SECvma: Virtualization-based Linux Kernel Protection for Arm

Teh Beng Yen
National Taiwan University
r12922178@csie.ntu.edu.tw

Joey Li
National Taiwan University
r10944062@csie.ntu.edu.tw

Shih-Wei Li
National Taiwan University
shihwei@csie.ntu.edu.tw

***Abstract*—A rootkit or an attacker that exploited a single vulnerability in a monolithic OS kernel like Linux could obtain full authority over the system. We introduce SECvma, a new system with Linux kernel protection for Arm-based platforms. SECvma employs a virtualization-based approach to transparently protect the kernel's code integrity in its lifetime. SECvma proposes a new design that extends current Linux KVM-based confidential virtual machine (CVM) frameworks to provide standalone Linux kernel protection with modest effort while preserving the safety of CVMs. SECvma leverages Arm's hardware virtualization extensions and addresses their limitations in supporting kernel protection. SECvma incorporates novel optimizations to reduce the overhead from the virtualization-based approach. SECvma significantly enhances Linux's security while retaining its performance efficiency and standard features, including dynamic kernel module loading and kernel page table isolation (KPTI).**

***Index Terms*—Operating Systems, Security, Virtualization**

I. Introduction

Monolithic OS kernels have become increasingly complex in satisfying escalating demands for functionality and performance. Linux, for example, has been deployed to heterogeneous computing environments. Linux has more than a million lines of code in its codebase. The growing complexity resulted hundreds of new common vulnerabilities and exposures (CVEs) are reported each year [1]. Monolithic kernels are a prime target for attackers. An attacker that installed kernel rootkits or exploited a kernel vulnerability could obtain full authority over the system to control and access all resources, compromising user safety.

Arm processors have experienced widespread adoption across computing platforms, including mobile and embedded devices, personal computers, and cloud servers [2]–[4]. As these different Arm-based environments widely adopt Linux, it is crucial to secure such deployments. This work introduced SECvma (Kernel Security-Enhanced CVM framework for Arm) , a new system that enhances the security of Linux-based Arm platforms. SECvma employs a virtualization-based approach to protect Linux's code integrity of its lifetime against a strong attacker who obtains kernel privileges. SECvma leverages Arm's virtualization extensions (VE) features commonly available across Arm-based platforms. Employing a virtualization approach allows SECvma to protect Linux with minimal instrumentation, preserving Linux's functionalities.

Confidential computing frameworks have been introduced to support confidential VMs (CVMs). Mechanisms were employed to protect CVMs from host attackers who obtain hypervisor privileges. Recently, Arm has announced hardware support for Armv9.2 processors to support the Confidential Compute Architecture (CCA) [5]. However, to our best knowledge, no existing hardware implementation supports CCA. The CVM frameworks [6]–[8] that support the existing Arm hardware rely on a software-based security monitor that leverages Arm VE to protect CVMs.

We make the following observations about these CVM frameworks. First, the existing CVM frameworks for Arm focus on extending KVM to protect CVMs from potentially compromised hypervisor hosts, which includes their full Linux kernels. Secondly, the security monitor in these frameworks provides essential functionalities, such as page table management and memory access controls that could be utilized for implementing kernel protection mechanisms. Building on these observations, we introduced a new system called SECvma that extends the security monitor in the KVM-based CVM frameworks for Arm to protect Linux's code integrity throughout its lifetime, independent of CVM presence. Specifically, SECvma protects KVM's host Linux, the bare-metal Linux that serves both host applications and VM functionality. SECvma prevents attackers from modifying the existing kernel code or injecting new code. It also ensures attackers cannot manipulate page tables or systems control registers to compromise kernel code integrity. Further, SECvma protects against kernel rootkits, ensuring only authenticated kernel modules can be installed and executed.

To prototype SECvma, we extended our previous work, SeKVM [7], [8], that hosts CVMs on Arm. The resulting SECvma prototype extended SeKVM's security monitor to enforce code integrity protection of the integrated Linux kernel. SECvma reuses the functionalities of SeKVM's monitor to simplify the implementation efforts required for kernel protection. Our virtualization-based approach avoids intrusive kernel instrumentation and preserves CVM protection and compatibility with Linux's standard features, such as dynamic module loading and KPTI [9]. The resulting SECvma implementation requires 4,355 LOC of changes to SeKVM, mostly to extend its security monitor to provide kernel protection.

The SECvma prototype addresses challenges that Arm VE hardware limitations pose in supporting kernel protection. Moreover, a virtualization-based approach could incur a high-performance overhead [10], [11]. For example, Arm platforms

may present a small TLB — using S2PTs can cause TLB contention, resulting in performance degradation. Furthermore, imposing system register protection causes a performance slowdown in Linux when utilizing features such as KPTI. To address the performance issues, SECvma incorporates a novel huge page optimization that addresses the TLB contention incurred by the extra level of page table translations. In addition, SECvma optimizes KPTI without compromising register protection. The evaluation results showed that SECvma retains Linux's standard features and performance while enhancing the security of Linux-based platforms on Arm.

SECvma is available at https://github.com/ae-acsac24-44/acsac24-paper240-ae.

II. Background

Introduction to Arm. Arm introduced Virtualization Extensions (VE) to support unmodified guest kernels to run in VMs. Arm VE adds a higher privileged Hyp mode (EL2) on top of the existing kernel (EL1) and user (EL0) mode to run hypervisors. Arm VE provides EL2 banked registers to allow the hypervisor to execute in an isolated address space from EL0 and EL1. It allows the hypervisor running in EL2 to configure EL2 registers to control the behaviors of the software running EL0 and EL1. Arm VE supports stage 2 memory translation to support nested paging. Arm provides an IOMMU, SMMU [12], to address DMA attacks.

When nested paging is effective, Arm's MMU first walks the stage 1 page tables (S1PTs) managed by VMs to translate guest virtual addresses (GVAs) to the intermediate or guest physical addresses (IPAs/GPAs). It traverses the stage 2 page tables (S2PTs) managed by the hypervisor to translate the IPAs/GPAs to the machine physical addresses (PAs). Linux for Arm64 uses four levels S1PTs and S2PTs: the top level is the page global directory (PGD). Below that comes the page upper directory (PUD), page middle directory (PMD), and page table entry (PTE).

Arm provides two S1PT base registers, TTBR0_EL1 and TTBR1_EL1. Each register specifies the base physical address of a PGD, which provides translation for different virtual address ranges. In Linux, the former translates user space virtual addresses, and the latter is used to translate addresses for the kernel space memory. User space programs use TTBR0_EL1, while the kernel uses both tables. These two registers also contain bits that specify the current Address Space Identifier (ASID) used to tag TLB entries. The Arm VE introduced the VTTBR_EL2 register to store the base of the S2PT. Figure 1 shows the virtual memory configuration used by Linux for Armv8. Arm supports various translation granules, including 4K, 2M, and 1G bytes (4KB, 2MB, 1GB). On Arm, the access permission of a CPU's code execution or a load/store from/to a given memory region is derived by combining the permissions set in all page entries of both stages. The MMU ensures that the CPU's memory operations satisfy the permission requirements. For instance, consider a page is set readable-writable (RW) in the S1PT and read-only (RO) and

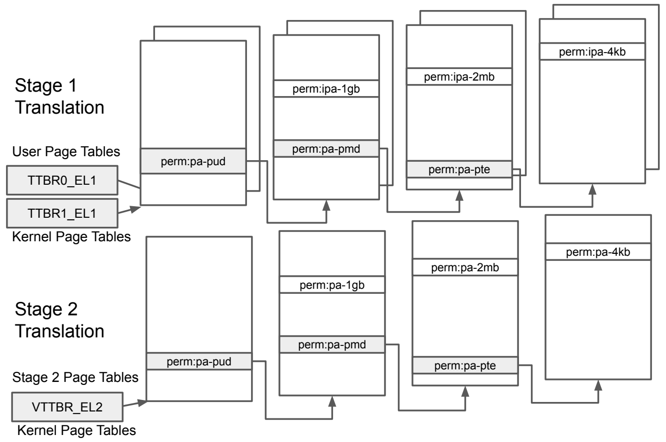

Fig. 1: Linux Armv8 Virtual Memory Systems

execute-never (XN) in the S2PT, the resulting permissions of the translation are RO and XN.

Arm VE traps page faults during stage 2 memory translation to EL2. This allows the hypervisor to handle the faults caused by either accesses to unmapped memory regions or permission violations. Page faults resulting from stage 1 translation are handled by the OS kernel running in EL1.

Similar to other implementations, Arm's TLB caches recently accessed page table translations in the MMU. When a processor makes a memory access, the MMU checks if the access can be satisfied by a cached translation in the TLB. If the requested address translation hits the TLB, the MMU retrieves the translated address from the TLB. If not, the MMU performs the page table walk to translate the intended address. The result of the page table walk can be refilled in the TLB for possible reuse if the walk does not cause a page fault. Arm's TLB caches single-stage, two-stage, or partial single-stage page table walk results. Two-stage paging translation consumes more entries and stresses the TLB. KVM uses huge page (2MB or 1GB) mappings to reduce the TLB pressure.

Arm-based Confidential Virtual Machines. The recent surge in confidential computing aims to isolate confidential VMs (CVMs) from a compromised hypervisor on the same host that could encompass a full OS kernel. Confidential computing frameworks on Arm [5]–[7], [13], [14] rely on a security monitor to protect CVMs. Arm has announced the Confidential Compute Architecture (CCA) [5] for Armv9.2 processors. To our knowledge, no existing hardware implementation supports CCA. To support CVMs on the current Arm hardware, SeKVM [7], [8] and pKVM [6] retrofit KVM [15] into an untrusted host that includes the Linux kernel and a trusted security monitor. The monitor isolates the host from accessing CVM data. The secure KVM implementations leverage Arm VE to isolate the security monitor in EL2 to utilize Arm VE features and deprivilege the host Linux in EL1. The monitor interposes CVMs' exits and stores their CPU registers in its private memory that the host cannot access. It lets the host Linux manage S1PTs and leverages Arm's

S2PT to restrict the host's memory access. The output of S1PTs (denoted as `PA`) is translated via the host S2PT when the host is running. The monitor uses an identity map in the host's S2PT to translate a `PA` to an identical address. It tracks the ownership of every physical page and ensures that the host's S2PT only maps to free memory but not CVMs' private memory. Finally, the monitor leverages the SMMU to protect VM memory against DMA attacks.

III. Threat Model and Assumption

SECvma protects the code integrity of the Linux kernel. We assume the system is initially benign but could be later compromised by a remote attacker, including administrators with remote access to the hardware. The attacker could exploit zero-day bugs from the Linux kernel. We assume that the attacker gains permission to write to arbitrary memory and aims to overwrite the existing Linux code, emit new code, or manipulate kernel page tables [16] to modify the access permissions or page mapping in page table entries. The attacker could control devices to perform DMAs to arbitrary memory. The attacker could also program Arm's system registers to control the MMU. Further, we assume the attacker also aims to install kernel rootkits to run malicious code with kernel privileges. We also assume the attacker attempts to conduct the ret2usr attack [17]. Protection against data-only and code-reuse attacks is out of scope. Mechanisms to secure control flow integrity [18]–[20] or compartmentalization [21] could be employed to defend against these attacks. We assume SECvma's TCB, KPCore, and hardware is trusted and bug-free. Denial of service attacks, side channels, and physical attacks against Linux are excluded from the threat model. Our work retains CVM protection of existing Arm-based frameworks. We make the same assumptions about attackers against CVMs as previous work [6], [7].

IV. SECvma Architecture

To enhance the safety of the Linux kernel running on Arm-based systems, SECvma protect Linux kernel code integrity throughout its lifetime. SECvma ensures the kernel only runs approved code. It prevents injected code or rootkits from being executed or installed to kernel memory. SECvma further ensures that attackers cannot corrupt Arm's system registers to manipulate Linux's virtual memory protection mechanisms. Instead of adopting a clean-slate approach, SECvma aims to extend the existing secure Linux KVM-based hypervisor that hosts CVMs for Arm platforms for standalone Linux kernel code integrity protection. As shown in Figure 2, SECvma incorporates a trusted KPCore to provide kernel protection. KPCore encompasses the trusted security monitor from the KVM-based CVM frameworks. KPCore builds on the monitor's features to protect Linux's code integrity.

SECvma leverages Arm VE features to transparently protect Linux with modest instrumentation so it can maintain compatibility with existing Linux functionalities. SECvma isolates KPCore in EL2 from Linux that runs in a less privileged EL1 mode. KPCore manages S2PTs to restrict Linux's memory access, preventing attackers who control the kernel from overwriting exiting kernel code or executing unverified kernel modules. KPCore traps attackers' illegal memory access and updates to Arm's system registers to EL2 to validate and enforce protection policies.

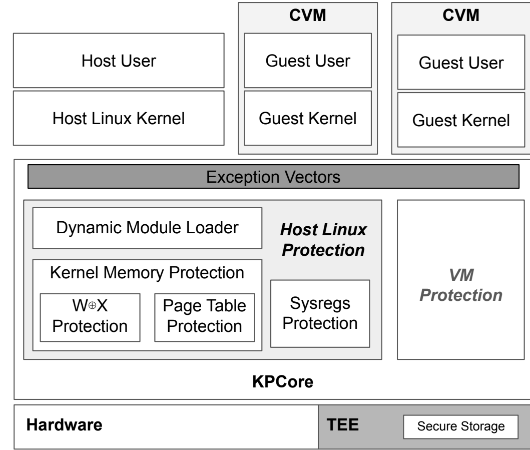

Fig. 2: SECvma System Architecture

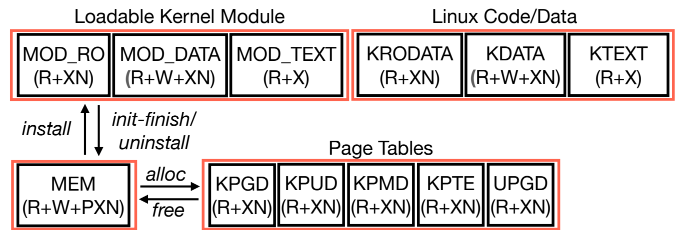

Fig. 3: Memory Usage and Access Permissions

A full Linux kernel and KPCore are linked into a single binary. SECvma relies on hardware secure boot such as Unified Extensible Firmware Interface (UEFI) firmware and its signing infrastructure with a hardware root of trust. The SECvma binary is signed and verified using secure storage keys to guarantee that a trusted SECvma binary is loaded. SECvma relies on the host Linux to bootstrap the hardware and install KPCore in EL2 early in the boot process. KPCore gains full hardware control after the installation completes. It ensures the Linux host can never turn off its protection.

A. Kernel Memory Protection

SECvma prevents attackers from modifying the existing code and executing maliciously injected code. SECvma leverages functionalities of the secure KVM to protect the Linux kernel's memory. It extends the integrated monitor's memory protection features, which restricts the host's memory access via the host S2PT, with host kernel memory protection. Similarly, the same host S2PT is used throughout the host's execution in EL0 or EL1. KPCore allocates S2PTs from its private memory that an attacker cannot access. Specifically, KPCore enforces ***(1) memory usage tracking***, ***(2) kernel and user page table protection***, and ***(3) DMA protection***.

1) Memory Usage Tracking: SECvma tracks Linux's memory usage and enforces access permissions. Figure 3 shows SECvma permission assignment to memory with all usage types. KPCore configures the access permission bits in page table entries according to the intended types. SECvma leverages S2PT to enforce the permission transparently to Linux.

SECvma enforces the "W⊕X" policy to achieve Data Execution Prevention [22] (DEP) at the hypervisor level to prevent Linux from executing code from writable memory. SECvma identifies all pages of the kernel text (`KTEXT`) and grants them read and execute (R+X) permission. For pages that belong to the kernel's data (`KDATA`) and rodata (`KRODATA`) section, SECvma enforces the execute never (XN) permission. SECvma write-protects memory that contains page tables with (R+XN) permission. Dynamic module loading poses unique challenges to the permission enforcement scheme. Linux loads the module's contents from the file system to memory and executes the module. As shown Figure 3, SECvma sets permissions accordingly for the module's text (`MOD_TEXT`), data (`MOD_DATA`), and rodata (`MOD_RO`) section to allow Linux to execute an authenticated module. We discuss SECvma 's support for loadable kernel modules further in Section IV-B. For the rest of the memory in the system (`MEM`), SECvma grants them read-write and privilege execute-never (R+W+PXN) permission. These pages contain the memory in the kernel data allocated from the heap and stack and the memory that Linux allocates to user applications and services. Enforcing the PXN permission prevents a privileged attacker from executing code injected into these regions.

2) Kernel page table protection: Attackers could exploit memory safety bugs in Linux to corrupt kernel page tables and tamper with the kernel space memory mappings. KPCore write-protects all kernel page tables (see Figure 3) in the host S2PT. Linux's writes to page tables trap to EL2. KPCore handles the write faults and validates if the update is legitimate; if yes, it performs the write on behalf of Linux to the respective page table; otherwise, KPCore rejects the illegal write. SECvma enforces the following policies to protect kernel page table updates.

P1: Kernel page tables structured as Directed Acyclic Graph (DAG). SECvma allows Linux to update kernel page tables during runtime to allocate new kernel page tables (from NULL to a new mapping) while ensuring their structure remains a DAG. SECvma allows multiple entries from the same page table level to point to the same next-level table. This means a given leaf or non-leaf (except the root) page table can have multiple parents. In addition, SECvma enforces page tables from the DAG to follow a strict hierarchical ordering requirement. As shown in Figure 1, a given non-leaf page table (e.g., PGD, PUD, PMD) can map to a next-level page table or to a memory page; the latter concludes a translation. For the former case, SECvma ensures that entries from a given non-leaf table only map to a strictly lower-level table in the hierarchical order. That is, PGD entries could only map to a PUD, PUD entries could only map to a PMD, and PMD entries could only map to a PTE.

P2: Restricting Updates to existing kernel page table entries. SECvma permits Linux updates to existing page table entries in two cases. First, SECvma only allows Linux to update status bits, such as the dirty bit in an allocated kernel page table entry. Updates to status bits do not influence kernel code safety. Second, SECvma allows Linux to zero out existing page table entries. SECvma rejects kernel updates to entries in a leaf or non-leaf page table that alter the address of a resulting page map or the next-level page table to a different non-zero value. For instance, SECvma prevents an attacker from adapting a PMD entry that maps an existing PTE to a newly allocated PTE that contains malicious entries that map to executable payloads.

P3: Page table scrubbing. SECvma imposes page table scrubbing to secure Linux's page table usages in two cases: (1) before a table is attached to the existing kernel page table DAG; (2) after the kernel frees a table. KPCore can interpose the two cases here because Linux's updates to page tables trap to KPCore. The protection is crucial because Linux uses an untrusted memory allocator to allocate page tables.

We next discuss how the policies secure Linux's common operations that require writes to kernel page tables.

Create new mappings. Linux updates kernel page tables to create a new page mapping, i.e., to support the translation of an unmapped virtual address to a physical address. Creating a new mapping involves updating the leaf or non-leaf page tables. To securely update to the leaf, SECvma enforces **P2** so that the update does not overwrite an existing entry. Updates to non-leaf page tables could happen if the tables used to perform the intended address translation are missing – new page tables are allocated. SECvma ensures the update to a non-leaf page table does not violate **P1** and **P2**. KPCore then applies **P3**. It write-protects and scrubs the next-level page table and updates the page table entry on behalf of Linux.

Update existing mappings. This operation is recognized when a non-zero page table entry is updated. SECvma enforces **P2** to secure the update.

Delete mappings. Linux zeros a page table entry during an unmap. Admittedly, although protection against availability is out of scope in this work, permitting the kernel to zero out page tables could affect the availability of the compromised system, as an attacker could wipe out kernel page tables. To mitigate such page table tampering, for each physical page containing a page table, KPCore maintains a reference count that stores the number of times the given page table is pointed to by an entry from a page table at a higher hierarchy. SECvma allows multiple different entries from the same page table level to point to the same next-level page table. KPCore increments the reference count of the page table page in each map and decrements the count on each unmap. If the count reaches zero, the table is not used anymore, and the page can now be repurposed. KPCore applies **P3** to scrub freed page tables.

3) User page table protection: Return to user (ret2usr) attack [17] allows an attacker to hijack the OS kernel to execute arbitrary code with kernel privileges. The ret2usr attacks also affected Linux [23], [24]. The attacker could

load malicious code to user memory and exploit the kernel to execute the code. To protect against ret2usr attacks, SECvma enforces PXN permission for pages allocated to user applications (categorized as MEM in Figure 3).

4) DMA Protection: SECvma extends protection mechanisms from the CVM frameworks to prevent attackers from performing malicious DMAs to compromise kernel code integrity. SECvma attaches DMA-capable devices to Arm's SMMU to restrict devices' memory access through their SMMU page tables. SECvma ensures these page tables do not map memory that contains kernel code. SECvma ensures an attacker cannot control the SMMU. KPCore unmaps the SMMU from the host S2PT to trap-and-emulate Linux's MMIO access to the SMMU. This allows KPCore to authorize updates to the SMMU. KPCore prevents an attacker from detaching DMA-capable devices from the SMMU, disabling paging for attached devices, or using a malicious SMMU page table. Like the CVM implementations [6], [7], KPCore should expose hypercalls to Linux to allocate and de-allocate an SMMU translation unit for a device, map/unmap a page to a device's SMMU page table, and walk the SMMU page tables. KPCore allocates SMMU page tables from its private memory and manages them so that devices cannot access KPCore's or Linux's protected memory.

B. Dynamic Module Loading

Modern OS kernels like Linux support Loadable Kernel Modules (LKMs). LKMs are commonly used by users to install driver modules to kernel space memory to run the driver's code without recompiling the kernel. SECvma supports loading Arm64 module binaries and requires no change to the module's source. SECvma ensures that the kernel can only install and execute authenticated modules. Authenticating loadable modules at runtime represents a challenge. Linux loads and installs kernel modules from the file system to memory. These modules consist of symbols unresolved at compile time and cannot be executed directly. To install a kernel module, Linux must first allocate memory to accommodate a given module, relocate the module's contents, and then link and resolve the module's symbols. Linking and symbol resolution require modification to the module's code loaded to memory. While the traditional hash-based authentication approach verifies kernel modules before memory installation, this approach is vulnerable to runtime attacks. An adversary could compromise module integrity by manipulating Linux's loading process to inject malicious code into memory regions containing previously authenticated modules.

To address the issue, one approach is to task KPCore for module authentication and installation. An intuitive approach is to port the existing module installation functionality from Linux to KPCore or reuse its module signing facility [25]. This guarantees safety but on the other hand, could significantly bloat the code size of KPCore. Alternatively, one could reinvent a new kernel module loading mechanism for KPCore. However, the clean-slate approach is likely incompatible with Linux's existing features.

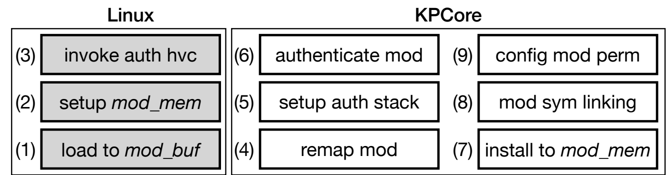

Fig. 4: Secure Dynamic Module Loading in SECvma

To support the installation of loadable kernel modules, SECvma *splits module loading to memory from authentication and relocation*. SECvma allows the untrusted host Linux to load kernel modules from the file system to memory but forbids the host from executing the module directly. KPCore only grants the memory pages that contain the module code execute permission after the module is authenticated. KPCore ensures that the code pages are read-only so Linux cannot tamper with the authenticated module contents. KPCore exposes a set of hypercalls as depicted in Table IV to Linux to support dynamic module loading. We detail the other hypercalls in Section V.

KPCore authenticates loadable kernel modules using public key cryptography. SECvma assumes that the module distributors (e.g., vendors) signed their modules using a private key to produce a signature. The distributors publish the respective public key and the module's signature. We assume the signatures and public keys are downloaded from the authenticated portals via secure channels to the platform running SECvma, sealed to the platform's secure storage, and loaded into KPCore's memory before an out-of-scope attack is conducted. KPCore later uses the public key to authenticate the module against the module's respective signature.

The secure module loading support in SECvma consists of the steps listed from Figure 4.

Module Loading. Linux handles the `insmod` system call made by users to install a loadable kernel module. Linux first copies the contents of the entire module file, including headers and all of the sections from user space, and stores them in a memory buffer (denoted as `mod_buf`) (Step 1 from Figure 4). The contents from `mod_buf` are shown in Figure 5.

Linux then allocates a new memory region from the kernel space, denoted `mod_mem`. The region consists of four virtually contiguous areas: `code`, `rodata`, `ro_after_init`, and `writable data`. Linux retrieves contents from sections in the kernel module with the `SHF_ALLOC` flag set and puts them to the respective areas in `mod_mem` according to their access permission and usage. Linux updates the module's section headers to specify the respective section's runtime addresses.

The mainline Linux resolves and links the symbol addresses for code and data in `mod_mem` and executes them at runtime after the module is installed. In SECvma, Linux makes the `mod_auth` hypercall (see Table IV) to KPCore to authenticate the loaded module before making it executable (enforced PXN permission). It passes the address to the start of `mod_buf`, which points to the ELF header of the module to KPCore via the first parameter `p_hdr` of the hypercall. Linux also passes the addresses of the percpu section data and the architecture-

specific data via `percpu` and `arch` parameters, respectively. The data will be used for relocation.

Pre Module Authentication. KPCore then performs step (4) from Figure 4 to remap pages from `mod_buf` to its address space. This is necessary because KPCore cannot access Linux's `mod_buf`. KPCore runs in the EL2 address space and cannot access Linux's EL1 address space. KPCore gets the module's contents from the remapped `mod_buf` (to EL2). Before authentication, KPCore ensures memory pages containing the module's contents (`mod_buf` and `mod_mem`) are read-only to Linux. This prevents an attacker from modifying the module's contents that KPCore has verified.

Module Authentication. SECvma verifies all executable sections, read-only data sections, and relocation sections to ensure the integrity of the module's code. The latter two contain essential information, such as the module's symbol table to module symbol resolution and linking. As shown in Table IV, the hypercall `mod_auth` takes `list` and `size` as the fourth and fifth arguments. `list` specifies a list of indices denoting sections in the module's section table that need to be verified, whereas `size` specifies the number of indices.

KPCore allocates a `stacking buf` to stack the module contents to be authenticated. It iterates through `list` and copies the section's header, data, or code from the module's `mod_buf` to `stacking buf`. As shown in Figure 5, the header[1] and code/data for each section are stacked after each other in `stacking buf`. KPCore authenticates the stacked contents against a pre-computed signature derived from the same stacking layout of the same module. Our stacking approach eliminates the need to validate each section individually.

Post Module Authentication. After the module is authenticated, KPCore proceeds to install the authenticated contents to `mod_mem` (see Figure 5). Since the allocation of `mod_mem` is supplied by the untrusted Linux memory allocator, the pages in `mod_mem` could contain injected exploits before the region was write-protected. Hence, KPCore first scrubs `mod_mem`. KPCore then copies the authenticated module contents to the respective regions in `mod_mem`. Next, it updates the symbol table based on the information stored in the module's verified section headers. KPCore then performs linking for the kernel module. It updates symbol references from the code and data in `mod_mem` to resolve their runtime addresses. Finally, KPCore configures the access permissions for pages used by `mod_mem`. As shown in Figure 3, KPCore grants the module's text region R+X permission and enforces R+W+XN and R+XN permission for the module's data and rodata region. See `MOD_TEXT`, `MOD_DATA`, and `MOD_RO` from Figure 3 about this setting.

C. System Register Protection

In addition to protecting page tables, SECvma prevents attackers from manipulating Arm's system registers that control virtual memory to compromise kernel code integrity.

[1]We masked members from respective section headers that Linux updated during module loading at runtime before authentication.

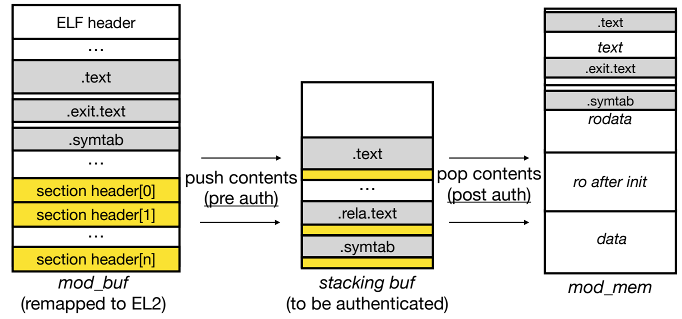

Fig. 5: Module Authentication in SECvma

TABLE I: Virtual Memory System Registers

Category	Registers
VMem Translation Registers	TCR_EL1, SCTLR_EL1, MAIR_EL1
Hardware-Managed Register	ESR_EL1
Misc Registers	AFSR0_EL0, AFSR1_EL1, AMAIR_EL1, CONTEXTIDR_EL1
Runtime-Updated Registers	FAR_EL1, TTBR0_EL1, TTBR1_EL1

We categorize these system registers into four categories shown in Table I. SECvma enforces two policies: *ro-after-boot* and *write-check* against Linux's updates to these system registers. SECvma applies the *write-check* policy to Runtime-Updated Registers. SECvma enforces the *ro-after-boot* policy to registers in the rest of the three categories; the values of these registers either are predetermined by the OS kernel and remain unchanged after the kernel boot, or are only updated by hardware during runtime. KPCore enables the `TVM` bit from Arm's HCR_EL2 register to trap every Linux's write to the registers in Table I to EL2. This allows KPCore to transparently interpose every register update and apply the corresponding strategies to protect different registers.

VMem Translation Registers: The registers in this category are vital for security and play a key role in defining MMU configuration, memory attribute representation, and page table translation methods. Manipulating these registers arbitrarily most likely results in kernel panic but could also result in dangerous impacts. Thus, KPCore ensures Linux cannot change these registers at runtime.

TCR_EL1: The register controls the stage 1 translation methods of memory accesses from EL0 or EL1. It dictates the page table format, translation granule, and memory access attributes of the MMU's page table walk. Arbitrarily modifying the register will most likely result in crashes. However, an attacker can manipulate the *TG1* bits to change the translation granule of the kernel page tables and the size of the page tables used in the translation regime. Linux, by default, uses the 4KB granule. If TCR_EL1 is unprotected, an attacker could change the translation granule and page table size to 64KB. Since SECvma write-protects 4KB page tables, this renders 60K bytes of the table potentially unprotected, allowing an attacker to inject malicious entries into the table [26].

SCTLR_EL1: This register controls the virtual memory system, such as paging or the endianness of data access. Both

can lead to unpredictable outcomes if changed during runtime. Since SECvma could rely on the PXN bit in kernel page table entries (see Section V-A) to protect memory, an attacker who disables paging could bypass the protection.

MAIR_EL1: This register defines a set of memory attributes (e.g., cacheability). The attributes are used by stage 1 translations at EL1. Entries from an S1PT specify an attribute index to MAIR_EL1 to pick an attribute set. Load and store instructions initiate the MMU page table walk that translates the virtual address via the page table entry use its specified attribute. On the other hand, the software programs the TCR_EL1 register to control the memory attribute for page table access from the MMU [27]. An attacker who gains the privilege to modify these registers could cause a mismatch of the attributes specified in MAIR_EL1 and TCR_EL1. Consider a scenario in which the attributes mismatch: TCR_EL1 specifies that the memory attribute of the MMU page table walk is set as cacheable, while the attribute set in the page table entry that maps to the page table is set as non-cacheable. The MMU accesses and caches the page table entry during a memory translation. Later on, Linux changes the page table. Because the memory attribute for the page table is set as non-cacheable, the change is carried through the main memory. This results in the cache holding an outdated page table entry. A subsequent MMU walk will use the stale entry rather than the updated one. The stock Linux kernel ensures that the memory attributes of the MMU page table walk and the load and store instruction accesses to the page table match, such that it can avoid cache maintenance when updating a page table entry. However, an attacker who causes an attribute mismatch could compromise kernel safety. For example, when Linux updates a page table entry with PXN permission, the MMU may continue using the outdated permission setting, potentially allowing the execution of privileged code.

Hardware-Managed Register: The hardware updates ESR_EL1 with exception syndrome information (e.g., exception causes) at runtime. Linux reads the register to retrieve such information for exception handling. An attacker who tampers with the register could result in unpredictable kernel behavior. Therefore, SECvma forbids Linux's updates to the register.

Misc Registers: Registers in the category are used for debugging and tracing or providing supplementary information. SECvma revokes write access to these registers because Linux does not currently use these registers.

Runtime-Updated Registers: Unlike the registers in the other three categories, Linux updates registers in this category at runtime in the exception vectors and during process context switches. In the former case, Linux updates the registers to support Kernel Page Table Isolation (KPTI) [9], a feature to mitigate Meltdown [28] and Kernel Address Space Layout Randomization (KASLR) [29] bypass. The KPTI implementation uses FAR_EL1 as a scratch register [30]. For TTBR1_EL1, KPTI provides two kernel page tables: K-PGD for the kernel and UK-PGD for the user space. The latter only maps to an exception trampoline to perform kernel enter. When KPTI is enabled, Linux points TTBR1_EL1 to UK-PGD before returning to the user space. The trampoline from UK-PGD programs TTBR1_EL1 with the address of K-PGD on exception enters. During process context switches, Linux first updates TTBR0_EL1 to a zero page then updates TTBR0_EL1 with the PGD of the next process to switch to. When KPTI is enabled, it also updates the ASID bits in TTBR1_EL1 with the ASID of the next process, so the TLB uses the ASID to tag the address translation results.

FAR_EL1 is either updated by hardware (e.g., fault address) or by the host as a scratch register at runtime during kernel exit to support KPTI [30]. In the latter case, KPCore traps Linux's updates the FAR_EL1 and operates on behalf of Linux. KPCore ensures that FAR_EL1 is updated with the saved value at kernel enter. For page table base registers, KPCore ensures that Linux can only update TTBR1_EL1 to the address of either K-PGD or UK-PGD at the entrance to the kernel or user. Both PGDs are allocated during Linux boot time. KPCore stores the respective addresses for runtime checks. For updates to TTBR1_EL1 during process context switches, KPCore ensures the updates only replace the ASID bits. Further, KPCore ensures that Linux cannot program TTBR0_EL1 with the address of K-PGD, UK-PGD, or an arbitrary address.

V. Implementation

To prototype SECvma, we extended the SeKVM [7], [8] hypervisor. KPCore leverages the functions provided by SeKVM's security monitor, KCore, for page table management (e.g., walk and set) to update page access permissions bits in the host S2PT entries to implement the permission enforcement scheme. KPCore extends KCore's exception handling framework with permission fault handling features. KPCore extends SeKVM's ownership-based memory access control scheme, which aims to isolate VM memory from the host to support Linux integrity protection. SECvma leverages the formally verified Ed25519 implementation from the HACL [31] library integrated with SeKVM to support module authentication. To facilitate practical deployments, we implemented a script for developers to compute the signature for their kernel modules in a stacking fashion described in Figure 5.

As listed in Table II, we added and modified 4,355 LOC in SeKVM for Linux 4.18 from the open-source artifact [32] to support SECvma. 3,419 LOC was added to new layer modules. Among them, 1,458 LOC was dedicated to kernel page table and static code protection, while 213 LOC was added for system register protection. 1,748 LOC was added to support secure kernel module loading. Another 663 LOC was to define data structures in header files.

273 LOC was added to or modified from SeKVM's existing codebase. 69 LOC was added to SeKVM to initialize data structures and store the information needed for kernel protection, such as sections' and kernel PGD's base address. We added 34 LOC to SeKVM's TrapDispatcher module to support the hypercalls listed in Table IV. 21 and 74 LOC were added to the FaultHandler and MemAux modules to handle stage 2

TABLE II: SECvma's Lines of Code

Category	Layer	LOC	Total
New Layer Modules	Page Table & Static Code Protection	1,458	3,419
	System Register Protection	213	
	Secure Kernel Module Loading	1,748	
SeKVM's Layer Modules	Initialization	69	273
	TrapDispatcher	34	
	FaultHandler	21	
	MemAux	74	
	PTWalk, NPTWalk, NPTOps	75	
C Headers		663	663
Grand Total			4,355

TABLE III: Permission Table[2]

usage	S2 perm	S1 perm	result perm
KPGD	R + XN	*	R + XN
KPUD	R + XN	*	R + XN
KPMD	R + XN	*	R + XN
KPTE	R + XN	*	R + XN
UPGD	R + XN	*	R + XN
KTEXT	R + X	*	R + X
KDATA	R + W + XN	*	R + W + XN
KRODATA	R + XN	*	R + XN
MOD_RO	R + XN	*	R + XN
MOD_TXT	R + X	*	R + X
MOD_DATA	R + W + XN	*	R + X
MEM	R + W + X	*+PXN	R + W + PXN

permission faults caused by the host Linux's updates to page tables. Lastly, we added or modified 75 LOC to three modules, PTWalk, NPTWalk, and NPTOps, to support the huge page optimization discussed in Section V-B.

*a) **Memory Usage and Protection.**:* SeKVM's KCore manages metadata called `s2page` for each physical memory page. This metadata stores the ownership information, which tracks the respective entity that currently owns the respective page and the sharing information, which logs whether a CVM shares the page with the Linux host. To support host kernel protection, SECvma extended `s2page` with a new `usage` field to specify Linux's current page usage. The column *usage* in Table III lists all memory usage types (shown in Figure 3) in SECvma. In addition, we extended `s2page` with a new `map count` field. It counts the number of times a given page table is mapped by an entry to secure the page table free (see Section IV-A2). We record the usage for memory pages at boot time and store the start physical address and the size of the ELF sections (e.g., text, data). When initializing KPCore, SECvma uses the information to set the `usage` in `s2page`. SECvma also traverses the kernel page tables from the root to all leaves. SECvma assigns the usage to a page table page according to its hierarchy level (e.g., the kernel page table root pages are assigned `KPGD` usage by SECvma). SECvma enforces the hierarchical order in kernel page tables (see Section IV-A2) based on the page table usages. SECvma assigns the `UPGD` usage to pages used as user PGDs and the `MEM` usage to the rest of the memory.

[2]R: readable; W: writable; X: execute; PX: privilege execute; PXN: privilege execute never; XN: execute never; *: value set by Linux

After KPCore is successfully installed at EL2, it updates VTTBR_EL2 with host S2PT then enables stage 2 translation before returning to Linux at EL1. The host S2PT initially contains no mapping. The kernel's access to an unmapped page causes a page fault that traps to EL2. KPCore gets the faulted address from Arm's HPFAR_EL2 register then queries a respective `s2page` of the faulting page. KPCore sets the access permission to the faulting S2PT entry based on the page's usage (from the column S2 perm in Table III).

*b) **Hypercalls.**:* KPCore exposes hypercalls listed in Table IV to Linux. `mod_auth` is used for module authentication that we discussed earlier. We discuss `opt_switch_mm` in Section V-B. KPCore exposes two hypercalls to Linux to support module memory freeing. Both hypercalls, `free_init` and `free_module`, pass one argument: `mod_id` to KPCore. `mod_id` is the module identifier that `mod_auth` returns to Linux after successfully installing an authenticated module. KPCore uses `mod_id` to identify the pages respective to a loaded module. KPCore ensures that Linux only uses these hypercalls to reclaim pages owned by the loaded kernel module but not protected kernel memory. An error is returned if an input `mod_id` does not correspond to an installed kernel module. After the module is installed, Linux first executes code from the module's init section and makes `free_init` to free the memory of the init section after the execution finishes. To handle the hypercall, for each of the pages in the init section, KPCore sets their usage to `MEM` and updates their access permission in the host S2PT. KPCore performs the same operations for all module's memory to handle `free_module`. Linux makes the hypercall when the user requests a module uninstall via the `rmmod` system call. `alloc_el0_pgd` and `free_el0_pgd` were introduced to support user page table protection. We extended Linux to make these hypercalls when creating and terminating a process, respectively, to allocate and free a PGD. In addition to hypercalls in Table IV, SECvma supports SeKVM's SMMU hypercalls (see Section IV-A4).

TABLE IV: Kernel Protection Hypercalls

Hypercall
`mod_auth(p_hdr, percpu, arch, list, size)`
`free_init(mod_id)`
`free_module(mod_id)`
`opt_switch_mm(ttbr0, asid)`
`alloc_el0_pgd(addr)`
`free_el0_pgd(addr)`

A. Addressing Limitations of Arm VE

PXN bit in S2PT entries. Arm processors without the FEAT_XNX extension [33] do not include a PXN bit in S2PT. KPCore uses the PXN bit in S2PT if is available. Otherwise, it enforces the PXN bits in S1PTs (see column S1 perm from Table III). KPCore sets the PXN bit when updating kernel page tables on behalf of Linux. For user page tables, KPCore sets the PXN bit in all PGD entries when serving the `alloc_el0_pgd` hypercall to protect against ret2usr attacks. SECvma write-protects kernel page tables and user PGDs to prevent an attacker from canceling PXN.

Incomplete Exception Information. Arm VE does not provide the faulting IPA (that equals the machine's physical address in SECvma) when permission fault occurs during stage 2 translation. For KPCore, the address is essential for emulating writes to page tables on behalf of Linux. To address the limitation, KPCore leverages an Arm instruction: `at` to translate the faulting virtual address to an IPA when handling a stage 2 permission fault to the physical address. We ensure page tables remain static during hardware traversal when executing the `at` instruction. First, Linux's S1PT is marked read-only; the kernel running on a different CPU cannot modify page tables while executing `at`. Second, although Linux cannot directly modify its S1PT, any modification attempts trap to KPCore. Upon detecting such a write fault, KPCore acquires a spinlock before walking and updating Linux's S1PT, ensuring these operations are performed atomically.

Further, when unmapping entries of the page tables, Linux uses Arm's `stxr` instruction to zero out page tables. Since page tables are write-protected, executing `stxr` causes a trap to KPCore. Arm VE does not provide information, such as the source and target register operands about a trapping `stxr` (in the ESR_EL2 register) due to a permission fault. To overcome the limitation, KPCore fetches and decodes the `stxr` instruction from memory via the faulting PC stored in Arm's ELR_EL2 register and gets the register operands.

B. Optimizations

a) Huge Page Optimization.: To achieve memory access control at the finest granule, SECvma uses the 4KB translation granule in the host S2PT by default. This approach incurs a significant performance slowdown on Arm hardware with a small TLB. To address the slowdown, we make pages that share the same S2 permission in Table III locate within the same 2MB aligned regions. This allows KPCore to map these pages in S2PTs with huge (2MB) pages to reduce TLB pressure. We modified Linux such that memory pages other than those with their `usage` set to `MEM` are backed by 2MB mappings in the host S2PT. To further reduce the TLB pressure, we introduced an additional optimization for KPCore to map all `MEM` memory pages with 1GB mappings in the host S2PT if the given 1GB region of physical memory contains no 2MB pages with different usage.

We modified Linux's linker script to ensure all kernel code and data are loaded to separated 2MB aligned regions. This also ensures kernel code and data do not co-locate on the same page. Linux, by default, allocates kernel pages from the kernel heap. We cannot map the heap memory in the host S2PT with 2MB pages because 4KB pages with different S2 permissions could co-locate within the same 2MB region. To incorporate huge page optimization, we introduced a buddy memory allocator to Linux to allocate 4KB-aligned pages from 2MB-aligned memory pools that share the same permission in the host S2PT. The enlightened Linux allocates kernel page tables from the pool with the R+XN permission in the host S2PT. The mappings in S2PTs are transparent to Linux. We did not change Linux's page granularity from 4KB to 2MB

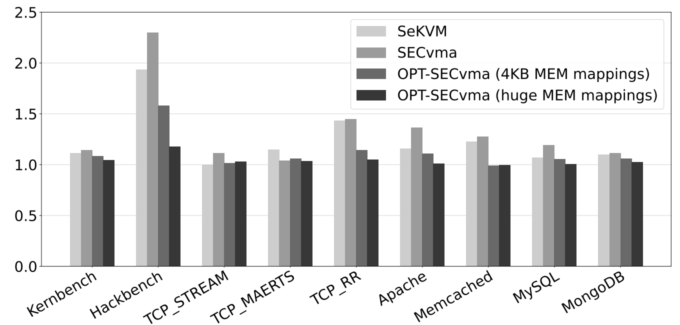

Fig. 6: Application Performance - Linux Host

and thus incurring no fragmentation or pressure in process creation and caching.

b) Optimize System Register Traps.: Without compromising security, we optimized system register protection in two scenarios to reduce traps to KPCore: (1) kernel/user enters and exits, and (2) process context switches. For (1), as mentioned earlier, Linux's KPTI implementation uses FAR_EL1 as scratch registers [30]. Due to register protection, the writes to FAR_EL1 result in traps to EL2 in SECvma. To avoid the trap, we replaced the write to FAR_EL1 with TPIDRRO_EL0. Note that this incurs no safety issue. Linux already uses TPIDRRO_EL0 as a scratch register. To further guarantee security, before returning to userspace, KPCore zeros the value of TPIDRRO_EL0 after use. For (2), we introduced a hypercall, `opt_switch_mm` to batch the three updates made by Linux to TTBR0_EL1 and TTBR1_EL1 to KPCore (see Section IV-C). Linux passes the base address of the next process' PGD, and its ASID as arguments to the hypercall.

VI. Evaluation

We evaluated the performance of the mainline Linux/KVM, SeKVM, and SECvma on an HP Moonshot m400 server with an 8-core 64-bit Armv8-A 2.4 GHz Applied Micro Atlas SoC, 64 GB of RAM, a 120 GB SATA3 SSD, and a Dual-port Mellanox ConnectX-3 10GbE NIC. For client-server workloads, clients ran on another m400 machine and connected to the server via a 10 GbE. We tested application benchmarks running on the host Linux kernel on the bare-metal and VMs on the m400 hardware. All configurations use the Linux 4.18 kernel and Ubuntu 20.04. The kernels were compiled with standard protection features, including KPTI. The bare-metal configurations use all hardware available. All VMs were configured with 4 virtual CPUs and 12 GB of RAM. All VMs used paravirtualized I/O. They were configured with its standard vhost virtio network and with `cache=none` for its virtual block storage devices [34], [35].

Figure 6 presents the performance of applications running on the Linux host on SECvma and SeKVM. The results are normalized against those of the same applications running on the mainline Linux v4.18 (lower is better). The applications used the configuration listed in Table V. We adopted four configurations: (1) SeKVM, (2) unoptimized SECvma

TABLE V: Description of benchmark application

Name	Description
Kernbench	Compilation of the Linux 4.18 kernel using allnoconfig for Arm with GCC 9.3.0.
Hackbench	Hackbench [36] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	Netperf v2.6.0 [37] running the netserver on the server and the client with its default parameters in three modes: TCP_STREAM (receive throughput), TCP_MAERTS (send throughput), and TCP_RR (latency).
Apache	Apache v2.4.41 server handling 100 concurrent requests from *two* remote ApacheBench [38] v2.3 clients on bare-metal and *one* for a VM, serving the 41 KB index.html of the GCC 4.4.7 manual. Measured the number of requests handled by the Apache server per second.
Memcached	Memcached v1.5.22 using the memtier benchmark [39] v1.2.3 with its default parameters, running 8 threads in the bare-metal experiment, 4 threads for the VM experiment.
MySQL	MySQL v8.0.39 running SysBench v.1.0.18 using the default configuration with 100 parallel transactions, tables=10, and table-size=100000.
MongoDB	MongoDB server v4.4.0 handling requests from a remote YCSB [40] v0.17.0 client running workload A with 16 concurrent threads, readcount=10000 and operationcount=500000

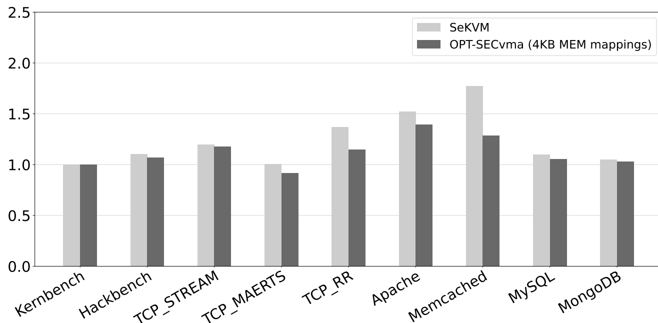

Fig. 7: Application Performance - VM

(SECvma), (3) OPT SECvma (4KB `MEM` mappings), and (4) OPT SECvma (huge `MEM` mappings). Configurations (3) and (4) both incorporate the system registers optimization and 2MB mappings in the host S2PT for memory that has a different usage from `MEM`; however, in the host S2PT, (3) maps `MEM` memory with 4KB mappings, while (4) maps `MEM` memory with both 2MB and 1GB mappings.

The results show that the unoptimized SECvma could significantly reduce performance slowdown in application performance. We found that the overhead stems from register protection and the extra stage of memory translation used in memory protection. The former was due to the extra traps incurred when the unoptimized SECvma updates the protected system registers during task context switches and kernel/user space entrance and exit to support KPTI. For the latter, the processors on the m400 machine have a tiny TLB [41]. The unoptimized SECvma employed 4KB mappings for all memory in the host S2PT and resulted in high TLB contention.

SECvma's overhead is more significant in Hackbench, TCP_RR, Apache, and Memcached. All these applications include system calls in critical paths and thus suffer from the overhead resulting from register protection in KPTI. Hackbench forks numerous processes to perform IPC via Unix pipes. Compared to others, Hackbench involves more frequent processes and kernel/user switches, suffering higher overhead from register protection. SECvma's system register optimizations effectively improved the performance of these workloads. Hackbench exhibited a much higher memory footprint than other applications. Thus, SECvma's huge page optimization for `MEM` memory reduced the overhead significantly in Hackbench for more than 40% (OPT-SECvma (4KB vs huge `MEM`)). SECvma also outperformed SeKVM, which suffered from TLB contention due to its adoption of a 4KB mapping scheme in host S2PT.

Figure 7 presents the performance of application workloads running in VMs on SeKVM and the optimized SECvma with 4KB `MEM` mappings. The results are normalized to the performance of VMs running on the mainline KVM v4.18 using the same configuration. Note that the optimized SECvma outperformed SeKVM. The performance improvement is more noticeable in I/O bound workloads. On KVM, the host Linux and QEMU provide I/O virtualization. The host's efficiency is critical to VM performance. SECvma reduced the stage 2 translation overhead during the host's execution with huge page mappings, resulting in optimized VM performance.

TABLE VI: Module Performance

		Linux 4.18		SECvma	
Name	**File Size**	`insmod`	`rmmod`	`insmod`	`rmmod`
crypto_virtio.ko	26.5 KB	$3ms$	$16ms$	$62ms$	$16ms$
xfs.ko	1.06 MB	$17ms$	$64ms$	$98ms$	$70ms$

insmod and rmmod Performance. We evaluated the performance of executing the `insmod` and `rmmod` commands on SECvma versus the mainline Linux. We tested two kernel modules: a cryptographic module (`virtio_crypto.ko`) and a file system driver (`xfs.ko`). Table VI shows the modules' respective sizes. As shown in Table VI, `insmod` and `rmmod` on SECvma took much longer than on Linux. Due to the memory access overhead from module authentication, `insmod` suffered a higher overhead than `rmmod`. The slowdown is a trade-off for much-enhanced security. `insmod` and `rmmod` are infrequent. We believe the resulting overhead (far less than 1s) should not affect user experience in practice.

A. Security Analysis

SECvma ensures that an attacker who gains kernel privileges cannot modify the metadata and S2PTs stored in KPCore's private memory. An attacker from the kernel mode cannot access or modify EL2's registers that KPCore uses to enable protection. For example, an attacker cannot modify the HCR_EL2 register to disable stage 2 memory translation or trapping on writes to system registers.

We analyzed Linux's CVEs in the past eight years and identified a class of Linux CVEs that enable out-of-bound writes to kernel memory. Many of these CVEs are from the kernel's

device drivers [42]–[49]. Some others were bugs reported in Linux's network subsystem [50]–[53] and file systems [54]–[56]. With an equivalent implementation of SECvma for vulnerable kernel versions and platforms, SECvma effectively protects Linux's code integrity against these malicious writes. SECvma prevents an attacker that exploits these vulnerabilities from modifying kernel memory that contains approved code, either from Linux's binary or an authenticated kernel module.

An attacker could also exploit vulnerabilities that enable out-of-bound memory writes to (1) emit malicious code to memory or (2) corrupt page tables. For (1), an attacker could output code to kernel heap and stack or a user application's memory. The latter could be part of the efforts to carry out the ret2usr attack [17]. SECvma cannot prevent code emission to writable memory (e.g., memory with the `MEM` type) regions. However, it enforces the PXN permission for these writable pages to prevent attackers from executing the emitted code.

For (2), an attacker could manipulate kernel page tables to create new entries or modify existing entries. SECvma ensures that an attacker cannot disrupt the page table ordering and lead to a faulty translation (*Policy P1*). For example, this prevents an attacker from inserting an existing PTE into the PUD, leading the MMU to treat the PTE as a PMD. An attacker can also modify the existing page table entries. For instance, she could cancel permission bits to disable Linux's DEP protection at the kernel page tables. Furthermore, an attacker could remap a page table entry to new memory pages or page tables. SECvma eliminates these arbitrary page table modifications (*Policy P2*). Finally, SECvma scrubs newly allocated page tables (*Policy P3*) before a map. This prevents an attacker from creating a new entry that maps to a page table that contains malicious entries that map to executable payloads to perform code injection.

In addition to the CVEs that compromise Linux's spatial memory safety, CVE-2023-6238 [57], a bug in Linux's NVM Express driver allows an unprivileged user to let the device perform DMA to overwrite kernel memory. The attack cannot compromise kernel code integrity on SECvma. DMA writes are restricted to data buffers but not memory that contains kernel code on SECvma.

An attacker could corrupt system registers to compromise kernel code integrity. She could update the page table base registers to specify a customized page table root that maps to malicious executable payloads or contains unsafe permission settings that mismatch with W$\oplus$X. An attacker could also manipulate VMem Translation system registers to compromise kernel safety. She could corrupt TCR_EL1 to bypass SECvma's page table protection or SCTLR_EL1 to disable paging to cancel SECvma's PXN protection. Further, she could control MAIR_EL1 and TCR_EL1 to cause a mismatch of page table translation settings to potentially bypass page permission settings. An attacker could cause cache incoherence in page table walks to trick the MMU into using outdated page table entries with unsafe permission settings. As discussed in Section IV-C, SECvma traps the kernel's writes to system registers to prevent the above malicious updates from compromising Linux's code integrity.

The Linux Integrity Measurement Architecture (IMA) maintains the integrity of the system's files by measuring and attesting to the integrity of files before they are accessed. IMA can ensure the integrity of module files before loading. However, a compromised Linux could alter the module file put into memory during module loading or load a malicious module instead of the authenticated one. SECvma addresses such TOCTTOU issue by tasking KPCore to authenticate the module contents loaded to memory at module installation.

VII. Limitations and Future Work

Other than dynamically installing kernel modules, SECvma does not allow new code to be added to the kernel space. Therefore, SECvma has to be extended to permit the execution of verified eBPF code [58]. SECvma currently supports executing static eBPF code in interpreter mode. In the future, we plan to extend SECvma to support eBPF's JIT mode. SECvma current cannot enable the OPT-SECvma (huge `MEM` mappings) optimization when running confidential VMs. VM memory allocated from the region with `MEM` usage fragments the huge page mappings in the host S2PT. As shown in Figure 7, SECvma still outperformed SeKVM without such optimization. The optimization is left for future work. In addition, we plan to upgrade SECvma to a more recent Linux version. The changes required for SECvma and SeKVM to mainline Linux are self-contained and require no intrusive changes. As long as SeKVM's patch is ported to a new Linux version, incorporating SECvma's kernel protection features should require modest porting efforts.

KPCore builds on SeKVM's formally verified KCore to support host kernel protection. SeKVM adopts a modular verification approach. Functions from modules can be verified independently to produce composable proofs. We minimized changes to existing functions from KCore to retain the existing proofs. Most of our updates to SeKVM are in the additional unverified modules. Our extensions are orthogonal to SeKVM's existing VM protection features. We thus expect the addition to preserve SeKVM's verified VM security guarantees. In the future, we plan to leverage the Spoq [59] to automate code verification for KPCore.

pKVM [6] adopts a similar Arm VE-based approach to SeKVM, relying on a security monitor to support CVMs on KVM. We expect that SECvma's design applies to pKVM to enable the host Linux kernel protection Like SeKVM, pKVM manages a host S2PT to isolate the host Linux's access to CVM memory. pKVM uses the unused bits in S2PT entries to track memory ownership and sharing information. SECvma should extend the mechanism to track the usage types (e.g., incorporate extra metadata) and manage the host S2PT accordingly to protect kernel memory. Further, pKVM should be extended to handle permission faults caused by Linux's updates to kernel page tables and enforce page table protection. In addition, we could utilize SECvma's approach to extend Arm-based type-1 hypervisors [60], [61] to leverage

VE features to secure the OS kernels running on the integrated privileged VM (e.g., Dom0) and enhance their safety.

VIII. Related Work

Previous work relies on a trusted hypervisor in the TCB to protect a monolithic OS kernel. Some relies on a hypervisor to detect kernel rootkits [62]–[66] in VMs. Others [67], [68] rely on the hypervisor [61] to isolate an OS kernel from untrusted components. Microsoft's Hypervisor-protected Code Integrity [69] framework relies on a full Windows hypervisor to protect the code integrity of the Windows OS kernel. These systems have a much larger TCB than SECvma.

Secvisor [11] relies on a tiny hypervisor that leverages x86's virtualization extensions to protect Linux's code integrity. Unlike SECvma, Secvisor incurred high performance overhead to hosted applications (2.19x in the worst case). Secvisor allows Linux to install kernel modules at runtime at the user's discretion. Unlike SECvma, Secvisor does not authenticate the contents of a loaded kernel module before an installation.

Sprobes [70] and TZ-RKP [10] rely on privileged firmware that leverages Arm's TrustZone extension to protect Linux. Unlike Arm VE, TrustZone does not provide features for a secure world software to proactively interpose and monitor sensitive system events. For instance, TrustZone does not support trap-and-emulate against Linux's accesses to write-protected memory (e.g., page tables) or writes to sensitive system registers. The TrustZone-based approaches thus require instrumentation to Linux to make secure monitor calls (SMCs) to the monitor to validate and perform the operation. The instrumentation efforts incur portability and compatibility issues with updated Linux versions. Unlike these systems, SECvma requires modest instrumentation to Linux: (1) make hypercalls to load/unload kernel modules and allocate/free user PGDs, (2) enable ret2usr protection, and (3) adopt optimizations.

KNOX [71] provides the Real-time Kernel Protection (RKP) [72] mechanism to ensure the code integrity of the Linux kernel integrated with the Android OS. The RKP employs a security monitor, either a dedicated hypervisor or a TrustZone-based monitor, depending on the device model, to protect the kernel. KNOX is proprietary. Previous efforts that reversed-engineered the RKP hypervisor [26] disclosed that it leverages Arm VE to support kernel memory protection. Similarly to SECvma, the RKP hypervisor runs in Arm's EL2 mode and uses S2PTs to protect memory. RKP also validates writes to virtual memory control system registers and secures dynamic module loading.

In contrast to KNOX's clean-slate approach, SECvma proposed to extend the existing CVM frameworks for Arm to significantly reduce the implementation complexity for protecting Linux's code integrity. Additionally, unlike SECvma, KNOX does not support CVMs. Further, compared to SECvma, KNOX does not support module authentication during dynamic module loading to memory. RKP could be vulnerable to a TOCTTOU attack that bypasses module authentication, allowing an attacker to install malicious modules to executable kernel memory (see Section VI-A). We cannot comprehensively compare SECvma's feature set with KNOX because KNOX is not opened source. At the time of writing, neither detailed design documentation nor performance benchmarks for KNOX are publicly available.

Nested kernel [73] splits the OS kernel into a trusted nested and a comprehensive outer kernel. Instead of adopting a virtualization-based approach, the nested kernel leverages Intel's hardware features, i.e., CR0's WP bit and privilege rings, to protect the integrity of the outer kernel's memory, prevent unauthorized updates to the kernel page tables and tamper with kernel data and code integrity. The approach is inapplicable to SECvma. Hardware features like the WP bit are unavailable on Arm.

IX. Conclusion

This work introduced SECvma, a new system that employs a virtualization-based approach to protect the code integrity of Linux on Arm-based platforms. SECvma ensures a privileged attacker cannot tamper with the existing kernel code or load unauthenticated kernel modules. SECvma proposed a new design that extends Arm's existing CVM framework to simplify development efforts while preserving CVM protection. The SECvma prototype extended a KVM-based secure hypervisor for Arm to achieve robust kernel security and retains Linux's performance and functionality.

Acknowledgement

We thank the anonymous reviewers for their insightful feedback. This research was supported by the National Science and Technology Council of Taiwan under research grants 112-2628-E-002-027- and 113-2628-E-002-030-.

References

[1] J. Li, S. Miller, D. Zhuo, A. Chen, J. Howell, and T. Anderson, "An incremental path towards a safer os kernel," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 183–190. [Online]. Available: https://doi.org/10.1145/3458336.3465277

[2] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: Nvidia's first 64-bit arm processor," *IEEE Micro*, vol. 35, no. 2, pp. 46–55, 2015.

[3] C. Williams, "Microsoft: Can't wait for ARM to power MOST of our cloud data centers! Take that, Intel! Ha! Ha!" *The Register*, Mar. 2017, https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup.

[4] Amazon Web Services, "Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors," Nov. 2018, https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances.

[5] ARM Ltd., "Introducing Arm Confidential Compute Architecture Version 1," May 2022, https://developer.arm.com/documentation/den0125/0100/What-is-Arm-CCA-.

[6] Jake Edge, "KVM for Android," LWN.net, Nov. 2020, https://lwn.net/Articles/836693/.

[7] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, May 2021.

[8] ——, "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, Aug. 2021.

[9] J. Corbet, "The current state of kernel page-table isolation," https://lwn.net/Articles/741878/, Dec. 2017.

[10] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 90–102. [Online]. Available: https://doi.org/10.1145/2660267.2660350

[11] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[12] ARM Ltd., "ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0," http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, Jun. 2016.

[13] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from commodity hypervisor and host operating system exploits," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, Santa Clara, CA, Aug. 2019, pp. 1357–1374.

[14] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 465–484. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/li

[15] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, Ottawa, ON, Canada, Jun. 2007.

[16] "Mwr labs. windows 8 kernel memory protections bypass," http://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memoryprotections-bypass, 2014.

[17] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against Return-to-User attacks," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 459–474. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis

[18] J. Corbet, "Control-flow integrity in 5.13," https://lwn.net/Articles/856514/, May 2021.

[19] ——, "Shadow stacks for 64-bit Arm systems," https://lwn.net/Articles/940403/, Aug. 2023.

[20] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 179–194.

[21] D. P. McKee, Y. Giannaris, C. Ortega, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs," *Proceedings 2022 Network and Distributed System Security Symposium*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248225227

[22] Microsoft, "Data Execution Prevention," Feb. 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[23] "Common Vulnerabilities and Exposures - CVE-2009-1897," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897, 2009.

[24] "Common Vulnerabilities and Exposures - CVE-2009-2908," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2908, 2009.

[25] "Kernel module signing facility," https://www.kernel.org/doc/html/v4.18/admin-guide/module-signing.html.

[26] "Lifting the (hyper) visor: Bypassing samsung's real-time kernel protection," https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html.

[27] A. Limited, "What memory attribute do I set for page table access from MMU Page Table Walk unit," https://developer.arm.com/documentation/ka005348/latest/, Jan. 2024.

[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[29] "Kernel address space layout randomization," https://lwn.net/Articles/569635/, 2013.

[30] "arm64: entry: Hook up entry trampoline to exception vectors," https://patchwork.kernel.org/project/linux-arm-kernel/patch/1512059986-21325-14-git-send-email-will.deacon@arm.com/, 2017.

[31] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[32] Columbia University, "SOSP 21: Artifact Evaluation: Verifying a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware," https://github.com/VeriGu/usenix-ae-linux/tree/bf0e5d0a4ec4f2b6d2524a5c8cd86e12cc0f6e0e, Sep. 2021.

[33] Arm Ltd. (2022) Arm® Architecture Reference Manual for A-profile architecture - DDI 0487I.a.

[34] "Tuning KVM," http://www.linux-kvm.org/page/Tuning_KVM [Accessed: Dec 16, 2020].

[35] S. Hajnoczi, "An Updated Overview of the QEMU Storage Stack," in *LinuxCon Japan 2011*, Yokohama, Japan, Jun. 2011.

[36] "Improve hackbench," http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, 2008.

[37] R. Jones, "Netperf," https://github.com/HewlettPackard/netperf, Accessed 2023.

[38] "Apache http server benchmarking tool," http://httpd.apache.org/docs/2.4/programs/ab.html.

[39] R. Labs, "Memtier benchmark," https://github.com/RedisLabs/memtier_benchmark, Accessed 2024.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[41] 7-CPU.COM, "Applied micro x-gene," https://www.7-cpu.com/cpu/X-Gene.html [Accessed: Apr 28, 2021].

[42] "Common Vulnerabilities and Exposures - CVE-2021-28660," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28660, 2021.

[43] "Common Vulnerabilities and Exposures - CVE-2019-17666," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17666, 2019.

[44] "Common Vulnerabilities and Exposures - CVE-2019-9500," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9500, 2019.

[45] "Common Vulnerabilities and Exposures - CVE-2023-3090," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3090, 2023.

[46] "Common Vulnerabilities and Exposures - CVE-2019-10126," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10126, 2019.

[47] "Common Vulnerabilities and Exposures - CVE-2022-47521," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47521, 2022.

[48] "Common Vulnerabilities and Exposures - CVE-2016-3955," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3955, 2016.

[49] "Common Vulnerabilities and Exposures - CVE-2020-12654," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12654, 2020.

[50] "Common Vulnerabilities and Exposures - CVE-2019-17133," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17133, 2019.

[51] "Common Vulnerabilities and Exposures - CVE-2022-27666," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-27666, 2022.

[52] "Common Vulnerabilities and Exposures - CVE-2023-31436," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-31436, 2023.

[53] "Common Vulnerabilities and Exposures - CVE-2023-32233," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-32233, 2023.

[54] "Common Vulnerabilities and Exposures - CVE-2019-19814," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19814, 2019.

[55] "Common Vulnerabilities and Exposures - CVE-2022-48423," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-48423, 2022.

[56] "Common Vulnerabilities and Exposures - CVE-2019-19378," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19378, 2019.

[57] "Common Vulnerabilities and Exposures - CVE-2023-6238," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-6238, 2023.

[58] D. Calavera and L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[59] X. Li, X. Li, W. Qiang, R. Gu, and J. Nieh, "Spoq: Scaling Machine-Checkable systems verification in coq," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 851–869. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/li-xupeng

[60] C. van Schaik, S. Gamiz, and J. Huang. Gunyah hypervisor. [Online]. Available: https://github.com/quic/gunyah-hypervisor?tab=readme-ov-file

[61] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Bolton Landing, NY, Oct. 2003, pp. 164–177.

[62] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based 'out-of-the-box'semantic view," in *14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (November 2007)*, vol. 10, no. 1315245.1315262, 2007.

[63] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 103–115.

[64] X. Wang, J. Zhang, A. Zhang, and J. Ren, "Tkrd: Trusted kernel rootkit detection for cybersecurity of vms based on machine learning and memory forensic analysis," *Mathematical Biosciences and Engineering*, vol. 16, no. 4, pp. 2650–2667, 2019.

[65] D. Tian, R. Ma, X. Jia, and C. Hu, "A kernel rootkit detection approach based on virtualization and machine learning," *IEEE Access*, vol. 7, pp. 91 657–91 666, 2019.

[66] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 279–290. [Online]. Available: https://doi.org/10.1145/1950365.1950398

[67] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity os from untrusted extensions," in *Network and Distributed System Security Symposium*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:16293986

[68] R. Nikolaev and G. Back, "Virtuos: an operating system with kernel virtualization," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 116–132. [Online]. Available: https://doi.org/10.1145/2517349.2522719

[69] "Virtualization based security," https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs, 2023.

[70] X. Ge and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the TrustZone architecture," in *Proceedings of the Mobile Security Technologies 2014 Workshop*, 2014.

[71] SAMSUNG. (2023) Samsung knox — secure mobile platforms and solutions. https://www.samsungknox.com/.

[72] ——. (2023, Jul.) Sreal-time kernel protection (rkp). https://docs.samsungknox.com/admin/fundamentals/whitepaper/core-platform-security/real-time-kernel-protection/.

[73] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 191–206.