國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

vSECvma：經優化與形式化驗證的 Arm 平台 Linux 核心保護框架

vSECvma: An Optimized and Formally Verified Linux Kernel Protection Framework on Arm

鄭明淵

Beng Yen Teh

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li Ph.D.

中華民國 114 年 10 月

October, 2025

doi:10.6342/NTU202504620

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

vSECvma：經優化與形式化驗證的 Arm 平台 Linux 核心保護框架

vSECvma: An Optimized and Formally Verified Linux Kernel Protection Framework on Arm

本論文係鄭明淵君（學號 R12922178）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 114 年 10 月 08 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 8 October 2025 have examined a Master's thesis entitled above presented by THE, BENG-YEN (student ID: R12922178) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

（指導教授 Advisor）

系主任/所長 Director: 陳祝嵩

doi:10.6342/NTU202504620

致謝

在論文完成之際，首先要獻上我最誠摯的謝意，感謝所有在研究過程中給予我幫助、指導與支持的人。

由衷感謝我的指導教授黎士瑋老師，在整個研究過程中老師給予我無私的指導與支持。不僅在技術層面上提供了精闢的見解與建議，更在研究方向的選擇與論文的撰寫上給予我極大的啟發與鼓勵。老師嚴謹的學術態度與對研究的熱情，將持續成為我未來學術道路上的榜樣。

同時，我也誠摯感謝陳郁方老師在研究過程中所給予的指導與協助。陳老師的建議與討論讓我對形式化驗證的研究內容有了更深入的理解，並對我思考問題的方式帶來了莫大的啟發。

謹向本論文的口試委員：黎士瑋老師、陳郁方老師、林忠緯老師以及王超老師致以最深的謝意。感謝各位老師在百忙之中撥冗審閱本論文，並於口試過程中提供了寶貴的建議與專業指導。老師們深入的提問與精闢的意見，不僅使本研究內容更加完整與嚴謹，也極大地拓展了我的研究視野與思考方向。

此外，我亦要特別感謝 Spoq 專案的作者們，感謝他們開放原始碼與相關資源，使我能夠在本研究中順利地運用 Spoq 進行形式化驗證。若無他們的傑出貢獻，本研究將難以順利完成。其中，尤其感謝 Jason Nieh、Xupeng Li 與 Xuheng Li，他們慷慨分享專案工具的使用經驗與技術細節，對我理解 Spoq 的運作原理與

doi:10.6342/NTU202504620

應用方法有極大的幫助。

同時，感謝 Joey Li 學長與黎老師促成了我們發表 SECvma 論文的機會，使我得以有機會參與學術研討會，拓展了我的研究視野與人脈網絡。在研究與開發過程中，這段交流與合作的歷程讓我獲益良多，充滿了挑戰與樂趣。

感謝所有與我共同討論、協助我解決研究問題，以及聆聽我的研究進度並給予寶貴建議的同學與朋友們。特別感謝朋友馮朗軒、周健烽以及湯士弘同學，在研究過程中的切磋與陪伴。我也要感謝實驗室的夥伴們，感謝你們在研究與生活上的相互扶持與協助，讓這段求學旅程充滿意義與美好的回憶。

最後，也是最重要的，感謝我的家人。感謝您們一路以來的無私支持與鼓勵，讓我在研究道路上得以心無旁慮、全心投入。謹向所有支持與幫助過我的人致以最誠摯的謝意。

doi:10.6342/NTU202504620

摘要

隨着單體式核心如 Linux 的規模與複雜度的不斷提升，其不僅造成了巨大的攻擊面，也引發了頻繁的安全性漏洞。雖然基於虛擬化的核心防護機制能透過硬體強制隔離提供強大的安全性，但它往往會帶來顯著的效能開銷，且僅僅是將信任邊界轉移到管理程式 (Hypervisor)。本論文提出 vSECvma，這是首個針對 Arm 架構、具備高效能且經過功能正確性形式化驗證的 Linux 核心防護框架。我們的工作證明了在強安全性、高效能與形式化保證之間能夠同時達成。在既有研究的基礎上，vSECvma 強化了安全性保證，並引入一系列新穎的最佳化技術，以緩解虛擬化方法所造成的效能負擔，從而維持 Linux 的效率與相容性。我們接著對系統核心進行功能正確性的形式化驗證。此驗證透過多重策略得以實現：重用來自 SeKVM（經過形式化驗證的 KVM 管理程式）的已驗證元件；運用自動化驗證框架 Spoq；並將系統系統化地重構為更適合驗證的模型。本研究最終實現了首個能同時在 Arm 上提供強安全性與高效能，並具備功能正確性機器檢證證明的 Linux 核心防護框架。

關鍵字：作業系統，資訊安全，虛擬化，形式化驗證

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

Abstract

Monolithic kernels such as Linux have grown vastly in complexity, creating a large attack surface with frequent vulnerabilities. While virtualization-based kernel protection offers strong, hardware-enforced isolation, it often incurs significant performance overhead and merely shifts the trust boundary to the hypervisor. This thesis presents vSECvma, the first high-performance Linux kernel protection framework on Arm to be formally verified for functional correctness. Our work demonstrates that it is possible to achieve strong security, high performance, and formal assurance simultaneously. Building on prior work, vSECvma strengthens security guarantees and incorporates a suite of novel optimizations to mitigate the performance overhead of virtualization-based approaches, thereby retaining Linux's efficiency and compatibility. We then formally prove the functional correctness of its system's core. Our verification is made tractable through a multi-faceted strategy: reusing verified components from the SeKVM, a formally verified KVM-based hypervisor; leveraging Spoq, an automated verification framework; and systematically

doi:10.6342/NTU202504620

restructuring the system into a verification-friendly model. Our work delivers the first framework for Linux on Arm that simultaneously achieves strong security and high performance, backed by a machine-checked proof of functional correctness.

Keywords: Operating Systems, Security, Virtualization, Formal Verification

doi:10.6342/NTU202504620

Contents

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

List of Figures

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

List of Tables

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

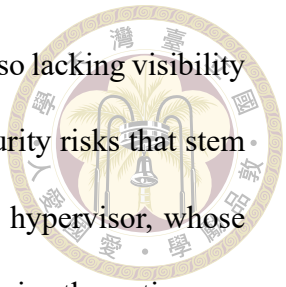
Chapter 1 Introduction

Monolithic OS kernels have become increasingly complex to meet the growing demands for functionality and performance. Linux, for example, is deployed across highly heterogeneous computing environments and has grown to tens of millions of lines of code. This complexity results in hundreds of new common vulnerabilities and exposures (CVEs) being reported each year [46], making the kernel a prime target for attackers. An attacker can exploit kernel vulnerabilities or install rootkits to gain full control over the system and unrestricted access to resources, thereby compromising user safety.

Virtualization-based kernel protection [8, 39, 52, 55, 58, 59, 61, 63, 71] has emerged as a practical and widely explored defense strategy. It leverages hardware virtualization features with a higher-privileged hypervisor to transparently monitor and restrict abnormal kernel behavior from outside the OS. Modern processors now provide virtualization support as a standard capability [64, 68, 69], including mechanisms such as nested page tables (NPTs) to enforce memory isolation between the host OS and guest virtual machines (VMs). The widespread availability of these features makes virtualization-based protection broadly deployable.

However, these approaches face two major challenges. First, establishing an additional protection layer at the hypervisor level often incurs non-trivial performance over-

doi:10.6342/NTU202504620



head from frequent traps and hypervisor-mediated operations, while also lacking visibility into certain OS-level contexts. Second, although this reduces the security risks that stem from kernel vulnerabilities, it merely shifts the trust boundary to the hypervisor, whose own vulnerabilities remain a single point of failure that can compromise the entire system.

Formal verification offers a principled way to address the second challenge. By mathematically proving that system software is correctly implemented and satisfies desired security properties, formal methods provide strong security guarantees that go beyond conventional testing and eliminate broad classes of implementation flaws. Over the years, the field has seen significant advancements, with projects demonstrating the feasibility and benefits of this approach in verifying system software [24, 26, 49], such as OS kernels [30, 31, 38, 54] and hypervisors [15, 42, 47, 48, 66]. However, applying formal methods to complex systems remains immensely difficult. It requires deep understanding of low-level details, substantial expertise in formal methods, and often person-years of effort. The most fundamental and challenging part is proving functional correctness, which involves rigorously demonstrating that the implementation faithfully satisfies an abstract high-level specification. The specification represents the intended behavior of the software as formalized in its formal model. It can then serve as the foundation for establishing high-level security properties.

Building on these insights, we present vSECvma, the first high-performance Linux kernel protection framework that has been formally verified for functional correctness. vSECvma employs a virtualization-based approach to preserve Linux kernel code integrity throughout its lifetime, even against a powerful attacker who has already obtained kernel privileges, while retaining Linux's performance and compatibility. Our work addresses

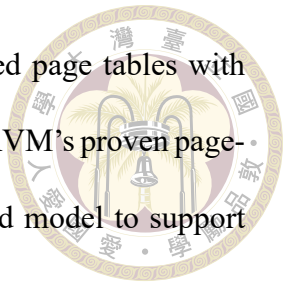
doi:10.6342/NTU202504620

the challenges of both optimization and verification in two steps.

We first design and implement SECvma [73], which builds on our prior work, K-Int [45]. K-Int is a virtualization-based kernel protection framework designed to enforce code integrity of the Linux kernel, prototyped on SeKVM [47, 48], a formally verified KVM-based hypervisor. Building on SeKVM, K-Int provides a strong security foundation for kernel protection while simplifying implementation efforts. However, K-Int has several limitations: it assumes critical system registers controlling the MMU remain benign, fails to prevent attacks such as ret2usr [37], and suffers from significant performance overheads due to nested paging. To address these shortcomings, SECvma strengthens security guarantees and introduces novel optimizations, such as huge-page support and optimized trap handling. Our evaluation shows that SECvma retains Linux performance and compatibility while maintaining kernel code integrity.

We next formally prove the functional correctness of SECvma's TCB. To make this large-scale effort feasible, we reuse verified components of SeKVM. For the novel and extended components introduced by SECvma, we adopt a modular verification strategy [32] and leverage Spoq [50], an automated verification framework that generates specifications and machine-checkable proofs from unmodified system software. However, SECvma's implementation contains complex semantics—nested loops, intricate control flow branches, and pervasive pointer manipulation—that exceed Spoq's direct capabilities. We address this by systematically restructuring SECvma into vSECvma, a verification-friendly model that preserves functionality while enabling automation. With these strategies, we successfully prove the correctness of the framework, generating a top-level specification formally captures the intended behavior of the implementation. Notably, our proof includes the most challenging and security-critical components: the paging and module loading

doi:10.6342/NTU202504620



subsystems. In particular, the paging subsystem manages host nested page tables with hardware-defined semantics. We prove its correctness by adapting SeKVM's proven page-table abstraction methodology [48] and further extending the verified model to support 1GB pages in addition to the existing 4KB and 2MB granularities.

Our work makes the following contributions:

- We design and implement SECvma, a virtualization-based kernel protection framework that enforces Linux kernel code integrity throughout its lifetime. SECvma builds on the K-Int design, strengthens security guarantees, and introduces performance optimizations such as huge-page support and optimized trap handling, while preserving Linux's performance and compatibility.
- We formally prove the functional correctness of SECvma's TCB. We employ a modular approach that reuses verified components from SeKVM and retrofit new components into a verification-friendly model, vSECvma, enabling automated reasoning with the Spoq framework.
- We successfully verified two of the most complex and security-critical components: the paging and module loading subsystems. Our work also extends the verified paging model to support 1GB huge pages.

doi:10.6342/NTU202504620

Chapter 2 Background

2.1 Formally Verified Systems

To address the growing complexity of modern system software, which potentially introduces subtle bugs and security vulnerabilities, formal verification has emerged as a promising solution to ensure the correctness and reliability of system implementations. Formal verification employs mathematical techniques to prove that a system's implementation satisfies its formal specification, guaranteeing key properties like safety, integrity, or confidentiality under all specified conditions.

This potential has been realized in practice through a number of verified systems, such as microkernels and hypervisors. Pioneering examples include the seL4 microkernel [38], the first formally verified OS kernel, which leveraged the Isabelle/HOL theorem prover [56] to formally prove the functional correctness of its C code implementation. CertiKOS [30, 31] builds a certified concurrent OS kernel using a layered verification methodology in Coq [4]. It leverages the CompCert verified C compiler [44] to extend its end-to-end correctness proofs reliably down to assembly level correctness. SeKVM [47, 48] retrofits a commodity multiprocessor KVM hypervisor and formally verifies its critical components that manage virtual machines (VMs) to ensure the integrity and confidentiality of VMs. Li et al. formally verified the correctness of SeKVM via mechanized

doi:10.6342/NTU202504620

refinement proofs in Coq, building a low-level Arm machine model that captures page table management, TLB management, and a coherent cache hierarchy with cache-bypass support. Tao et al. [66] introduced the VRM framework and formally verified SeKVM, ensuring that proofs established under sequential consistency (SC) also hold on Arm's relaxed memory hardware.

2.2 Spoq

Formal verification of complex system software is inherently difficult, particularly when dealing with low-level implementations in languages like C. Reasoning about intricate behaviors such as pointer arithmetic, type casting, low-level memory and register operations, and inline assembly code is inherently complex. Bridging the large semantic gap between low-level implementations and high-level formal specifications frequently requires developers to invest significant manual effort in writing specifications and proofs, which is difficult and error-prone. Any changes to the verified implementation can further invalidate existing proofs, imposing substantial engineering overhead for re-verification. These difficulties have motivated the development of verification frameworks [16, 18, 32, 34, 43, 50, 60, 65, 72] that aim to reduce manual effort and improve the scalability of formal verification.

Spoq [50] is an automated verification framework designed to generate machine-checkable proofs for system software. It significantly reduces the verification costs by automating the process of generating specifications and proofs directly from software implementations. Traditionally, specifications are manually written to describe the intended abstract behavior of a system and to capture the system's invariants. However, construct-

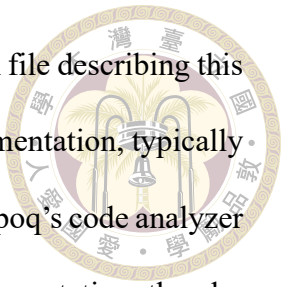
doi:10.6342/NTU202504620

ing and maintaining such specifications for complex system requires substantial manual effort, and the process is prone to human error. In contrast, Spoq's specifications are implementation-driven. They are automatically derived from the system implementation, capturing most of its operational behavior. This approach greatly reduces manual modeling effort and the likelihood of human error, while still allowing developers to introduce additional abstract higher-level abstractions when reasoning about functional correctness or security properties. Unlike manually written specifications, Spoq-generated specifications faithfully capture the implementation's behavior. Consequently, any unintended behavior in the implementation will also be inherited by the generated specifications unless explicitly ruled out or constrained by separately proven invariants.

Spoq's verification process requires only minor code modifications and structural adjustments, guided by a principled methodology. To achieve large-scale automation, Spoq adopts a layering proof strategy based on Concurrent Certified Abstraction Layers (CCAL) [32]. It allows the decomposition of the complex verification process into multiple, independent layers, thereby simplifying the overall proof. Following the CCAL methodology, the system implementation is modularized into distinct layers, where each layer encapsulates a set of functions. Functions within these modules can be verified independently, yielding composable proofs. Each lower layer exposes a high-level, abstract interface to the layers above it, enabling higher layers to reason about functionality using these abstract interfaces, without being burdened by the intricate implementation details of the layers below. However, a key constraint of this approach is that functions within a layer are prohibited from invoking functions in the same or higher layers.

Spoq's Layered Verification Workflow. To scale and automate the verification process, a system's implementation is first refactored to conform to the layer structure

doi:10.6342/NTU202504620



defined by CCALs. The developer provides Spoq with a configuration file describing this layer hierarchy. The workflow begins by translating the system implementation, typically written in C, into LLVM IR using the Clang compiler. Subsequently, Spoq's code analyzer transforms the LLVM IR into a Coq Abstract Syntax Tree (AST) representation, thereby eliminating the complexities of directly handling C semantics.

Following this, Spoq utilizes the provided configuration file to automatically generate a set of specifications and proofs for each defined layer. Based on the CCAL methodology, each proof module comprises three key components: *a layer implementation*, *a low-level interface*, and *a high-level interface*. To construct the proof for each module, Spoq introduces two fundamental types of refinement proofs for each function within the layer: the *identical refinement* and the *lifting refinement*. The former proves that the layer implementation built on top of the low-level interface correctly refines a low-level specification. The lifting refinement proves that this low-level specification ultimately refines the high-level specification with the refinement relation. The bottom layer serves as a machine model layer, which is a trusted layer that models low-level LLVM IR semantics, encompassing aspects such as register states, memory operations, and object pointers. Spoq provides a suite of Coq libraries to support these semantics as parsed from LLVM IR, simplifying the modeling of hardware and low-level software behaviors. The top-most layer culminates in a high-level specification of the entire system's intended behavior, which can then be used to formally prove critical properties, such as security invariants.

Specifications and Proofs Generation. Spoq automatically generates both specifications and proofs. To generate low-level specifications, Spoq performs a single-step transition from each instruction within the parsed Coq AST, progressively aggregating these individual instruction behaviors to form a comprehensive specification for the entire

doi:10.6342/NTU202504620

function that is close to actual code behavior. Spoq then applies a set of transformation rules to systematically lift these low-level specifications to their corresponding high-level specifications, leveraging the Z3 SMT solver [25] to discharge symbolic execution obligations and perform mathematical simplifications. The high-level specifications are ” self-contained”, meaning they contain no calls to low-layer functions, as those calls are unfolded during the lifting process. Neither Spoq’s libraries nor the generator needs to be trusted, as incorrect generation of specifications and proofs will be automatically detected and rejected by the Coq proof checker.

Spoq employs static proof strategies to generate proofs for each function, building them from generic and reusable templates. Distinct templates are used for functions with and without loops. For loop-free functions, Spoq applies a case analysis strategy to recursively decompose each conditional branch function into two sub-proofs, proving each execution path under a specific condition. For functions containing loops, Spoq applies an induction-based proof template. To assist in proving loop termination, the user is required to provide a ”ranking function” in the configuration file. However, auto-generated proofs are not always sufficient, especially when dealing with complex functions that involve intricate branching, loop bodies with additional conditions, or early-exit constructs (e.g., `break`, `continue`). Such functions often span multiple basic blocks with complex control flow, making generic proof templates difficult to reason about. Even so, they provide a robust starting point. Although highly complex functions require manual refinement, most proofs can be completed with only minor user intervention or small adjustments to the proof template.

doi:10.6342/NTU202504620

2.3 Introduction to K-Int

Kernel integrity protection is a fundamental security property ensuring that the kernel code and critical data are not maliciously altered. Monolithic OS kernels, such as Linux, serve as the core component of an operating system, managing system resources and performing privileged tasks. Protecting their integrity is therefore essential to prevent full system compromise. One widely adopted defense strategy is to rely on a privileged system software, such as a hypervisor, cooperating with hardware features, to enforce robust protection even after the kernel's privileges have been breached.

K-Int [45] is a system that resides at the hypervisor level (EL2 on Arm) and leverages Arm Virtualization Extensions (VE) to enforce kernel code integrity. It aims to integrate with commodity hypervisors, reducing implementation effort while preserving essential hypervisor functionality, such as support for virtual machines. Currently, K-Int enforces several key protections, including **Kernel Code Integrity**, **Data Execution Prevention (DEP)**, and **Kernel Page Table Protection**. It also supports **Loadable Kernel Modules (LKMs)**, enabling users to install drivers as kernel extensions while prohibiting the installation of unauthorized modules.

Kernel Code Integrity and Data Execution Prevention. To enforce kernel code integrity and DEP, K-Int adopts the memory usage tracking mechanism that assigns a specific usage type to each memory page during system initialization (e.g., kernel code or kernel data). K-Int leverages Arm VE by enabling stage 2 paging for the host Linux. When the Linux host accesses memory via a virtual address (VA), the VA first translates to an intermediate physical address (IPA) through the Stage 1 Page Table (S1PT). This IPA is then translated to a physical address (PA) through the Stage 2 Page Table (S2PT), so

doi:10.6342/NTU202504620

that the host cannot directly access physical memory. This allows K-Int to transparently interpose on all host memory accesses via host S2PT. K-Int handles S2 page faults from the host and employs identity mapping (where IPA is identical to PA), simplifying address management and minimizing TCB complexity. Based on the assigned memory usage type, K-Int enforces the appropriate S2 permissions (e.g., read-execute for kernel code, non-executable for kernel data) of mapped memory in host S2PT.

Kernel Page Table Protection. While K-Int effectively protects static kernel memory regions initialized during system boot, it also addresses threats stemming from dynamically allocated pages. The Linux kernel allocator allocates pages from the kernel heap, which can be subsequently used for various purposes such as S1PT pages, kernel data, or user memory. Since K-Int lacks specific knowledge of the intended use for these newly allocated pages (as they are not assigned a distinct usage type during runtime), these pages are granted full permissions in S2PT. An attacker could potentially perform page table manipulation attacks by tampering with S1PT entries, for example, by enabling Privileged Executable (PX) permissions, leading to arbitrary code execution attacks. To mitigate such threats, K-Int implements strict policies and write-protects the kernel page table (read-only in the S2PT). Any update to the kernel page table triggers a trap to K-Int, where the modification is validated against the predefined policies. These policies ensure that all non-executable memory pages mapped by the kernel are enforced as Privileged Non-eXecute (PXN). Furthermore, they also ensure that kernel page mappings form a directed acyclic graph (DAG) structure and prevent attackers from altering existing mappings to arbitrary non-zero addresses. These enforcements are critical to prevent bypassing the protections enforced by K-Int.

Loadable Kernel Modules. To support loadable kernel modules, K-Int uses public-

doi:10.6342/NTU202504620

key cryptography to ensure the integrity of loaded modules. It relies on a whitelist of valid signatures and keys, which are assumed to be securely sealed in secure storage before the kernel is compromised. In contrast to Linux's built-in module signing infrastructure [2], K-Int minimizes its TCB on handling module loading. It achieves this by delegating non-security-critical tasks like memory allocation to Linux, while retaining security-critical operations such as authentication and relocation within the K-Int. K-Int preserves Linux's functionality while ensuring that only authorized modules can be installed and executed.

K-Int's Limitations. Although K-Int effectively enforces kernel code integrity throughout its lifetime, it still relies on several trusted assumptions that introduce significant limitations and potential vulnerabilities. First, K-Int does not protect critical system registers, allowing an attacker to manipulate register values that control virtual memory and thereby bypass its protection. Second, K-Int does not provide protection for user page tables, leaving them vulnerable to direct manipulation. An attacker could exploit kernel vulnerabilities to alter user page table entries with elevated permissions, enabling attacks such as return-to-user (ret2usr [37]), where user code is executed with kernel privileges. Third, enabling stage 2 paging introduces significant runtime overhead due to additional page walks and increased pressure on the TLB cache, which is a hardware cache commonly employed across system architectures to accelerate page translation. Finally, K-Int's implementation is potentially vulnerable to flaws since it has not been formally verified.

doi:10.6342/NTU202504620

Chapter 3 Threat Model And Assumptions

We protect the code integrity of the Linux kernel. We assume the system is intially benign but may later be compromised by a remote attacker, including administrators with privileged remote access to the hardware. The attacker may exploit zero-day vulnerabilities in the Linux kernel to gain arbitrary memory write capabilities. Once compromised, the attacker can attempt to overwrite the existing kernel code, emit new code, or manipulate kernel page tables [6, 41] to modify the access permissions or page mapping in page table entries. The attacker may also control devices to perform arbitrary DMA operations or reprogram Arm system registers to manipulate the MMU. In addition, the attacker may install kernel rootkits or launch ret2usr attacks [37] to run malicious code with kernel privileges. Protection against data-only or code-reuse attacks is out of scope. Mechanisms to secure control flow integrity [22, 23, 29] or compartmentalization [51] could be employed to defend against these attacks. Denial of service attacks, side channels, and physical attacks against Linux are excluded from the threat model. We trust the Arm hardware platform, the machine model, the Clang/LLVM toolchain (including IR translation), the Spoq verification framework [50], and the Coq proof checker [4].

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

Chapter 4 Design

This chapter details the design of our system, which was developed in two primary stages: (1) **SECvma**, an optimized kernel protection framework, and (2) **vSECvma**, its formally verified model. Figure 4.1 illustrates the overall system architecture. KPCore serves as a trusted hypervisor core that enforces Linux kernel code integrity while preserving the functionality of commodity hypervisor such as virtual machine support.

SECvma extends K-Int, addressing its limitations and strengthening its security guarantees to ensure kernel code integrity. However, expanding security enforcement inherently adds runtime overhead. To overcome this, SECvma incorporates a suite of novel optimizations—such as mitigating nested paging overhead and reducing trap costs from register protection—aimed at preserving Linux's performance while upholding its protection guarantees.

The second stage is to formally verify SECvma's TCB (KPCore). Specifically, our primary goal is to prove its functional correctness, demonstrating that the implementation correctly refines a high-level formal specification. To manage the complexity of this task, we adopt modular verification based on the CCALs methodology, in which KPCore is decomposed into multiple independent, layered modules. Functions within these modules can be verified independently, yielding composable proofs that build towards verifying

doi:10.6342/NTU202504620

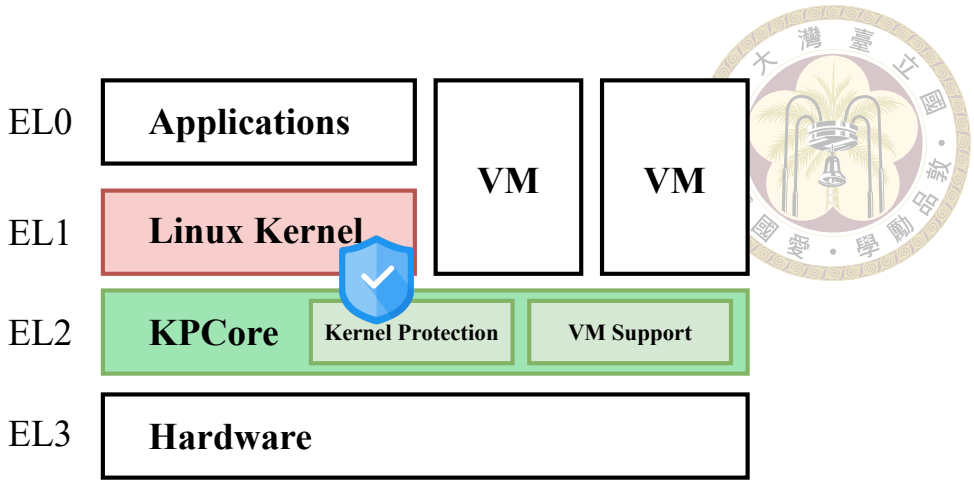


Figure 4.1: vSECvma System Architecture

the entire TCB.

We build our prototype on the formally verified SeKVM hypervisor, which already employs the same layering strategy. This allows us to leverage SeKVM's existing proof efforts and focus on the novel and extended modules. For these new components, we use the Spoq framework to automate the generation of specifications and proofs. We introduce restructuring strategies to systematically transform SECvma into vSECvma to make the system amenable to automated verification. The final, composed proof establishes the TCB's functional correctness and yields a verified high-level specification, which serves as a trusted foundation for subsequently proving desired invariants. Spoq's specifications are implementation-driven, they will inherently reflect any unintended behaviors present in the code. Nevertheless, our framework still provides fundamental correctness guarantees built upon our machine model and several manually-written specifications. For example, we provide assertions in the machine model to enforce well-formedness conditions such as valid pointer dereferences, checks against out-of-bound memory accesses. In addition, we introduce data abstractions, such as representing S2PTs as well-formed tree structures that preserve essential properties. These guarantees will be formalized and detailed in the following discussion and in Chapter 5.

doi:10.6342/NTU202504620

4.1 Security Enhancements

KPCore, served as SECvma's TCB, builds upon the foundation of K-Int, providing essential kernel protections, such as data execution prevention, kernel code integrity enforcement, and kernel page table protection. It supports loadable kernel modules, enabling users to install authorized drivers as kernel extensions and execute driver code without kernel recompilation. As Section 2.3 mentioned, K-Int's design still relies on certain trusted assumptions that can potentially compromise kernel code integrity. The following sections describe the limitations that SECvma addresses, providing substantially stronger guarantees for maintaining kernel integrity.

4.1.1 System Register Protection

K-Int assumes that system register values are benign. However, malicious manipulation of these registers can compromise kernel code integrity. For instance, an attacker could manipulate the value of the Arm register `TTBR1_EL1`, which stores the base address of the kernel page table, to bypass K-Int's enforced kernel page table protection. SECvma addresses this vulnerability by employing a trap-and-emulate approach, preventing malicious manipulation of Arm's system registers that control virtual memory to compromise kernel code integrity.

SECvma categorizes these system registers into four categories shown in Table 4.1. 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

doi:10.6342/NTU202504620

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

Table 4.1: Virtual Memory System Registers

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 4.1 to EL2. This allows KPCore to transparently interpose every register update and apply the appropriate protection strategies based on the register's category.

4.1.2 Supervisor Mode Execution Prevention

K-Int write-protects the kernel page table and applies privileged non-executable (PXN) permissions to all non-executable memory pages to prevent malicious attackers from executing arbitrary code mapped by it. However, it lacks comprehensive protection for user page tables. A malicious attacker might manipulate user page table entries—for instance, by granting privileged executable (PX) permissions—and subsequently perform attacks such as return-to-user (ret2usr) attacks that execute arbitrary user code with kernel privileges. However, a significant challenge arises from the dynamic allocation of user page tables by the kernel allocator each time a new user process is created. Unlike the kernel page table, which is statically initialized at boot and shared across all processes, directly applying the same monitoring strategy to all user page table updates becomes infeasible with a dynamic kernel allocator. A naive approach would involve SECvma implementing its own page allocator for user page table allocation. However, this would significantly

doi:10.6342/NTU202504620

increase the TCB complexity. Furthermore, directly write-protecting all user page tables would incur an excessive number of traps during frequent updates, leading to severe performance degradation.

To address these limitations, SECvma introduces a novel approach to monitor user page table updates, while incurring minimal trap overhead. To enforce Supervisor Mode Execution Prevention (SMEP) [27], SECvma write-protects the first-level user page tables and enforces the PXN bit, ensuring that all subsequent page table entries inherit PXN protection. Then, SECvma introduces two hypercalls to interpose Linux's user page table allocation and deallocation processes. When the kernel allocates the first-level user page table, it invokes the hypercall, causing a trap to SECvma. SECvma first verifies that the provided page is empty with no prior usage. It then assigns a new usage type (denoted as `UPGD`) to the page, explicitly marking it as a user page table.

It is known that when a user process is scheduled to run, the scheduler switches the active page table to that of the running process by modifying the `TTBR0_EL1` register. Through the mechanism introduced in Section 4.1.1, SECvma intercepts write accesses to `TTBR0_EL1`. Upon such access, SECvma could validate that the target page table has a valid `UPGD`s identity assigned during the earlier allocation phase. As a result, the host kernel is compelled to invoke the SECvma hypercalls for legitimate page table setup. Otherwise, an invalid page table cannot be installed via `TTBR0_EL1` and used for execution. By doing so, SECvma achieves comprehensive supervisor mode execution prevention.

doi:10.6342/NTU202504620

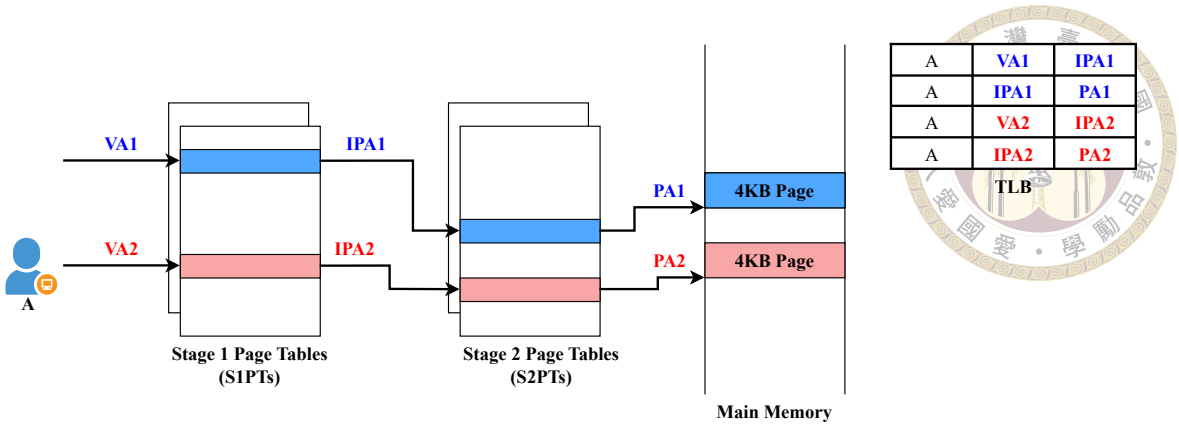
4.2 Optimization

SECvma's design introduces performance overhead to enforce Linux kernel integrity throughout its runtime. SECvma adopts stage 2 paging and applies different permissions based on different memory usage types to protect kernel memory. To achieve memory access control at the finest granularity, SECvma uses the minimal (i.e., 4KB) translation granule in the host stage 2 page table (S2PT) by default. This approach incurs a significant performance slowdown due to the requirement for extra page walks and increased pressure on the TLB cache, as shown example in Figure 4.2a. Additionally, the system register protection also introduces significant overhead. The registers that are frequently runtime-updated by Linux can cause an excessive number of traps to KPCore for validation. We introduce optimizations to mitigate the trade-off caused by these approaches.

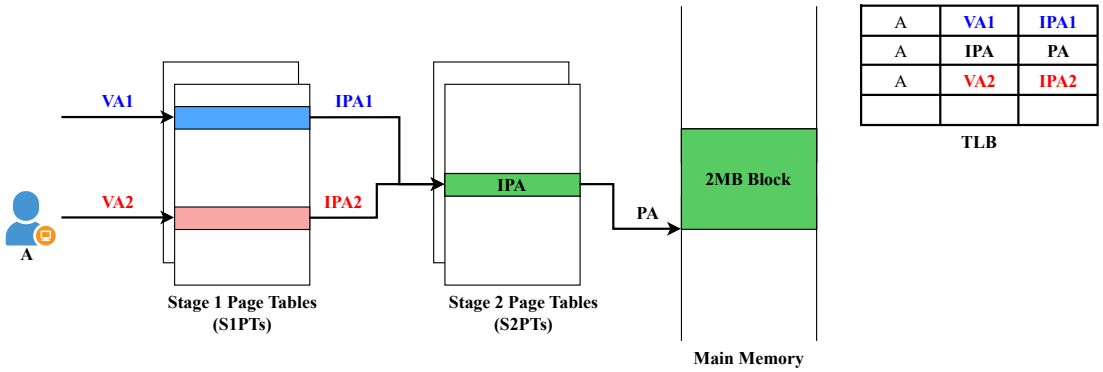
Huge Page Optimization. As illustrated in Figure 4.2b, we adopt huge page optimization to address the slowdown caused by S2 paging. However, this approach presents several challenges. First, to enforce precise memory access control on huge page mappings, we must ensure that all memory pages within a huge block share the same S2 permissions, thereby preventing unintended permission overlaps with different memory regions. Second, we need to minimize external fragmentation that could arise from the use of huge pages.

KPCore manages S2 mappings with different granularity sizes (e.g., Arm's 4KB, 2MB, and 1GB mappings), determined by the mapped region and its associated memory usage. To effectively apply these optimizations, we modified Linux's linker script to ensure all static memory within kernel binaries (i.e., kernel code and data) is loaded and aligned to 2MB regions during boot. This prevents kernel code and data from co-locating

doi:10.6342/NTU202504620



(a) When stage 2 paging is enabled, both stage 1 (VA2|IPA2) and stage 2 (IPA2|PA2) translations are cached in TLB. Using 4KB mappings significant pressure on the TLB cache during runtime.



(b) VA1 and VA2 reside within the contiguous 2MB block (IPA) and share stage 2 permissions, SECvma maps both entries to a single 2MB S2 entry, reducing TLB pressure.

Figure 4.2: Huge Page Optimization.

within the same 2MB huge page, concurrently minimizing external fragmentation in these static areas.

By default, Linux allocates kernel pages from the kernel heap. We cannot directly map heap memory in the host S2PT with huge pages because 4KB pages with different S2 permissions could co-locate within the same huge page. To incorporate huge page optimization for dynamic kernel memory, we introduced a buddy memory allocator to Linux. This allocator is designed to provide 4KB-aligned pages from 2MB-aligned memory pools that are guaranteed to share the same permission in the host S2PT. For instance, the enlight-

doi:10.6342/NTU202504620

ened Linux kernel can allocate page table pages from these specifically managed pools. When handling mapping traps to KPCore, KPCore can then apply the corresponding (e.g., read-only) permissions to these huge mappings.

We further extend huge page optimizations to support 1GB mappings. Currently, 1GB mappings are applied only to memory without specific usage types (i.e., user memory), with an explicit enforcement that no other memory pages with defined usages are co-located within the same 1GB region. These optimizations are transparent to Linux and do not change its page granularity, thus incurring no additional fragmentation or pressure during process creation and caching operations.

By enabling huge page optimizations, we introduce modifications to Linux, such as aligning static memory regions and creating specialized memory pools, with minimal impact on existing Linux's functionality. However, it is important to note that Linux itself may apply optimizations to the S1 mappings. SECvma supports S1 mapping optimizations performed by Linux but prevents attackers from tampering with huge page entries in the S1PTs that potentially lead to privilege escalation and bypass KPCore's preventions.

System Register Optimization. As previously discussed, KPCore employs a trap-and-emulate mechanism to validate Linux's write accesses to system registers. We analyzed the usage patterns of these system registers across different scenarios and applied optimizations to reduce traps in two main contexts:

Batching System Register Updates: In scenarios where Linux performs multiple system register updates within a single process, KPCore introduces batching optimizations to consolidate these operations into a single hypercall. Rather than trapping and validating each update individually, KPCore validates all updates collectively upon handling the

doi:10.6342/NTU202504620

hypercall. For instance, during process context switches, Linux introduced three updates to replace the stage 1 user and kernel page tables (i.e., `TTBR0_EL1` and `TTBR1_EL1`) of the next active process. By batching these updates, SECvma reduces the number of traps to KPCore from multiple individual traps to a single hypercall, reducing overhead during frequent context switches.

Replacing scratch registers: For specific mechanisms and scenarios, we observed that Linux may use system registers as scratchpads for temporary storage. To address the frequent traps caused by system register protection in these cases, we modified Linux to replace the use of such registers with alternative registers while ensuring the modifications do not compromise system behavior or security guarantees. For example, Linux's KPTI [21] implementation uses `FAR_EL1` as scratch registers during kernel/user mode switches [7]. Consequently, the writes to `FAR_EL1` result in traps to KPCore. To avoid the trap, we replaced the write to `FAR_EL1` with `TPIDRRO_EL0`. This incurs no safety issue since 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.

4.3 Adapting KPCore to Spoq Framework

SECvma is prototyped on SeKVM [47, 48], a formally verified KVM-based hypervisor for Arm. Spoq has also evaluated the correctness proofs of SeKVM through its open-source artifact [19]. We reuse and extend SeKVM's verified functionality, including S2PT management, exception handling, and memory usage mechanisms. Building on these components, we apply similar proof strategies to re-verify the extensions introduced by KPCore.

doi:10.6342/NTU202504620

To adapt KPCore to Spoq framework, we first refactor its architecture into Spoq's layered verification model. In this model, functions in higher layers can only invoke functions in lower layers, and not vice versa. For example, the complex call patterns, such as recursion, are prohibited. Moreover, KPCore contains complex routines, such as kernel module loading, that are difficult to verify automatically. To manage these complexities, we applied several restructuring strategies, including encapsulating pointers, modularizing branches and loops, simplifying generated specifications through directives, and introducing manual data abstractions. These adaptations are essential for meeting the requirements of the Spoq framework and, just as importantly, for improving its performance in automatically generating complete specifications and composable proofs for every layer of KPCore.

Type-Aware Pointer Validation. In low-level systems like SECvma, memory pointers are extensively used to access and pass structured data. A key challenge is bridging the semantic gap between typed pointers preserved in the Coq AST and their representation in the machine model. Spoq provides Coq libraries for modeling low-level primitives such as object pointers as `(base, ofs)` pairs. However, Spoq's layering strategy introduces challenges when pointers are passed across different layers. If a pointer originates in the same or a lower layer, its source can be tracked and verified through the layer's exposed state and invariants. In contrast, when a pointer is passed from an upper layer, Spoq cannot assume what the pointer references. To ensure soundness, Spoq generates specifications using case analysis over all modeled object pointers and constructs conditional branches to check whether the `(base, ofs)` falls within a valid object footprint. If none of the cases match when the pointer source is revealed, the pointer evaluates to `None` in the specification, and any subsequent operations on it will raise a fault in Spoq, exposing a potential

doi:10.6342/NTU202504620

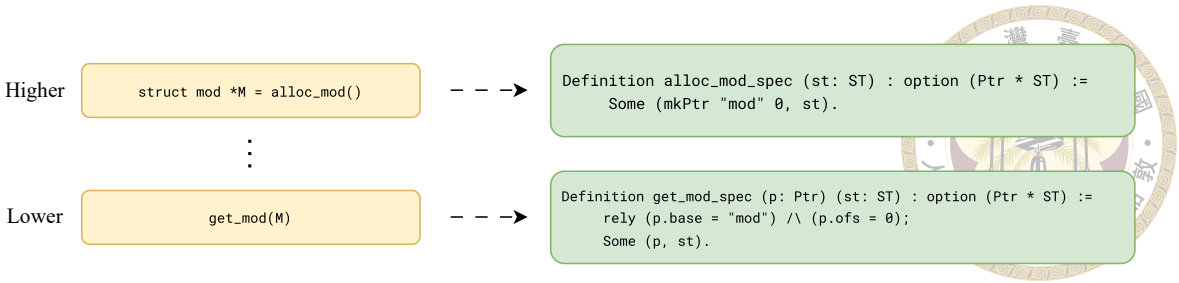


Figure 4.3: Encapsulating raw pointers from higher-layers with a getter wrapper.

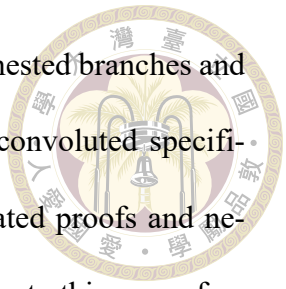
bug. While this approach ensures correctness, it leads to bloated specifications and slows down proof automation.

To address this, we encapsulate the pointers from upper layers within our machine model and mediate them through type-specific ”getter” functions. As illustrated in Figure 4.3, a higher-layer function constructs a pointer `*M` and passes it to a lower layer for further use. Since the low-level specification represents such pointers generically as `Ptr`, without type information, we introduce a getter `get_mod()` to mediate these accesses. However, we cannot simply assume that the pointer is valid (`*M`) and of the correct type (`struct mod`). It is unsafe, as an invalid pointer passed from the upper layer can cause the proof to succeed under a false assumption. Instead, we introduce assertions using the `rely` construct, previously leveraged in SeKVM's correctness verification with Spoq.

The `rely` construct is implemented in Coq as a pattern-matching expression (`match ... with`) that checks the validity of conditions. These assertions are automatically validated by Spoq through symbolic execution with the Z3 SMT solver during the generation of high-level specifications. If a condition evaluates to `False`, the specification for that execution path collapses to `None`, meaning that the subsequent function behavior is not executed. As a result, when an expected execution path incorrectly leads to `None`, it immediately exposes a potential bug in the implementation.

Modularizing Branches and Loops. In processes such as kernel module load-

doi:10.6342/NTU202504620



ing, several functions exhibit intricate control flow, involving deeply nested branches and loops with non-trivial logic. These structures often lead to highly convoluted specifications generated by Spoq, potentially overwhelming its auto-generated proofs and necessitating manual intervention to complete the verification. To mitigate this, we refactored several functions by modularizing complex branches and loops. For example, as illustrated in Listing 1, we addressed functions with intricate branching and duplicated conditions by extracting the logic for dynamically constructing function arguments into dedicated getter functions. Instead of directly passing different arguments within each branch, we factored out argument construction into a separate `get_arg0` function. These getter functions calculate and return the necessary structured arguments, which are then passed to a unified function call. As a result, the switch-case structure becomes more concise to select the function to call, while the argument retrieval is encapsulated in a single function wrapper. Similarly, for loops with complex bodies, we encapsulated the loop's logic into multiple helper functions at a lower layer, so that we can significantly facilitate Spoq's auto-generated proof templates, allowing them to automatically prove the containing function with minimal manual effort.

```
switch (cond) {
    case A:
        funcA(a, ...)
    case B:
        funcB(b, ...)
    case C:
        funcB(c, ...)
    case D:
        funcA(b, ...)
    ...
}
```

```
arg0 = get_arg0(cond);

switch (cond) {
    case A:
    case D:
        funcA(arg0, ...)
    case B:
    case C:
        funcB(arg0, ...)
    ...
}
```

Listing 1: Modularizing complex branch logic.

Simplifying Specifications for Scalable Proofs. As previously discussed, the high-level specifications generated by Spoq are self-contained, derived by unfolding lower-layer specifications that contain no further calls to lower layers. However, for a large-scale

doi:10.6342/NTU202504620

system like SECvma, these specifications often remain too complex to prove efficiently. This is because fully unfolding all low-level details can lead to a combinatorial explosion of states that overwhelm automated theorem provers. To manage this complexity, we employ two key, complementary strategies, depending on how a function is used across the system:

- *Manual Abstract Specifications:* For low-level and widely reused functions, such as general-purpose helpers that involve complex routines, we manually write abstract high-level specifications rather than relying solely on Spoq-generated ones. For example, stage 2 paging is widely used by higher-layer modules to enforce memory protection, yet its implementation involves intricate routines that traverse multi-level page tables to query or update entries. To manage this complexity, we manually define an abstract specification that captures the function's intended behavior and then construct refinement relations to formally prove that the low-level implementation correctly refines this specification. By doing so, we can focus on verifying essential properties without getting bogged down in implementation details. However, manually written specifications carry the inherent risk of containing errors by human efforts, which could compromise the soundness of the overall proof.

- *Selective Use of NoUnfold:* For higher-layer functions that are not helpers but instead provide relatively complete and self-contained functionality, we selectively apply the `NoUnfold` directive to prevent their high-level specifications from being automatically unfolded during proof construction. A prime example is the module loading process, which consists of sequential subprocesses like symbol resolution and relocation. Since each subprocess has been independently verified, operates

doi:10.6342/NTU202504620

with minimal shared state, and exposes a well-defined interface, higher layers can safely invoke them as trusted, opaque operations without re-verifying their underlying implementation. This abstraction reduces proof size and complexity. To mitigate the risk of masking behaviors critical for system-wide reasoning, we apply `NoUnfold` conservatively and only when such structural properties are present.

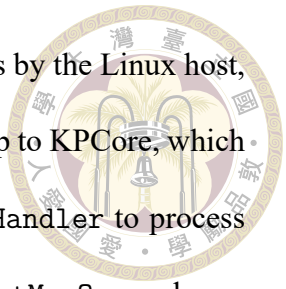
These two approaches help us manage proof complexity in practice and were also applied in SeKVM's correctness proof. While each carries potential risks to soundness, we apply them conservatively and only when necessary, making them a practical compromise for reasoning about a full system implementation.

4.3.1 Layered Design

Figure 4.4 illustrates the layered design of KPCore. Building on SeKVM, KPCore reuses and extends SeKVM's existing modules. It also introduces new modules to enforce kernel code integrity. The newly introduced modules are organized into three main security components: **Kernel Memory Protection**, **Secure Module Loading**, and **System Register Protection**. Each component is further decomposed into multiple layers, with each layer encapsulating a group of functions responsible for specific tasks. The top-layer module, `TrapHandler`, is responsible for handling exceptions and hypercalls (as summarized in Table 4.2) from the host kernel. It then dispatches these events by invoking the top-level interfaces exposed by the three primary security components.

Kernel Memory Protection. KPCore provides kernel memory protection through two key modules: `HostMemHandler` and `MemoryOps`. These modules are responsible for handling stage 2 permission faults and stage 2 page faults caused by the host, respectively.

doi:10.6342/NTU202504620



Since KPCore write-protects the stage 1 page table to monitor updates by the Linux host, any write access to these page tables will trigger a permission fault trap to KPCore, which is captured by `TrapHandler`. `TrapHandler` then invokes `HostMemHandler` to process the fault. `HostMemHandler` retrieves relevant information from `HostMemOps`, a lower layer providing helper functions related to memory operations, such as fetching the faulting instruction, and the page table entry being written. Concurrently, it tracks the memory usage of the faulting address by invoking `PageManager`, determining whether it belongs to a kernel or user page table. According to the memory usage, it dispatches the request to the appropriate handler within `HostPtHandler`. `HostPtHandler` then interprets the operation performed by the host (i.e., mapping, unmapping, or updating a page table entry) and further delegates the task to functions in `HostPtOps` and `HostPtAux`. If the stage 1 page table operation is validated, `HostNptOps` is invoked to perform the corresponding stage 2 remapping, ensuring the nested page tables remain synchronized with the updated memory usage and its associated permissions. `HostInsnHandler` and `HostInsnAsmHandler` responsible to emulate the instructions issued by the host that caused the trap. Between this, `HostMapOps` is called by upper layers to perform map permission checks and updates as needed. To support user page table updates, KPCore introduces two hypercall handlers `alloc_el0_pgd` and `free_el0_pgd` in `HostPtOps`. These are triggered by `TrapHandler` when the host explicitly requests allocation or deallocation of user page tables.

When a stage 2 page fault occurs, `MemoryOps` is responsible for handling the fault. It first invokes `PageManager` and `HostMapOps` to track the memory usage of the faulting address, retrieving information to determine the final mapped permission and the mapping level. Subsequently, `S2PTOps` and `S2PTTreeOps` are triggered for handling stage 2 page table updates. During this process, `S2PTWalk` is used to traverse the stage 2 page table

doi:10.6342/NTU202504620

across multiple levels. If any intermediate page table is missing, `S2PTAlloc` is responsible for managing the allocation for the stage 2 page table.

Secure Module Loading. KPCore exposes three hypercalls, `mod_auth`, `free_init`, and `free_module`, to support dynamic kernel module loading. These hypercalls are invoked by the host kernel during module installation or removal, intercepted by `TrapHandler`, and dispatched to corresponding handlers in the `ModHandler` layer. To support secure module loading, KPCore adopts a layered design that divides the loading process into multiple stages. Each stage exposes a top-level interface, invoked by `ModLoadOps`, which coordinates a top-down approach across submodules responsible for specific tasks, including module initialization, authentication, and symbol resolution with relocation. The process begins with the `ModInit*` layer, which collects and organizes essential metadata passed from the kernel. Next, similar to the Linux kernel, the `ModLayout` layer constructs the module's memory layout, such as handling text sections, read-only sections, and other segments. These sections are then stacked in preparation for signature verification. Then, `ModAuth` performs module authentication to the stacked module content. Once authorized, KPCore proceeds to the `ModSym*` layer to resolve undefined symbols. Then, the `ModReloc*` layer applies relocations to patch the module contents according to the resolved symbols. The lower-level module `ModCore` provides reusable utilities and helpers to support the various stages of this process. Meanwhile, `ModCoreInfo` stores persistent information about the loaded module, such as the address of its protected sections and its exported symbol table. `ModPage*` manages both stage 1 and stage 2 page-level operations. For example, before authentication, KPCore write-protects the module pages to prevent any unauthorized access by the host. Only after the module is successfully authenticated does KPCore allow it to be installed and executed.

doi:10.6342/NTU202504620

`mod_auth(info, list, size)`
`free_init(mod_id)`
`free_module(mod_id)`
`opt_switch_mm(ttbr0, asid)`
`alloc_el0_pgd(addr)`
`free_el0_pgd(addr)`

Table 4.2: Kernel Protection Hypercalls

System Register Protection. KPCore interposes on every host write to system registers listed in Table 4.1. `SysRegHandler` handles the fault and only permits writes to registers categorized as *Runtime-Updated Registers*, rejecting all other accesses. It then delegates to the corresponding register handlers defined in `SysRegOps`. `SysRegOps` invoke `PageManager` and `SysRegAux` to validate the written value and emulate the register update. As discussed in Section 4.2, KPCore also exposes a hypercall `opt_switch_mm` for batching register updates during process context switches. `HostTTBRSwitch` handles this hypercall, batching updates to the user and kernel page table base registers. All values are validated without compromising the functionality of context switches.

doi:10.6342/NTU202504620

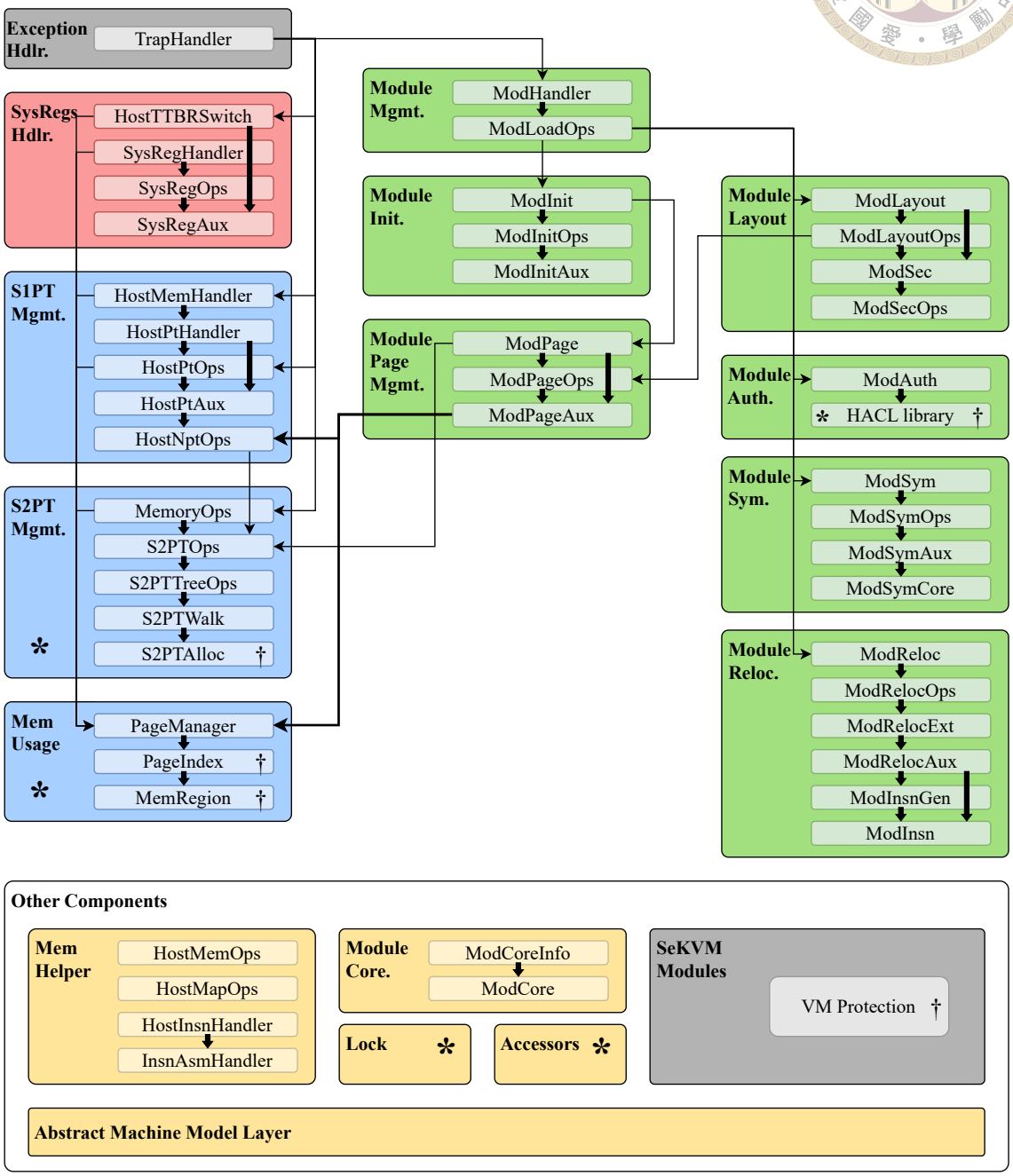


Figure 4.4: KPCore Layered Design. (1) Blue: Kernel Memory Protection. (2) Green: Secure Module Loading. (3) Red: System Register Protection. (4) Yellow: Common/ Helper/Machine Model Modules. (5) Grey and modules marked with ∗: SeKVM's existing modules [48] and the HACL cryptographic library [74]. † indicates the original implementations are fully reused without modification.

doi:10.6342/NTU202504620

Chapter 5 Verifying Functional Correctness of KPCore

In this chapter, we present the methodologies used to prove the functional correctness of KPCore. Figure 5.1 illustrates the main security components of KPCore, including the paging subsystem, the module loading subsystem, the system register handler, and existing SeKVM modules for VM protection. These components were built on top of the abstract machine model. At the top layer, the exception handler served as the entry point to KPCore, handling faults and hypercalls and delegating them to the corresponding security components. Among these, the SeKVM modules dedicated to VM protection were reused without modification and were treated as trusted. We therefore focused our verification efforts on the remaining modules that are dedicated to host protection and proved their functional correctness.

In our implementation, we extended SeKVM's stage 2 page table (S2PT) management beyond the original 4 KB and 2 MB mappings to support 1 GB pages, along with additional permission configurations to enforce host memory protection. However, the hardware-defined semantics of multi-level page tables, when exposed to higher layers, could further complicate the specifications and dramatically increase the complexity of the proof. To simplify the proof, we proved the correctness of our extended S2PT man-

doi:10.6342/NTU202504620

agement by adapting SeKVM's page table abstraction strategy. Specifically, we showed that the multi-level page table, represented as a tree structure in the low-level specification, refines multiple flat maps in the high-level specification.

The module loading subsystem and stage 1 page table management (S1PT) were built on top of the S2PT management, leveraging its verified high-level abstract interface. We were therefore able to rely on this simplified abstraction and focus our verification efforts on proving the correctness of their main routines. We did not introduce additional data abstractions for S1PTs; instead we relied on Spoq to directly generate their specifications that capture the intended behavior of their functions, since S1PT component already reside at a higher layer and are not extensively used by other components. For the system register handler, the restructuring strategies described in Section 4.3 were sufficient for Spoq to generate specifications and proofs establishing functional correctness automatically.

In the remainder of this chapter, we describe our machine model layer (Section 5.1), the verification of the paging subsystem (Section 5.2) and the module loading subsystem (Section 5.3), the system invariants ensured by correctness proofs (Section 5.4), and the limitations we encountered when using Spoq (Section 5.5).

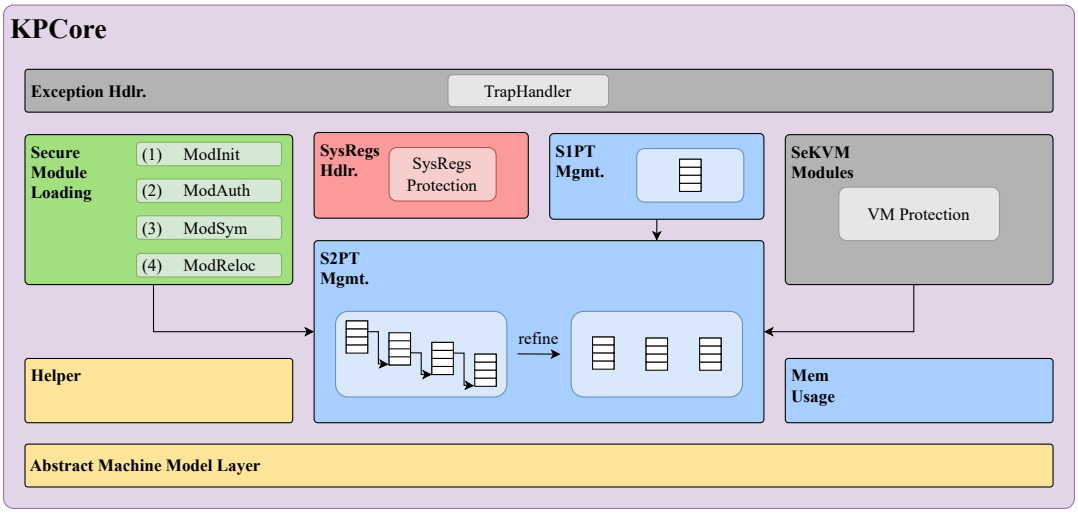


Figure 5.1: Security Components in KPCore

doi:10.6342/NTU202504620

5.1 Abstract Machine Model Layer

The abstract machine model layer served as the trusted layer upon which all other modules were built. Following the methodology of Spoq, we modeled the low-level primitives essential for KPCore, such as memory load/store, register operations, and memory pointer accesses, using the Coq libraries provided by Spoq. To support CPU-local concurrency reasoning within our model, we incorporated key architectural ideas introduced by Spoq. This involved manually defining concurrency-related structures such as an event log and an oracle. This event-based approach, which was similar to methods used in prior work [31], enabled us to reason about multi-core execution by analyzing individual CPU behaviors, thereby laying the groundwork for composing these proofs to eventually verify global invariants under a sequential consistency memory model.

5.2 Functional Correctness of Page Tables

We formally verified the functional correctness of vSECvma's page table management. vSECvma's page table management architecture encompassed both stage 1 and stage 2 mappings. The former, residing in a higher layer, was built upon and leveraged the verified layer functions exposed by the S2PT modules. To handle S1PT fault traps from the host, KPCore validated and updated the S1PTs according to the policies described in Section 2.3 and Section 4.1.2, updating the corresponding S2 mapping when necessary. The S2PT management included walking the S2PT, allocating S2PT pages as needed, and mapping entries in S2PT at the specified level with appropriate permission configurations.

doi:10.6342/NTU202504620

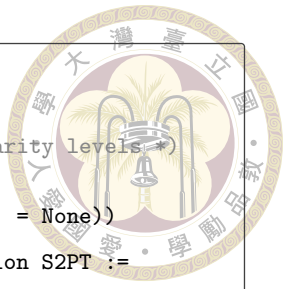
5.2.1 Stage 2 Page Table Management

As discussed earlier, Spoq evaluated SeKVM's implementation and proved the functional correctness of its S2PT management. Building on this, we reused the S2PT management proof from Spoq's artifact and extended it to support new functionalities introduced by SECvma. Similar to SeKVM, we modeled the S2PT as a four-level hierarchical tree (i.e., `pgd`, `pud`, `pmd`, and `pte`) within the machine model layer. SeKVM ensured correct S2PT management and supported mappings with page sizes of 4KB and 2MB. To incorporate the performance optimizations in SECvma from Section 4.2, we extended this support to include 1GB mappings in the S2PT.

Modeling the S2PT as a tree structure made its specifications inherently complex. This complexity, in turn, made proving functional correctness more challenging. Adding 1 GB mappings further increased this challenge, especially when generating comprehensive proofs across the different layers. To simplify the proof, we followed SeKVM's approach: **we proved that the multi-level page table, represented as a tree structure in the low-level specification, refines flat map representations in the high-level specifications**, as shown in Figure 5.1. We reused the original refinement proof strategy. However, extending it to support 1GB mappings still posed challenges, especially in ensuring several properties within the refinement.

Listing 2 illustrates the high-level specification for handling S2 mappings, where the S2PT is represented as multiple abstract flat maps. In our machine model, `npt` serves as a lock-synchronized object that models the host S2PT. We abstract three flat maps (i.e., `npt.lv1pt`, `npt.lv2pt`, and `npt.lv3pt`) representing 1GB, 2MB, and 4KB mappings, respectively. All S2PT queries and updates in the low-level machine model ultimately

doi:10.6342/NTU202504620



```
1  (* Allocate new page table pages from the npt.pool *)
2  Definition alloc_pt (npt : S2PT) : option S2PT := ...
3
4  (* Validity checks for overlapping mappings across different granularity levels *)
5  Definition is_valid_l1 (addr : Z) : Prop :=
6      forall (addr': Z), (addr[47:30] = addr'[47:30] ->
7          (npt.lv2pt # addr'[47:21] = None /\ npt.lv3pt # addr'[47:12] = None))
8
9  Definition s2pt_set (addr: Z) (level: Z) (pte: Z) (npt: S2PT) : option S2PT :=
10
11     let L1_index := addr[47:30] in
12     let L2_index := addr[47:21] in
13     let L3_index := addr[47:12] in
14
15     npt <- alloc_pt npt;
16     if level =? 1 then
17         rely (is_huge pte = true /\ is_valid_l1 addr);
18         Some (npt.lv1pt # L1_index <- pte) else
19     if level =? 2 then
20         rely (is_huge pte = true /\ is_valid_l2 addr);
21         Some (npt.lv1pt # L2_index <- pte)
22     else
23         rely (is_valid_l3 addr);
24         Some (npt.lv3pt # L3_index <- pte).
```

Listing 2: `s2pt_set` models the operation for setting an S2PT entry. The function takes the intermediate physical address (`addr`), the target mapping level (`level` for 1GB, 2MB, or 4KB), the page table entry (`pte`) to be set, and the abstract object `npt`, which represents the host S2PT.

refine simple load/store operations on these flat maps, thereby simplifying higher-level proofs that reason about the S2PT. Each map is indexed by the specific range of the intermediate physical address and loads/stores the corresponding page table entry from/to the corresponding abstract map.

s2pt_set. As shown in Listing 2, `s2pt_set` (line 9) models the operation for setting an S2PT entry. It first invokes `alloc_pt` (line 2, 15) to abstractly represent the allocation of necessary page table pages from `npt.pool`. Following this, `s2pt_set` proceeds to handle the S2 mapping based on the target level. The function also includes `rely` assertions (line 17, 20, 23) that guarantee no overlapping mappings across different granularity levels (`is_valid_lX`). If the assertions fail, the function returns `None` without performing updates. In SeKVM, it is ensured that if a 2MB mapping exists, then no 4KB mappings exist within that same huge block. Likewise, by extending the support to 1GB mappings,

doi:10.6342/NTU202504620

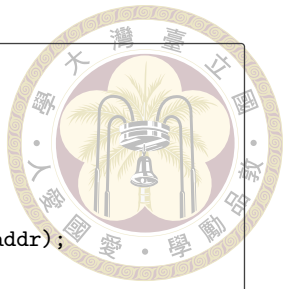
`is_valid_l1` enforced that neither 4KB nor 2MB mappings can exist within a 1GB-mapped region (line 5), and conversely, finer-grained mappings must not overlap with existing 1GB regions.

We constructed refinement relations to show that the multi-level page table refines the abstract flat maps. Several key properties are inherently validated by the refinement proof itself, ensuring that the structure is a well-formed tree and maintained in a hierarchical order. Within the refinement relations, the following properties are ensured: (1) each lower-level page table page is uniquely referenced by a single parent entry; (2) page tables allocated from the page pool are always free and empty; and (3) if a higher-level non-leaf entry exists, then there must have been, or currently exists, a corresponding next-level table entry or leaf entry. Property (1) rules out aliasing, i.e., where two distinct parent entries point to the same lower-level page table, which would cause unintended sharing of page table nodes. Property (2) guarantees that only a fresh page table can be inserted during allocation, meaning the page table has not already been referenced elsewhere. Properties (1) and (2) together ensure that the multi-level page table maintains a tree structure. Property (3) maintains hierarchical order: every non-leaf entry points only to the next lower-level page table or to a leaf entry, but not otherwise.

Listing 5.2 illustrates the refinement relations used to enforce these properties. To support 1GB mappings, we extended two additional refinement relations, as shown in Listing 5.2b and Listing 5.2c.

P1, P2. The `P1` states that when an `addr` is accessed in `npt.parent_pool`, the entry is either empty (`val = 0`) or allocated from the child pool. In the latter case, there exist a unique inverse mapping `inv_map` that maps the page back to exactly one parent `addr`. The `P2` states that unused pages in the child pool remain empty.

doi:10.6342/NTU202504620



```
P1:
    exists (inv_map: Z -> Z),
    forall addr,
        let val := npt.parent_pool[addr] in
        let base := npt.child_pool_base in
        let used := npt.child_pool_used in
        val = 0 \/
        (base <= val <= base + used * PAGE_SIZE) /\ (inv_map val = addr);
P2:
    forall addr,
        let base := npt.child_pool_base in
        let used := npt.child_pool_used in
        addr >= base + used * PAGE_SIZE -> npt.child_pool[addr] = 0;
```

(a) **P1** and **P2** relations.

```
P3: (* Omitted for brevity *)
P3_lv1:
    forall pud,
    is_tbl pud = true ->
    exists addr' pmd' pte',
        pgd_idx[addr] = pgd_idx[addr'] /\
        pud_idx[addr] = pud_idx[addr'] /\
        pmd' = npt.pmd # (pmd_idx[addr'] + pud) /\
            is_huge pmd' = true \/
                (is_tbl pmd' = true
                /\ pte' = npt.pte # (pte_idx[addr'] + pmd')
                /\ pte' <> 0);
```

(b) **P3** relation.

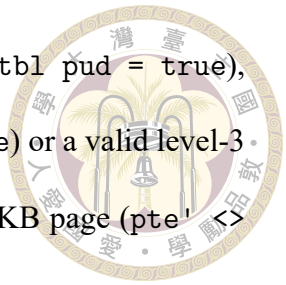
```
lvXpt_rel: (* Omitted for brevity *)
lv1pt_rel:
    let hnpt := (* S2PT in high spec *) in
    let lnpt := (* S2PT in low spec*) in
    forall addr pud,
        hnpt.lv1pt # addr[47:30] = Some pud <->
            let pgd := lnpt.pgd # (pgd_idx[addr] + lnpt.ttbr) in
            pgd <> 0 /\ pud = lnpt.pud # (pud_idx[addr] + pgd)
                /\ is_huge pud = true;
```

(c) **lvXpt_rel** relation.

Listing 5.2: Refinement relations used in the proof: (a) **P1** and **P2** enforces properties (1) and (2); (b) **P3** enforces property (3); and (c) **lvXpt_rel** connects entries in the high-level flat map to their corresponding multi-level page walks in the low-level specification. (b) and (c) indicate only the added relations for 1GB mappings, with other cases omitted for brevity.

P3, P3_lv1. In the original proof, the `P3` relation ensured that if a level-2 entry (`pmd`) was a table, it exclusively led to a valid `pte` leaf entry. With 1GB mapping support, however, a level-1 entry (`pud`) could reference either a huge page or another table. The `P3_lv1`

doi:10.6342/NTU202504620



relation captures this behavior: if a level-1 entry (pud) is a table (`is_tbl pud = true`), then it must lead to either a 2MB huge block (`is_huge pmd' = true`) or a valid level-3 page table (`is_tbl pmd' = true`) that contains at least one valid 4KB page (`pte' <> 0`).

lv1pt_rel. The `lv1pt_rel` relation connects the high-level flat map to the low-level specification. Specifically, if a 1GB mapping exists at a given address in `hnpt.lv1pt`, then a level-by-level traversal in the low-level specification from the pgd to the pud must yield a pud entry marked as a 1GB huge block (`is_huge pud = true`).

5.2.2 Stage 1 Page Table Management

S1PT management was built upon the established and formally verified S2PT management layers, allowing S1PT operations to directly invoke the simplified high-level specifications exposed by the S2PT management as needed. By formally proving the functional correctness of S1PT management, we transformed its implementation into high-level specifications in Coq.

However, modeling the S1PT within the machine model layer, using the same tree-structural approach as S2PT, was challenging. First, KPCore did not manage memory pools for S1PT. Instead, these pages were dynamically allocated by the untrusted kernel allocator. As a result, KPCore had no control or visibility over how memory was allocated or reused, making it infeasible to enforce precise structural constraints in the same manner as S2PT. Second, the host kernel could also dynamically map memory using various granularities (e.g., 1GB, 2MB, or 4KB) at runtime, for example, to optimize performance or maintain memory efficiency. Unlike a static structure, S1PTs grew dynamically without

doi:10.6342/NTU202504620

a fixed layout.

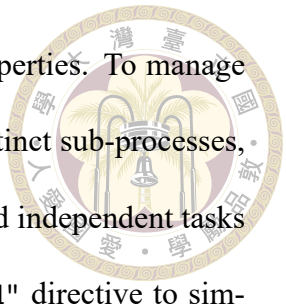
From a memory perspective, S1PT updates could be treated as reads or writes to arbitrary 8-byte-aligned physical memory addresses. Accordingly, we modeled S1PTs as a flat memory region (i.e., `Map.t Z` in Coq), where updates were interpreted as array modifications, similar to the high-level specifications of S2PTs, as illustrated in Figure 5.1. This simplified model allowed us to reason about the S1PTs as simple memory updates exposed to higher-level specifications. KPCore implements a lock for managing the write-protected S1PTs. This lock is required when handling permission fault traps from the host and during updates. Since the host cannot directly modify the protected page tables, we model S1PTs as lock-synchronized objects, with all updates performed under this lock to maintain consistency.

5.3 Functional Correctness of Secure Module Loading

SECvma adopted public-key cryptography to authenticate loaded kernel modules, ensuring that only authorized modules could be installed and executed. While SECvma's design migrated security-critical tasks from the kernel to KPCore, much of the inherent complexity of these operations was preserved within KPCore to maintain existing system properties and Linux features. This significantly increased the difficulty of formally proving the functional correctness of the module loading implementation.

We applied the design discussed in Section 4.3 to simplify the implementation and ultimately proved the functional correctness of secure module loading. By simplifications, we refer to modifications made to facilitate proof automation rather than changes to functionality. In the complex process of module loading, it was evident that it comprised many

doi:10.6342/NTU202504620



independent tasks with limited interdependencies regarding their properties. To manage this complexity, we decomposed the module loading process into distinct sub-processes, each fulfilling a specific role. We identified functions that represented independent tasks or were not common helper routines, then applied `"Hint NoUnfold"` directive to simplify Spoq's generated high-level specifications. This allowed higher-layer modules to call these functions directly instead of unfolding the exposed abstract high-level interface, as discussed earlier. The module loading routine was decomposed into three key stages: (1) **Module Initialization**, (2) **Module Authentication**, and (3) **Symbol Resolution and Relocation**.

Module Initialization. This stage consists of two main tasks: (1) *memory remapping* and (2) *parsing and preparing module loading information*. The kernel module is an ELF file that contains an ELF header and a section header table describing the module's layout and metadata, followed by the main contents such as code and data sections. Since the kernel module resides in a different address space from KPCore, it first remaps the entire module file into the EL2 address space. During this process, the entire module file mapped in the kernel address space is set to write-protected in the S2PT, preventing the kernel from modifying its contents once it is passed to KPCore for authentication. KPCore then defines and allocates a temporary structure for storing extracted module information. In our machine model, we modeled the extracted module information as a static, lock-synchronized structure `ModData`. The associated lock was already implemented in KPCore specifically for module loading, and it was distinct from other locks in the system. Unlike page tables, which grow dynamically and follow hardware-defined semantics, the module object was presented as a fixed structure. As a result, we could directly model its access behavior in the machine model using the Coq libraries provided by Spoq. KPCore

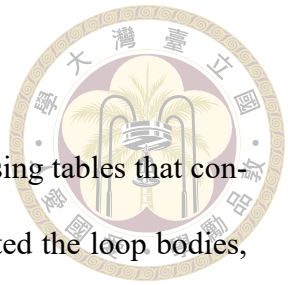
doi:10.6342/NTU202504620

maintained the actual base address of this structure and passed it as a pointer to successive functions. Similar to the approach in Figure 4.3, we encapsulated the pointer with getter and setter functions and modeled member access relative to the specific offset.

Module Authentication. SECvma's implementation builds upon SeKVM. We reuse the formally verified Ed25519 implementation from the HACL cryptographic library [74] which is already integrated in SeKVM to authenticate the signature of a loaded kernel module. Prior to authentication, KPCore stacks the module headers and contents into a pre-allocated buffer. In our machine model, we manually modeled two memory copy operations, `stack_buff` and `pop_buff`, which handle pushing data into and retrieving data from the buffer, respectively. To ensure memory safety, we verified that the source content resided within the remapped memory region and that the destination lay strictly within the bounds of the pre-allocated buffer. This guaranteed that the copy operation could not overwrite arbitrary memory.

Symbol Resolution and Relocation. This phase is the most complex and challenging part of the module loading process. It involves parsing multiple ELF sections and analyzing their contents in depth. The process begins by traversing the symbol table section and assigning resolved values to each symbol. For undefined symbols, KPCore searches the exported symbols provided by the kernel, such as the kernel symbol table and symbols exported from other loaded modules. After resolving symbols, KPCore iteratively processes relocation entries from the relocation sections. It updates the corresponding instructions or data in target sections by patching symbol references with the resolved values. Unlike other parts of the module loading process, symbol resolution and relocation involve multi-layered parsing, starting from high-level section structures down to the instruction level. KPCore decodes instructions, resolves unresolved symbols, and patches

doi:10.6342/NTU202504620



them with the computed values.

Both symbol resolution and relocation involve iteratively traversing tables that containing loops. To simplify the loop refinement proofs, we encapsulated the loop bodies, as discussed in Section 4.3. During the relocation process, the Linux implementation simply iteratively parses the target Arm instructions and performs updates, which include an excessive number of conditional branches. As mentioned previously, Spoq leverages a case analysis approach for conditionals that would generate two sub-proofs for each branch, making the proof prohibitively challenging. To address this challenge, we applied a similar modularization strategy, as illustrated in Listing 1. We transformed 41 conditional branches into only four unified functions within the relocation routine. Each of these functions constructs the appropriate arguments based on the relocation type before delegating the task to a subprocess that performs the actual relocation.

5.4 Ensured Invariants and Correctness Guarantees

In our work, we leveraged Spoq to formally prove the functional correctness of KPCore by demonstrating that the implementation refines the top-level specification. Beyond the Spoq-generated specifications, we additionally establish and prove essential system invariants by manually introducing constraints into the machine model and providing manually written abstract specifications in place of the automatically generated ones when necessary, as discussed in Section 4.3.

Abstract Machine Model. All verified components are built upon our machine model, which formalizes low-level memory accesses, pointer manipulations, and register operations. Spoq provides a suite of Coq libraries for modeling these low-level primitives.

doi:10.6342/NTU202504620

Any incorrect modeling—such as accessing an invalid offset within a memory structure—would either produce incorrect access behavior or yields a `None` result in the generated specification, causing the specification generation or the subsequent proofs to fail.

However, some low-level primitives cannot be statically modeled using the provided libraries, such as dynamic memory structures (e.g., page tables), raw memory pointers, and dynamically allocated buffers. For these primitives, we manually model their abstract behavior and insert `rely` assertions to specify preconditions and establish essential invariants, such as valid pointer ranges (Figure 4.3), before modeling the corresponding low-level semantics. These `rely` assertions encode local preconditions required for low-level functions in the machine model and are exposed to the higher-layer modules that invoke them. If any of these assertions are invalid, the corresponding execution path collapses to `None`, indicating that the subsequent function behavior is undefined and excluded from the model.

Building on this foundation, the machine model enforces several critical guarantees: (1) correct access to static global memory objects and registers is ensured through the Coq libraries provided by Spoq; (2) the type-aware pointer validation mechanism (Section 4.3) ensures that all local pointers passed from upper layers across function boundaries are valid; (3) bounds checks are enforced on both memory buffers and array accesses, such as dynamically allocated buffers used during module authentication and the section header table accessed during module file parsing; (4) address-space checks restrict memory accesses to their modeled regions, ensuring that operations occur within the correct address space—for example, KPCore accesses modules remapped into the EL2 reserved region, while relocation routines compute the corresponding EL1 addresses for modules executing at EL1; and (5) all shared objects—including S1PTs, S2PTs, and module structures—

doi:10.6342/NTU202504620

are accessed only when their corresponding locks are held.

Manual Abstract Specifications. In addition to the machine model, we manually write abstract high-level specifications to capture the intended behavior of functions. We then establish refinement relations and prove that the low-level implementation correctly refines these specifications. Through the correctness proof, several system invariants are also derived and formally verified. The manually written specifications encompass the memory usage and S2PT management components. The former defines an ownership-based memory access control scheme that assigns each host page a specific usage category (e.g., kernel code or data, user memory), ensuring that every page maintains a valid and consistent ownership state throughout execution. The latter models stage 2 page tables as well-formed hierarchical trees, preserving their refinement relations and enforcing structural consistency across updates, as discussed in Section 5.2.1.

5.5 Addressing Limitations of Spoq

Although Spoq provided substantial automation for verifying vSECvma, we encountered several limitations that required additional modeling effort or source-level refactoring. This section discusses three representative challenges, i.e., LLVM intrinsic functions, jump tables, and unsupported Arm instructions, and presents our solutions to each of these limitations.

Intrinsic Functions generated by LLVM. As is known, Spoq's automation began by parsing LLVM IR into a Coq AST. However, the LLVM toolchain automatically introduced intrinsic functions that were not explicitly defined in the source code, for purposes such as optimization, during IR generation. Since these intrinsics were treated as exter-

doi:10.6342/NTU202504620

nal functions to be resolved by the LLVM backend during code generation, Spoq's parser could not analyze their bodies in the LLVM IR and therefore could not automatically produce corresponding specifications. To address this, rather than manually writing specifications for each intrinsic, we refactored the source code to express the same behavior in a form that prevented LLVM from lowering it into an intrinsic. We ensured that the function's semantics were fully preserved in the generated IR and could be automatically captured by Spoq.

Jump Tables in Conditional Branches. Considering the helper function `get_arg0` in Listing 1, `cond` may be a sequential or closely related set of numeric values, and the function simply returns a corresponding value `arg0` based on `cond`. In such cases, LLVM optimized the conditional branches by automatically generating a static, global jump table indexed by the input. From a memory perspective, this jump table was accessed by loading the memory address of the target entry. To model this behavior in our machine model, we manually modeled jump-table accesses as a flat map, similar to how other memory accesses were represented.

Unsupported Arm Instructions. While Spoq supports most Arm instructions, including inline assembly, it has limitations for certain specialized instructions used by KPCore. In KPCore, when a permission fault occurs during S1PT updates, the Arm VE provides the faulting VA rather than the faulting IPA. The faulting IPA is essential for KPCore to emulate writes to page tables on behalf of Linux. To address this, KPCore used Arm's address translation instruction `at`, which triggered a hardware traversal of the S1PT and retrieved the faulting IPA from a given VA. Spoq cannot automatically reason about the `at` instruction due to its specialized hardware semantics. As mentioned earlier, since we did not model the S1PT as a tree structure, we did not perform an actual page walk to de-

doi:10.6342/NTU202504620

fine `at`'s behavior, which we left as future work. Instead, we assumed a flat map indexed by VA, with values corresponding to the IPA, analogous to the flat map representations used for the S2PT. This flat map was treated as a lock-synchronized object, ensuring that updates were performed atomically. In addition, KPCore also used the exclusive store instruction `stxr`, which is not directly supported by Spoq. Unlike a normal memory store, `stxr` conditionally writes to memory depending on the state of the exclusive monitor and returns a status flag indicating whether the store succeeded. In Spoq, we modeled this status flag with a `Parameter`, which could take values representing success or failure. If the flag indicated success, we then modeled the store operation as a normal memory store instruction.

doi:10.6342/NTU202504620

Chapter 6 Evaluation

Our implementation is integrated with the formally verified hypervisor SeKVM [48], based on its publicly available artifact for Linux 4.18 [66]. We proved the functional correctness of KPCore. By default, vSECvma's page table abstraction proof was established for four-level hierarchical host S2PTs. However, the Arm processors used in our experiments [9] employ a three-level host S2PT configuration. To support this, we applied the same verification methodology to extend the proof to the three-level configuration. The following presents the system's and KPCore's lines of code (LoC) and proof effort.

System's Lines of Code. Our work built on the prototype K-Int [45], which introduced about 4.2K LoC into SeKVM to preserve kernel code integrity. We subsequently developed SECvma [73], which refined K-Int's design with additional security mechanisms and optimizations. Despite these extensions, the codebase remained stable at around 4.3K LoC. Our final prototype, vSECvma, the formally verified model of SECvma, consists of approximately 5.4K LoC added and modified to the original SeKVM codebase. Compared to SECvma, about 1.1K LoC in vSECvma are dedicated to modeling data-structure operations and jump-table accesses, implemented mainly as simple getter and setter functions to support verification, in addition to restructuring some existing routines. In the kernel-space codebase, fewer than 100 LoC were modified relative to SECvma.

doi:10.6342/NTU202504620

Category	Security Components	LoC	Total
New Layer Module for Kernel Protection	Stage 1 Page Table Management	0.3K	**3.3K**
	System Registers Handler	0.1K	
	Secure Module Loading	1.2K	
	Helper (e.g., Mem., ModCore, Struct Access.)	1.5K	
	AbstractMachine	0.2K	
SeKVM's Layer Modules	Exception Handler	0.1K	**4.1K**
	Stage 2 Page Table Management	0.4K	
	Memory Usage Tracking	0.1K	
	SeKVM's Modules for VM protection	3.5K	
Grand Total			**7.4K**

Table 6.1: KPCore's Lines of Code

KPCore's Lines of Code. KPCore comprises a total of 7.4K LoC, as summarized in Table 6.1. Among these, 3.9K LoC are dedicated to enforcing kernel code integrity, while the remaining 3.5K LoC support VM protections. Within the 3.9K LoC, about 0.6K originate from existing SeKVM modules. Of these, roughly half remain unchanged and can be directly reused together with their original proofs. The rest are re-verified to cover extensions to exception handling for additional hypercalls and stage 2 page faults, and S2PT management for 1 GB mappings and host-memory permission control.

The remaining 3.3K LoC are newly introduced in KPCore modules. Specifically, 0.3K LoC implement page table and static code protection, 0.1K LoC provide system register protection, and 1.2K LoC support secure module loading. Another 1.7K LoC are distributed across helper modules and the abstract machine layer, implementing low-level operations such as memory access, module helpers, getter/setter methods for structured data access, and pointer encapsulation.

Proof Effort. We proved the functional correctness of 3.9K LoC in KPCore dedicated to kernel protection, including 3.3K from newly introduced layer modules and 0.6K from SeKVM's existing modules. Table 6.2 summarizes the lines of code in Coq automatically

doi:10.6342/NTU202504620

LoC in Coq	Autogen	Manual
Layer Configuration	-	0.7K
Machine Model	0.4K	1.4K
Low-level Specifications	12.3K	-
High-level Specifications	11.0K	0.1K
Identical Refinement Proofs	11.2K	1.3K
Lifting Refinement Proofs	32.2K	5.5K
Grand Total	**67.1K**	**9.0K**

Table 6.2: Specification and proof generation by Spoq, with manual proof effort.

generated by Spoq and the manual proof effort built on top of it. By leveraging Spoq and adopting proof strategies similar to those used in SeKVM, our verification required only about half a person-year. In total, Spoq generated approximately 67.1K LoC of specifications and proofs.

On the manual side, 0.7K LoC of layer configuration was provided to define KPCore's layered structure. Another 1.4K LoC describes the machine model, including load/store primitives, data structure definitions, and pointer encapsulation, largely building on Coq libraries already provided by Spoq. As discussed in Section 4.3, we also manually wrote 0.1K LoC of high-level specifications for data abstraction, including the stage 2 page table model and the memory usage tracking scheme.

Among the refinement proofs, 1.3K LoC for identical refinements and 0.6K LoC for lifting refinements required only minor effort, often just a few lines of tactics. Functions with more complex semantics or with loops still require manual proofs. Since our goal was to retrofit vSECvma into a verification-friendly model for Spoq, most proofs could be completed by applying tactic libraries and induction templates already provided by Spoq. The remaining 4.9K LoC of refinement proofs were devoted to manually written high-level data abstractions, with about 4.4K for the page table abstraction and the rest for the memory usage scheme. Compared to the manual proof effort required for these modules

doi:10.6342/NTU202504620

in the original SeKVM, our extensions introduced about 130 LoC of proofs for the host memory usage tracking scheme and 2.6K LoC of proofs for supporting 1 GB mappings, both built by extending and modifying SeKVM's original proofs evaluated in Spoq.

To support three-level S2PT configurations, the modifications to both implementation and specifications were minimal, involving only a few lines of change. We adapted the original four-level proof model for the three-level configuration, and most proofs could be directly reused. Overall, approximately 2.7K LoC of refinement proofs were verified for the three-level S2PT, of which 1.2K LoC were newly added or modified, most being constant-scale adjustments from the four-level model.

6.1 Performance Evaluation

We conducted our performance evaluation 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 on the m400 hardware. All configurations use the Linux 4.18 kernel and Ubuntu 20.04 with standard protection features such as KPTI enabled. The bare-metal configurations use all hardware available.

To measure the overhead of our protections and the effectiveness of our optimizations, we evaluated four primary system configurations: (1) K-Int, (2) an unoptimized version of SECvma (NOPT-SECvma), (3) the optimized SECvma, and (4) the verified model vSECvma. The NOPT-SECvma incorporates only the security enhancements from

doi:10.6342/NTU202504620

Name	Description
Kernbench	Compilation of the Linux 4.18 kernel using allnoconfig for Arm with GCC 9.3.0.
Hackbench	Hackbench [5] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	Netperf v2.6.0 [36] 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 [1] v2.3 clients on bare-metal, 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[40] v1.2.3 with its default parameters, running 8 threads in the bare-metal 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 [20] v0.17.0 client running workload A with 16 concurrent threads, readcount=10000 and operationcount=500000

Table 6.3: Description of benchmark application

Section 4.1, without the performance optimizations from Section 4.2. The applications benchmarks used the configuration listed in Table 6.3.

Figure 6.1 presents the performance of applications running on the Linux host under different configurations. All results are normalized against those of the same applications running on the mainline Linux v4.18 where lower values indicate better performance.

The results show that K-Int imposes a significant performance overhead, which is further increased by the additional security enhancements introduced in the unoptimized SECvma. We found that the overhead stems from the extra stage of memory translation used in memory protection and from the new register protection mechanism. The memory overhead arises from high TLB contention. On the m400 processors [9], which have a limited TLB capacity, both systems' reliance on 4KB page mappings in the host S2PT

doi:10.6342/NTU202504620

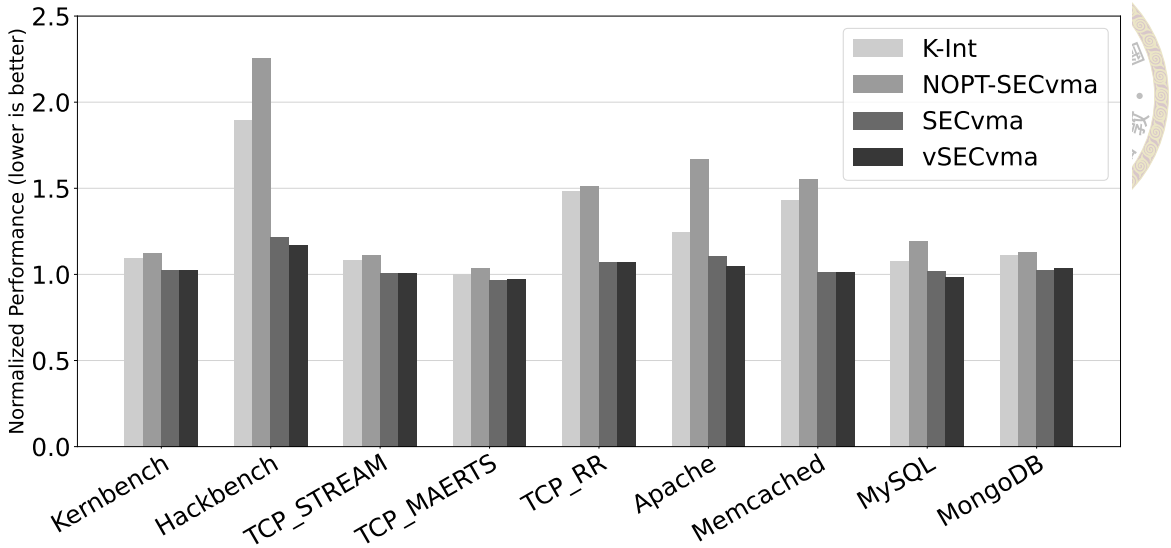


Figure 6.1: Application Performance of Linux Host

leads to frequent and costly TLB misses. The register protection overhead stems from 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.

The performance 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, in particular, forks numerous processes to perform IPC via Unix pipes. Compared to other workloads, 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 and Memcached also exhibited a much higher memory footprint than other applications. In these cases, SECvma's huge page optimization significantly reduced TLB pressure and improved performance. For Hackbench, this optimization reduced the overhead by more than 50% (K-Int vs SECvma).

Finally, the performance of vSECvma closely matches that of SECvma, preserving both functionality and performance while remaining formally verified. This is expected,

doi:10.6342/NTU202504620

as the restructuring required for verification does not alter the system's behavior. For instance, some loop bodies were encapsulated during refactoring, which could increase control flow complexity. However, since these changes occur primarily in module loading routines, they do not affect the performance-critical mechanisms identified earlier. In addition, although numerous getter and setter methods were introduced for verification purposes, they are unfolded and optimized away by the compiler, resulting in negligible additional cost.

insmod and rmmod Performance. We evaluated the performance of executing the `insmod` and `rmmod` commands on the mainline Linux, SECvma, and vSECvma. We tested three kernel modules: a cryptographic module `sha256_generic.ko`, a network device driver (`tun.ko`) and a file system driver (`btrfs.ko`). Table 6.4 shows the modules' respective sizes. The results show that both SECvma and vSECvma incurred higher latency than mainline Linux, with vSECvma introducing the most overhead. This additional cost stemmed primarily from the restructuring strategies used to enable verification, which increased the control flow complexity during module loading. In particular, loop bodies were encapsulated as function calls to simplify the induction proof. Since module loading involves nested loops that repeatedly query entries in the symbol and relocation tables and perform symbol resolution and relocation for each entry, this restructuring leads to an extensive number of additional function invocations. Besides, the modularized branching used in vSECvma—where arguments were constructed through several helper functions rather than being passed directly as constant flags—also contributed to the performance degradation.

Overall, `insmod` incurred measurable overhead in both SECvma and vSECvma, with the impact becoming more pronounced as the module size increased, whereas `rmmod` intro-

doi:10.6342/NTU202504620

		Linux 4.18		SECvma		vSECvma	
Name	**File Size**	`insmod`	`rmmod`	`insmod`	`rmmod`	`insmod`	`rmmod`
sha256_generic.ko	16.8 KB	$3ms$	$17ms$	$4ms$	$17ms$	$8ms$	$16ms$
tun.ko	66.3 KB	$4ms$	$15ms$	$5ms$	$16ms$	$39ms$	$18ms$
btrfs.ko	1.55 MB	$25ms$	$88ms$	$47ms$	$88ms$	$180ms$	$91ms$

Table 6.4: Module Performance

duced negligible overhead and maintained performance comparable to native Linux. The primary cause is that `insmod` involves more memory-intensive operations: it performs significantly more host mapping updates for page remapping and copies the module's contents for authentication. 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.

6.2 Bugs found in SECvma

During the verification process, we found several implementation bugs in our SECvma artifact [53], as summarized in Listing 3. These bugs were found by restructuring the code for proof automation, revealed by inspecting the generated specifications, or triggered by failures during specification generation, for example when operations evaluated to `None` due to undefined behavior. It is important to note that our current effort establishes only functional correctness, i.e., the implementation refines a top-level specification, so a buggy implementation will still refine a specification that reflects the same buggy behavior. This top-level specification can then be used to verify desired security invariants, which could identify some possible bugs in the underlying implementation. This is left as our future work.

We nevertheless found several bugs in SECvma during the current verification effort, as detailed below. While the last two bugs were discovered during code restructuring, they

doi:10.6342/NTU202504620

could also be detected through formal proofs of security properties or concurrency correctness.

Undefined behavior in PLT entry generation. During module relocation, SECvma generates PLT entries, which are small code sequences used for calling external symbols. These entries are emitted into the module code as needed. However, the relocation routine in SECvma was incorrectly implemented and did not generate the `mov1` instruction (A:line 3). As a result, the value of `mov1` remained undefined, which could potentially cause undefined behavior when the code was executed. In Spoq, this issue was exposed during the transformation from the Coq AST to the low-level specification generation. In the specification, `plt[i]` is modeled as a pointer `Ptr`, and we use an encapsulated setter function, `set_plt_entry(plt_i, insn)`, to handle the write to the PLT entry. In the Coq AST, the undefined value `mov1` had already been recognized and encoded as `VUndef` type. When generating the low-level specification, Spoq failed to translate this `VUndef` value into a valid construct, raising a fault when `mov1` was passed as an argument to the setter function.

Incorrect offset computation in branch relocation. During relocation of branch instructions, SECvma computes the relative offset between the caller PC and the target function address. If the target lies within the allowed `range`, it generates a branch-immediate instruction with that constant offset, which is checked by `branch_imm_common` (B: line 2). However, since SECvma operates in the EL2 address space, we found that a caller function incorrectly passed the PC as an EL2 address while the target `addr` was an EL1 address, resulting in an incorrect offset computation. In Spoq, this bug is exposed through the `rely` constraints, which required that neither the PC nor the target address reside in the EL2 address range. When this condition is violated, the `pc` evaluates to `None`, and the

doi:10.6342/NTU202504620

subsequent offset computation on it results in a fault during specification generation.

TOCTTOU vulnerability in page permission fault handling. When handling a permission fault on a S1PT update, SECvma acquires the page table lock and translates the faulting VA to obtain IPA, then releases the lock (C: line 2-4) and later re-acquires to apply the S1PT updates (C: line 6-8). This creates a window where the S1PT entry may change between translation and updates. We fix this by performing both translation and update within a single critical section protected by the page-table lock.

Privilege escalation via huge block mappings. SECvma grants privileged execution permission `PX` only when the mapped memory is classified as executable (D: line 8, 10). However, an attacker can exploit huge page mappings to escalate privileges. Specifically, a kernel attacker may construct a huge block mapping `huge_pte` (e.g., a 1GB mapping) that spans both executable and non-executable regions, then invoke `set_pud` to update the corresponding entry in the kernel page table. This write triggers a trap to KPCore. Since KPCore tracks memory usage at 4KB granularity using `get_page_subid()`, it observes that the base of `huge_pte` lies in an executable region and incorrectly grants execute permission to the entire block, thereby allowing unrelated memory to inherit execute rights. To address this, our policy restricts execute permissions to leaf mappings that fall entirely within authorized regions (i.e., static kernel code and loaded module code). Since KPCore already maintains a record of these regions, we added a range containment check before granting execute permission (D: line 9). For example, the static kernel code is mapped in 2 MB-aligned regions at boot, so KPCore permits only 4 KB or 2 MB executable mappings within this range. In other words, a huge block is granted execute permission only if all covered pages are executable. Otherwise, KPCore rejects the update, as such behavior is abnormal even under Linux.

doi:10.6342/NTU202504620

(A) Undefined behavior

```
mov0 = aarch64_insn_gen_movewide(...);
// mov1 = aarch64_insn_gen_movewide(...);
...

plt[i] = (struct plt_entry) {
    cpu_to_le32(mov0),
    cpu_to_le32(mov1),
    ...
};
```

(B) Incorrect offset computation

```
long branch_imm_common(u64 pc, u64 addr,
    long range)
{
    long offset = ((long)addr - (long)pc);

    if (offset < -range || offset >= range)
        return range;   // overflow
    else
        return offset;
}
```

(C) TOCTTOU vulnerability

```

acquire_host_pt_lock();
fault_ipa = trans_to_phys(va);
release_host_pt_lock();
...
acquire_host_pt_lock();
handle_host_update(fault_ipa, value);
release_host_pt_lock();

```

(D) Privilege escalation

```

Host:
    set_pud(pudp, huge_pte);
    // Trap to KPCore

KPCore:
    subid = get_page_subid(huge_pte);
    if (subid == KTEXT || subid == MOD_TEXT)
        // <= Mitigation here
        update_perm_px(huge_pte);
    ...
    handle_host_update(pudp, huge_pte);

```

Listing 3: Bugs found in SECvma.

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

Chapter 7 Related Work

Kernel Protection Framework. 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 [33, 35, 57, 67, 70] in VMs. Others [55, 71] rely on the hypervisor [11] to isolate an OS kernel from untrusted components. Microsoft's Hypervisor-Protected Code Integrity [8] framework relies on a full Windows hypervisor to protect the code integrity of the Windows OS kernel. In contrast, vSECvma builds on a much smaller hypervisor codebase, making it amenable to formal verification.

SecVisor [63] relies on a tiny hypervisor that leverages x86's virtualization extensions to protect Linux's code integrity but incurs high performance overhead to hosted applications (2.19x in the worst case). SecVisor permits runtime module loading, authenticating modules with SHA-1 functions and performing relocation. Unlike vSECvma, SecVisor does not authenticate all critical contents of a module (e.g., section headers or symbol resolution), leaving room for manipulation and incorrect relocation. Moreover, vSECvma hardens module loading by write-protecting module contents during installation, preventing TOCTTOU-based attacks where the kernel could tamper with a module after it has been authenticated.

Sprobes [28] and TZ-RKP [10] rely on privileged firmware that leverages Arm's

doi:10.6342/NTU202504620

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, vSECvma 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 [62] provides the Real-time Kernel Protection (RKP) [61] 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 [3] disclosed that it leverages Arm VE to support kernel memory protection. Similar to vSECvma, 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, vSECvma proposed to extend the existing commodity hypervisors for Arm to significantly reduce the implementation complexity for protecting Linux's code integrity. Further, compared to vSECvma, KNOX does not support module authentication during dynamic module loading to memory. RKP could be vulnerable to a TOCTTOU-based attack that bypasses module authentication, allowing an attacker to install malicious modules to executable kernel memory. We cannot com-

doi:10.6342/NTU202504620

prehensively compare vSECvma'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.

Verified Systems and Verification Frameworks. Previous work has built formally verified OS kernels that are proven functionally correct with respect to their specifications. The seL4 microkernel [38], the first formally verified OS kernel, required nearly 20 person-years of development and verification to achieve a comprehensive functional correctness proof. CertiKOS [30, 31] introduced an extensible architecture that adopts the CCAL methodology in Coq to build a certified concurrent kernel. Its primary artifact, the mC2 kernel, comprises 6.5K lines of C and x86 assembly and runs on stock x86 multicore machines. NrOS [12] and Atmosphere [17] explore the design of verified kernels written in Rust. Despite their high assurance, these systems are typically designed as microkernels. This design choice makes formal verification tractable by deliberately minimizing the TCB, but comes at the cost of lacking the rich functionality and broad driver support of monolithic kernels such as Linux.

Beyond verified kernels, several frameworks aim to make program verification more tractable for system developers. Verification-aware programming languages such as Dafny [43] and F* [65] integrate specifications directly into high-level programs and use SMT solvers to automatically discharge verification conditions, enabling proofs of functional correctness and security properties. Tools such as VCC [18] and VeriFast [34] build on separation logic, allowing modular reasoning about memory ownership and concurrency in C programs. While these approaches reduce manual effort compared to interactive theorem proving, they often require substantial annotation overhead and enlarge the trusted computing base by relying on complex, external SMT solvers.

doi:10.6342/NTU202504620

TPot [16] is a verification framework that adopts C-based ”proof-oriented tests” as specifications and leverages symbolic execution built on KLEE [13] to verify the correctness of critical system components. Rather than targeting full-system verification, TPot focuses on proving properties of individual critical components such as page tables and memory allocators. Its design aims to reduce the annotation burden on developers, allowing them to write specifications in C without needing to understand specialized verification languages. TPot is also designed to support integration with continuous integration (CI) workflows to promote practical deployment.

vSECvma consists of multiple security components that, in principle, could be verified using frameworks like TPot. TPot's key advantage is its automation, which abstracts low-level details through symbolic execution and allows developers to write specifications as C-based tests rather than as formal models in a proof assistant such as Coq [4]. However, constructing proof-oriented tests and identifying appropriate invariants still requires manual effort from the developer. Moreover, TPot does not scale to full-system or multi-threaded verification, and symbolic-execution-based approaches inherently suffer from path explosion, which limits their applicability to complex subsystems.

doi:10.6342/NTU202504620

Chapter 8 Limitations and Future Work

Currently, vSECvma supports dynamic installation of authorized kernel modules but does not allow arbitrary new code to be added into the kernel space. Therefore, vSECvma has to be extended to permit the execution of verified eBPF code [14]. vSECvma currently supports executing static eBPF code in interpreter mode. In the future, we plan to extend SECvma to support eBPF's JIT mode.

Our current prototype is based on Linux v4.18. In future work, we plan to upgrade vSECvma to a more recent Linux version, starting with Linux kernel v5.4.55, which has already been evaluated with SeKVM in Spoq's artifact [19]. Since the required modifications to vSECvma and SeKVM are self-contained and non-intrusive, we expect the porting and proof integration effort to be modest.

In this work, our verification effort focuses on certifying the correctness of host kernel protection features introduced by vSECvma, while SeKVM's original VM protection guarantees are provisionally assumed. As these features extend existing SeKVM modules, any upper-layer components that depend on them must also be re-verified to preserve end-to-end proof consistency. Our extensions are minimal and largely orthogonal to SeKVM's VM protection features. We therefore expect that the re-verification effort

doi:10.6342/NTU202504620

should be modest. In the future, we also plan to complete this process to deliver comprehensive correctness guarantees for both kernel and VM protection.

In addition, our current model abstracts S1PTs as flat memory, unlike S2PTs which are modeled as a tree structure reflecting Arm hardware semantics. We plan to refine our specifications to fully model S1PTs and construct corresponding refinement proofs. While Spoq effectively automates much of the code verification for KPCore, hardware-defined semantics such as page table management still require substantial manual effort. For example, extending SeKVM's proofs to support 1 GB mappings required an additional 2.6K lines of Coq proofs. Looking forward, we aim to explore new verification methodologies and tooling that can reduce the manual burden of reasoning about these complex page table semantics.

Our current work establishes only the first step toward proving the correctness of system implementations. In the future, we plan to leverage the generated specifications to prove higher-level security properties. For example, a non-interference approach can be used to reason about information flow security. VRM [66] provides methods to verify that programs satisfy a set of synchronization and memory-access conditions, ensuring that proofs established under sequential consistency (SC) model are preserved under relaxed memory models.

doi:10.6342/NTU202504620

Chapter 9 Conclusion

This thesis presented vSECvma, the first high-performance Linux kernel protection framework on Arm that has been formally verified for functional correctness. vSECvma extends K-Int, using a virtualization-based approach to protect kernel code integrity. It introduces new protection mechanisms and novel optimizations, strengthening K-Int's security guarantees while preserving Linux's performance and functionality. We formally proved the functional correctness of the system's core, KPCore, demonstrating that its implementation faithfully refines a high-level specification in Coq. Our verification strategy simplified the proof effort by reusing verified components from SeKVM and leveraging the Spoq framework to automate the verification of new modules. We restructured the KPCore's security components into a verification-friendly model that preserves functionality while enabling automation. Our effort culminated in the successful verification of two of the most complex and security-critical components: the paging and module loading subsystems, delivering a robust foundation for trustworthy and correct system software.

doi:10.6342/NTU202504620

doi:10.6342/NTU202504620

References

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

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

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

[4] The Coq Proof Assistant. http://coq.inria.fr. [Accessed on Sep 6, 2025].

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

[6] Windows 8 kernel memory protections bypass. https://labs.withsecure.com/publications/windows-8-kernel-memory-protections-bypass, 2014.

[7] 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.

doi:10.6342/NTU202504620

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

[9] 7-CPU.COM. Applied micro x-gene. https://www.7-cpu.com/cpu/X-Gene.html [Accessed: Apr 28, 2021].

[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, pages 90–102, 2014.

[11] 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), pages 164–177, Bolton Landing, NY, Oct. 2003.

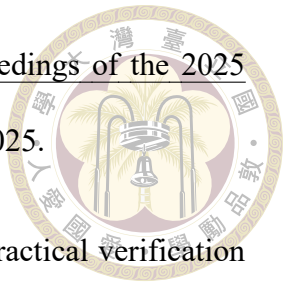
[12] M. Brun, R. Achermann, T. Chajed, J. Howell, G. Zellweger, and A. Lattuada. Beyond isolation: Os verification as a foundation for correct applications. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems, pages 158–165, 2023.

[13] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

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

[15] C. Castes, F. Costa, N. Foster, T. Bourgeat, and E. Bugnion. Lightweight hypervisor

doi:10.6342/NTU202504620



verification: Putting the hardware burger on a diet. In Proceedings of the 2025 Workshop on Hot Topics in Operating Systems, pages 27–33, 2025.

[16] C. Cebeci, Y. Zou, D. Zhou, G. Candea, and C. Pit-Claudel. Practical verification of system-software components written in standard c. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, pages 455–472, 2024.

[17] X. Chen, Z. Li, L. Mesicek, V. Narayanan, and A. Burtsev. Atmosphere: Towards practical verified kernels in rust. In Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification, pages 9–17, 2023.

[18] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In International Conference on Theorem Proving in Higher Order Logics, pages 23–42. Springer, 2009.

[19] Columbia University. OSDI 23: Artifact Evaluation: Spoq: Scaling Machine-Checkable Systems Verification in Coq. https://github.com/columbia/osdi23-paper114-ae, Sept. 2021.

[20] 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, pages 143–154, 2010.

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

[22] J. Corbet. Kernel support for hardware-based control-flow integrity. https://lwn.net/Articles/900099/, July 2022.

doi:10.6342/NTU202504620

[23] J. Corbet. Shadow stacks for 64-bit Arm systems. https://lwn.net/Articles/940403/, Aug. 2023.

[24] Z. Dai, S. Liu, V. Sjoberg, X. Li, Y. Chen, W. Wang, Y. Jia, S. N. Anderson, L. Elbeheiry, S. Sondhi, et al. Verifying rust implementation of page tables in a software enclave hypervisor. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 1218–1232, 2024.

[25] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.

[26] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 287–305, 2017.

[27] S. Fischer. Supervisor mode execution protection. In NSA Trusted Computing Conference, 2011.

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

[29] 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), pages 179–194, 2016.

[30] R. Gu, Z. Shao, H. Chen, J. Kim, J. Koenig, X. Wu, V. Sjöberg, and D. Costanzo.

doi:10.6342/NTU202504620

Building certified concurrent os kernels. Communications of the ACM, 62(10):89–99, 2019.

[31] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. {CertiKOS}: An extensible architecture for building certified concurrent {OS} kernels. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 653–669, 2016.

[32] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. ACM SIGPLAN Notices, 53(4):646–661, 2018.

[33] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. SIGARCH Comput. Archit. News, 39(1):279–290, mar 2011.

[34] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In NASA formal methods symposium, pages 41–55. Springer, 2011.

[35] 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), volume 10, 2007.

[36] R. Jones. Netperf. https://github.com/HewlettPackard/netperf, Accessed 2025.

[37] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against Return-to-User attacks. In 21st USENIX Security Symposium

doi:10.6342/NTU202504620

(USENIX Security 12), pages 459–474, Bellevue, WA, Aug. 2012. USENIX Association.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 207–220, 2009.

[39] I. Korkin. Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel. arXiv preprint arXiv:1805.11847, 2018.

[40] R. Labs. Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark, Accessed 2025.

[41] J. Lee, H. Ham, I. Kim, and J. Song. Poster: Page table manipulation attack. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 1644–1646, 2015.

[42] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with vcc. In International Symposium on Formal Methods, pages 806–809. Springer, 2009.

[43] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In International conference on logic for programming artificial intelligence and reasoning, pages 348–370. Springer, 2010.

[44] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert-a formally verified optimizing compiler. In ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress, 2016.

doi:10.6342/NTU202504620

[45] J. LI. K-Int, Kernel Code Integrity Enforcer. Master's thesis, National Taiwan University, June 2023.

[46] 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, HotOS '21, page 183 – 190, New York, NY, USA, 2021. Association for Computing Machinery.

[47] 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.

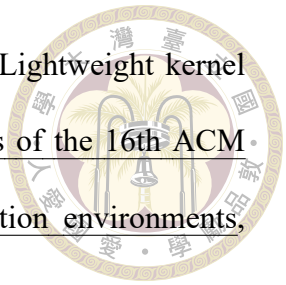
[48] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021), Aug. 2021.

[49] 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), pages 465–484, 2022.

[50] 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), pages 851–869, Boston, MA, July 2023. USENIX Association.

[51] 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.

doi:10.6342/NTU202504620



[52] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev. Lightweight kernel isolation with virtualization and vm functions. In Proceedings of the 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments, pages 157–171, 2020.

[53] National Taiwan University. ACSAC 24: Artifact Evaluation: SECvma: Virtualization-based Linux Kernel Protection for Arm. https://github.com/ae-acsac24-44/acsac24-paper240-ae, Dec. 2024.

[54] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-button verification of an os kernel. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 252–269, 2017.

[55] R. Nikolaev and G. Back. Virtuos: An operating system with kernel virtualization. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 116–132, 2013.

[56] T. Nipkow, M. Wenzel, and L. C. Paulson. Isabelle/HOL: a proof assistant for higher-order logic. Springer, 2002.

[57] 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, pages 103–115, 2007.

[58] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In International Workshop on Recent Advances in Intrusion Detection, pages 1–20. Springer, 2008.

[59] M. Salaün. Hypervisor-enforced kernel integrity (heki) for kvm. In Linux Plumbers Conference, 2023.

doi:10.6342/NTU202504620

[60] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 158–174, 2021.

[61] SAMSUNG. Real-time kernel protection (rkp), July 2023. https://docs.samsungknox.com/admin/fundamentals/whitepaper/core-platform-security/real-time-kernel-protection/.

[62] SAMSUNG. Samsung knox | secure mobile platforms and solutions, 2023. https://www.samsungknox.com/.

[63] 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, pages 335–350, 2007.

[64] G. Strongin. Trusted computing using amd "pacifica" and "presidio" secure virtual machine technology. Information Security Technical Report, 10(2):120–132, 2005.

[65] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, et al. Dependent types and multi-monadic effects in f. In Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 256–270, 2016.

[66] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu. Formal verification of a multi-processor hypervisor on arm relaxed memory hardware. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 866–881, New York, NY, USA, 2021. Association for Computing Machinery.

doi:10.6342/NTU202504620

[67] D. Tian, R. Ma, X. Jia, and C. Hu. A kernel rootkit detection approach based on virtualization and machine learning. IEEE Access, 7:91657–91666, 2019.

[68] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. Computer, 38(5):48–56, 2005.

[69] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In Proceedings of the Second Asia-Pacific Workshop on Systems, pages 1–5, 2011.

[70] 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, 16(4):2650–2667, 2019.

[71] X. Xiong, D. Tian, P. Liu, et al. Practical protection of kernel integrity for commodity os from untrusted extensions. In NDSS, volume 11, 2011.

[72] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive os kernels. In International Conference on Computer Aided Verification, pages 59–79. Springer, 2016.

[73] T. B. Yen, J. Li, and S.-W. Li. Secvma: Virtualization-based linux kernel protection for arm. In 2024 Annual Computer Security Applications Conference (ACSAC), pages 579–592, 2024.

[74] 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, pages 1789–1806, 2017.

doi:10.6342/NTU202504620