019)
28. Li, M., Wilke, L., Wichelmann, J., Eisenbarth, T., Teodorescu, R., Zhang, Y.: A systematic look at ciphertext side channels on amd sev-snp. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 337–351. IEEE (2022)
29. Mai, H., Zhao, J., Zheng, H., Zhao, Y., Liu, Z., Gao, M., Wang, C., Cui, H., Feng, X., Kozyrakis, C.: Honeycomb: Secure and efficient {GPU} executions via static validation. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). pp. 155–172 (2023)
30. Mattioli, M.: Rome to milan, amd continues its tour of italy. IEEE Micro **41**(4), 78–83 (2021)
31. Mofrad, S., Zhang, F., Lu, S., Shi, W.: A comparison study of intel sgx and amd memory encryption technology. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–8 (2018)
32. Narayanan, V., Carvalho, C., Ruocco, A., Almási, G., Bottomley, J., Ye, M., Feldman-Fitzthum, T., Buono, D., Franke, H., Burtsev, A.: Remote attestation of sev-snp confidential vms using e-vtpms (2023)
33. Orenbach, M., Lifshits, P., Minkin, M., Silberstein, M.: Eleos: Exitless OS services for SGX enclaves. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 238–253 (2017)
34. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 241–254 (2019)

35. Park, T., Dhondt, K., Gens, D., Na, Y., Volckaert, S., Franz, M.: Nojitsu: Locking down javascript engines. In: Proceedings 2020 Network and Distributed System Security Symposium. Internet Society (2020)
36. Pecholt, J., Wessel, S.: Cocotpm: Trusted platform modules for virtual machines in confidential computing environments. In: Proceedings of the 38th Annual Computer Security Applications Conference. pp. 989–998 (2022)
37. Potter, S., Nieh, J., Selsky, M.: Secure isolation of untrusted legacy applications. In: LISA. vol. 7, pp. 1–14 (2007)
38. Qin, H., Song, Z., Zhang, W., Huang, S., Yao, W., Liu, G., Jia, X., Du, H.: Protecting encrypted virtual machines from nested page fault controlled channel. In: Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy. pp. 165–175 (2023)
39. Rajagopalan, V.L., Kleftogiorgos, K., Göktas, E., Xu, J., Portokalidis, G.: Syspart: Automated temporal system call filtering for binaries. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1979–1993 (2023)
40. Schrammel, D., Weiser, S., Sadek, R., Mangard, S.: Jenny: Securing syscalls for {PKU-based} memory isolation systems. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 936–952 (2022)
41. Schrammel, D., Weiser, S., Steinegger, S., Schwarzl, M., Schwarz, M., Mangard, S., Gruss, D.: Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1677–1694 (2020)
42. Sev-Snp, A.: Strengthening vm isolation with integrity protection and more. White Paper, January p. 8 (2020)
43. Shen, Z., Dharsee, K., Criswell, J.: Fast execute-only memory for embedded systems. In: 2020 IEEE Secure Development (SecDev). pp. 7–14. IEEE (2020)
44. Song, S., Suneja, S., Le, M.V., Tak, B.: On the value of sequence-based system call filtering for container security. In: 2023 IEEE 16th International Conference on Cloud Computing (CLOUD). pp. 296–307. IEEE (2023)
45. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: Erim: Secure, efficient in-process isolation with protection keys. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1221–1238 (2019)
46. Voulimeneas, A., Vinck, J., Mechelinck, R., Volckaert, S.: You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 266–282 (2022)
47. Wang, X., Yeoh, S., Olivier, P., Ravindran, B.: Secure and efficient in-process monitor (and library) protection with intel mpk. In: Proceedings of the 13th European workshop on Systems Security. pp. 7–12 (2020)
48. Weisse, O., Bertacco, V., Austin, T.: Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. ACM SIGARCH Computer Architecture News **45**(2), 81–93 (2017)
49. Zhang, M., Polychronakis, M., Sekar, R.: Protecting cots binaries from disclosure-guided code reuse attacks. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 128–140 (2017)