Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and
John Zhuang Hui, *Columbia University*

https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor

Shih-Wei Li   Xupeng Li   Ronghui Gu   Jason Nieh   John Zhuang Hui
Department of Computer Science
Columbia University
{shihwei,xupeng.li,rgu,nieh,j-hui}@cs.columbia.edu

Abstract

Hypervisors are widely deployed by cloud computing providers to support virtual machines, but their growing complexity poses a security risk, as large codebases contain many vulnerabilities. We present SeKVM, a layered Linux KVM hypervisor architecture that has been formally verified on multiprocessor hardware. Using layers, we isolate KVM's trusted computing base into a small core such that only the core needs to be verified to ensure KVM's security guarantees. Using layers, we model hardware features at different levels of abstraction tailored to each layer of software. Lower hypervisor layers that configure and control hardware are verified using a novel machine model that includes multiprocessor memory management hardware such as multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with cache bypass support. Higher hypervisor layers that build on the lower layers are then verified using a more abstract and simplified model, taking advantage of layer encapsulation to reduce proof burden. Furthermore, layers provide modularity to reduce verification effort across multiple implementation versions. We have retrofitted and verified multiple versions of KVM on Arm multiprocessor hardware, proving the correctness of the implementations and that they contain no vulnerabilities that can affect KVM's security guarantees. Our work is the first machine-checked proof for a commodity hypervisor using multiprocessor memory management hardware. SeKVM requires only modest KVM modifications and incurs only modest performance overhead versus unmodified KVM on real application workloads.

1 Introduction

Cloud computing providers rely on commodity hypervisors to securely host and protect user applications and data in virtual machines (VMs). However, commodity hypervisors are complex pieces of software, in some cases integrated with an entire host operating system (OS) kernel to leverage existing kernel functionality. This complexity poses a significant security risk as more complex software has more bugs, allowing attackers to exploit hypervisor vulnerabilities to compromise VMs [14–18].

Theoretically, formal verification offers a solution by proving that a system is correctly implemented. However, previously verified systems, such as CertiKOS [33], seL4 [43,53], Komodo [28], and Serval [54], were not verified using realistic hardware models that resemble what can be found in a cloud computing setting. Most of them are limited to uniprocessor settings, and none of them model common hardware features such as multi-level shared page tables, tagged TLBs, or writeback caches. In other words, these verified implementations cannot be deployed to handle cloud applications and workloads, and even if they could, their proofs may not hold for hardware used in a cloud computing setting.

We present SeKVM, the first hypervisor that has been formally verified on multiprocessor hardware with shared page tables, tagged TLBs, and writeback caches. This is made possible by introducing a layered hypervisor architecture and verification methodology. We use layers in three ways. First, we use layers to reduce the trusted computing base (TCB) by splitting the hypervisor into two layers, a higher layer consisting of a large set of untrusted hypervisor services and a lower layer consisting of a small core that serves as the hypervisor's TCB. We build on our previous work on HypSec [46] to retrofit the Linux KVM hypervisor in this manner without compromising its functionality. Reducing the hypervisor's TCB reduces the amount of code that needs to be trusted, thereby reducing code complexity and vulnerabilities.

Second, we use layers to modularize the implementation and proof of the TCB. We structure the TCB's implementation as a hierarchy of modules that build upon the hardware and each other. Modularity enables us to decompose the verification of the TCB into simpler components that are easier to prove. Once we prove that a lower layer module of the implementation refines its specification, we can then hide its implementation details and rely on its abstract specification in proving the correctness of higher layer modules that rely on the lower layer module. Furthermore, we can prove the correctness of the lower layer module once and then rely on it in proving higher layer modules instead of needing to verify its implementation each time it is used by a higher layer module. We leverage our previous work on security-preserving layers [48] to provide a deep specification of each layer of the hypervisor implementation, and verify that the implementation refines a stack of layered specifications. Using layers allows us to reduce the proof of a complex implementation by composing a set of simpler proofs, one for each implementation module, reducing proof effort overall. As software is updated, layers also help with proof maintainability, as only the proofs for the implementation modules that change need to be updated while the other proofs can remain the same.

Third, we use layers to modularize the model of the hardware used for verification. We introduce a layered hardware model that is accurate enough to model multiprocessor hardware features yet simple enough to be used to verify real software by tailoring the complexity of the hardware model to the software using it. Lower layers of the hypervisor tend to provide simpler, hardware-dependent functions that configure and control hardware features. We verify these layers using all the various hardware features provided by the machine model, allowing us to verify low-level operations such as TLB shootdown. Higher layers of the hypervisor tend to provide complex, higher-level functions that are less hardware dependent. We verify these layers using simpler, more abstract machine models that hide lower-level hardware details not used by the software at higher layers, reducing proof burden for the more complex parts of the software. We extend our layered verification approach to construct an appropriately abstract machine model for each respective layer of software. This allows us to verify the correctness of the multiprocessor hypervisor TCB while accounting for and taking advantage of widely-used multiprocessor features, including multi-level shared page tables, tagged TLBs, and multi-level caches with cache bypass support.

We have implemented and verified a SeKVM prototype by retrofitting KVM on Armv8 multiprocessor hardware [19, 23–25]. The implementation requires only modest modifications to Linux and has a TCB of only a few thousand lines of code, yet retains KVM's full-featured commodity hypervisor functionality, including multiprocessor, full device I/O, multi-VM, VM management, and broad Arm hardware support. SeKVM improves KVM security by verifying the correctness of its TCB and the security guarantees of the entire hypervisor. Our verification also accounts for multi-level shared page tables, tagged TLBs, and multi-level caches. Furthermore, the verification has been done for multiple versions of KVM, specifically those in versions v4.18 and v5.4 of the Linux kernel. Both the machine model and the proofs that build upon it were formalized using the Coq proof assistant [3]. We show that SeKVM provides its strong security while providing similar performance to unmodified KVM, with only modest overhead for real application workloads and similar scalability when running multiple VMs.

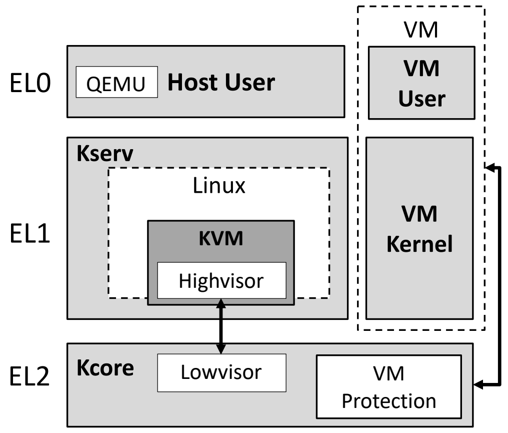


Figure 1: SeKVM Design

Although SeKVM shares the same security properties as HypSec, both the correctness of SeKVM's TCB implementation and its security guarantees are formally verified. While HypSec's TCB may contain vulnerabilities that compromise its security properties, we have proven that SeKVM's TCB contains no vulnerabilities. Furthermore, while HypSec is designed to provide security properties to ensure VM confidentiality and integrity, SeKVM has been proven to guarantee those security properties on multiprocessor hardware. Our work is the first, machine-checked correctness proof of the TCB of a commodity hypervisor on a realistic hardware model with shared page tables, tagged TLBs, and writeback caches, and the first, machine-checked security proof of a commodity hypervisor using multiprocessor memory management hardware.

2 Threat Model and Assumptions

Our threat model is primarily concerned with hypervisor vulnerabilities that may be exploited to compromise a VM's private data. For each VM we are trying to protect, an attacker may control other VMs and exploit any hypervisor vulnerabilities. We protect each VM from attacks by other compromised VMs, but do not protect VMs that voluntarily reveal their own private data. Attackers may control peripherals to perform malicious memory accesses via DMA [61]. Side-channel attacks [6, 38, 51, 55, 71, 72] are beyond the scope of the paper.

We assume a secure persistent storage to store keys. We assume the hardware is bug-free and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We trust the machine model, compiler, and Coq.

3 SeKVM Design

SeKVM uses HypSec's design to retrofit the Linux KVM hypervisor, reducing its TCB while protecting the confidentiality and integrity of VMs. As shown in Figure 1, we split KVM into two layers, a small trusted and privileged KCore that is the TCB with full access to VM data, and an untrusted and deprivileged KServ delegated with most hypervisor functionality including the Linux kernel integrated with KVM. The result is a hypervisor with a significantly smaller TCB that still supports KVM's rich hypervisor features.

KCore is kept small by only performing VM data access control, including saving and restoring CPU register state and page table management to limit access to a VM's CPU state and memory to only KCore and the VM itself. Other hypervisor functionality, including I/O and interrupt virtualization and resource management such as CPU scheduling and memory allocation, are delegated to KServ. SeKVM leverages hardware virtualization support to enforce this separation. SeKVM runs KCore at a higher privilege CPU mode designed for running hypervisors, giving it full control of hardware, including virtualization hardware mechanisms such as nested page tables (NPTs) [8]; KServ runs at a lower privilege mode.

KCore configures virtualization hardware to enforce its access control. KCore enables NPTs for KServ and VMs so that they do not have direct access to physical memory. KCore can limit KServ's or a VM's access to pages of physical memory by unmapping those pages from the respective NPT. KCore ensures its own memory is not mapped into any of the NPTs, protecting its memory by making it inaccessible to KServ and other VMs. KCore also uses NPTs to make each VM's memory inaccessible to KServ and other VMs.

KCore interposes on all VM transitions, namely exiting or entering a VM. When a VM exits, KCore saves the VM's execution context from CPU hardware registers to its private memory, then restores KServ's execution context to the hardware before switching to KServ. KServ therefore cannot access a VM's CPU state from the hardware or memory, which the state is saved in KCore memory inaccessible to KServ. Since KServ must run to switch a CPU from running one VM to another, a VM's CPU state is also not accessible by any other VM. A compromised KServ or VM can neither control hardware virtualization mechanisms nor access KCore memory and thus cannot disable SeKVM.

Specifically, SeKVM uses Arm Virtualization Extensions (VE) to run KCore in hypervisor (EL2) mode while KServ runs in a less privileged kernel (EL1) mode. VM operations that need hypervisor intervention trap to EL2 and run KCore. KCore either handles the trap directly to protect VM data or world switches the hardware to EL1 to run KServ if more complex handling is necessary, KCore context switches to KServ. When KServ finishes its work, it makes a hypercall to trap to EL2 so KCore can securely restore the VM state to hardware. KCore interposes on every switch between the VM and KServ, thus protecting the VM's execution context. SeKVM ensures that KServ cannot invoke arbitrary KCore functions via hypercalls.

KCore leverages Arm VE's stage 2 memory translation support, Arm's NPTs, to virtualize both KServ and VM memory. Stage 2 page tables translate from guest physical addresses (gPAs) in a VM to the actual physical memory addresses on the host (PAs). Free physical memory is mapped into KServ's stage 2 page tables so KServ can allocate it to VMs. Once it is allocated to a VM, KCore maps the memory into the VM's stage 2 page tables and unmaps the memory from KServ's stage 2 page tables to make the physical memory inaccessible to KServ. KCore routes stage 2 page faults to EL2 and rejects illegal KServ and VM memory accesses. KCore allocates KServ's and VMs' stage 2 page tables from its own protected physical memory and manages the page tables, preventing KServ from accessing them. When a VM is terminated and is done with its allocated memory, KCore scrubs the memory before mapping it back into KServ's stage 2 page tables as free memory which can be allocated again to another VM. Further details are described in [46].

SeKVM by default ensures that KServ has no access to any VM memory. However, a VM may want to share its memory with KServ in some cases. For example, a VM may encrypt its data for use with paravirtualized I/O, in which a memory region owned by the VM has to be shared with KServ for communication and efficient data copying since KServ handles paravirtualized I/O. SeKVM provides `GRANT_MEM` and `REVOKE_MEM` hypercalls which a guest OS can use to share its memory with KServ. The VM passes the start of a guest physical frame number, the size of the memory region, and the specified access permission to KCore via the hypercalls. KCore enforces the access control policy by controlling the memory region's mapping in stage 2 page tables. Only VMs can use these two hypercalls; KServ cannot use them to gain access to VM pages.

SeKVM delegates device management to KServ. Devices are untrusted and KCore ensures that devices cannot compromise VM data using DMA protection. Like HypSec, SeKVM assumes VMs do not voluntarily leak data, and assumes that they encrypt I/O data for end-to-end security.

Using hardware features. Like KVM, SeKVM leverages standard multiprocessor hardware features for its functionality and performance, including multi-level shared page tables, tagged TLBs, caches, and IOMMU hardware. KCore supports multi-level shared NPTs to support standard KVM functionality. KCore supports dynamically allocated 4-level NPTs as used in KVM, which is essential on Arm 64-bit hardware. KCore supports huge (2MB) and regular (4KB) pages, also standard in KVM, which is crucial for virtualization performance. KCore supports shared NPTs that can be concurrently accessed by multiple CPUs as this is a requirement for multiprocessor VMs, each of which has a shared NPT.

KCore uses Arm's tagged TLBs to improve paging performance, avoiding the need to flush TLBs on context switches between VMs and KServ. KCore assigns an identifier to each VM and KServ which it uses to tag TLB entries so address translation can be properly disambiguated on TLB accesses from multiple VMs and KServ. When updating a page table entry, KCore flushes corresponding TLB entries to ensure the TLB does not include stale page table entries that could potentially compromise VM security. For instance, when a VM page is evicted from its stage 2 page tables, KCore has to flush the TLB entries correlated to the translation used for the evicted page. Otherwise, a VM could use the cached TLB entry to access the evicted page that KServ may now allocate to the other VMs. Correct TLB maintenance while avoiding unnecessary TLB flushes is crucial for VM security and performance.

KCore takes advantage of Arm's hardware cache coherence architecture to maximize system performance, but needs to ensure that caching does not violate the confidentiality and integrity of VM data. Architectures like Arm allow software to manage cached data. In particular, Arm's hardware cache coherence ensures that all cached memory accesses across different CPUs and different level caches get the same synchronized value, but it does not guarantee that what is in the cache is the same as main memory. Memory accesses that are configured to bypass the cache may therefore obtain stale data if the latest value is cached. To ensure this does not result in any possible leakage of VM data, when KCore scrubs memory pages, it executes cache management instructions to force those writes to cached data to also be written back to main memory to ensure there is no way for any VMs or KServ to access VM data directly from main memory.

KCore leverages the System Memory Management Unit (SMMU) [4], Arm's IOMMU, to ensure that a VM's private memory cannot be accessed by devices assigned to KServ or other VMs, including protecting against DMA attacks. KCore ensures the SMMU is unmapped from all NPTs so it can fully control the hardware to ensure devices can only access memory through the SMMU page tables it manages. It uses the SMMU page tables to enforce memory isolation. KCore validates all SMMU operations by only allowing the driver in KServ to program the SMMU through Memory Mapped IO (MMIO) accesses, which trap to KCore, and SMMU hypercalls. MMIO accesses are trapped by unmapping the SMMU from KServ's stage 2 page tables. SMMU hypercalls (1) allocate/deallocate an SMMU translation unit, and its associated page tables, for a device, and (2) map/unmap/walk the SMMU page tables for a given device. As part of validating a KServ page allocation proposal for a VM, KCore also ensures that the page being allocated is not mapped by any SMMU page table for any device assigned to KServ or other VMs.

Layered implementation. While SeKVM's design significantly reduces the size of the its TCB and therefore also reduces the proof effort to verify the TCB, proving the correctness of the smaller hypervisor TCB, KCore, still remains a challenge, especially on Arm multiprocessor hardware. To further reduce the proof burden, KCore itself uses a layered architecture to facilitate a layered approach to verification. The implementation is constructed as a set of layers such that functions defined in higher layers of the implementation can only call functions at lower layers of the implementation. Layers can then be verified in an incremental and modular way. Once we verify the lower layers of the implementation, we can compose them together to simplify the verification of higher layers.

The specific layers in KCore's implementation are not determined in a vacuum, but with verification in mind based on the following layer design principles. First, we introduce layers to simplify abstractions, when functionality needed by lower layers is not needed by higher layers. Second, we introduce layers to hide complexity, when low-level details are not needed by higher layers. Third, we introduce layers to consolidate functionality, so that such functionality only needs to be verified once against its specification. For instance, by treating a module used by other modules as its own separate layer, we do not have to redo the proof of that module for all of the other modules, simplifying verification. Finally, we introduce layers to enforce invariants, which are used to prove high-level properties. Introducing layers modularizes verification, reducing proof effort and maintenance.

Figure 2 shows the KCore layered architecture. The top layer is `TrapHandler`, which defines KCore's interface to KServ and VMs, such as KServ hypercalls and VM exit handlers. Exceptions caused by KServ and VMs cause a context switch to KCore, calling `CtxtSwitch` to save CPU register state to memory, then `TrapDispatcher` or `FaultHandler` to handle the respective exception. On a KServ hypercall, `TrapDispatcher` calls `VCPUOps` to handle the `VM_ENTER` hypercall to execute a VM, and `MemHandler`, `BootOps` and `SmmuOps` to use their respective hypercall handlers. On a VM exit, `TrapDispatcher` calls functions at lower layers if the exception can be handled directly by KCore, otherwise `CtxtSwitch` is called again, protecting VM CPU data and switching to KServ to handle the exception. On other KServ exceptions, `FaultHandler` calls `MemOps` to handle KServ stage 2 page faults and `SmmuOps` to handle any KServ accesses to SMMU hardware. `FaultHandler` also calls `MemOps` to handle VM `GRANT_MEM` and `REVOKE_MEM` hypercalls. KCore implements basic page table operations in the layers in `MMU PT`, including page table walk, map or clear a `pfn` in page table, and page table allocation. KCore implements ownership tracking for each page in `PageMgmt`, `PageIndex`, and `Memblock` for memory access control. `MemOps` and `MemAux` provide memory protection APIs to other layers. KCore provides SMMU page table operations in layers in `SMMT PT`. KCore provides VM boot protection in `BootOps`, `BootAux`, and `BootCore`. `BootOps` calls the Ed25519 libary from the

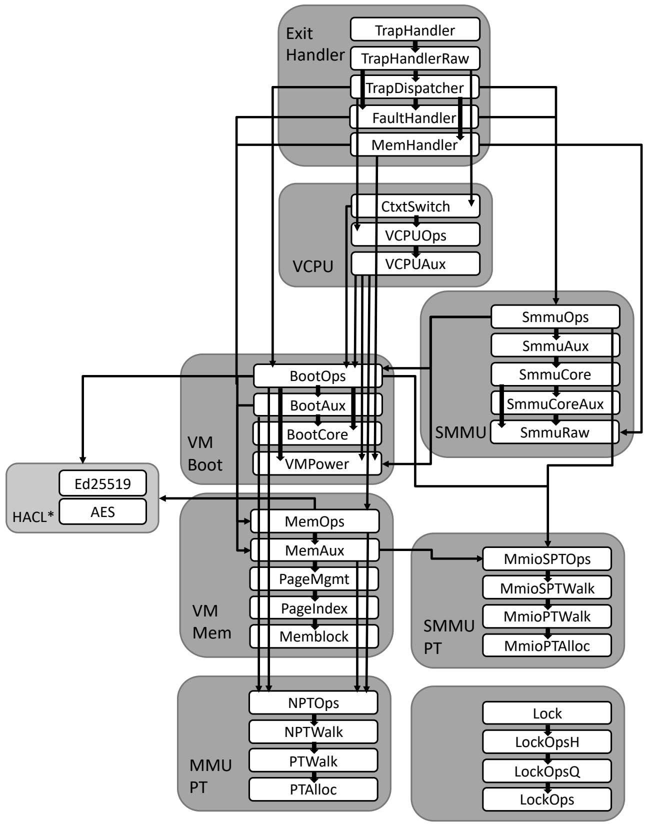


Figure 2: KCore Layered Implementation

verified Hacl* [74] library to authenticate signed VM boot images. `BootOps` and `MemOps` call to the AES implementation in Hacl* to encrypt or decrypt VM data to support VM management. Finally, four layers implement locks.

4 SeKVM Verification

We combine the layered implementation of SeKVM's TCB, KCore, with a layered hardware model to verify its correctness using Coq. We start with a bottom machine model that supports real multiprocessor hardware features such as multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with bypass support. We use layers to gradually refine the detailed low-level machine model to a higher-level and simpler abstract model. Finally, we verify each layer of software by matching it with the simplest level of machine model abstraction, reducing proof burden to make it possible for the first time to verify commodity software using these hardware features.

Each abstraction layer [31, 34] consists of three components: the *underlay interface*, the layer's *implementation*, and its *overlay interface*. Each interface exposes abstract *primitives*, encapsulating the implementation of lower-level routines, so that each layer's implementation may invoke the primitives of the underlay interface as part of its execution.

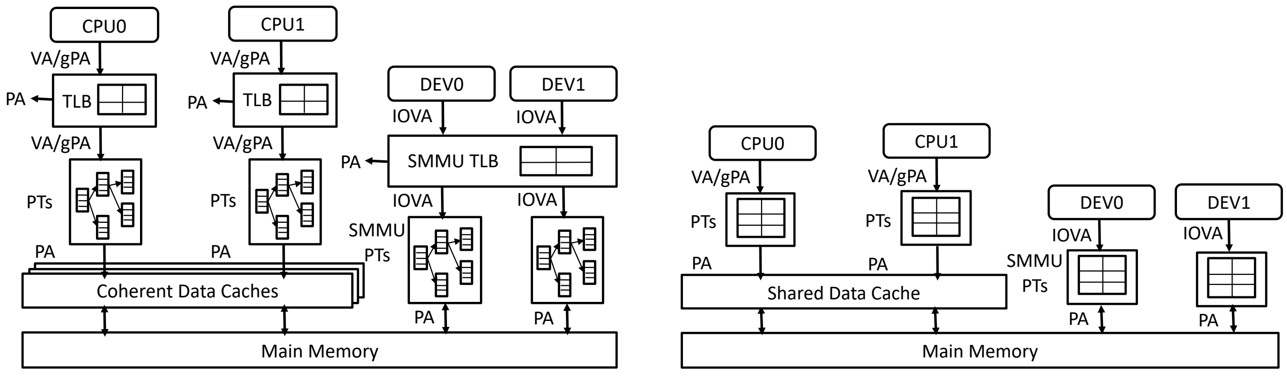
For each layer I of KCore's implementation, we prove that I running on top of the underlay interface L refines its (overlay) specification S, $I@L \sqsubseteq S$. Because the layer refinement relation $\sqsubseteq$ is transitive, we can incrementally refine KCore's entire implementation as a stack of layer specifications. For example, given a system comprising of layer implementations I_3, I_2, and I_1, their respective layer specifications S_3, S_2, and S_1, and a base machine model specified by S_0, we prove $I_1@S_0 \sqsubseteq S_1$, $I_2@S_1 \sqsubseteq S_2$, and $I_3@S_2 \sqsubseteq S_3$. We compose these layers to obtain $(I_3 \oplus I_2 \oplus I_1)@S_0 \sqsubseteq S_3$, proving that the behavior of the system's linked modules together refine the top-level specification S_3.

All KCore interface specifications and refinement proofs are manually written in Coq, with 34 interface specifications matching the layers in Figure 2. We use CompCert [45] to parse each layer of the C implementation into Clight representation, an abstract syntax tree defined in Coq; the same is done manually for assembly code. We then use that Coq representation to prove that the layer implementation refines its respective interface specification at the C and assembly level. Note that the C functions that we verify may invoke primitives implemented in assembly and introduced in the bottom machine model. We enforce that these assembly primitives do not violate C calling conventions and parameters are correctly passed. For example, we verify the correctness of TLB maintenance code, which is implemented in C, but invokes primitives implemented in assembly.

We prove, layer by layer, that the KCore implementation using a detailed machine model refines its top-level specification using a simpler abstract model. We then use the top-level specification to prove that KCore guarantees VM confidentiality and integrity for any KServ implementation, thereby proving security guarantees for the entire SeKVM hypervisor.

4.1 AbsMachine: Abstract Hardware Model

Each of KCore's layer modules successively builds upon AbsMachine, our bottom machine model. This abstract multiprocessor hardware model constitutes the foundation of our correctness proof. As shown in Figure 3a, AbsMachine includes multiple CPUs and a shared main memory. AbsMachine models general purpose and systems registers for each CPU. It also models Arm hardware features relevant to modern hypervisor implementation, including stage 1 and stage 2 page tables, a physically indexed, physically tagged (PIPT) shared data cache, and SMMU page tables, and TLBs. The shared data cache is semantically equivalent to Arm's multi-level cache hierarchy with coherent caches. KCore uses stage 2 page tables to translate guest physical addresses to actual physical addresses on the host, and uses its own EL2 stage 1 page table to translate its virtual addresses to physical addresses. AbsMachine models the particular hardware configuration of KCore which we verify. For example, although Arm supports 1GB, 2MB, and 4KB mappings in stage 2 page tables, KCore



(a) The bottom machine model: `AbsMachine`

(b) The machine model after the layer refinement

Figure 3: Refinement of machine models. (a) The bottom machine model that includes TLBs, multi-level page tables, and PIPT writeback caches. (b) The layered refinement of machine models abstracts away TLBs, consolidates multi-level page tables into a single-level flat page map, and enforces the well-formedness of data caches.

only uses 4KB and 2MB mappings in stage 2 page tables, since 1GB mappings result in fragmentation. Thus, we model a VM's memory load and store accesses in AbsMachine over stage 1 and stage 2 page tables using 4KB and 2MB mappings.

Our abstract machine is formalized as a transition system, where each state transition is the result of some atomic computational step by a single CPU, such as executing a single machine instruction or invoking a primitive; concurrency is realized by the nondeterministic interleaving of each CPU's steps. To simplify reasoning about all possible interleavings, we borrow the ideas of CertiKOS to lift multiprocessor execution to a *CPU-local* model [33]. The machine state σ for our model consists of per-physical CPU private state (e.g., CPU registers) and a global *logical log*, a serial list of *events* generated by all CPUs throughout their execution. σ does not explicitly model shared objects, including anything stored in physical memory. Instead, events incrementally convey interactions with shared objects, whose state may be calculated by *replaying* the logical log. An event is emitted by a CPU and appended to the log whenever that CPU invokes a primitive that interacts with a shared object. All effects coming from the environment are encapsulated by and conveyed through an *event oracle*, which yields events emitted by other CPUs when queried. To account for all possible concurrent interleaving, how the event oracle synchronizes these events is left abstract, its behavior constrained only by rely-guarantee conditions [40]. CPUs need only query the event oracle (a query move) before interacting with shared objects, since its private state is not affected by these events. Querying the event oracle will result in a composite event trace of the events from other CPUs interleaved with events from the local CPU. A local CPU makes a step via the CPU-local move.

For simplicity, we describe AbsMachine as a sequentially consistent model – writes always take effect in program order, and reads always read from the most recent write. Although Arm supports relaxed memory, we prove that all shared memory accesses in the KCore implementation are correctly protected by spinlocks. Because the spinlocks use barriers that prevent memory accesses from being reordered beyond their critical sections, we can show that KCore only exhibits sequentially consistent behavior. As a result, our guarantees over KCore verified using a sequentially consistent model still hold on Arm's relaxed memory model. This proof is beyond the scope of this paper.

Although the abstract machine model is specified in the bottom machine model of our proof, each successive layer implicitly has a machine model which is used to express how events at that layer affect machine state. For example, each layer has some notion of memory to support memory load and store primitives. For many layers, most primitives and their effect on the machine model at the overlay interface are the same as those at the underlay interface. These *passthrough* primitives and their effects on machine state do not need to be respecified for each higher layer. On the other hand, each layer may define new primitives based on a higher-level machine model, so long as a refinement can be proven between the layer's implementation over the underlay interface and the overlay interface.

A key aspect of our proofs is to abstract away the low-level details of the machine model, layer by layer, by proving refinement between the software implementation using a lower-level machine model and its specification based on a higher-level machine model. Specifically, by proving refinement relations between adjacent layers, we successively verify that KCore's implementation over AbsMachine refines the abstract top-level specification defined by `TrapHandler`, as shown in Figure 2. For example, we verify that the TLB behavior exposed by AbsMachine is wholly encapsulated by our implementation, and is thus abstracted from `TrapHandler`'s specification.

4.2 Page Table Management

As shown in Figure 3a, AbsMachine models Arm hardware's multi-level page tables. A page table can include up to four levels, referred to using Linux terminology as pgd, pud, pmd, and pte. AbsMachine models both regular and huge page table mappings, as used by KVM and also employed by KCore. KCore maintains stage 2 page tables — one per VM and one for KServ — as well as its own EL2 stage 1 page table. The functions for KCore to manipulate page tables are implemented and verified at the four layers of the MMU PT module, shown in Figure 2. The PTAlloc layer dynamically allocates page table levels, e.g.,pud, pmd, and pte. The PTWalk layer provides helper functions for walking an individual level of the page table, e.g., walk_pgd, walk_pud, etc. The NPTWalk layer uses PTWalk's primitives to perform a full page table walk. The NPTOps layer grabs and releases page table locks to perform page table operations, such as the map_page function that maps a VM's guest physical frame number (gfn) to a physical frame number (pfn) by calling the set_s2pt function in the NPTWalk layer to create a new mapping in the VM's stage 2 page table:

```
void map_page(u32 vmid, u64 gfn, u64 pfn, u64 attr) {
  acq_lock_s2pt();
  if (!get_s2pt(vmid, gfn)) {
    set_s2pt(vmid, gfn, pfn, 4K, attr);
  }
  rel_lock_s2pt();
}

void set_s2pt(u32 vmid, u64 gfn, u64 pfn, u32 size,
              u64 attr) {
  u64 pgd, pud, pmd, pte;
  pgd = walk_pgd(vmid, gfn);
  pud = walk_pud(vmid, pgd, gfn);
  pmd = walk_pmd(vmid, pud, gfn);
  if (size == 2M) {
    /* make sure pmd is not mapped to a pte */
    if (pmd_table(pmd) != PMD_TYPE_TABLE)
      set_pmd(vmid, pmd, gfn, pfn, attr);
  } else if (size == 4K) {
    if (pmd_table(pmd) == PMD_TYPE_TABLE) {
      pte = walk_pte(vmid, pud, gfn);
      set_pte(vmid, pte, gfn, pfn, attr);
    }
  }
}
```

We need to prove that KCore correctly manages its own stage 1 page table and all stage 2 page tables to enforce memory isolation among VMs, regardless of how KServ and VMs manage their own stage 1 page tables. To simplify this proof and the reasoning related to page tables at higher layers, we first abstract away the underlying implementation details and refine the multi-level page table into a flat map, as shown in Figure 3b, at the layer NPTWalk. For example, we refine the stage 2 page table into a flat map from gfn to a physical frame tuple (pfn, size, attr), where size is the size of the page, 4KB or 2MB, and attr encompasses attributes of the page, such as its memory access permissions.

This refinement is proven by first showing that the multi-level page table managed by KCore always forms a *tree* data structure—every page table at lower levels (pud, pmd, and pte) is referenced by only one page table entry at higher levels. We verify that KCore enforces the following two properties: (1) a lower-level page table can only be allocated and inserted during the page table walk when the target page table level does not exist, and (2) the allocated page table is a free and empty page. The allocated page is free such that no page table entry references it before the insertion. The allocated page is empty such that it does not contain any existing page tables. In this way, if the page table initially forms a tree, inserting this allocated page still results in a tree. The first property ensures that each edge of the tree before insertion remains unchanged after the insertion.

We then verify that the tree structure can be refined to a flat map by showing that updating the mapping for a gfn does not affect the mapping for any other $\texttt{gfn'} \neq \texttt{gfn}$. Suppose both gfn and gfn' are regular or huge pages. If the page walks for gfn and gfn' diverge at some level, they will fall into different leaf nodes due to the tree structure. If gfn and gfn' have the same page walk path, their pte indices will be different if they are regular pages, and their pmd indices will be different if they are huge pages, since $\texttt{gfn'} \neq \texttt{gfn}$.

The proof becomes more complicated when one page is a regular page and the other is a huge page. We have to prove that, once a pmd is allocated to store huge page mappings, it cannot be used to store lower-level pte pointers for regular pages, and vice versa. This is ensured by checking the size argument and the type of pmd during the page walk, as shown in the above example.

To unify the representation for the flat map at higher layers, we logically treat a 2MB huge page as 512 4KB pages. Changing one mapping for a 2MB huge page will cause updates to the mappings for all of its 512 4KB pages.

After the refinement proof at the layer NPTWalk, all the modules and their properties at higher layers can be reasoned about using this flat map without the need to deal with the implementation details of the multi-level page tables. For example, the memory isolation proof can be simplified significantly using the flat page map.

4.3 TLB Management

As shown in Figure 3a, AbsMachine models Arm's tagged TLB for each CPU, which caches page translations to regular and huge pages. In AbsMachine, each CPU is associated with an abstract TLB mapping, which maps VMIDs as tags to a set of TLB entries.

Arm TLBs cache three types of entries: (1) a stage 1 translation from a VM's virtual address to a gPA, (2) a stage 2 translation from a gPA to a PA, and (3) a translation from a VM's virtual address to a PA that combines stage 1 and stage 2 translations. AbsMachine models all three types of TLB entries, respectively, as: (1) a mapping from a virtual page number vpn to a tuple (gfn, size, attr), and (2) a mapping from a gfn to a physical frame tuple (pfn, size, attr), and (3) a mapping from a vpn to a gfn to a physical frame tuple (pfn, size, attr), where size and attr are used the same way as in AbsMachine's page tables, described in Section 4.2. Mappings are aligned to size (4KB or 2MB) of the mapped page. AbsMachine provides the following four basic TLB operations reflecting Arm's hardware behavior:

- *TLB lookup.* For a given memory load or store made by a VM VMID to access an address addr (gfn or vpn), AbsMachine searches the running CPU's TLB tagged with VMID, and checks if any entry translates addr. AbsMachine first checks if addr maps to an exact 4KB pfn, If no such mapping exists, it then checks if addr maps to a 2MB pfn by aligning addr to its 2MB base, pfn_2m, and searching the TLB using pfn_2m. If a matching entry is found, a TLB hits, the TLB returns the respective physical frame number if the VM memory operation is permitted, otherwise generates a permission fault. If no matching entry is found, the TLB returns None to indicate a TLB miss, and AbsMachine will then perform the address translation using page tables directly.
- *TLB refill.* If a TLB miss occurs on a memory access, AbsMachine refills the TLB with information from the ensuing page table walk, either a 4KB or 2MB translation to the CPU's tagged TLB. As previously mentioned, the refilled pfn must be aligned to the corresponding mapping size.
- *TLB eviction.* In AbsMachine, a memory load or store operation randomly invalidates a TLB entry before the actual memory access to account for all possible TLB eviction policies.
- *TLB flush.* Like Arm, AbsMachine exposes two primitives, mmu_tlb_flush1 and mmu_tlb_flush2, to flush TLB entries. mmu_tlb_flush2 takes a gfn and a VMID as arguments and invalidates the second type of TLB entry that maps the gfn. mmu_tlb_flush1 takes a VMID as an argument and invalidates all TLB entries associated with VMID that are either the first or third type of TLB entry. Hypervisors like KVM must use mmu_tlb_flush1 to conservatively flush all of a VM's TLB entries related to stage 1 translations when they update stage 2 page tables because they do not track how VMs manage their own stage 1 page tables. Like KVM, KCore uses both primitives to flush TLB entries as needed when updating a VM's stage 2 page tables. For simplicity, we use mmu_tlb_flush to refer to a call to both mmu_tlb_flush1 and mmu_tlb_flush2.

Note that the first three operations, TLB lookup, refill, and eviction, model Arm's TLB hardware behavior during the memory access, while the last operation, TLB flush, provides a set of primitives for the KCore software to perform TLB maintenance, implemented and verified at the NPTOps layer of the MMU PT module shown in Figure 2.

At the layer NPTOps, we verify that TLB entries are correctly maintained by KCore and that no principal, a VM or KServ, can use the TLB to access a physical page that does not belong to it, regardless of the behavior of KServ or any VM. In this way, we can hide TLB and TLB-related operations from all the layers above NPTOps, as shown in Figure 3b, to simplify the reasoning at higher layers.

This verification step introduces a concept of *page observers* to represent the set of all possible principals that can observe a pfn through TLBs or page tables. We write {pfn: n kserv}@TLB to denote that VM n and KServ are page observers to pfn through TLBs. As an example, consider the unmap_pfn_kserv primitive in NPTOps. When a page pfn is allocated by KServ to a VM *n*, KCore first calls unmap_pfn_kserv to remove the pfn from KServ's stage 2 page table, then inserts pfn in *n*'s stage 2 page table. The page observers before and after each step can be computed as follows:

```
// {pfn: kserv}@TLB   {pfn: kserv}@PT
unmap_pfn_kserv (pfn);
// {pfn: kserv}@TLB   {pfn: _}@PT
mmu_tlb_flush (pfn, kserv);
// {pfn: _}@TLB   {pfn: _}@PT
map_page (n, gfn, pfn, attr);
// {pfn: n}@TLB   {pfn: n}@PT
```

A TLB can be refilled using page tables' contents at any point due to a memory access on another CPU, so the (possible) page observers through TLBs must be a *superset* of the ones through page tables. That is why VM n can observe pfn through TLBs right after inserting pfn to n's page table. Intuitively, the superset relationship is because a TLB can contain the earlier and current cached page table translations while page tables contains only the current translations. The TLB flush collapses all possible (cached) observers to pfn to the observers defined by the page table.

The above example generates the following sequence of page observers through TLB:

```
{pfn: kserv}, {pfn: kserv}, {pfn: _}, {pfn: n}
```

If we merge consecutive identical page observers into a page observer group, we get the following page observer groups:

```
{pfn: kserv}, {pfn: _}, {pfn: n}        (1)
```

To prove that TLBs are maintained correctly and can be hidden at higher layers, we just need to show that TLBs and page tables generate the same sequence of page observer groups, even if page tables' observers are a subset of TLBs' observers. In the above example, the page observers through page tables are:

```
{pfn: kserv}, {pfn: _}, {pfn: _}, {pfn: n}
```

which can be merged to the same sequence of page observer groups shown in Eq. (1).

This property can be generally proven as follows. Starting with the same observer group through TLBs and page tables, the resulting observer groups produced by operations such as memory accesses, creating new page mappings in page tables, and TLB flushes are still the same. The only non-trivial case is unmapping pages, which introduces a new observer group through page tables, while a TLB would still show the old observer group. To avoid missing this new observer group, the TLBs must be invalidated by KCore calling `mmu_tlb_flush`.

Using this approach, incorrect maintenance of TLBs can be detected by a mismatch of page observer groups. Consider the following insecure implementation that invalidates the TLB before unmapping `pfn`.

```
// {pfn: kserv}@TLB   {pfn: kserv}@PT
mmu_tlb_flush (gfn, kserv);
// {pfn: kserv}@TLB   {pfn: kserv}@PT
unmap_pfn_kserv (pfn);
// {pfn: kserv}@TLB   {pfn: _}@PT
map_page (n, gfn, pfn, attr);
// {pfn: kserv n}@TLB   {pfn: n}@PT
```

Since TLBs can be refilled by page tables' contents, the page observers through TLBs remain the same after the TLB flush. The subsequent page unmapping does not invalidate TLBs such that the sequence of page observer groups through TLB for this insecure implementation is as follows:

```
{pfn: kserv}, {pfn: kserv n}
```

which is different from the one in Eq. (1), meaning that more information can be released through TLBs than page tables.

4.4 Cache Management

As shown in Figure 3a, AbsMachine includes PIPT writeback caches. Arm adopts MESI/MOESI cache coherence protocols, guaranteeing that all levels' of cache are consistent, meaning the hardware can retrieve the same contents from the cache located at different levels, and the updates to the cache are synchronized to the cache at different levels. Arm's multi-level caches can be modeled by AbsMachine as a uniform global cache. To model hardware that will invalidate and write back cached entries unbeknownst to software, for example, due to cache line replacement, AbsMachine exposes a `cache-sync` primitive that randomly evicts a cache entry and writes it back to memory. In KCore's specification, memory load and store operations call `cache-sync` before the actual memory accesses to account for all possible cache eviction policies. While caches are coherent, Arm hardware does not guarantee that cached data is always coherent with main memory; caches may write back dirty lines at any time. Like other architectures, Arm provides cache maintenance instructions to allow software to flush cache lines to ensure what is stored in main memory is up-to-date with what is stored in cache. AbsMachine provides a `cache-flush` primitive that models Arm's clean and invalidate instruction. The primitive takes a `pfn` as an argument, copies the `val` of `pfn` from cache to main memory if the entry is present in the cache, then removes `pfn`'s entry from the cache. Cache mismanagement could result in security vulnerabilities, so hypervisors must use these instructions to ensure that data accesses across all of its cores remain coherent, preventing stale data leaks.

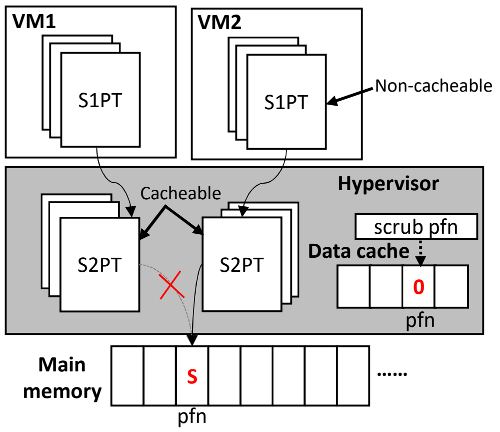


Figure 4: Attack Based on Mismatched Memory Attributes

Figure 4 shows how a malicious VM could leverage cache mismanagement on Arm hardware to potentially obtain confidential data of another VM from main memory. Suppose the hypervisor decides to evict a VM1's page `pfn`. It unmaps the page from VM1 and scrubs the page by zeroing out any residual data. Since the page no longer can be used by VM1, the hypervisor is free to reassign it to another VM, VM2, by mapping `pfn` to VM2's stage 2 page tables (S2PT). Arm hardware guarantees the scrubbing is synchronized across all CPU caches, but does not guarantee it is written back to main memory. Arm allows software to mark whether a page is cacheable or not by setting the memory attributes in the respective page table entry. When stage 2 translation is enabled, Arm combines memory attribute settings in stage 1 and stage 2 page tables. For a given mapping, caching is only enabled when both stages of page tables enable caching. Hypervisors allow VMs to manage their own stage 1 page tables for performance reasons. Although KCore always enables caching in stage 2 page tables, an attacker in VM2 could disable caching for the mapping to `pfn` in its stage 1 page table, allow it to bypass the caches and directly access `pfn` in main memory, which could contain VM1's confidential data. To protect VM memory against this attack, the hypervisor should flush `pfn`'s associated cache line after scrubbing the page to ensure that the changes are written back to main memory. This ensures VM2 can never retrieve VM1's secret in main memory.

To ensure that KCore correctly manages caches, we verify it over AbsMachine, which models writeback caches and cache bypass. AbsMachine models both cache and main memory as partial maps `pfn`$\mapsto$`val`, where `val` is the content stored in a given `pfn`. As a `pfn` moves between cache and main memory, AbsMachine propagates its content with it. For example, on a cacheable memory access, AbsMachine checks if the cache contains a mapping for `pfn`. If it does not, AbsMachine populates the cache with `val` from main memory. It then returns `val` for memory loads, and updates the cached value for memory stores. Similarly, on a cache-flush or cache-sync, AbsMachine flushes the pfn to main memory, populating main memory with the respective val from the cache.

Using AbsMachine, we prove that KCore always sets the memory attributes in the page tables that it manages to enable caching, maximizing performance. We then prove that KCore flushes caches in the primitives that can change page ownership, verifying that KCore's implementation refines its specification. Finally, we use KCore's specification to prove that KCore's cache management has no security vulnerabilities and does not compromise VM data. We discuss the first two proofs here, but defer the latter proof to Section 4.6.

We first prove that KCore always sets the memory attributes in the stage 2 page tables for VMs and KServ to enable caching. KCore updates stage 2 page table entries by calling the verified map_page primitive, as discussed in Section 4.2. map_page is passed the attr parameter to set the page table entry attributes. We verify the primitives that call map_page pass in the correct attr to enable caching. Specifically, we verify the implementation of map_pfn_vm and map_pfn_host in the MemAux layer, which call map_page to map a pfn to a VM's and KServ's stage 2 page tables, respectively refine their specifications that pass an attr value with caching enabled to map_page. We also prove that KCore always sets the memory attributes in its own EL2 stage 1 page tables to enable caching. Similar to map_page, NPTOps provides a map_page_core primitive for updating EL2 stage 1 page tables, which in turn calls set_s1pt in NPTWalk to update the multi-level page tables — we prove the correctness of these primitives similarly to the proofs for map_page and set_s2pt. We then verify the primitives that call map_page_core pass in the correct attr to enable caching.

We then prove that KCore correctly flushes the cache in the primitives that change page ownership. In the MemAux layer, we prove the correctness of assign_pfn_vm and clear_vm_page. assign_pfn_vm unmaps pfn from KServ and assigns the owner of a newly allocated pfn to a VM. clear_vm_page reclaims a pfn from a VM upon the VM's termination, scrubs the pfn, and assigns the owner of the pfn to KServ. We prove that the implementations of both primitives refine their specifications that call cache-flush.

4.5 SMMU Management

As shown in Figure 3a, AbsMachine models Arm's SMMU, which supports a shared SMMU TLB and SMMU multi-level page tables, that can be allocated for each device dev_k. The TLB is tagged, and page tables can support up to four levels of paging with regular and huge page support, similar to the page tables and TLBs discussed in Sections 4.2 and 4.3. Unlike memory accesses from CPUs, there are no caches involved in memory accesses through the SMMU. For simplicity, we only describe the SMMU stage 2 page tables, used by the SMMU implementation [5] on the Arm Seattle server hardware we used for evaluation in Section 6. AbsMachine also provides dev_load and dev_store operations to model memory accesses of DMA-capable devices attached to the SMMU.

KCore controls the SMMU and maintains the SMMU TLB and SMMU page tables for each dev_k. TLB entries are tagged by VMID. The parts of KCore that manipulate page tables are the four layers of SMMU PT shown in Figure 2. Similar to how we refine multi-level page tables in NPTWalk as discussed in Section 4.2, we refine the SMMU multi-level page table and its multi-level page table walk in MmioSPTWalk in SMMU PT into a layer specification with a partial map that maps an input page frame from device address space, devfn $\mapsto$ (pfn, size, attr), where size is the size of the page, 4KB or 2MB, and attr encompasses attributes of the page. Once we prove this refinement, higher layers that depend on SMMU page tables can be verified against the abstract page table, enabling us to prove the correctness of KCore's SMMU page table management.

Similar to how we refine CPU TLBs as discussed in Section 4.3, we refine the SMMU TLB in MmioSPTOps so that it is abstracted away from higher layers. We model the SMMU TLB as a set of partial maps, each map identified by VMID and mapping devfn $\mapsto$ (pfn, size, attr). AbsMachine models SMMU TLB invalidation by exposing a smmu-tlb-flush primitive to flush all entries associated with a VMID [5]. We prove the correctness of KCore with the SMMU TLB by verifying it correctly flushes entries to ensure consistency with the SMMU page tables, then abstract away the TLB by proving that the MmioSPTOps implementation using the SMMU TLB refines a simpler, higher-level specification without the SMMU TLB. We prove unmap_spt in MmioSPTOps calls smmu-tlb-flush after unmapping a pfn from the SMMU page table.

4.6 Security Guarantees

By proving that KCore's implementation refines its top-level Coq specification, we can then use the high-level specification to prove higher-level security guarantees. Proving security guarantees is much easier using the specification because we can avoid being inundated with the details of KCore's entire implementation, and we can use the simplified machine model refined from the lower layers. For instance, to prove the security properties for VM's memory accesses, we can reason over the memory load and store primitives at KCore's top layer based on the abstract single-level page tables without TLB, instead of the primitives defined in AbsMachine using multi-level page tables with TLB. We ensure the specification soundly captures all behaviors of the KCore implementation so the proven guarantees hold on the implementation.

We prove that SeKVM protects their VMs' data *confidentiality*—adversaries should not be privy to private VM data—and *integrity*—adversaries should not be able to tamper with private VM data. For some particular VM, potential adversaries are other VMs hosted on the same physical machine, as well as the hypervisor itself—specifically, HypSec's untrusted KServ. Our goal here is to verify that, irrespective of how any principal, KServ or another VM, behaves, KCore protects the security of each VMs' data. We formulate confidentiality and integrity as *noninterference assertions* [30]—invariants on how principals' behavior may influence one another. For confidentiality, we show the behaviors of all other VMs and KServ remains unaffected despite any changes the VM made to its data. For integrity, we prove that a VM's behavior acting upon its data is unaffected by other VMs' or KServ's behaviors, therefore its data is intact.

We can prove noninterference by showing state indistinguishability, which means that two machine states observable to a principal are the same. Machine states include a principal's data in CPU registers and memory. Data in memory includes data in main memory and caches as well as metadata in the principal's page tables. We want to prove that starting from any two indistinguishable states to principal p, the abstract machine should only transition to a pair of states that are still indistinguishable to p. We leverage previous work to prove this for data in CPU registers [48] and focus our discussion on proving this for memory. In particular, we need to prove that primitives that are part of the top-level specification that affect the management of page tables, caches, and SMMU preserve state indistinguishability since they can all affect memory. Since we have proven that TLBs do not need to be considered as part of the abstract machine model used by the top-level specification, TLBs do not need to be considered as part of our noninterference proofs. Note that pages explicitly shared via `GRANT_MEM` are not considered private and not included in the VM data protected by SeKVM until sharing is revoked using the `REVOKE_MEM` hypercall. While our proofs do account for this dynamically changing sharing of pages [48], we omit further discussion of `GRANT_MEM` and `REVOKE_MEM` for simplicity and focus on protecting memory private to each principal.

We prove that the use of page tables by top-level primitives preserves state indistinguishability by first proving a page table isolation invariant that any page mapped by a principal's stage 2 page table must be owned by itself. As discussed in Section 3, KCore assigns an owner for each page. Since each page has at most one owner, page tables, and address spaces, are isolated. With this invariant, we can prove that a principal p's states are not changed by any other principal q's operations on q's own address space. In a similar vein, we also prove that primitives that cause a page to be transferred from principal p to another principal q also do not affect state indistinguishability; one principal must be KServ on all transfers. If the transfer is from KServ to a VM, KCore ensures that such a page is first unmapped from KServ's stage 2 page table before the page's ownership is changed to the VM, and is only mapped to the VM's stage 2 page table after the ownership is changed to the VM. If the transfer is from a terminated VM to KServ, KCore clears the contents of the page before it is transferred so VM data is not leaked to KServ. Since KServ never has private VM data, it cannot leak such data when transferring a page to another VM. As a result, the use of page tables preserves state indistinguishability with respect to VM memory.

We prove that the use of caches by top-level memory load and store primitives preserves state indistinguishability so that the potential attack shown in Figure 4 cannot happen. We first prove noninterference when the ownership of a page does not change. If a principal p always owns a page `pfn`, only p can access that page. If only p can access `pfn`, `pfn` will only be cached as a result of being accessed by p. Based on the page table isolation invariant, the pages owned by p that can be in the cache must be a subset of the pages mapped in p's stage 2 page table. Since page tables and address space are isolated, so are each principal's entries in the cache. We can thereby prove that a principal p's states are not changed by any other principal's q load and store operations on q's own address space even if those operations involve the cache.

We then prove noninterference when KCore changes the principal associated with a `pfn`, which occurs when KServ allocates a new page to handle a VM's page fault, and reclaims the pages from a terminated VM. The former occurs when KServ calls the `run_vcpu` hypercall to execute a VM's VCPU after allocating a new `pfn` to the faulting VM, in which KCore unmaps the `pfn` from KServ's stage 2 page table, calls `assign_pfn_vm` to assign the owner of the `pfn` to the faulting VM, and maps the `pfn` to the VM's stage 2 page table before switching to the VM. The latter occurs when KServ calls the `clear_vm` hypercall to reclaim all `pfns` from a terminated VM, in which KCore calls `clear_vm_page` to scrub and assign the owner of these `pfns` to KServ.

When allocating a new page to handle a VM's page fault, KCore calls `cache-flush` on the `pfn` in `assign_pfn_vm` before mapping the `pfn` to p stage 2 page table. If `pfn` is cached, this causes `pfn` to be invalidated in the cache and its content is synchronized to main memory; otherwise it has no effect. We prove noninterference for KServ. Starting from two indistinguishable states for KServ, `run_vcpu` in two executions will unmap the same `pfn` from KServ's stage 2 page tables; thus, the resulting states remain indistinguishable to KServ since it cannot access `pfn` after the unmap. We prove noninterference for VMs other than p. Consider a VM q different from VM p. We prove that KServ never allocates the `pfn` owned by VM q to p, and executing `run_vcpu` does not affect q's states. Therefore, the resulting states of q remain indistinguishable. Finally, we prove noninterference for the VM p. Starting from two indistinguishable states for p, the resulting cache will not contain an entry for `pfn`, and `pfn`'s contents in memory contains the same value, while the `pfn` is mapped to p's stage 2 page tables. The resulting states are indistinguishable to p.

When reclaiming pages from a terminated VM p, KCore scrubs each reclaimed `pfn` and calls `cache-flush` on the `pfn` in `clear_vm_page`, which invalidates the `pfn` in the cache and writes the scrubbed `pfn` to main memory; `cache-flush` has no effect if the pfn is not cached. We prove noninterference for KServ. From two states indistinguishable to KServ, after making the hypercall, the pfns reclaimed from *p* will be owned by KServ. These pages will not be cached, and their contents in memory are scrubbed. The resulting states remain indistinguishable to KServ. This ensures that an attacker in KServ that bypasses the cache, as shown in Figure 4, cannot access VM *p* data. We prove noninterference for all VMs other than *p*. Consider a VM *q* different from VM *p*, starting from two indistinguishable states, KCore does not change any of *q*'s states when handling the hypercall for KServ, thus the resulting states of *q* remain indistinguishable.

We prove that the use of SMMU page tables by top-level primitives preserves state indistinguishability. Similar to page tables, we verify an SMMU page table isolation invariant that any page mapped by a device's SMMU page table must be owned by the device's owner. With this invariant, we prove that a principal *p*'s states are not changed by load and store operations from a device owned by any other principal *q* using their SMMU page tables. Similarly, we prove that SMMU primitives that transfer page ownership also do not affect state indistinguishability. The transfer only happens when KServ calls the SMMU hypercall to map a pfn to the SMMU page table used by a VM *p*'s device. KCore ensures the pfn is unmapped from KServ's stage 2 page table before transferring the owner of pfn from KServ to *p*. We thus ensure that use of SMMU page tables preserves state indistinguishability with respect to VM memory.

Although SeKVM's implementation was based on the codebase of HypSec, we have verified the correctness of KCore, SeKVM's TCB, and verified the security guarantees of SeKVM. We verified that KCore contains no vulnerabilities and that any vulnerabilities in KServ cannot compromise SeKVM's guarantees of VM confidentiality and integrity. In fact, while verifying SeKVM, we found various bugs in HypSec's TCB that affect HypSec's security guarantees. For example, we found a TLB management bug in which HypSec did not flush the SMMU TLB after unmapping a page from the SMMU page tables. We fixed the bug in KCore by adding a SMMU TLB flush after the unmap. As another example, we found a cache management bug in HypSec in which a VM boot image may be cached when loaded from the file system but not written back to main memory. As VMs are booted with paging and caching disabled, it is possible that the VMs access the page content in memory, thereby not using the correct VM images. We fixed the bug in KCore by flushing the corresponding cache lines for memory that contain the pre-loaded VM image before booting the VM, ensuring the use of the correct VM image loaded in memory.

5 Implementation

We refactored KVM into SeKVM, starting with the HypSec codebase and structuring its TCB into layers. We first did this with KVM in the v4.18 Linux kernel, which involved modifying or adding roughly 15K lines of code (LOC) across both KCore and KServ. Most of the added code was 10.1K LOC in KCore for the implementation of Ed25519 and AES from the verified HACL* crypto library [74]. Other than HACL*, KCore consisted of 3.8K LOC, of which 3.4K LOC was in C and 0.4K was in Arm assembly. Table 1 shows the 3.8K LOC categorized by the modules shown in Figure 2 (C+Asm).

Component	C+Asm	Spec	Code	Refine	CodeAll	RefineAll
Exit Handler	0.4K	1.7K	0.2K	1.1K	1K	1.4K
VCPU	0.8K	0.5K	2.4K	0.9K	3.3K	1.3K
VM Boot	0.9K	1.0K	0.6K	1.1K	2.8K	1.5K
SMMU	0.5K	0.7K	0.2K	1.0K	1.8K	1.4K
VM Mem	0.5K	0.9K	0.6K	2.2K	2.3K	2.6K
SMMU PT	0.2K	0.5K	0.1K	2.3K	1.6K	2.7K
MMU PT	0.4K	0.5K	0.1K	4.3K	1.7K	4.7K
Lock	0.1K	0.2K	1.2K	1.8K	2K	2.2K
Total	3.8K	6.0K	5.4K	14.7K	16.5K	17.8K

Table 1: KCore Implementation and Proof Effort in Lines of Code

We then retrofitted KVM in the v5.4 Linux kernel, which involved reusing much of the same 15K LOC. Of the 15K LOC, less than 100 LOC needed to be changed in KServ going from v4.18 to v5.4, mostly to support installing and initializing KCore on a different codebase before KCore starts running in EL2. No code changes were required in KCore in going from v4.18 to v5.4. These results indicate that the changes needed to retrofit a widely-used, commodity hypervisor so it can be verified and integrated with multiple versions of a commodity host kernel were modest overall.

We verified all of KCore's C and assembly code. Table 1 shows the LOC in Coq for proving the correctness of KCore's code, categorized by the modules shown in Figure 2. The proof effort for each module consists of writing the Coq specifications (Spec), code proofs (Code) to verify the C and assembly code refines the Coq specifications, and layer refinements (Refine) to verify at each layer the implementation on the underlay interface refines the overlay interface, thereby linking the layers together to refine the top-level specification.

Some modules required much more manual effort than others. For the specifications, the LOC for the Exit Handler module is higher than other modules because it includes the top layer TrapHandler specification that encompasses all of KCore's behavior. For code proofs, the LOC for the VCPU module is higher than other modules because it has both loops and assembly code. This is because we used automated reasoning to reduce manual effort, but our methods do not support automating loop verification or assembly code. For layer refinement, the LOC for the MMU PT proof is higher than other modules because refining the multi-level page table implementation to a flat map specification was the most complex refinement proof.

Table 1 also shows all of the resulting code in Coq for code proofs (CodeAll) and layer refinement (RefineAll), by adding automatically generated LOC to the manually written LOC. For some modules, the use of automated reasoning significantly simplified the manual effort, such as for the code proofs for the `MMU PT`, `SMMU PT`, and `SMMU` modules. However, we did not apply automated reasoning uniformly for all modules because different parts of the system were verified by different authors who took different approaches. For example, we did not use Coq tactics to automate the proofs for the `Lock` module, resulting in more LOC for its code proofs, but this could have been done. While automated tools helped significantly with code proofs, they did not help much with layer refinement, as shown by comparing the manually written versus total LOC for each in Table 1.

In addition to the Coq code for proving the correctness of each module, we implemented the machine model and proved the security guarantees in Coq. 1.8K LOC were used to implement AbsMachine, which models the multiprocessor hardware behaviors including multi-level page tables for the MMU and SMMU, TLBs, and write-back caches with bypass support. AbsMachine primitives used by higher layers were passed through to those layers then verified as part of each layer. The security proofs, including the invariant and noninterference proofs, consist of 4.8K LOC. Roughly 1K LOC were used to verify the isolation invariants mentioned in Section 4.6 for the MMU and SMMU page tables. The rest of the 3.8K LOC were noninterference proofs for KCore's top-level primitives; for example, these proofs involved proving state indistinguishability with respect to caches. We did not link HACL's F* proofs with our Coq proofs, or our Coq proofs for C code with those for Arm assembly code. The latter requires a verified compiler for Arm multiprocessor code; no such compiler exists. No changes were required to the proofs used to verify KVM in the Linux kernel v4.18 versus v5.4.

6 Performance

We quantify the performance of SeKVM against unmodified KVM as well as HypSec highlighting how a commodity hypervisor with a verified TCB performs against unverified versions. All experiments were run on a 64-bit Armv8 AMD Seattle (Rev.B0) server with 8 Cortex-A57 CPU cores, 16 GB of RAM, a 512 GB SATA3 HDD for storage, an AMD 10 GbE (AMD XGBE) NIC device. The hardware we used supports Arm VE, but not VHE [21, 22]. For client-server experiments, the clients ran on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The clients and the server communicated via a 10 GbE network connection.

To provide comparable measurements across the systems, we kept the software environments across all platforms the same as much as possible. We tested unmodified KVM, HypSec, and SeKVM based on two different versions of mainline Linux, 4.18.0 and 5.4.0, both with QEMU 2.3.50. VMs used the same kernel version as the host, and all hosts

Name	Description
Kernbench	Compilation of the Linux 4.9 kernel using `allnoconfig` for Arm with GCC 5.4.0.
Hackbench	`hackbench` [56] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	`netperf` v2.6.0 [41] running `netserver` on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_MAERTS (throughput), and TCP_RR (latency).
Apache	`Apache` v2.4.18 Web server running `ApacheBench` [1] v2.3 on the remote client, which measures number of handled requests per second when serving the 41 KB `index.html` file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.25 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.7.26) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 2: Application Benchmarks

and VMs ran Ubuntu 16.04.06. We modified virtio front-end drivers in the VM kernel on SeKVM and HypSec to use the `GRANT_MEM` and `REVOKE_MEM` hypercalls to enable shared memory communication with back-end drivers in KServ. All VMs used paravirtualized I/O (virtio), typical of cloud infrastructure deployments such as Amazon EC2.

We ran benchmarks in each VM and compared their performance to native hardware. Each native or VM instance was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. Specifically, we used the following configurations: (1) native Linux capped at 4 cores and 12 GB RAM, and (2) a VM using KVM with 8 cores and 16 GB RAM, with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. We measured multi-core configurations to reflect real-world server deployments. For VMs, we pinned each VCPU to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU [20,21,49,50]. For client-server benchmarks, the clients ran natively on Linux and used the full hardware available.

We ran real application workloads to compare SeKVM with HypSec and unmodified KVM. Table 2 lists the workloads, a mix of widely-used CPU and I/O intensive benchmarks. For the v4.18 configuration, we compared the following five system configurations with HypSec: (1) Native unmodified Linux host kernel without Full Disk Encryption (FDE), (2) Unmodified KVM and guest kernel with FDE (KVM), (3) HypSec and paravirtualized guest kernel with FDE (HypSec), (4) SeKVM and paravirtualized guest kernel with FDE (SeKVM), (5) SeKVM and paravirtualized guest kernel with FDE and TLB flushes during world switches (SeKVM-TLB-FLUSH).

We compared VM performance with FDE to bare-metal execution without FDE, to conservatively quantify the performance overhead in the presence of end-to-end I/O protection. We also compared the performance of SeKVM versus SeKVM while flushing all entries from the TLB in

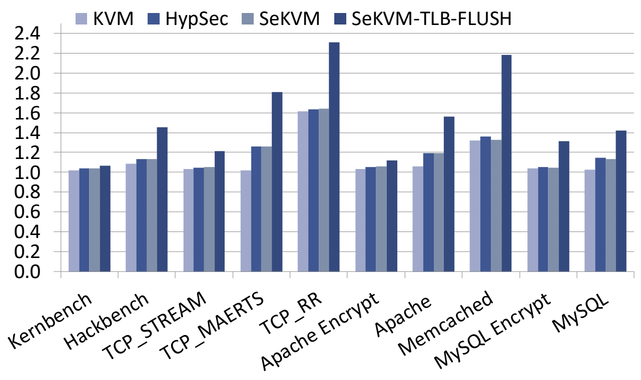


Figure 5: Application Benchmark Performance - Linux v4.18

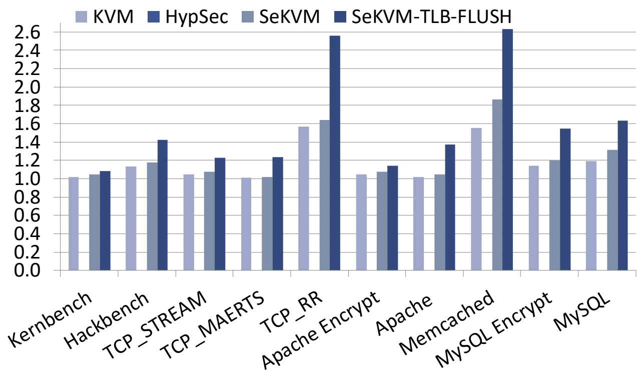


Figure 6: Application Benchmark Performance - Linux v5.4

each world switch to remove all cached entries used by EL0 and EL1, to measure the performance impact of a verified implementation that models tagged TLB behavior versus one that does not (and must therefore perform additional TLB flushes for verified correctness).

Figures 5 and 6 show the relative overhead of executing in a VM in our v4.18 and v5.4 hypervisor configurations. We normalize the performance results to native execution on the respective unmodified Linux kernel, with 1.0 indicating the same performance as native hardware. Lower numbers mean less overhead. We report results for Apache and MySQL both with and without TLS/SSL to show performance with network encryption as well. Both figures show that SeKVM has only modest performance overhead compared to unmodified KVM. Figure 5 also shows that SeKVM has comparable performance to HypSec, but the HypSec implementation was not available for v5.4, so Figure 6 shows no HypSec v5.4 measurements. Overall, the measurements show that a commodity hypervisor with a verified TCB on multiprocessor hardware can achieve excellent performance.

As shown in Figures 5 and 6, flushing the TLB during each world switch results in significant performance overhead. The overhead is especially pronounced in I/O intensive workloads, where frequent world switches between VMs and

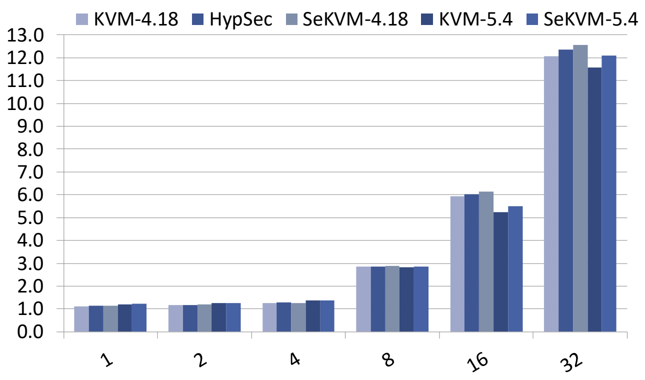


Figure 7: Multi-VM Performance with Hackbench

KServ result in more frequent TLB flushes. This comparison quantifies the cost of not modeling a tagged TLB, which would force TLB flushes on each world switch to ensure correctness. Our measurements show that this can result in an additional 70% overhead for some application workloads such as Memcached compared to using a tagged TLB as is standard practice for commodity hypervisors.

To provide a measure of multi-VM performance, we also measured the performance of SeKVM compared to HypSec and unmodified KVM running multiple VMs, each running Hackbench. We tested five hypervisor configurations: KVM and SeKVM on Linux v4.18 and v5.4, and HypSec on only v4.18. To scale the experiment on the same Seattle Arm server host, we made some changes to the VM and Hackbench configurations. Each VM was configured in a similar manner as our previous experiments, except we reduced the number of cores and RAM of each VM to two cores and 256 MB of RAM, respectively. We changed the parameters in Hackbench from the previous setup to run 20 process groups in 500 loops, so that it could run successfully in the more resource-constrained VMs. In addition, we did not use FDE given the limited memory assigned to each VM. We measured the performance of 1, 2, 4, 8, 16, and 32 VMs.

Figure 7 shows the average results from each VM running on HypSec and SeKVM normalized to native execution of one instance of Hackbench using the respective Linux version with the same configuration, though there was minimal difference in native execution performance between kernel versions. The results show that SeKVM incurs modest performance overhead over KVM and HypSec, even as the number of VMs scales. The overhead versus one instance of Hackbench natively executed is of course higher when running many instances of Hackbench instead of just one, but the relative overhead of SeKVM versus KVM remains small. Note that although KCore's data race-free implementation does not take full advantage of Armv8 relaxed memory behavior, the performance impact on SeKVM is minimal.

7 Related Work

Previous work has verified uniprocessor systems, including seL4 [43], Nickel [59], Serval [54], and Komodo [28]. None of these approaches can be directly applied to verify multiprocessor systems such as SeKVM. CertiKOS has verified a series of uniprocessor and multiprocessor OS kernels [9, 10, 13, 31–34], but like previous verified uniprocessor systems, did not model common hardware features including shared page tables, tagged TLBs or caches. In contrast, SeKVM is verified on a multiprocessor abstract machine that models these widely-used hardware features.

Various verified systems can be used as hypervisors, but are limited in their functionality and what has been verified. A version of seL4 verifies the functional correctness of some hypervisor features, but not the MMU functionality [2, 42]. CertiKOS verifies the correctness of the mC2 kernel that provides some virtualization functionality. Both of these systems lack common hypervisor features such as support for multiprocessor VMs. The üXMHF hypervisor [65, 66] verifies simple properties, such as memory integrity of their multiprocessor microhypervisor implementation, but does not verify its functional correctness. Unlike SeKVM, the proofs were reasoned on a simple abstract hardware that does not model concrete MMU features. The Verisoft team [44] applies the VCC framework [12] to verify Hyper-V. VCC does not include a realistic hardware model. Only 20% of the hypervisor code is verified for function contracts and type invariants at the source code level, with no correctness guarantees of the overall hypervisor's behavior. In contrast, SeKVM's security guarantees and its TCB are fully verified while supporting commodity hypervisor features inherited from KVM.

We build on our previous work [47, 48] that introduced security-preserving layers and microverification to verify the security guarantees of a KVM hypervisor. We describe here for the first time (1) a new layered hardware model, (2) the construction of a layered implementation of SeKVM's TCB, KCore, (3) how the layered hardware can be used in conjunction with the layered software to verify KCore's functional correctness in the presence of widely-used multiprocessor hardware features such as tagged TLBs and coherent caches, and (4) how to account for all of these hardware features in verifying the security guarantees of SeKVM. We also demonstrate for the first time how both the implementation and verification of SeKVM can be extended to integrate with multiple versions of Linux as a host kernel with modest effort.

Formal shim verification [39] reduces the proof effort in verifying security guarantees about a large and untrusted code. Their techniques focus on proving that a small, sequential browser kernel, consisting of a few hundred LOC, enforces noninterference properties between components running in sandboxes. This approach is insufficient for SeKVM, whose multiprocessor core consists of a few thousand LOC, and leverages hardware virtualization features to implement hypervisor functionality.

Some work [57, 64, 73] has verified the MMU subsystem within an OS kernel. Unlike SeKVM, the verified component does not make any guarantees about the overall behavior of the system. Other work [62, 63] integrates the specifications of their abstract TLB into the Cambridge Arm model [29], but only uses it for proving the program logic of the system's execution, not the correctness of the actual implementation.

Microhypervisors [35, 60] take a microkernel approach to build clean-slate small hypervisors from scratch. These architectures mitigate vulnerabilities, but are not verified to be correct. In contrast, SeKVM retrofits KVM using microkernel principles to reduce its TCB and verifies its implementation, providing verified correctness and security guarantees with full-featured commodity hypervisor functionality. Nested virtualization [70] and special hardware features [7, 37, 68] have been used to protect VM data in memory against an untrusted hypervisor. Privileged code, such as a hypervisor, has been used to protect OS kernels [26, 58, 67] or applications [11, 27, 36, 52, 69] against untrusted software components. Unlike SeKVM, none of these systems verify their TCBs or prove the security properties of their designs.

8 Conclusions

We have presented SeKVM, the first Linux KVM hypervisor that has been formally verified. This is made possible using a layered design and verification methodology. We use layers to isolate KVM's TCB into a small core, then construct the core with layers such that we can modularize the proofs to reduce proof effort, modeling hardware features at different levels of abstraction tailored to each layer of software. We can then gradually refine detailed hardware and software behaviors at lower layers into simpler abstract specifications at higher layers, which can in turn be used to prove security guarantees for the entire hypervisor. Using this approach, we prove the correctness of KVM across two versions of Linux, using a novel layered machine model that accounts for realistic multiprocessor features including multi-level shared page tables, tagged TLBs, and a coherent cache hierarchy with cache bypass support. The layering requires only modest modifications to KVM and only incurs modest overhead versus unmodified KVM on real application workloads. Our work is the first machine-checked proof of the correctness and security of a commodity hypervisor on multiprocessor server hardware.

9 Acknowledgments

Xuheng Li helped with assembly code and layer refinement proofs. Nathan Dautenhahn provided helpful comments on earlier drafts. This work was supported in part by a Guggenheim Fellowship, DARPA contract N6600121C4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html [Accessed: Mar 8, 2021].

[2] seL4 Supported Platforms. https://docs.sel4.systems/Hardware [Accessed: Mar 8, 2021].

[3] The Coq Proof Assistant. https://coq.inria.fr [Accessed: Dec 16, 2020].

[4] ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0, June 2016.

[5] ARM Ltd. ARM CoreLink MMU-401 System Memory Management Unit Technical Reference Manual, July 2014.

[6] Michael Backes, Goran Doychev, and Boris Kopf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In *20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, February 2013.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 267–283, Broomfield, CO, October 2014.

[8] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[9] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–447, 2016.

[10] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. *Journal of Automated Reasoning*, 61(1):141–189, 2018.

[11] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 2–13, Seattle, WA, March 2008.

[12] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, pages 23–42, Munich, Germany, August 2009.

[13] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664, Santa Barbara, CA, June 2016.

[14] CVE. CVE-2009-3234. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3234, September 2009.

[15] CVE. CVE-2010-4258. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4258, November 2010.

[16] CVE. CVE-2013-1943. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1943, February 2013.

[17] CVE. CVE-2016-9756. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9756, December 2016.

[18] CVE. CVE-2017-17741. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17741, December 2017.

[19] Christoffer Dall. *The Design, Implementation, and Evaluation of the Linux ARM Hypervisor*. PhD thesis, Columbia University, February 2018.

[20] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. ARM Virtualization: Performance and Architectural Implications. *ACM SIGOPS Operating Systems Review*, 52(1):45–56, July 2018.

[21] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[22] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[23] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[24] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[25] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[26] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 191–206, Istanbul, Turkey, March 2015.

[27] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 1441–1458, Baltimore, MD, August 2018.

[28] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 287–305, Shanghai, China, October 2017.

[29] Anthony Fox. Formal specification and verification of arm6. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 25–40, Rome, Italy, September 2003.

[30] Joseph A Goguen and José Meseguer. Unwinding and Inference Control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (IEEE S&P 1984)*, pages 75–86, Oakland, CA, April 1984.

[31] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.

[32] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, September 2019.

[33] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669, Savannah, GA, November 2016.

[34] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.

[35] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-pacific Workshop on Workshop on Systems (APSys 2010)*, pages 19–24, New Delhi, India, August 2010.

[36] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 265–278, Houston, TX, March 2013.

[37] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing. vTZ: Virtualizing ARM Trustzone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 2017)*, pages 541–556, Vancouver, BC, Canada, August 2017.

[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 591–604, San Jose, CA, May 2015.

[39] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 113–128, Bellevue, WA, August 2012.

[40] Cliff Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5:596–619, October 1983.

[41] Rick Jones. Netperf. `https://github.com/HewlettPackard/netperf` [Accessed: Mar 8, 2021].

[42] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally Verified Software in the Real World. *Communications of the ACM*, 61(10):68–77, September 2018.

[43] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.

[44] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, pages 806–809, Eindhoven, The Netherlands, November 2009.

[45] Xavier Leroy. The CompCert Verified Compiler. `https://compcert.org` [Accessed: Mar 8, 2021].

[46] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[47] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Microverification of the Linux KVM Hypervisor: Proving VM Confidentiality and Integrity. Technical Report CUCS-003-20, Department of Computer Science, Columbia University, June 2020.

[48] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang 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.

[49] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 201–217, Shanghai, China, October 2017.

[50] Jin Tack Lim and Jason Nieh. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 557–574, Lausanne, Switzerland, March 2020.

[51] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 605–622, San Jose, CA, May 2015.

[52] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P 2010)*, pages 143–158, Oakland, CA, May 2010.

[53] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for Operating System Kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP 2012)*,

pages 126–142, Kyoto, Japan, December 2012.

[54] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, Huntsville, Ontario, Canada, October 2019.

[55] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Chicago, IL, November 2009.

[56] Rusty Russell, Zhang Yanmin, Ingo Molnar, and David Sommerseth. Improve hackbench. `http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c`, January 2008.

[57] Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with DMA. In *Proceedings of the 10th International Haifa Verification Conference (HVC 2014)*, pages 236–251, Haifa, Israel, November 2014.

[58] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 335–350, Stevenson, WA, October 2007.

[59] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.

[60] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, pages 209–222, Paris, France, April 2010.

[61] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, pages 21–41, Heraklion, Crete, Greece, July 2013.

[62] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation. In *Proceedings of the 2018 International Conference on Interactive Theorem Proving (ITP 2018)*, pages 542–559, Oxford, United Kingdom, July 2018.

[63] Syeda Hira Taqdees and Gerwin Klein. Reasoning about Translation Lookaside Buffers. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2017)*, pages 490–508, Maun, Botswana, May 2017.

[64] Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In *Proceedings of the NICTA Foraml Methods Workshop on OS Verification*, pages 73–97, Sydney, Australia, October 2004.

[65] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (IEEE S&P 2013)*, pages 430–444, San Francisco, CA, May 2013.

[66] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 2016)*, pages 87–104, Austin, TX, August 2016.

[67] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: A Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 2015)*, pages 347–360, Santa Clara, CA, July 2015.

[68] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 441–453, Vienna, Austria, February 2018.

[69] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 71–80, Seattle, WA, March 2008.

[70] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 203–216, Cascais, Portugal, October 2011.

[71] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 305–316, Raleigh, NC, October 2012.

[72] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in Paas Clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS 2014)*, pages 990–1003, Scottsdale, AZ, November 2014.

[73] Yongwang Zhao and David Sanán. Rely-Guarantee Reasoning About Concurrent Memory Management in Zephyr RTOS. In *Proceedings of the 31st International Conference (CAV 2019)*, pages 515–533, New York, NY, July 2019.

[74] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, Dallas, TX, October 2017.