# A Secure and Formally Verified Linux KVM Hypervisor

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

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00049

***Abstract*—Commodity hypervisors are widely deployed to support virtual machines (VMs) on multiprocessor hardware. Their growing complexity poses a security risk. To enable formal verification over such a large codebase, we introduce *microverification*, a new approach that decomposes a commodity hypervisor into a small core and a set of untrusted services so that we can prove security properties of the entire hypervisor by verifying the core alone. To verify the multiprocessor hypervisor core, we introduce *security-preserving layers* to modularize the proof without hiding information leakage so we can prove each layer of the implementation refines its specification, and the top layer specification is refined by all layers of the core implementation. To verify commodity hypervisor features that require dynamically changing information flow, we introduce *data oracles* to mask intentional information flow. We can then prove noninterference at the top layer specification and guarantee the resulting security properties hold for the entire hypervisor implementation. Using microverification, we retrofitted the Linux KVM hypervisor with only modest modifications to its codebase. Using Coq, we proved that the hypervisor protects the confidentiality and integrity of VM data, while retaining KVM's functionality and performance. Our work is the first machine-checked security proof for a commodity multiprocessor hypervisor.**

## I. Introduction

Cloud computing has enabled increasing numbers of companies and users to move their data and computation off-site into virtual machines (VMs) running on hosts in the cloud. Cloud computing providers deploy commodity hypervisors [1], [2] to support these VMs on multiprocessor hardware. The security of a VM's data hinges on the correctness and trustworthiness of the hypervisor. However, modern hypervisors are huge, complex, and imperfect pieces of software, often integrated with an entire operating system (OS) kernel. Attackers that successfully exploit hypervisor vulnerabilities may gain unfettered access to VM data in CPU registers, memory, I/O data, and boot images, and compromise the confidentiality and integrity of VMs—an undesirable outcome for both cloud providers and users [3].

Theoretically, formal verification offers a solution to this problem, by proving that the hypervisor protects VM data under all circumstances. However, this approach is largely intractable for commodity hypervisors—existing systems verification research has yet to demonstrate how one might feasibly reason about the security properties of full-featured, multiprocessor hypervisors. Most verified systems are specifically designed to be verified, meaning they lack basic features and are far simpler than their commodity counterparts. This suggests that the proof effort potentially required to fully verify a commodity system is far beyond feasible. It took seL4 [4] ten person-years to verify 9K lines of code (LOC), and CertiKOS [5] three person-years to verify 6.5K LOC. For comparison, KVM [1], a full-featured, multiprocessor hypervisor integrated with Linux, is more than 2M LOC. It remains unknown how security properties of such a vast system, not written with verification in mind, may be verified in its entirety.

To address this problem, we introduce *microverification*, a new approach for verifying commodity systems, based on the hypothesis that small changes to these systems can make key properties much easier to verify while preserving their overall functionality and performance. Microverification reduces the proof effort for a commodity system by retrofitting the system into a small core and a set of untrusted services, so that it is possible to reason about properties of the entire system by verifying the core alone. Based on microverification, we introduce MicroV, a new framework for verifying the security properties of large, multiprocessor commodity systems. MicroV further reduces proof effort by providing a set of proof libraries and helper functions to modularize proofs using a layered verification approach, abstracting detailed C and assembly implementations into higher-level specifications using the Coq proof assistant [6]. Using MicroV, we verify a retrofitted commodity system by first proving the functional correctness of its core, showing that its implementation refines its Coq specification, then use the specification to prove security properties of the entire system. Because the specification is easier to use for higher-level reasoning, it becomes possible to prove security properties that would be intractable if attempted directly on the implementation.

As shown in Figure 1, we use MicroV to prove, for the first time, the security properties of the Linux KVM hypervisor. First, we retrofit KVM into a small, verifiable core, KCore, and a rich set of untrusted hypervisor services, KServ. KCore mediates all interactions with VMs and enforces access controls to limit KServ access to VM data, while KServ provides complex virtualization features. Building on our previous work [7], we retrofit KVM/ARM [8], [9], the Arm implementation of KVM, given Arm's increasing popularity in server systems [10], [11], [12]. Arm provides Virtualization Extensions (Arm VE) [13] to support virtual machines. We leverage Arm VE to protect and run KCore at a higher privilege level than KServ, which encapsulates the rest of the KVM implementation, including the host Linux kernel. Retrofitting required only modest modifications to the original KVM code. Upon retrofitting, KCore ends up consisting of 3.8K LOC (3.4K LOC in C and 400 LOC in assembly), linked with a verified crypto library [14].

Second, we prove that the KCore implementation refines its layered specification. A key challenge we address is ensuring that the refinement between implementation and specification preserves security properties, such as data confidentiality and integrity. For example, this may not hold in a multiprocessor setting [15], [16] because intermediate updates to shared data within critical sections can be hidden by refinement, yet visible across concurrent CPUs. To



DOI 10.1109/SP40001.2021.00049

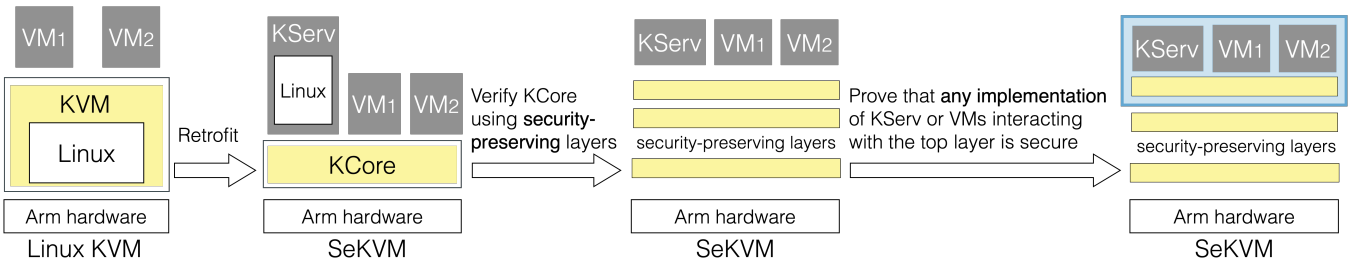



**Fig. 1:** Microverification of the Linux KVM hypervisor.

reason about KCore in a multiprocessor setting, MicroV introduces *security-preserving layers* to express KCore's specification as a stack of layers, so that each module of its implementation may be incrementally proven to refine its layered specification and preserve security properties. Security-preserving layers employ *transparent trace refinement*, a new technique to track updates to shared data and ensure that a critical section with multiple such updates that may be visible across CPUs is not refined into a single atomic primitive. This ensures that refinement does not hide information release. We use transparent trace refinement to verify, for the first time, the functional correctness of a multiprocessor system with shared page tables. Using security-preserving layers, we can ensure that the composition of layers embodied by the top-level KCore specification reflects all intermediate updates to shared data across the entire KCore implementation. We can then use the top-level specification to prove the system's information-flow security properties and ensure those properties hold for the implementation.

Finally, we use KCore's specification to prove that any malicious behavior of the untrusted KServ using KCore's interface cannot violate the desired security properties. We prove VM confidentiality and integrity using KCore's specification, formulating our guarantees in terms of noninterference [17] to show that there is no information leakage between VMs and KServ. However, a strict noninterference guarantee is incompatible with commodity hypervisor features, including KVM's. For example, a VM may send encrypted data via shared I/O devices virtualized via untrusted hypervisor services, thereby not actually leaking private VM data. This kind of intentional information release, known as declassification [18], does not break confidentiality and should be distinguished from unintentional information release. MicroV introduces *data oracles*, logical integer generators, which stand in as proxies for intentionally released data. The value returned by a data oracle is guaranteed to only depend on the state of the principal, a VM or KServ, reading the value. For example, the value of encrypted data that a VM sends to KServ is specified as the next integer returned by the data oracle. The integer masks the information flow of the encrypted data because it does not depend on the behavior of other VMs. After this masking, any outstanding information flow is unintentional and must be prevented, or it will affect the behavior of KServ or VMs. To show the absence of unintentional information flow, we prove noninterference assertions hold for *any behavior* by the untrusted KServ and VMs, interacting with KCore's top layer specification. The noninterference assertions are proven over this specification, for any implementation of KServ, but since KCore's implementation refines its specification via security-preserving layers, unintentional information flow is guaranteed to be absent for the entire KVM implementation.

While verifying KCore, we found various bugs in our initial retrofitting. Most bugs were discovered as part of our noninterference proofs, demonstrating a limitation of verification approaches that only prove functional correctness via refinement alone: *the high-level specifications may themselves be insecure*. In other words, these bugs were not detected by just verifying that the implementation satisfies its specification, but by ensuring that the specification guarantees the desired security properties of the system.

All the security-preserving layer specifications, transparent trace refinement proofs, and noninterference proofs were implemented using Coq [6]. Verification took two person-years to complete. Our verified KVM, SeKVM, incurs only modest performance overhead compared to unmodified KVM on real application workloads, and supports KVM's wide range of commodity hypervisor features, including running multiple multiprocessor VMs with unmodified commodity OSes, shared multi-level page tables with huge page support, standardized virtio I/O virtualization with vhost kernel optimizations, emulated, paravirtualized, and passthrough I/O devices with IOMMU protection against direct memory access (DMA) attacks, and compatibility with Linux device drivers for broad Arm hardware support. This is the first-ever multiprocessor hypervisor, and first-ever retrofitting of a widely-deployed commodity hypervisor, to provably guarantee VM data confidentiality and integrity.

## II. Threat Model and Assumptions

Our threat model is primarily concerned with hypervisor vulnerabilities that may be exploited to compromise private VM data. For each VM we are trying to protect, an attacker may control other VMs, control KServ, and attempt to exploit KCore vulnerabilities. We assume VMs do not voluntarily reveal their own private data, whether on purpose or by accident. We do not provide security features to prevent or detect VM vulnerabilities, so a compromised VM that involuntarily reveals its own data is out of the scope of our threat model. However, we do protect each VM from attacks by other compromised VMs. Attackers may control peripherals to perform malicious memory accesses via DMA [19]. Side-channel attacks [20], [21], [22], [23], [24], [25] are beyond the scope of the paper.

We assume a secure persistent storage to store keys. We assume 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.

## III. Overview of MicroV

Hypervisors must protect 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, SeKVM's untrusted KServ. Each of these *principals* run on one or more CPUs, with their execution and communication mediated by KCore. Our goal here is to verify that, irrespective of how any principal behaves, KCore protects the security of each VMs' data.

To do so, we formulate confidentiality and integrity as *noninterference assertions* [17]—invariants on how principals' behavior may influence one another. Intuitively, if the confidentiality of one VM's private data is compromised, then its adversaries' behavior should vary depending on that data. Thus, if the behavior of all other VMs and KServ remains the same, in spite of any changes made to private data, then that data is confidential. Integrity is the dual of confidentiality [26], [27]: if the behavior of a VM, acting upon its own private data, is not affected by variations in other VMs' or KServ's behavior, then its data is intact.

Using KCore's C and assembly code implementation to prove noninterference assertions is impractical, as we would be inundated by implementation details and concurrent interleavings. Instead, we use MicroV to show that the implementation of the multiprocessor KCore incrementally *refines* a high-level Coq specification. We then prove any implementation of KServ or VMs interacting with the top-level specification satisfies the desired noninterference assertions, ensuring that the entire SeKVM system is secure regardless of the behavior of any principal. To guarantee that proven top-level security properties reflect the behavior of the implementation of KCore, we must ensure that each level of refinement fully preserves higher-level security guarantees.

### A. Security-preserving Refinement

To enable incremental and modular verification, MicroV introduces security-preserving layers:

**Definition 1 (Security-preserving layer).** *A layer is security-preserving if and only if its specification captures all information released by the layer implementation.*

Security-preserving layers build on Certified Concurrent Abstraction Layers (CCAL) [28] to verify the correctness of multiprocessor code. Security-preserving layers retain the compositionality of CCALs, but unlike CCALs and other previous work, ensure refinement preserves security guarantees in a multiprocessor setting.

For each module $M$ of KCore's implementation, we construct a security-preserving layer $M@L \sqsubseteq S$, which states that $M$, running on top of the lower layer $L$, refines its interface specification $S$. Because the layer refinement relation $\sqsubseteq$ is transitive, we can incrementally refine KCore's entire implementation as a stack of security preserving layers. For example, given a system comprising of modules $M_3$, $M_2$, and $M_1$, their respective layer specifications $L_3$, $L_2$, and $L_1$, and a base machine model specified by $L_0$, we prove $M_1@L_0 \sqsubseteq L_1$, $M_2@L_1 \sqsubseteq L_2$, and $M_3@L_2 \sqsubseteq L_3$. In other words, once a module of the implementation is proven to refine its layer specification, we can use that simpler specification, instead of the complex module implementation, to prove other modules that depend on it. We compose these layers to obtain $(M_3 \oplus M_2 \oplus M_1)@L_0 \sqsubseteq L_3$, proving that the behavior of the system's linked modules together refine the top-level specification $L_3$.

All interface specifications and refinement proofs are manually written in Coq. We use CompCert [29] to parse each module of the C implementation into an abstract syntax tree defined in Coq; the same is done manually for assembly code. We then use that Coq representation to prove that each module refines its respective interface specification at the C and assembly level. The overall KCore implementation thereby refines a stack of security-preserving layers, such that the top layer specifies the entire system by its functional behavior over its machine state.

MicroV's security-preserving layer library provides facilities to soundly abstract away complications arising from potential concurrent interference, so that we may leverage sequential reasoning to simplify layer refinement proofs. The key challenge is handling objects shared across multiple CPUs, as we must account for how concurrent operations interact with them while reasoning about the local execution of any given CPU.

*Example 1 (Simple page table).* We illustrate this problem using a page table example. On modern computers with hardware virtualization support, the hypervisor maintains a nested page table (NPT) [30] for each VM, to manage the VM's access to physical memory. NPTs translate guest physical memory addresses (gPAs) to host physical memory addresses (hPAs), mapping each guest frame number (gfn) to a physical frame number (pfn).

In our example, the NPT of a VM is allocated from its own page table pool. The pool consists of page table entries, whose implementation is encapsulated by a lower layer interface that exposes functions `pt_load(vmid, ofs)` to read the value at offset `ofs` from the page table pool of VM `vmid` and `pt_store(vmid, ofs, val)` to write the value `val` at offset `ofs`. To keep the example simple, we use a simplified version of the real NPT verified in MicroV, ignore dynamic allocation and permission bits, and assume two-level paging denoted with `pgd` and `pte`. Consider the following two implementations, which map `gfn` to `pfn` in VM `vmid`'s NPT:

```
void set_npt(uint vmid, uint gfn, uint pfn) {
  acq_lock_npt(vmid);
  // load the pte base address
  uint pte = pt_load(vmid, pgd_offset(gfn));
  pt_store(vmid, pte_offset(pte,gfn), pfn);
  rel_lock_npt(vmid);
}
void set_npt_insecure(uint vmid, uint gfn, uint pfn) {
  acq_lock_npt(vmid);
  uint pte = pt_load(vmid, pgd_offset(gfn));
  pt_store(vmid, pte_offset(pte,gfn), pfn+1); // BUG
  pt_store(vmid, pte_offset(pte,gfn), pfn);
  rel_lock_npt(vmid);
}
```

Since an NPT just maintains a mapping from gfns to pfns, we can specify the NPT as a logical map $\texttt{gfn} \mapsto \texttt{pfn}$, then prove that a correct NPT implementation refines its specification. However, among the two implementations, only `set_npt` is correct, while `set_npt_insecure` is not. A sound refinement proof should only admit `set_npt`, while rejecting `set_npt_insecure` for its extraneous intermediate mapping from `gfn` to `pfn+1`.

Figure 2 illustrates the vulnerability of `set_npt_insecure`. The problem arises because a multiprocessor VM supports shared memory among its virtual CPUs (VCPUs). This requires its NPT to also be shared among its VCPUs, potentially running on different physical CPUs. Even though the NPT is protected by software spinlocks, the hardware memory management unit (MMU) will still perform page translations during `set_npt_insecure`'s critical section. When the hypervisor runs `set_npt_insecure` to map a physical page used by VM $m$ to VM $n$'s NPT, VM $m$'s secrets can be leaked to VM $n$ accessing the page on another physical CPU.

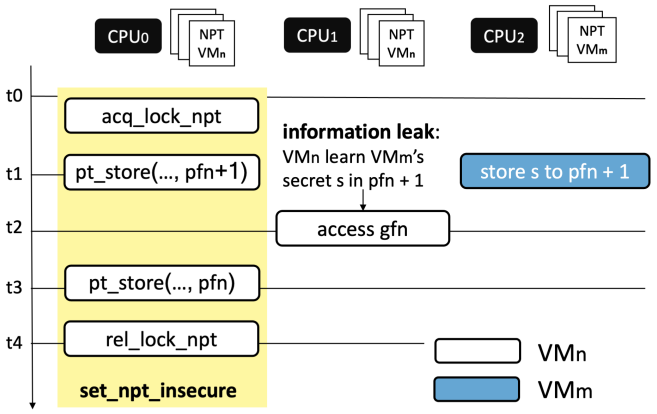



**Fig. 2: Insecure page table updates.** VM $n$ runs on CPUs 0 and 1, while VM $m$ runs on CPU 2. Physical page `pfn` is free, but `pfn+1` is mapped to VM $m$'s NPT and contains its private data, so only VM $m$ should have access to it. At time $t_0$, KCore handles VM $n$'s page fault and invokes `set_npt_insecure(n, gfn, pfn)` on CPU 0 to map guest page `gfn` in VM $n$'s NPT. At time $t_2$, `gfn` is transiently but erroneously mapped to `pfn+1` until $t_4$, allowing VM $n$ on CPU 1 to concurrently access VM $m$'s private data using this temporary mapping.

Previous refinement techniques [5], [28], [31] would incorrectly deem `set_npt` and `set_npt_insecure` functionally equivalent, failing to detect this vulnerability. For example, although CertiKOS proves that its own software is data-race free (DRF), it does not support, nor model, the MMU hardware feature allowing an untrusted principal to concurrently access a shared page table, so the above two implementations would erroneously satisfy the same specification.

To address this problem, MicroV introduces *transparent trace refinement*, which forbids hidden information flow in a multiprocessor setting. To explain this technique, we first describe our multiprocessor machine model, and how shared objects are modeled using event traces. We then describe how transparent trace refinement can be used to enable sequential reasoning for a write data-race-free system, where shared objects are protected by write-locks, but reads on shared objects (which may lead to information leakage) may occur at any time.

*1) Multiprocessor model:* We define an abstract multiprocessor machine model, whose machine state $\sigma$ 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. $\sigma$ does not explicitly model shared objects. 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. For example, the page table pool used by our NPT implementation is accessible from KCore running on each CPU via the `pt_store(vmid, ofs, val)` primitive, which generates the event (`P_ST vmid ofs val`). Similarly, the NPT itself is a shared object, so the `set_npt(vmid, gfn, pfn)` primitive defined in our layer specification generates the event (`SET_NPT vmid gfn pfn`).

Our abstract machine is formalized as a transition system, where each step models some atomic computation taking place on a single CPU; concurrency is realized by the nondeterministic interleaving of steps across all CPUs [32]. To simplify reasoning about all possible interleavings, we lift multiprocessor execution to a *CPU-local* model, which distinguishes execution taking place on a particular CPU from its concurrent environment [5].

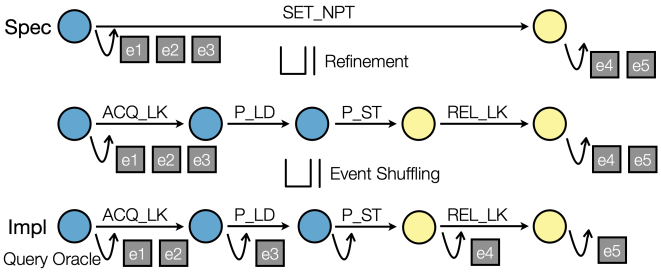



**Fig. 3: Querying the event oracle to refine `set_npt`.** The bottom trace shows events produced by `set_npt`'s implementation as it interacts with the shared lock and page table pool it uses. The query move before `ACQ_LK` yields all events from other CPUs prior to `ACQ_LK`; the query move before `P_LD` yields all events from other CPUs since the last query up until `P_LD`. The middle trace shows how we would like to shuffle events in the bottom trace to match those in the top trace.

All effects coming from the environment are encapsulated by and conveyed through an *event oracle*, which yields events emitted by other CPUs when queried. How the event oracle synchronizes these events is left abstract, its behavior constrained only by rely-guarantee conditions [33]. Since the interleaving of events is left abstract, our proofs do not rely on any particular interleaving of events and therefore hold for all possible concurrent interleavings. A CPU captures the effects of its concurrent environment by querying the event oracle, a *query move*, before its own CPU step, a *CPU-local move*. A CPU only needs to query the event oracle before interacting with shared objects, since its private state is not affected by these events. Figure 3 illustrates query and CPU-local moves in the context of the event trace produced by `set_npt`'s implementation to refine its specification. The end result of its execution is a composite event trace of the events from other CPUs, interleaved with the events from the local CPU.

Interleaving query and CPU-local moves still complicates reasoning about `set_npt`'s implementation. However, if we can guarantee that events from other CPUs do not interfere with the shared objects used by `set_npt`, we can safely *shuffle* events from other CPUs to the beginning or end of its critical section. For example, if we could prove that `set_npt`'s implementation is DRF, then other CPUs will not produce events within `set_npt`'s critical section that interact with the locked NPT. We would then only need to make a query move before the critical section, not within the critical section, allowing us to sequentially reason about `set_npt`'s critical section as an atomic operation.

Unfortunately, as shown by `set_npt_insecure`, even if `set_npt` correctly uses locks to prevent concurrent NPT accesses within KCore's own code, it is not DRF because KServ or VMs executing on other CPUs may indirectly read the contents of their NPTs through the MMU hardware. This prevents us from soundly shuffling event queries outside of the critical section and employing sequential reasoning to refine the critical section to an atomic step. If `set_npt` cannot be treated as an atomic primitive, sequential reasoning would then be problematic to use for any layer that uses `set_npt`, making their refinement difficult. Without sequential reasoning, verifying a large system like KCore is infeasible.

*2) Transparent trace refinement:* We observe that information leakage can be modeled by *read events* that occur arbitrarily

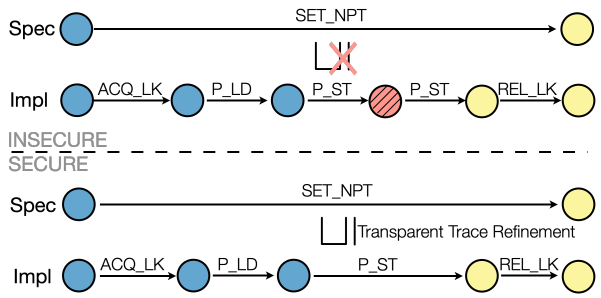



**Fig. 4: Transparent trace refinement of insecure and secure `set_npt` implementations.** Each node represents an event observation. Nodes of the same color constitute an event observer group. The insecure example does not satisfy the transparency condition because there is an intermediate observation (shown in red) that cannot map to any group in the specification.

throughout critical sections, without regard for locks. To ensure that refinement does not hide this information leakage, transparent trace refinement treats read and write events separately. We view shared objects as *write data-race-free* (WDRF) objects—shared objects with unavoidable concurrent observers. For these objects, we treat their locks as *write-locks*, meaning that query moves that yield write events may be safely shuffled to the beginning of the critical section. Query moves in the critical section may then only yield read events from those concurrent readers.

To determine when read events may also be safely shuffled, each WDRF object must define an *event observer* function, which designates what concurrent CPUs may observe: they take the current machine state as input, and produce some observed result, with consecutive identical event observations constituting an *event observer group*. Event observer groups thus represent all possible intermediate observations by concurrent readers. Since the event observations are the same in an event observer group, read events from other CPUs will read the same values anywhere in the group and can be safely shuffled to the beginning, or end, of an event observer group, reducing the verification effort of dealing with interleavings. Our security-preserving layers enforce that any refinement of WDRF objects must satisfy the following condition:

**Definition 2 (Transparency condition).** *The list of event observer groups of an implementation must be a sublist of that generated by its specification. That is, the implementation reveals at most as much information as its specification.*

This condition ensures that the possibility of concurrent readers and information release is preserved through each layer refinement proof. In particular, if a critical section has at most two distinct event observer groups, read events can be safely shuffled to the beginning or end of the critical section. Query moves are no longer needed during the critical section, but can be made before or after the critical section for both read and write events, making it possible to employ sequential reasoning to refine the critical section. Transparent trace refinement can thereby guarantee that events from other CPUs do not interfere with shared objects in critical sections. Figure 4 illustrates how this technique fares against our earlier counterexample, as well as to our original, secure implementation. `set_npt_insecure` has three event observer groups that can observe three different values, before the first `pt_store`, between the first and second `pt_store`, and after the second `pt_store`. Read events after the first `pt_store` cannot be shuffled before the critical section. On the other hand, `set_npt` has only two event observer groups, one that observes the value before `pt_store`, and one that observes the value after `pt_store`, so query moves are not needed during the critical section. The implementation can therefore be refined to an atomic `set_npt` specification. Refinement proofs for higher layers that use `set_npt` can then treat `set_npt` as an atomic primitive, simplifying those proofs since `set_npt` can be viewed as just one atomic computation step instead of many CPU-local moves with intervening query moves.

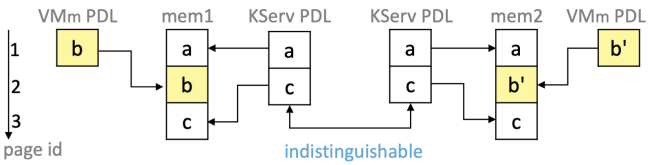
### B. Noninterference Assertions

Since transparent trace refinement ensures that KCore's top-level specification hides only its implementation details, but not any potential information flow, we can now soundly prove its security properties using its specification. We express the security properties as noninterference assertions, and want to show that one principal cannot affect the private data of another, ensuring VM confidentiality and integrity. For each principal, we define a *private data lens* (PDL), denoted by $\mathcal{V}$, which returns the subset of machine state $\sigma$ that is private to the principal. For example, the private data of VM $p$, denoted as $\mathcal{V}(\sigma, p) \subseteq \sigma$, includes the contents of its CPU registers and memory. A principal should not be able to infer the state in any other principal's PDL, and its own PDL should be unaffected by other principals. Such an *isolation property* can be proven using noninterference by showing state indistinguishability:

**Definition 3 (State indistinguishability).** *Two states $\sigma_1$ and $\sigma_2$ are indistinguishable from the perspective of principal $p$ if and only if $\mathcal{V}(\sigma_1, p) = \mathcal{V}(\sigma_2, p)$.*

In other words, a pair of distinct machine states are indistinguishable to some principal $p$ if the differences fall beyond the scope of $p$'s PDL. We want to prove that, starting from any two states indistinguishable to a principal $p$, the abstract machine should only transition to a pair of states that are still indistinguishable to $p$. Such transitions are said to preserve state indistinguishability.

*Example 2 (Proving VM confidentiality).* Consider KServ and a VM $m$, where VM $m$ has only gfn 1 in its address space, mapped to pfn 2. We prove VM $m$'s data confidentiality by showing that any change in VM $m$'s private data is not visible to KServ during execution. Suppose VM $m$ writes content $b$ to its gfn 1 in one execution (leading to state $\sigma$), and writes content $b'$ in an alternate execution (leading to state $\sigma'$); we must show that these executions are indistinguishable to KServ's PDL:




To keep our example simple, we use a simplified $\mathcal{V}$ consisting of only the contents stored in a principal's guest page frames. For instance, in $\sigma$, after VM $m$ writes $b$ to gfn 1, $\mathcal{V}(\sigma,\ m)$ is the partial map $\{1 \mapsto b\}$. Yet in both executions, whether VM $m$ writes $b$ or $b'$, KServ's PDL to the two states are identical: $\mathcal{V}(\sigma, \text{KServ}) = \mathcal{V}(\sigma', \text{KServ}) = \{1 \mapsto a, 2 \mapsto c\}$. This means that the two states are *indistinguishable* to KServ—it cannot observe VM $m$'s update to pfn 2.

Although previous work used noninterference assertions to verify information-flow security [34], [35], [36], they do not address two key issues that we solve in MicroV, concurrency and intentional information flow.

*1) Concurrency:* We extend previous work on noninterference in a sequential setting [35] to our multiprocessor specification. We prove noninterference for *big steps*, meaning that some principal always steps from an active state to its next active state, without knowledge of concurrent principals. We say a state is *active* if we are considering indistinguishability with respect to some principal $p$'s PDL, and $p$ will make the next step on CPU $c$; otherwise the state is *inactive*. We decompose each big step noninterference proof into proofs of a set of auxiliary lemmas for a given principal $p$:

**Lemma 1.** *Starting from any two active, indistinguishable states $\sigma_1$ and $\sigma_1'$, i.e., $\mathcal{V}(\sigma_1, p) = \mathcal{V}(\sigma_1', p)$, if $p$ makes CPU-local moves to states $\sigma_2$ and $\sigma_2'$, then $\mathcal{V}(\sigma_2, p) = \mathcal{V}(\sigma_2', p)$.*

**Lemma 2.** *Starting from any inactive state $\sigma$, if some other principal makes a CPU-local move to inactive state $\sigma'$, then $\mathcal{V}(\sigma, p) = \mathcal{V}(\sigma', p)$.*

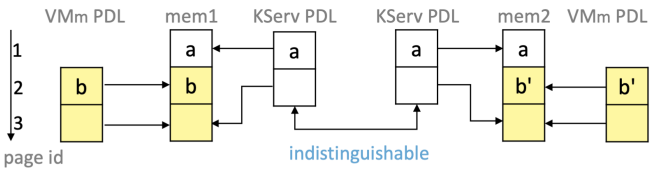
**Lemma 3.** *Starting from any two inactive, indistinguishable states $\sigma_1$ and $\sigma_1'$, i.e., $\mathcal{V}(\sigma_1, p) = \mathcal{V}(\sigma_1', p)$, if some other principal makes CPU-local moves to active states $\sigma_2$ and $\sigma_2'$, respectively, then $\mathcal{V}(\sigma_2, p) = \mathcal{V}(\sigma_2', p)$.*

**Lemma 4.** *Starting from any two indistinguishable states $\sigma_1$ and $\sigma_1'$, i.e., $\mathcal{V}(\sigma_1, p) = \mathcal{V}(\sigma_1', p)$, if query moves result in states $\sigma_2$ and $\sigma_2'$, respectively, then $\mathcal{V}(\sigma_2, p) = \mathcal{V}(\sigma_2', p)$.*

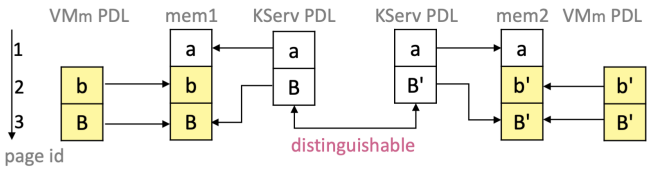
In other words, we prove in the first three lemmas that state indistinguishability is preserved in each possible step of execution due to CPU-local moves, then based on rely-guarantee reasoning [33], show in the last lemma that state indistinguishability must also be preserved due to query moves.

*2) Intentional information release:* Because commodity systems such as KVM may allow information release when explicitly requested, to support KVM's various virtualization features, we must model this intentional information release and distinguish it from unintentional data leakage. We call data that is intentionally released *non-private data*.

*Example 3 (Supporting declassification).* To illustrate the challenge of supporting intentional information release, suppose VM $m$ grants KServ access to gfn 3 for performing network I/O. Since KServ may copy gfn 3's data to its private memory, gfn 3's data should be included in KServ's private data lens. Private pages of VM $m$ (e.g., gfn 2) are handled the same as the previous example—the content of gfn 2, whether $b$ or $b'$, are not included in KServ's PDL, and do not affect state indistinguishability:

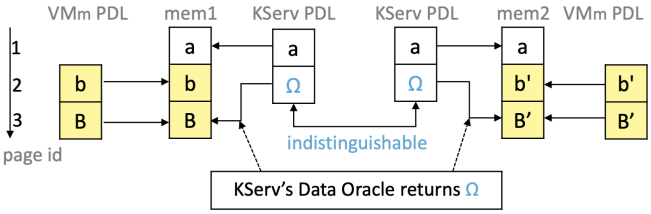



VM $m$ may encrypt the data in gfn 2 and share it with KServ through gfn 3, so that KServ can send the encrypted data via a backend paravirtual network I/O driver. Yet starting from two indistinguishable states to KServ with different gfn 2 contents $b$ and $b'$, writing encrypted data $B$ and $B'$ to gfn 3 leads to distinguishable states, since the differences between $B$ and $B'$ are exposed to KServ's PDL:




The problem is that this information release does not preserve state indistinguishability, even though it does not break confidentiality since KServ cannot decrypt the private data. In general, what is declassified is not known in advance; any page of a VM's memory may be declassified at any time, and the data may be declassified for an arbitrary amount of time.

To address this problem, MicroV introduces *data oracles* to model intentionally released information, such as encrypted data. Instead of using the actual value of such information, the value is specified by querying the data oracle. A data oracle is a deterministic function that generates a sequence of integers based on some machine state. We use a function based on a principal's PDL, guaranteeing that the value returned depends only on the state of the principal reading the value. Each principal logically has its own data oracle, ensuring that the returned value does not depend on the behavior of other principals. For example, if the encrypted data is masked using a data oracle and the next integer returned by KServ's data oracle is $\Omega$, the value of the next shared, encrypted data read by KServ will always be specified as $\Omega$, whether the encrypted data is $B$ or $B'$. This way, intentional information release can be masked by data oracles and will not break state indistinguishability:




Data oracles are only used to mask reading non-private data, decoupling the data read from the writer of the data. Integer sequences returned by data oracles are purely logical, and can yield any value; thus, our noninterference proofs account for all possible values read from a data oracle. A data oracle is applied dynamically, for example, masking a page of memory when it is declassified, then no longer masking the page when it is used for private data. Note that unintentional information leakage, such as sharing unencrypted private data, is not modeled by data oracles and will be detected since it will break state indistinguishability.

## IV. Retrofitting KVM into SeKVM

To use microverification, we observe that many parts of a hypervisor have little to do with security. If we can extract the core components that enforce security and isolate them from the rest of the system, we may focus our proof effort on the core while obtaining security guarantees for the entire system. Based on this observation, we retrofit KVM/ARM into a small core, KCore, and a set of untrusted services, KServ, according to the following desired security policies:

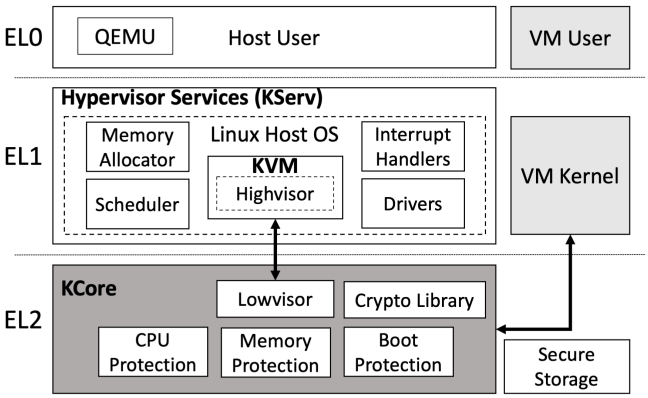



**Fig. 5:** Architecture of SeKVM.

**Policy 1.** ***vmdataiso:***

- ***vm-cpuiso****: a given VM's data in CPU registers is isolated from KServ and all other VMs.*
- ***vm-memiso:*** *a given VM's data in private memory is isolated from KServ and all other VMs.*
- ***vm-memdeviso:*** *a given VM's data in private memory is isolated from all devices assigned to KServ and all other VMs.*

**Policy 2.** ***vm-iodataiso****: the confidentiality and integrity of a given VM's I/O data is protected from KServ and all other VMs.*

**Policy 3.** ***data-declassification****: a given VM's non-private data may be intentionally released to support virtualization features.*

Figure 5 shows the retrofitted KVM, SeKVM. KCore runs in Arm VE's higher-privilege hypervisor level, EL2. It implements the functions that enforce access control or need direct access to VM data, as well as the VM interface, which can logically be thought of as a set of trap handlers. KServ runs at a lower-privilege kernel level, EL1. It includes the Linux host OS and most of the KVM codebase. Just like vanilla KVM, VMs run user-level code in EL0, the lowest privilege level, and kernel code in EL1. Further details are described in [7].

When a VM raises an exception, it traps to KCore, which can handle the exception directly or invoke KServ for more complex functionality. Under the **data-declassification** policy, KServ provides functionality that does not require access to private VM data, such as resource allocation, bootstrapping and scheduling VMs, and management features like snapshots and migration. KCore isolates KServ's access to VM resources to protect data confidentiality and integrity. KServ does not have the privilege to update VM data directly. Instead, KCore provides a restricted hypercall interface to KServ for operations which require VM data or EL2 privileges. The interface allows KServ to write updated data to a memory page shared with KCore, which validates the update before applying it to the real VM data. KCore also validates cryptographic signatures of VM images and encrypts VM data before it is exported for snapshots or migration.

*Policy vm-cpuiso*. When a VM is running on a CPU, its VCPU registers are stored in CPU registers only accessible to the VM. When a VM exits, and before running KServ, KCore saves the VM's VCPU registers in KCore's private memory, inaccessible to KServ or other VMs, and context switches the CPU so the VM's VCPU state is no longer available in the CPU registers. KCore may share non-private data from VM general purpose registers to KServ in accordance with the **data-declassification** policy.

*Policy vm-memiso*. KCore ensures that a VM's private memory cannot be accessed by KServ or other VMs. In its own private memory, KCore tracks the ownership and sharing status of each page. KCore ensures that any allocated page is owned by either itself, KServ, or a VM. KCore manages stage 2 page tables, Arm's NPTs, for KServ and VMs to virtualize and restrict their memory access, which in turn manage their own stage 1 page tables. By default, KServ owns unused physical pages. KCore delegates complex VM memory allocation to KServ, but validates its page allocation proposals before performing the mapping itself, to ensure that the page is not already owned by another VM or itself. KCore unmaps the newly allocated page from KServ's stage 2 page tables before mapping the page to the VM. When a page is freed, KCore zeroes out that page before allowing KServ to reallocate it, ensuring that reclaimed memory does not leak VM data.

*Policy vm-memdeviso*. KCore leverages the System Memory Management Unit (SMMU) [37], 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. By taking full control of the SMMU, KCore ensures 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.

*Policy vm-iodataiso*. To avoid additional complexity in KCore, we rely on KServ to support I/O virtualization. Similar to previous work [7], [38], [39], we assume VMs employ end-to-end encryption to protect I/O data against KServ.

*Policy data-declassification*. By default, KServ has no access to VM data. KCore provides `grant` and `revoke` hypercalls that a VM may use to voluntarily grant and revoke KServ access to the VM's pages. KCore also allows intentional information flow via a VM's general purpose registers (GPRs) to KServ to support MMIO.

## V. Verifying SeKVM

We verify SeKVM by decomposing the KCore codebase into 34 security-preserving layers. From KCore's modular implementation, we craft its layer specification based on four 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. Fourth, we introduce layers to enforce invariants, which are used to prove high-level properties. Introducing layers modularizes verification, reducing proof effort and maintenance.

| **Hypercall Handlers:** |
|---|
| *VM management*: `register_vm`, `register_vcpu`, `set_boot_info`, `remap_boot_image_page`, `verify_vm_image`, `clear_vm`, `encrypt_vcpu`, `decrypt_vcpu`, `encrypt_vm_mem`, `decrypt_vm_mem` |
| *Timer*: `set_timer` |
| *SMMU*: `smmu_alloc_unit`, `smmu_free_unit`, `smmu_map`, `smmu_unmap`, `smmu_iova_to_phys` |
| *VM run*: `run_vcpu` |
| *VM-only*: `grant`, `revoke`, `psci_power` |
| **Exception Handlers:** |
| *Page Fault:* `host_page_fault`, `vm_page_fault` |
| *Interrupts:* `handle_irq` |
| *WFI/WFEs:* `handle_wfx` |
| *SysReg Access:* `handle_sysreg` |
| **Memory Operations:** |
| `mem_load`, `mem_store`, `dev_load`, `dev_store` |

**TABLE I: `TrapHandler` interface.** KServ calls VM management hypercalls to boot and terminate VMs and obtain encrypted VM data for migration and snapshots, the timer hypercall to set timers, SMMU hypercalls to configure the SMMU, and the `run_vcpu` hypercall to run a VCPU. VMs call VM-only hypercalls, not available to KServ, for power management and to grant and revoke KServ access to VM (encrypted) data for full KVM I/O support. Exception handlers handle stage 2 page faults and other VM exceptions. Memory operations are not part of the interface for KCore's implementation, but are included in its specification to logically model page-translated memory accesses issued by KServ and VMs over our abstract hardware model.

We incrementally prove that each module of KCore's implementation transparently refines its layer specification, starting from the bottom machine model until reaching the top layer, then prove noninterference using the top layer specification. The bottom machine model, `AbsMachine`, defines the machine state, an instruction set, and a set of trusted primitives, such as cryptographic library routines. The top layer specification, `TrapHandler`, is the interface KCore exposes to its VMs and KServ. We prove noninterference for *any behavior* of KServ and VMs interacting with the `TrapHandler` interface, so that proven security properties hold for the whole SeKVM system with any implementation of KServ.

### A. Proving KCore Refines `TrapHandler`

Table I presents hypercalls and exception handlers provided by the top-level interface `TrapHandler`; Appendix A has more details. All proofs are developed and checked in Coq. We briefly describe some of the refinement proofs, but omit details due to space constraints.

*1) Synchronization:* KCore's spinlocks use an Arm assembly lock implementation from Linux. This implementation uses acquire and release barriers to prevent instruction reordering within the lock routines, such that we can soundly verify that spinlocks correctly enforce mutual exclusion in a similar manner to CertiKOS [5]. We prove that all shared memory accesses in the code are correctly protected by spinlocks. Because the barriers also prevent memory accesses from being reordered beyond their critical sections, we can easily show that KCore only exhibits sequentially consistent behavior, such that our guarantees over KCore verified using a sequentially consistent model still hold on Arm's relaxed memory model. The lock proofs required 4 layers.

*2) VM management:* KCore tracks the lifecycle of each VM from boot until termination, maintaining per-VM metadata in a `VMInfo` data structure. For example, KCore loads VM boot images into protected memory and validates signed boot images using an implementation of Ed25519. The implementation is verified by porting the HACL* [14] verified crypto library to EL2. Because no C standard library exists for EL2, this involved replacing the C library function for `memcpy` used by HACL* with a standalone `memcpy` implementation for EL2, which we verified. Similarly, KCore provides crypto hypercalls using an implementation of AES, also verified by using HACL*. Beyond HACL*, verifying VM management required 4 layers.

*3) VM CPU data protection:* KCore saves VM CPU registers to KCore memory in a `VCPUContext` data structure when a VM exits, and restores them to run the VM. KCore does not make VM CPU data directly available to KServ, and validates any updates from KServ before writing them to the VM CPU data it maintains in KCore memory. Many of the functions for saving, restoring, and updating VM CPU data involving looping over registers to initialize, copy, and process data. We verify these functions by first showing that the loop body refines an atomic step using transparent trace refinement, then use induction to prove that arbitrary many iterations of the loop refine an atomic step. Finally, we use induction to prove the loop condition is monotonic so that it will eventually terminate. Verifying protection of the VM CPU data required 3 layers.

*4) Multi-level paging with huge page support:* KCore manages its own stage 1 page table and a stage 2 page table per principal. KCore assigns a dedicated page pool for allocating stage-2 page tables to each principal, each capped with a page count quota, ensuring each principal's allocation cannot interfere with others'. Like KVM, KCore's page tables support multiprocessor VMs, 3- and 4-level multi-level paging with dynamically allocated levels, and huge pages to optimize paging performance; normal pages are 4KB and huge pages are 2MB.

Because page tables can be shared across multiple CPUs, a common feature of multiprocessor hypervisors, we use transparent trace refinement to prove that their implementation refines their specification (a logical map). Transparent trace refinement was essential for verifying other layers that use stage 2 page tables, simplifying their refinement by allowing the proofs to treat the primitives of the stage 2 page table map specification as atomic computation steps. Appendix B has more details. Using layers allowed most of the refinement proofs to be written with the page table implementation details abstracted away, making it possible to prove, for the first time, the functional correctness of a system that supports shared page tables.

To prove that the multi-level page table implementation refines the abstract map, we first show that the page table is a tree structure with page table entries stored in the leaves, thereby guaranteeing that multiple gfns cannot share the same page table entries. We then prove the tree is equivalent to storing page table entries in an array indexed by gfn. To verify huge page support, our proof additionally involves considering four possible invariant combinations: (1) changing a regular page mapping does not affect any regular page mappings; (2) changing a regular page mapping does not affect any huge page mappings; (3) changing a huge page mapping does not affect any regular page mappings; and (4) changing a huge page mapping does not affect any huge page mappings. The basic idea behind the proof reduces the problem of the multiple page sizes to dealing with just the one 4KB page size by treating the 2MB huge page as the equivalent of 512 4KB pages. Overall, the page table refinement proofs required 4 layers.

*5) Memory protection:* KCore tracks metadata for each physical

page, including its ownership and sharing status, in an `S2Page` data structure, similar to the `page` structure in Linux. KCore maintains a global `S2Page` array for all valid physical memory to translate from a pfn to an `S2Page`. This can be specified as an abstract `pfn` $\mapsto$ (`owner`, `share`, `gfn`) mapping, where `owner` designates the owner of the physical page, which can be KCore, KServ, or a VM, and `share` records whether a page is intentionally shared between KServ and a VM.

Many of the functions involved in memory protection to virtualize memory and enforce memory isolation involve nested locks. Transparent trace refinement enables us to prove that the implementation with nested locks refines an atomic step, by starting with the code protected by the inner most nested locks and proving that it refines an atomic step, then successively proving this while moving out to the next level of nested locks until it is proven for all nested locks. The refinement proofs for memory protection required 5 layers, including 3 layers for the `S2Page` specification.

*6) SMMU:* KCore manages the SMMU and its page tables to provide DMA protection. We changed KVM's SMMU page table walk implementation to an iterative one to eliminate recursion, which is not supported by our layered verification approach, since each layer can only call functions in a lower layer. This is not a significant issue for hypervisor or kernel code, in which recursion is generally avoided due to limited kernel stack space. As with KVM, SMMU page tables for a device assigned to a VM do not change after the VM boots, but can be updated before the VM boots. We leverage transparent trace refinement to prove the page table implementation refines its specification to account for pre-VM boot updates. These proofs required 9 layers: 4 for SMMU page tables, and 5 for SMMU hypercalls and handling SMMU accesses.

*7) Hypercalls:* On top of all above modules, KCore implements hypercall dispatchers and trap handlers with 5 layers. As shown in Table I, the top layer specification refined by KCore provides 20 hypercalls and 5 fault handlers. It also exposes basic memory operations like `mem_load` and `mem_store` passed through from the bottom layer, to model the memory access behavior of KServ and VMs.

### B. Formulating Noninterference over `TrapHandler`

Because of security-preserving layers, noninterference only needs to be formulated and proven over `TrapHandler`, the top-level specification of KCore. We define the PDL $\mathcal{V}$ for each principal based on policy **vmdataiso** and **vm-iodataiso**. $\mathcal{V}(\sigma, p)$ on a given CPU $c$ contains all of its private data, including (1) CPU $c$'s registers if $p$ is active on $c$, (2) $p$'s saved execution (register) context, and (3) contents of the memory pages owned by $p$, which we call the address space of $p$, including those mapped to $p$'s stage 2 and SMMU page tables. $\mathcal{V}$ for a VM also contains metadata which can affect its private data, including the sharing status of its own memory pages. $\mathcal{V}$ for KServ also contains metadata, including ownership metadata of all memory pages, VM execution metadata, and SMMU configuration. We then use per-principal data oracles to model the only three types of non-private data that a VM may intentionally release according to policy **data-declassification**: (1) data that VMs explicitly share and retain using the `grant` and `revoke` hypercalls, (2) MMIO data that KServ copies from/into VM GPRs, used for hardware device emulation, and (3) VCPU power state requests that VMs explicitly make using the `psci_power`

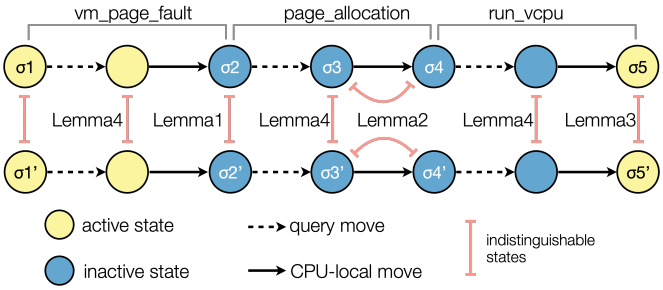



**Fig. 6: Noninterference proof for a VM NPT page fault.** Upon a page fault, VM $p$ calls `vm_page_fault` in two indistinguishable states ($\sigma_1$ and $\sigma_1'$) and traps to KServ, transitioning from active to inactive in both executions. By Lemmas 4 and 1, $\sigma_2$ and $\sigma_2'$ are indistinguishable. As KServ performs page allocation, the executions respectively reach states $\sigma_3$ and $\sigma_3'$ through query moves, then $\sigma_4$ and $\sigma_4'$ through CPU-local moves. By Lemma 4, $\sigma_3$ and $\sigma_3'$ are indistinguishable. By Lemma 2, since VM $p$, and all other VMs, remain inactive in states $\sigma_3$ and $\sigma_3'$, (1) $\sigma_3$ and $\sigma_4$ are indistinguishable in the first execution, and (2) $\sigma_3'$ and $\sigma_4'$ are indistinguishable in the second; transitively, $\sigma_4$ and $\sigma_4'$ are also indistinguishable. Finally, KServ invokes `run_vcpu`, transitioning VM $p$ from inactive to active; the executions reach states $\sigma_5$ and $\sigma_5'$, respectively. By Lemmas 4 and 3, $\sigma_5$ and $\sigma_5'$ are indistinguishable.

hypercall to control their own power management [40]. We also use data oracles to model data intentionally released by KServ when providing its functionality: (1) physical page indices proposed by KServ for page allocation, written to the page table of a faulting VM, (2) contents of pages proposed by KServ for page allocation, and (3) scheduling decisions made by KServ. Intuitively, if KServ has no private VM data, any release of information by KServ cannot contain such data and can therefore be modeled using data oracles.

We prove noninterference by proving the lemmas from Section III-B, for the primitives shown in Table I, with respect to the PDL for each principal. Data oracles are used for proving noninterference lemmas for the `grant` and `revoke` hypercalls, for `vm_page_fault` when it is called due to MMIO accesses, and for hypercalls such as `run_vcpu` which may declassify data from KServ to a VM. Our proofs not only verify that the execution of a principal does not interfere with another's data, but that one principal does not interfere with another's metadata. For instance, we proved that a VM cannot interfere with KServ's decision about VM scheduling, or affect the memory sharing status of another VM. Furthermore, since we prove that there is no unintentional release of private VM data to KServ, we also prove that any release of information by KServ cannot contain such data.

For example, Figure 6 shows how we prove noninterference for the big step execution of a stage 2 page fault caused by the need to allocate a physical page. VM $p$ causes KCore's `vm_page_fault` trap handler to run, KCore then switches execution to KServ to perform page allocation, then KServ calls the `run_vcpu` hypercall to cause KCore to switch execution back to the VM with the newly allocated page. We want to use the lemmas in Section III-B, but we must first prove them for each primitive. We briefly describe the proofs, but omit details due to space constraints.

We prove Lemmas 1 to 3 for `vm_page_fault`, which saves VM $p$'s execution context and switches execution to KServ. For Lemma 1, an active principal must be the VM $p$. Starting from two indistinguishable states for VM $p$, the local CPU's registers must be for VM $p$ since it is active and running on the CPU, and must be the

same in two executions. vm_page_fault in two executions will therefore save the same execution context for VM $p$; so Lemma 1 holds. For Lemma 2, an inactive principal $q$ must be a VM other than $p$. $q$'s PDL will not be changed by VM $p$'s page fault, which only modifies VM $p$'s execution context and the local CPU's registers; so Lemma 2 holds. For Lemma 3, only KServ can become active after executing vm_page_fault. Its PDL will then include the local CPU's registers after the execution. Since KServ's saved execution context must be the same in indistinguishable states, the restored registers will then remain the same; so Lemma 3 holds.

We prove Lemmas 1 to 3 for KServ's page allocation, which involves KServ doing mem_load and mem_store operations in its address space, assuming no KServ stage 2 page faults for brevity. For Lemma 1, KServ is the only principal that can be active and executes the CPU-local move, which consists of KServ determining what page to allocate to VM $p$. Starting from two indistinguishable states for KServ, the same set of memory operations within KServ's address space will be conducted and the same page index will be proposed in two executions, so Lemma 1 holds. For Lemma 2, all VMs are inactive. We prove an invariant for page tables stating that any page mapped by a principal's stage 2 page table must be owned by itself. Since each page has at most one owner, page tables, and address spaces, are isolated. With this invariant, we prove that VMs' states are not changed by KServ's operations on KServ's own address space; so Lemma 2 holds. Lemma 3 does not apply in this case since KServ's operations will not make any VM active.

We prove Lemmas 1 to 3 for run_vcpu. For Lemma 1, KServ is the only principal that can invoke run_vcpu as a CPU-local move, which consists of KCore unmapping the allocated page from KServ's stage 2 page table, mapping the allocated page to VM $p$'s stage 2 page table, and saving KServ's execution context so it can switch execution back to VM $p$. Starting from two indistinguishable states for KServ, run_vcpu in two executions will transfer the same page from KServ's page table to VM $p$'s page table. KServ's resulting address spaces remain indistinguishable as the same page will be unmapped from both address spaces. For Lemma 2, If a principal $q$ stays inactive during the run_vcpu CPU-local move, it must not be VM $p$. By the page table isolation invariant, the transferred page cannot be owned by $q$ since it is initially owned by KServ, and such a transfer will not affect $q$'s address space. For Lemma 3, VM $p$ is the only inactive principal that becomes active during the run_vcpu CPU-local move. Thus, the page will be transferred to $p$'s address space. Starting from two indistinguishable states for $p$, the page content will be masked with the same data oracle query results. $p$'s resulting address spaces remain the same and indistinguishable to $p$, so Lemma 3 holds.

By proving Lemmas 1 to 3 for all primitives, Lemma 4 holds for each primitive based on rely-guarantee reasoning. We can then use the proven lemmas to complete the indistinguishability proof for the big step execution of the stage 2 page fault, as shown in Figure 6.

As another example, a similar proof is done to show noninterference for MMIO accesses by VMs. Addresses for MMIO devices are unmapped in all VMs' stage 2 page tables. VM $p$ traps to KCore when executing a MMIO read or write instruction, invoking vm_page_fault. KCore switches to KServ on the same CPU to handle the MMIO access. On an MMIO write, KCore copies the write data from VM $p$'s GPR to KServ so it can program the virtual hardware, which we model using KServ's data oracle in proving Lemma 3 for vm_page_fault. Lemmas 4 and 1 are then used to show that indistinguishability is preserved between two different executions of VM $p$. Lemmas 4 and 2 are used to show indistinguishability as KServ runs. Finally, KServ calls run_vcpu. On an MMIO read, KServ passes the read data to KCore so KCore can copy to VM $p$'s GPR, which we model using VM $p$'s data oracle in proving Lemma 3 for run_vcpu. Lemmas 4 and 3 are then used to show that indistinguishability holds as KCore switches back to running the VM.

### C. Verified Security Guarantees of SeKVM

Our noninterference proofs over TrapHandler guarantee that KCore protects the confidentiality and integrity of VM data against both KServ and other VMs, and therefore hold for all of SeKVM.

*Confidentiality*. We show that the private data of VM $p$ cannot be leaked to an adversary. This is shown by noninterference proofs that any big step executed by VM $p$ cannot break the state indistinguishability defined using the PDL of KServ or any other VM. There is no unintentional information flow from VM $p$'s private data to KServ or other VMs.

*Integrity*. We show that the private data of VM $p$ cannot be modified by an adversary. This is shown by noninterference proofs that any big step executed by KServ or other VMs cannot break the state indistinguishability defined using the PDL of VM $p$. VM $p$'s private data cannot be influenced by KServ or other VMs.

In other words, SeKVM protects VM data confidentiality and integrity because we prove that KCore has no vulnerabilities that can be exploited to compromise VM confidentiality and integrity, and any vulnerabilities in KServ, or other VMs, that are exploited also cannot compromise VM confidentiality and integrity.

### D. Bugs Found During Verification

During our noninterference proofs, we identified bugs in earlier unverified versions of SeKVM, some of which were found in the implementation used in [7]:

*1) Page table update race:* When proving invariants for page ownership used in the noninterference proofs, we identified a race condition in stage 2 page table updates. When allocating a physical page to a VM, KCore removes it from KServ's page table, assigns ownership of the page to the VM, then maps it in the VM's page table. However, if KCore is processing a KServ's stage 2 page fault on another CPU, it could check the ownership of the same page before it was assigned to the VM, think it was not assigned to any VM, and map the page in KServ's page table. This race could lead to both KServ and the VM having a memory mapping to the same physical page, violating VM memory isolation. We fixed this bug by expanding the critical section and holding the S2Page array lock, not just while checking and assigning ownership of the page, but until the page table mapping is completed.

*2) Overwrite page table mapping:* KCore initially did not check if a gfn was mapped before updating a VM's stage 2 page tables, making it possible to overwrite existing mappings. For example, suppose two VCPUs of a VM trap upon accessing the same unmapped gfn. Since KCore updates a VM's stage 2 page table whenever a VCPU traps on accessing unmapped memory, the same page table entry will be updated twice, the latter replacing the former. A compromised KServ could leverage this bug and allocate two different physical pages, breaking VM data integrity. We fixed this

| Retrofitting Component | LOC | Verification Component | LOC |
|---|---|---|---|
| QEMU additions | 70 | 34 layer specifications | 6.0K |
| KVM changes in KServ | 1.5K | `AbsMachine` machine model | 1.8K |
| HACL in KCore | 10.1K | C code proofs | 3.6K |
| KVM C in KCore | 0.2K | Assembly code proofs | 1.8K |
| KVM assembly in KCore | 0.3K | Layer refinements | 14.7K |
| Other C in KCore | 3.2K | Invariant proofs | 1.1K |
| Other assembly in KCore | 0.1K | Noninterference proofs | 3.7K |
| Total | 15.5K | Total | 32.7K |

**TABLE II:** LOC for retrofitting and verifying SeKVM.

bug by modifying KCore to update stage 2 page tables only when a mapping was previously empty.

*3) Huge page ownership:* When KServ allocated a 2MB page for a VM, KCore initially only validated the ownership of the first 4KB page rather than all the 512 4KB pages, leaving a loophole for KServ to access VM memory. We fixed this bug by accounting for this edge case in our validation logic.

*4) Multiple I/O devices using same physical page:* KCore initially did not manage memory ownership correctly when a physical page was mapped to multiple KServ SMMU page tables, with each page table controlling DMA access for a different I/O device, allowing KServ devices to access memory already assigned to VMs. We fixed this bug by having KCore only map a physical page to a VM's stage 2 or SMMU page tables when it is not already mapped to an SMMU page table used by KServ's devices.

*5) SMMU static after VM boot:* KCore initially did not ensure that mappings in SMMU page tables remain static after VM boot. This could allow KServ to modify SMMU page table mappings to compromise VM data. We fixed this bug by modifying KCore to check the state of the VM that owned the device before updating its SMMU page tables, and only allow updates before VM boot.

## VI. Implementation

The SeKVM implementation is based on KVM from mainline Linux 4.18. Table II shows our retrofitting effort, measured by LOC in C and assembly. 1.5K LOC were modified in existing KVM code, a tiny portion of the codebase, such as adding calls to KCore hypercalls. 70 LOC were also added to QEMU to support secure VM boot and VM migration. 10.1K LOC were added for the implementation of Ed25519 and AES in the HACL* [14] verified crypto library. Other than HACL*, KCore consisted of 3.8K LOC, 3.4K LOC in C and .4K LOC in assembly, of which .5K LOC were existing KVM code. The entire retrofitting process took one person-year. These results demonstrate that a widely-used, commodity hypervisor may be retrofitted with only modest implementation effort.

All of KCore's C and assembly code is verified. Table II shows our proof effort, measured by LOC in Coq. 6K LOC were for KCore's 34 layer specifications; 1.7K LOC were for the top layer which defines all of KCore's behavior while the rest were for the other 33 layers to enable modular refinement. Although using layers requires additional effort to write 33 more layer specifications, this is more than made up for by the reduction in proof effort from decomposing refinement into simpler proofs for each layer that can be reused and composed together. 1.8K LOC were for the machine model. 20.1K LOC were for refinement proofs, including proofs between KCore's C and assembly code modules and their specifications, and proofs between layer specifications. 4.8K LOC were for noninterference proofs, including invariant proofs for data structures maintained by KCore and proofs to verify security properties. 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. The Coq development effort took two person-years. These results show that microverification of a commodity hypervisor can be accomplished with modest proof effort.

| Name | Description |
|---|---|
| Hypercall | Transition from the VM to the hypervisor and return to the VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations. |
| I/O Kernel | Trap from the VM to the emulated interrupt controller in the hypervisor OS kernel, then return to the VM. Measures base cost of operations that access I/O devices supported in kernel space. |
| I/O User | Trap from the VM to the emulated UART in QEMU and then return to the VM. Measures base cost of operations that access I/O devices emulated in user space. |
| Virtual IPI | Issue a virtual IPI from a VCPU to another VCPU running on a different CPU, both CPUs executing VM code. Measures time between sending the virtual IPI until the receiving VCPU handles it. |

**TABLE III:** Microbenchmarks.

| Microbenchmark | unmodified KVM | verified KVM |
|---|---|---|
| Hypercall | 2,896 | 3,720 |
| I/O Kernel | 3,831 | 4,864 |
| I/O User | 9,288 | 10,903 |
| Virtual IPI | 8,816 | 10,699 |

**TABLE IV:** Microbenchmark performance (cycles).

## VII. Performance

We compare the performance of unmodified Linux 4.18 KVM versus our retrofitted KVM, SeKVM, both integrated with QEMU 2.3.50 to provide virtual I/O devices. We kept the software environments across all platforms as uniform as possible. All hosts and VMs used Ubuntu 16.04.06 with the same Linux 4.18 kernel. All VMs used paravirtualized I/O, typical of cloud deployments [41]. In both cases, KVM was configured with its standard virtio [42] network, and with `cache=none` for its virtual block storage devices [43], [44], [45]. For running a VM using SeKVM, we modified its guest OS virtio frontend driver to use `grant`/`revoke` to explicitly enable shared memory communication with the backend drivers in KServ.

We ran benchmarks in VMs using unmodified KVM or SeKVM. Unless otherwise indicated, all VM instances were configured with 4 VCPUs, 12 GB RAM, and huge page support enabled. 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 [46], [47]; SeKVM does not yet support VHE.

### A. Microbenchmarks

We ran KVM unit tests [48] to measure the cost of common micro-level hypervisor operations listed in Table III. Table IV shows the microbenchmarks measured in cycles for unmodified KVM and SeKVM. SeKVM incurs 17% to 28% overhead over KVM, but provides verified VM protection. The overhead is highest for the simplest operations because the relatively fixed cost of KCore protecting VM data is a higher percentage of the work that must be done. These results provide a conservative measure of overhead since real hypervisor operations will invoke actual KServ functions, not just measure overhead for a null hypercall.

| Name | Description |
|---|---|
| Kernbench | Compilation of the Linux 4.9 kernel using `allnoconfig` for Arm with GCC 5.4.0. |
| Hackbench | `hackbench` [49] using Unix domain sockets and 100 process groups running in 500 loops. |
| Netperf | `netperf` v2.6.0 [50] running `netserver` on the server and the client with its default parameters in three modes: TCP_STREAM (receive throughput), TCP_MAERTS (send throughput), and TCP_RR (latency). |
| Apache | `Apache` v2.4.18 server handling 100 concurrent requests from remote `ApacheBench` [51] v2.3 client, serving the 41 KB `index.html` of the GCC 4.4 manual. |
| 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. |
| MongoDB | `MongoDB` v4.0.20 server handling requests from a remote `YCSB` [52] v0.17.0 client running workload A with 16 concurrent threads, readcount=500000, and operationcount=100000. |

**TABLE V:** Application benchmarks.

### B. *Application Benchmarks*

We evaluated performance using real application workloads listed in Table V. For client-server experiments, the clients ran natively on an x86 Linux machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The clients communicated with the server via a 10 GbE network connection. To evaluate VM performance with end-to-end I/O protection, all VMs are configured with Full Disk Encryption (FDE) in their virtual disk. Using FDE did not have a significant impact on performance overhead, so results without FDE are omitted due to space constraints. We used dm-crypt to create a LUKS-encrypted root partition of the VM filesystem. To evaluate the extra costs in VM performance, we normalized the VM results to native hardware without FDE.

We ran application workloads using the following six configurations: (1) native hardware, (2) multiprocessor (SMP) VM on unmodified KVM (KVM), (3) SMP VM on SeKVM (SeKVM), (4) SMP VM on SeKVM without vhost (SMP-no-vhost), and (5) SMP VM on SeKVM without vhost or huge page support (SMP-no-vhost-no-huge), and (6) uniprocessor VM on SeKVM without vhost or huge page support (UP-no-vhost-no-huge). We ran benchmarks on native hardware using 4 CPUs and the same amount of RAM to provide a common basis for comparison. SMP-no-vhost, SMP-no-vhost-no-huge, and UP-no-vhost-no-huge were used to quantify the performance impact of not having verified kernel support for virtual I/O (vhost in KVM) [53], huge pages, and multiprocessor VM execution on multiple CPUs. For VMs, we pinned each VCPU to a specific physical CPU and ensured that no other work was scheduled on that CPU [46], [54], [55], [56].

To conservatively highlight microverification's impact on VM performance, we measured application performance without full network encryption, because its cost would mask any overhead between SeKVM and unmodified KVM. However, for applications that provide an option to use end-to-end encryption, specifically Apache and MySQL which have TLS/SSL support, we measured their performance with and without that option enabled, to show how the option affects overhead.

Figure 7 shows the overhead for each hypervisor configuration, normalized to native execution. On real application workloads, SeKVM incurs only modest overhead compared to unmodified KVM. In most cases, the overhead for SeKVM is similar to unmodified KVM and less than 10% compared to native. The worst overhead for SeKVM versus unmodified KVM is for TCP_MAERTS, which measures bulk data send performance from the VM to a client. Unmodified KVM achieves near native performance here because virtio batches packet sends from the VM without trapping. The cost is greater for SeKVM because the guest OS virtio driver must trap to KCore to grant KServ access for each packet, though this can be optimized further. TCP_STREAM, which measures bulk data receive performance, does not have this overhead because the virtio backend driver batches packet processing for incoming traffic, resulting in the additional traps happening less often.

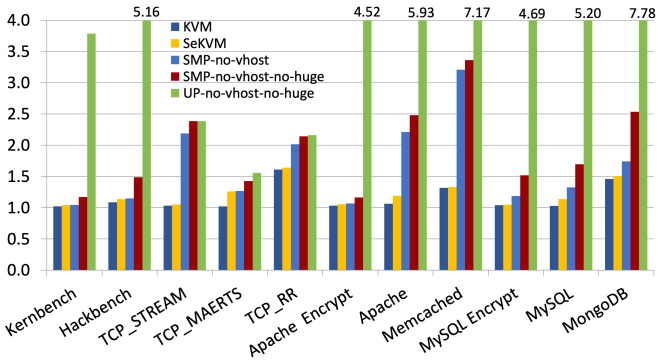



**Fig. 7: Application benchmark performance.** Overhead for each hypervisor relative to native execution. A lower score indicates less overhead; 1 means the performance in a VM is the same as native hardware.

In contrast, the performance of SMP-no-vhost, SMP-no-vhost-no-huge and UP-no-vhost-no-huge is much worse than KVM and SeKVM. SMP-no-vhost shows that lack of kernel-level virtual I/O support can result in more than two times worse performance for network I/O related workloads such as TCP_STREAM, TCP_RR, Apache, and Memcached. SMP-no-vhost-no-huge shows that lack of huge page support adds between 35% to 50% more overhead versus SMP-no-vhost for hackbench, MySQL, and MongoDB. UP-no-vhost-no-huge shows that lack of multiprocessor VM support results in many benchmarks having more than 4 times worse performance than SeKVM. TCP_STREAM and TCP_MAERTS are bandwidth limited and TCP_RR is latency bound, so the performance loss due to using only 1 CPU is smaller than other benchmarks. The results suggest that a verified system without support for commodity hypervisor features such as kernel-level virtual I/O, huge pages, and multiprocessor VMs will have relatively poor performance.

## VIII. Related Work

*Hypervisor verification.* seL4 [4], [57] and CertiKOS [5], [58] are verified systems with hypervisor functionality, so we compared their virtualization features against SeKVM. We compare the verified versions of each system; there is little reason to use unverified versions of seL4 or CertiKOS (mC2) instead of KVM. Table VI shows that SeKVM provides verified support for all listed virtualization features while seL4 and CertiKOS do not. A key verified feature of SeKVM is page tables that can be shared across multiple CPUs, which became possible to verify by introducing transparent trace refinement. This makes it possible to provide verified support for multiprocessor VMs on multiprocessor hardware as well as DMA protection. Another key verified feature of SeKVM is noninterference in the presence of I/O through shared devices paravirtualized using virtio, made possible by introducing data

| Feature | SeKVM | seL4 | CertiKOS |
|---|---|---|---|
| **VM boot protection** | Verified+FC | | |
| **VM CPU protection** | Verified+FC | | |
| **VM memory protection** | Verified+FC | | |
| **VM DMA protection** | Verified+FC | | |
| **Server hardware** | Verified | | |
| **SMP hardware** | Verified | | Unverified |
| **SMP VMs** | Verified | | |
| **Multiple VMs** | Verified | | |
| **Shared page tables** | Verified+FC | | |
| **Multi-level paging** | Verified+FC | Unverified | |
| **Huge pages** | Verified | | |
| **Virtio** | Verified | Unverified | Unverified |
| **Device passthrough** | Verified | | |
| **VM migration** | Verified | | |
| **Linux ease-of-use** | Verified | | |

**TABLE VI: Comparison of hypervisor features.** For each feature, Verified+FC means it is supported with verified VM data confidentiality and integrity and its functional correctness is also verified, Verified means it is supported with verified VM data confidentiality and integrity, and Unverified means it is supported but unverified. A blank indicates the feature is not available or so severely limited to be of little practical use.

oracles. None of these verified features are available on seL4 or CertiKOS. Unlike seL4, SeKVM does verify Arm assembly code, but CertiKOS also links C and x86 assembly code proofs together by using a verified x86 compiler to produce correct assembly; no such compiler exists for Arm multiprocessor code. Unlike seL4 and CertiKOS, SeKVM supports standard Linux functionality.

While various versions of seL4 exist, noninterference properties and functional correctness have only been verified on a single uniprocessor version [59]; bugs have been discovered in other seL4 versions [60]. The verified version only supports Armv7 hardware and has no virtualization support [59]. Another seL4 Armv7 version verifies the functional correctness of some hypervisor features, but not MMU functionality [59], [61], which is at the core of a functional hypervisor. seL4 does not support shared page tables [62], and verifying multiprocessor and hypervisor support remain future work [63]. It lacks most features expected of a hypervisor. Its device support via virtio is unverified and also needs to be ported to its platform, limiting its virtio functionality. For example, seL4 lacks support for virtio block devices and has no vhost optimization. Building a system using seL4 is much harder than using Linux [63].

CertiKOS proves noninterference for the sequential mCertiKOS kernel [35] without virtualization support and goes beyond seL4 in verifying the functional correctness of the mC2 multiprocessor kernel with virtualization. However, mC2 provides no data confidentiality and integrity among VMs. Like seL4, CertiKOS also cannot verify shared page tables, so it does not provide verified support for multiprocessor VMs. The verified kernel does not work on modern 64-bit hardware. It lacks many hypervisor features, including dynamically allocated page tables for multi-level paging, huge page support, device passthrough, and VM migration. Its virtio support does not include vhost, is limited to only certain block devices, and requires porting virtio to its platform, making it difficult to keep up with virtio improvements and updates.

Others have only partially verified their hypervisor code to reduce proof effort. The VCC framework has been used to verify 20% of Microsoft's Hyper-V multiprocessor hypervisor, but global security properties remain unproven [64], [65]. überSpark has been used to verify the üXMHF hypervisor, but their architecture does not support concurrent hardware access, and their verification approach foregoes functional correctness [66], [67]. In contrast, KCore verifiably enforces security properties by leveraging its verified core while inheriting the comprehensive features of a commodity hypervisor.

*Information-flow security verification*. Information-flow security has previously been proven for a few small, uniprocessor systems using noninterference [27], [34], [35], [36], [68], [69], [70]. None of the techniques used generalize to multiprocessor environments, where refinement can hide unintentional information leakage to concurrent observers. Information-flow security has been verified over a high-level model of the HASPOC multicore hypervisor design [71], [72], but not for the actual hypervisor implementation. Furthermore, the strict noninterference previously proven is of limited value for hypervisors because information sharing is necessary in commodity hypervisors like KVM. While some work has explored verifying information-flow security in concurrent settings by requiring the use of programming language constructs [73], [74], [75], [76], they require writing the system to be verified in their respective proposed languages, and have not been used to verify any real system. In contrast, SeKVM is written and verified in C without additional annotations, and information-flow security is proven while permitting dynamic intentional information sharing, enabling VMs to use existing KVM functionality, such as paravirtual I/O, without compromising VM data.

*Virtualization for security*. Various approaches [61], [77], [78], [79], [80] divide applications and system components in VMs, and rely on the hypervisor to safeguard interactions among secure and insecure components. In contrast, SeKVM decomposes the hypervisor itself to achieve the first security proof for a commodity multiprocessor hypervisor.

## IX. Conclusions

We have formally verified, for the first time, guarantees of VM data confidentiality and integrity for the Linux KVM hypervisor. We achieve this through microverification, retrofitting KVM with a small core that can enforce data access controls on the rest of KVM. We introduce security-preserving layers to incrementally prove the functional correctness of the core, ensuring the refinement proofs to the specification preserve security guarantees. We then use the specification to verify the security guarantees of the entire KVM hypervisor, even in the presence of information sharing needed for commodity hypervisor features, by introducing data oracles. We show that microverification only required modest KVM modifications and proof effort, yet results in a verified hypervisor that retains KVM's extensive commodity hypervisor features, including support for running multiple multiprocessor VMs, shared multi-level page tables with huge pages, and standardized virtio I/O virtualization with vhost kernel optimizations. Furthermore, our verified KVM performs comparably to stock, unverified KVM, running real application workloads in multiprocessor VMs with less than 10% overhead compared to native hardware in most cases.

## X. Acknowledgments

Xuheng Li helped with proofs for assembly code and layer refinement. Christoffer Dall, Deian Stefan, and Xi Wang provided helpful comments on drafts of this paper. This work was supported in part by NSF grants CCF-1918400, CNS-1717801, and CNS-1563555.

## References


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

[2] "Hyper-V Technology Overview," Microsoft, Nov. 2016, https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[3] S. J. Vaughan-Nichols, "Hypervisors: The cloud's potential security Achilles heel," *ZDNet*, Mar. 2014, https://www.zdnet.com/article/hypervisors-the-clouds-potential-security-achilles-heel.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, Oct. 2009, pp. 207–220.

[5] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, Savannah, GA, Nov. 2016, pp. 653–669.

[6] "The Coq Proof Assistant." http://coq.inria.fr [Accessed: Dec 16, 2020]

[7] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, Santa Clara, CA, Aug. 2019, pp. 1357–1374.

[8] C. Dall and J. Nieh, "KVM/ARM: Experiences Building the Linux ARM Hypervisor," Department of Computer Science, Columbia University, Technical Report CUCS-010-13, Jun. 2013.

[9] ——, "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)*, Salt Lake City, UT, Mar. 2014, pp. 333–347.

[10] "Cloud companies consider Intel rivals after the discovery of microchip security flaws," *CNBC*, Jan. 2018, https://www.cnbc.com/2018/01/10/cloud-companies-consider-intel-rivals-after-security-flaws-found.html.

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

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

[13] "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile," ARM Ltd., ARM DDI 0487A.a, Sep. 2013.

[14] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS 2017)*, Dallas, TX, Oct. 2017, pp. 1789–1806.

[15] J. Graham-Cumming and J. W. Sanders, "On the Refinement of Non-interference," in *Proceedings of Computer Security Foundations Workshop IV*, Franconia, NH, Jun. 1991, pp. 35–42.

[16] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazieres, "Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, ser. ACM SIGPLAN Notices, vol. 47, no. 9, Sep. 2012, pp. 201–214.

[17] J. A. Goguen and J. Meseguer, "Unwinding and Inference Control," in *Proceedings of the 1984 IEEE Symposium on Security and Privacy (SP 1984)*, Oakland, CA, Apr. 1984, pp. 75–86.

[18] A. Sabelfeld and A. C. Myers, "A Model for Delimited Information Release," in *Proceedings of the 2nd International Symposium on Software Security (ISSS 2003)*, Tokyo, Japan, Nov. 2003, pp. 174–191.

[19] P. Stewin and I. Bystrov, "Understanding DMA Malware," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, Heraklion, Crete, Greece, Jul. 2013, pp. 21–41.

[20] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS 2009)*, Chicago, IL, Nov. 2009, pp. 199–212.

[21] Y. Zhang, A. Juels, M. K. Reiter, and T. 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)*, Raleigh, NC, Oct. 2012, pp. 305–316.

[22] G. Irazoqui, T. Eisenbarth, and B. 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 (SP 2015)*, San Jose, CA, May 2015, pp. 591–604.

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in Paas Clouds," in *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS 2014)*, Scottsdale, AZ, Nov. 2014, pp. 990–1003.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*, San Jose, CA, May 2015, pp. 605–622.

[25] M. Backes, G. Doychev, and B. 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, Feb. 2013.

[26] K. J. Biba, "Integrity Considerations for Secure Computer Systems," MITRE, Technical Report MTR-3153, Jun. 1975.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, Shanghai, China, Oct. 2017, pp. 287–305.

[28] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro, "Certified Concurrent Abstraction Layers," in *Proceedings of the 39th ACM Conference on Programming Language Design and Implementation (PLDI 2018)*, Philadelphia, PA, Jun. 2018, pp. 646–661.

[29] X. Leroy, "The CompCert Verified Compiler." https://compcert.org [Accessed: Dec 16, 2020]

[30] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization*, ser. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, Feb. 2017.

[31] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, and H. Zhang, "Deep Specifications and Certified Abstraction Layers," in *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, Mumbai, India, Jan. 2015, pp. 595–608.

[32] R. Keller, "Formal Verification of Parallel Programs," *Communications of the ACM*, vol. 19, pp. 371–384, Jul. 1976.

[33] C. Jones, "Tentative Steps Toward a Development Method for Interfering Programs." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, pp. 596–619, Oct. 1983.

[34] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP 2013)*, San Francisco, CA, May 2013, pp. 415–429.

[35] D. Costanzo, Z. Shao, and R. 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)*, Santa Barbara, CA, Jun. 2016, pp. 648–664.

[36] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. 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)*, Carlsbad, CA, Oct. 2018, pp. 287–305.

[37] "ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0," ARM Ltd., Jun. 2016.

[38] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. 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)*, Seattle, WA, Mar. 2008, pp. 2–13.

[39] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. 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)*, Houston, TX, Mar. 2013, pp. 265–278.

[40] "ARM Power State Coordination Interface," ARM Ltd., ARM DEN 0022D, Apr. 2017.

[41] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual Remote I/O," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, Atlanta, GA, 2016, pp. 49–65.

[42] R. Russell, "virtio: Towards a De-Facto Standard for Virtual I/O Devices," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

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

[44] "Disk Cache Modes," in *SUSE Linux Enterprise Server 12 SP5 Virtualization Guide*. SUSE, Dec. 2020, ch. 15. https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-cachemodes.html

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

[46] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos, "ARM Virtualization: Performance and Architectural Implications," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, Seoul, South Korea, Jun. 2016, pp. 304–316.

[47] C. Dall, S.-W. Li, and J. Nieh, "Optimizing the Design and Implementation of the Linux ARM Hypervisor," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, Santa Clara, CA, Jul. 2017, pp. 221–234.

[48] "KVM Unit Tests." https://www.linux-kvm.org/page/KVM-unit-tests [Accessed: Dec 16, 2020]

[49] R. Russell, Z. Yanmin, I. Molnar, and D. Sommerseth, "Improve hackbench," Linux Kernel Mailing List (LKML), Jan. 2008, http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[50] R. Jones, "Netperf." https://github.com/HewlettPackard/netperf [Accessed: Dec 16, 2020]

[51] "ab - Apache HTTP server benchmarking tool." http://httpd.apache.org/docs/2.4/programs/ab.html [Accessed: Dec 16, 2020]

[52] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, Indianapolis, IN, Jun. 2010, pp. 143–154.

[53] "UsingVhost - KVM." https://www.linux-kvm.org/page/UsingVhost [Accessed: Dec 16, 2020]

[54] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier, "NEVE: Nested Virtualization Extensions for ARM," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, Shanghai, China, Oct. 2017, pp. 201–217.

[55] C. Dall, S.-W. Li, J. T. Lim, and J. Nieh, "ARM Virtualization: Performance and Architectural Implications," *ACM SIGOPS Operating Systems Review*, vol. 52, no. 1, pp. 45–56, Jul. 2018.

[56] J. T. Lim and J. 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)*, Lausanne, Switzerland, Mar. 2020, pp. 557–574.

[57] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–70, Feb. 2014.

[58] R. Gu, Z. Shao, H. Chen, J. Kim, J. Koenig, X. Wu, V. Sjöberg, and D. Costanzo, "Building Certified Concurrent OS Kernels," *Communications of the ACM*, vol. 62, no. 10, pp. 89–99, Sep. 2019.

[59] "seL4 Supported Platforms." https://docs.sel4.systems/Hardware [Accessed: Dec 16, 2020]

[60] J. Oberhauser, R. L. de Lima Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis, "VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, Detroit, MI, Apr. 2021.

[61] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser, "Formally Verified Software in the Real World," *Communications of the ACM*, vol. 61, no. 10, pp. 68–77, Sep. 2018.

[62] "seL4 Reference Manual Version 11.0.0," Data61, Nov. 2019.

[63] "Frequently Asked Questions on seL4." https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html [Accessed: Dec 16, 2020]

[64] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A Practical System for Verifying Concurrent C," in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, Munich, Germany, Aug. 2009, pp. 23–42.

[65] D. Leinenbach and T. Santen, "Verifying the Microsoft Hyper-V hypervisor with VCC," in *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, Eindhoven, The Netherlands, Nov. 2009, pp. 806–809.

[66] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP 2013)*, San Francisco, CA, May 2013, pp. 430–444.

[67] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. 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)*, Austin, TX, Aug. 2016, pp. 87–104.

[68] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad Apps: End-to-End Security via Automated Full-System Verification," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, Broomfield, CO, Oct. 2014, pp. 165–181.

[69] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. 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)*, Huntsville, Ontario, Canada, Oct. 2019, pp. 225–242.

[70] D. Jang, Z. Tatlock, and S. Lerner, "Establishing Browser Security Guarantees through Formal Shim Verification," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, Bellevue, WA, Aug. 2012, pp. 113–128.

[71] C. Baumann, M. Näslund, C. Gehrmann, O. Schwarz, and H. Thorsen, "A High Assurance Virtualization Platform for ARMv8," in *Proceedings of the 2016 European Conference on Networks and Communications (EuCNC 2016)*, Athens, Greece, Jun. 2016, pp. 210–214.

[72] C. Baumann, O. Schwarz, and M. Dam, "On the verification of system-level information flow properties for virtualized execution platforms," *Journal of Cryptographic Engineering*, vol. 9, no. 3, pp. 243–261, May 2019.

[73] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016)*, Lisbon, Portugal, Jun. 2016, pp. 417–431.

[74] T. Murray, R. Sison, and K. Engelhardt, "COVERN: A Logic for Compositional Verification of Information Flow Control," in *Proceedings of the 2018 IEEE European Conference on Security and Privacy (EuroS&P 2018)*, London, United Kingdom, Apr. 2018, pp. 16–30.

[75] G. Ernst and T. Murray, "SecCSL: Security Concurrent Separation Logic," in *Proceedings of the 31st International Conference (CAV 2019)*, New York, NY, Jul. 2019, pp. 208–230.

[76] D. Schoepe, T. Murray, and A. Sabelfeld, "VERONICA: Expressive and Precise Concurrent Information Flow Security," in *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF 2020)*, Boston, MA, Jun. 2020, pp. 79–94.

[77] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-based Platform for Trusted Computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Bolton Landing, NY, Oct. 2003, pp. 193–206.

[78] R. Strackx and F. Piessens, "Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, Raleigh, NC, Oct. 2012, pp. 2–13.

[79] R. Ta-Min, L. Litty, and D. Lie, "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, Nov. 2006, pp. 279–292.

[80] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation," in *Proceedings of the 2015 ACM Conference on Computer and Communications Security (CCS 2015)*, Denver, CO, Oct. 2015, pp. 1607–1619.

## APPENDIX A
## KCORE'S TOP-LEVEL INTERFACE AND HYPERCALLS

We describe in more detail the hypercalls and exception handlers in Table I, provided by KCore's top-level interface `TrapHandler`.

### A. Hypercalls

- **register_vm**: Used by KServ to request KCore to create new VMs. KCore allocates a unique VM identifier, which it returns to KServ. It also allocates the per-VM metadata `VMInfo` and a stage 2 page table root for the VM.
- **register_vcpu**: Used by KServ to request KCore to initialize a new VCPU for a specified VM. KCore allocates the `VCPUContext` data structure for the VCPU.
- **set_boot_info**: Used by KServ to pass VM boot image information, such as the image size, to KCore.
- **remap_boot_image_page**: Used by KServ to pass one page of the VM boot image to KCore. KCore remaps all pages of a VM image to a contiguous range of memory in its address space so it can later authenticate the image.
- **verify_vm_image**: Used by KServ to request KCore to authenticate a VM boot image. KCore authenticates each binary of the boot image, and refuses to boot the VM if authentication fails.

Before authenticating a VM image, KCore unmaps all its pages from KServ's stage 2 page table to guarantee that the verified image cannot be later altered by KServ. If authentication succeeds, KCore maps the boot image to the VM's stage 2 page table.

- **clear_vm**: Used by KServ to request KCore to reclaim pages from a terminated VM. KCore will scrub all pages of the terminated VM and set their ownership to KServ.
- **encrypt_vcpu**: Used by KServ to request KCore to export encrypted VM CPU data; for VM migration and snapshots.
- **decrypt_vcpu**: Used by KServ to request KCore to import encrypted VM CPU data; for VM migration and snapshots. KCore copies the data to a private buffer before decrypting it.
- **encrypt_vm_mem**: Used by KServ to request KCore to export encrypted VM memory data; for VM migration and snapshots.
- **decrypt_vm_mem**: Used by KServ to request KCore to import encrypted VM memory data; for VM migration and snapshots. KCore copies the data to a private buffer before decrypting it.
- **set_timer**: Used by KServ to request KCore to update a privileged EL2 timer register with a timer counter offset; for timer virtualization since SeKVM offloads timer virtualization to KServ.
- **smmu_alloc_unit**: Used by KServ to request KCore to allocate an SMMU translation unit for a given device. KCore sets the owner of the SMMU translation unit to the owner of the device.
- **smmu_free_unit**: Used by KServ to request KCore to deallocate an SMMU translation unit previously used by a device. If a device was owned by a VM, KCore ensures that deallocation can succeed when the VM is powered off.
- **smmu_map**: Used by KServ to request KCore to map a 4KB page pfn to a device's SMMU page table, from a device virtual address (iova) to the hPA of the pfn. KServ rejects the request if the owner of the pfn is different from the device. KServ is allowed to map a page to the SMMU page table of a VM owned device before the VM boots.
- **smmu_unmap**: Used by KServ to request KCore to unmap an iova in a device's SMMU page table. KServ is only allowed to do so after the VM that owns the device is powered off.
- **smmu_iova_to_phys**: Used by KServ to request KCore to walk a device's SMMU page table. Given an iova, KCore returns the corresponding physical address.
- **run_vcpu**: Used by KServ to request KCore to run a VM's VCPU on the current CPU. KServ passes the VM and VCPU identifiers to KCore. KCore context switches to the VCPU and resolves prior VM exceptions before returning to the VM.
- **grant**: Used by a VM to grant KServ access to its data in a memory region. KCore sets the `share` field in the `S2Page` structure for each page in the memory region.
- **revoke**: Used by a VM to revoke KServ's access to a previously shared memory region. KCore clears the `share` field in the `S2Page` structure for each of the pages in the memory region and unmaps the pages from KServ.
- **psci_power**: Used by a VM to request KCore to configure VM power states via Arm's PSCI power management interface [40]; the request is passed to KServ for power management emulation.

### B. Exception Handlers

- **host_page_fault**: Handles stage 2 page faults for KServ. KCore builds the mapping for the faulted address in KServ's stage 2 page table if the access is allowed. An identity mapping is used for hPAs in KServ's stage 2 page table, allowing KServ to implicitly manage all free physical memory.
- **vm_page_fault**: Handles stage 2 page faults for a VM, which occur when a VM accesses unmapped memory or its MMIO devices. KCore context switches to KServ for further exception processing, to allocate memory for the VM and emulate VM MMIO accesses. KCore copies the I/O data from KServ to the VM's GPRs on MMIO reads, and vice versa on MMIO writes.
- **handle_irq**: Handles physical interrupts that result in VM exits, KCore context switches to KServ for the interrupt handling.
- **handle_wfx**: Handles VM exits due to WFI/WFE instructions. KCore context switches to KServ to handle the exception.
- **handle_sysreg**: Handles VM exits due to accessing privileged system registers, handled directly by KCore.

### C. Memory Operations

- **mem_load/store**: Used to specify regular memory accesses. A memory access from a currently running principal will walk its stage 2 page table to translate from a gPA to hPA. If the physical page is not found or the access permission is violated, a page fault occurs. If KServ caused the fault, the program counter (PC) is set to the `host_page_fault` primitive. If a VM caused the fault, the PC is set to the `vm_page_fault` primitive.
- **dev_load/store**: Used to specify memory accesses by devices. A memory access from a currently running principal's device will walk its SMMU page table to translate from an iova to hPA.

## Appendix B
## Example Proofs

We present a more detailed but still simplified proof of the primitives involved in handling a VM's stage 2 page fault, shown in Figure 8, expanding on the examples in Sections III-A and V-B with Coq definitions. We describe the refinement proofs for primitives across four layers which handle the page fault after KServ has proposed a pfn to allocate and called `run_vcpu` in `TrapHandler`, the top layer. `run_vcpu` calls `assign_to_vm` in `MemOps` to unmap the page from KServ's stage 2 page table, update its owner to the VM, and map the page to the VM's stage 2 page table. `MemOps` updates these stage 2 page tables using `set_npt` in `NPTOps`, which in turn invokes basic primitives provided by `NPTWalk`.

**Layer 1:** *`NPTWalk`*. This layer specifies a set of basic *primitives*, verified in lower layers and passed through to higher layers, and an *abstract state* upon which they act. The abstract state consists of a `log` of shared object events, the local CPU identifier `cid`, the currently running VM `vmid` on CPU `cid`, and a map from `vmid` to each VM's local state. We define the events and log:

```
Inductive Event :=
| ACQ_NPT (vmid: Z)          | REL_NPT (vmid: Z)
| P_LD (vmid ofs: Z)         | P_ST (vmid ofs val: Z)
| ACQ_S2PG                   | REL_S2PG
| GET_OWNER (pfn: Z)         | SET_OWNER (pfn owner: Z)
| SET_MEM (pfn val: Z).

(* Log is a list of events and their CPU identifiers *)
Definition Log := list (Event * Z).
```

We then define a VM's local state and `NPTWalk`'s abstract state:

```
(* VM local state *)
Record LocalState := {
  data_oracle: ZMap.t Z; (* data oracle for the VM *)
```

```
// Primitives provided by NPTWalk
extern void acq_lock_npt(uint vmid);
extern void rel_lock_npt(uint vmid);
extern uint pt_load(uint vmid, uint ofs);
extern void pt_store(uint vmid, uint ofs, uint value);
extern void acq_lock_s2pg();
extern void rel_lock_s2pg();
extern uint get_s2pg_owner(uint pfn);
extern void set_s2pg_owner(uint pfn, uint owner);

// Primitive provided by NPTOps
void set_npt(uint vmid, uint gfn, uint pfn) {
  acq_lock_npt(vmid);
  uint pte = pt_load(vmid, pgd_offset(gfn));
  pt_store(vmid, pte_offset(pte,gfn), pfn);
  rel_lock_npt(vmid);
}

// Primitive provided by MemOps
uint assign_to_vm(uint vmid, uint gfn, uint pfn) {
  uint res = 1;
  acq_lock_s2pg();
  uint owner = get_s2pg_owner(pfn);
  if (owner == KSERV) {
    set_npt(KSERV, pfn, 0);
    set_s2pg_owner(pfn, vmid);
    set_npt(vmid, gfn, pfn);
  } else res = 0; // pfn is not owned by KSERV
  rel_lock_s2pg();
  return res;
}

// Primitive provided by TrapHandler
void run_vcpu(uint vmid) {
  ...
  assign_to_vm(vmid, gfn, pfn);
  ...
}
```

**Fig. 8: Simplified implementation for handling a stage 2 page fault.** Primitives from four layers are called. For simplicity, other primitives in the layers are omitted and only two levels of paging are shown.

```
  doracle_counter: Z; (* data oracle query counter *)
  ...
}.

(* Abstract state *)
Record AbsSt := {
  log: Log;
  cid: Z; (* local CPU identifier *)
  vid: Z; (* vmid of the running principal on CPU cid *)
  lstate: ZMap.t LocalState; (* per-VM local state *)
}.
```

The abstract state does not contain shared objects, which are constructed using the log through a *replay function*:

```
(* Shared objects constructed using replay function *)
Record SharedObj := {
  mem: ZMap.t Z; (* maps addresses to values *)
  s2pg_lock: option Z; (* s2pg lock holder *)
  pt_locks: ZMap.t (option Z); (* pt lock holders *)
  pt_pool: ZMap.t (ZMap.t Z); (* per-VM page table pool *)
  (* s2pg_array maps pfn to (owner, share, gfn) *)
  s2pg_array: ZMap.t (Z * Z * Z);
}.

Fixpoint replay (l: Log) (obj: SharedObj) :=
  match l with
  | e::l' => match replay l' obj with
             | Some (obj', _) => replay_event e obj'
             | None => None
             end
  | _ => Some st
  end.
```

The `replay` function recursively traverses the log to reconstruct the state of shared objects, invoking `replay_event` to handle each event and update shared object state; this update may fail (i.e., return `None`) if the event is not valid with respect to the current state. For example, the replay function returns the load result for a page table pool load event `P_LD`, but the event is only allowed if the lock is held by the current CPU:

```
Definition replay_event (e: Event) (obj: SharedObj) :=
  match e with
  | (P_LD vmid ofs, cpu) =>
    match ZMap.get vmid (pt_locks obj) with
    | Some cpu' => (*the pt lock of vmid is held by cpu'*)
      if cpu =? cpu' (* if cpu = cpu' *)
      then let pool := ZMap.get vmid (pt_pool obj) in
           Some (obj, Some (ZMap.get ofs pool))
      else None (* fails when held by a different cpu *)
    | None (* fails if not already held *)
    end
  | ... (* handle other events *)
  end.
```

We then define `NPTWalk`'s layer interface as a map from function names to their specifications defined upon the abstract state:

```
Definition NPTWalk: Layer AbsSt :=
  acq_lock_npt          ↦ csem acq_lock_npt_spec
  ⊕ rel_lock_npt        ↦ csem rel_lock_npt_spec
  ⊕ pt_load             ↦ csem pt_load_spec
  ⊕ pt_store            ↦ csem pt_store_spec
  ⊕ acq_lock_s2pg       ↦ csem acq_lock_s2pg_spec
  ⊕ rel_lock_s2pg       ↦ csem rel_lock_s2pg_spec
  ⊕ get_s2pg_owner      ↦ csem get_s2pg_owner_spec
  ⊕ set_s2pg_owner      ↦ csem set_s2pg_owner_spec.
```

These primitives are defined in a language-independent way and are lifted to C-level semantics through a wrapper function `csem`, so that arguments and return values are passed according to C calling conventions. As discussed in Section III-A, MicroV supports CPU-local reasoning, using event oracles to encapsulate events generated by other CPUs. A primitive accessing shared objects can be specified in a CPU-local manner. For example, `pt_load`'s specification queries event oracle `o` to obtain events from other CPUs, appends them to the logical log (producing `l0`), checks the validity of the load and calculates the load result using `replay`, then appends a load event to the log (producing `l1`):

```
(* Event Oracle takes the current log and produces
   a sequence of events generated by other CPUs *)
Definition EO := Log -> Log.

Definition pt_load_spec
    (o: EO) (st: AbsSt) (vmid ofs: Z) :=
  let l0 := o (log st) ++ log st in (*query event oracle*)
  (* produce the P_LD event *)
  let l1 := (P_LD vmid ofs, cid st) :: l0 in
  match replay l1 with
    (* log is valid and P_LD event returns r *)
  | Some (_, Some r) => Some (st {log: l1}, r)
  | _ => None
  end.
```

`pt_load_spec` relies on the assumption that events generated by other CPUs are valid with respect to the replay function—a *rely condition*. When rely conditions hold for all other CPUs, we can prove that all events generated by the local CPU are also valid—a *guarantee condition*. Since each CPU's rely condition follows from the guarantee conditions of other CPUs, the execution of all CPUs can be soundly composed.

Data oracles can be used for primitives that declassify data, as discussed in Section III-B. For example, `set_s2pg_owner` changes the ownership of a page. When the owner is changed from KServ to a VM `vmid`, the page contents owned by KServ is declassified to VM `vmid`, so a data oracle is used in the specification of `set_s2pg_owner` to mask the declassified contents:

```
Definition set_s2pg_owner_spec
    (o: E0) (st: AbsSt) (pfn vmid: Z) :=
  let l0 := o (log st) ++ log st in
  let l1 := (SET_OWNER pfn vmid, cid st) :: l0 in
  match replay l1 with
  | Some _ => (* log is valid and lock is held *)
    let st' := st {log: l1}) in
    if vid st =? KSERV && vmid != KSERV
    then (* pfn is transferred from KServ to a VM *)
      mask_with_doracle st' vmid pfn
    else Some st'
  | _ => None
  end.
```

We introduce an auxiliary Coq definition `mask_with_doracle` to encapsulate the masking behavior:

```
Definition mask_with_doracle (st: AbsSt) (vmid pfn: Z) :=
  let local := ZMap.get vmid (lstate st) in
  let n := doracle_counter local in
  let val := data_oracle local n in
  let l := (SET_MEM pfn val, cid st) :: log st in
  let local' := local {doracle_counter: n+1} in
  st {log: l, lstate: ZMap.set vmid local' (lstate st)}
```

`mask_with_doracle` queries the local data oracle of VM `vmid` with a local query counter, generates an event to mask the declassified content with the query result, then updates the local counter. Since each principal has its own data oracle based on its own local state, the behavior of other principals cannot affect the query result. `set_s2pg_owner_spec` only queries the data oracle when the owner is changed from KServ to a VM. When the owner is changed from a VM to KServ, the page is being freed and KCore must zero out the page before recycling it; masking is not allowed. We also introduce auxiliary definitions to mask other declassified data, such as page indices and scheduling decisions proposed by KServ, which are not shown in this simplified example.

**Layer 2: `NPTOps`.** This layer introduces the `set_npt` primitive for higher layers to update page table entries, and hides page table structures and page walk details by removing `NPTWalk`'s primitives related to page table pools. Other primitives are passed through.

```
Definition NPTOps: Layer AbsSt :=
  set_npt               ↦ csem set_npt_spec
  ⊕ acq_lock_s2pg       ↦ csem acq_lock_s2pg_spec
  ⊕ rel_lock_s2pg       ↦ csem rel_lock_s2pg_spec
  ⊕ get_s2pg_owner      ↦ csem get_s2pg_owner_spec
  ⊕ set_s2pg_owner      ↦ csem set_s2pg_owner_spec.
```

`set_npt`'s specification simply generates an atomic event `SET_NPT`:

```
Definition set_npt_spec
    (o: E0) (st: AbsSt) (vmid gfn pfn: Z) :=
  let l0 := o (log st) ++ log st in
  let l1 := (SET_NPT vmid gfn pfn, cid st) :: l0 in
  match replay l1 with
  | Some _ => Some (st {log: l1})
  | _ => None
  end.
```

To show `set_npt` meets the above specification, we prove that its C code implementation running over `NPTWalk` faithfully implements `set_npt_spec` for all possible interleavings across CPUs:

$\forall$`E0`, $\exists$`E0'`, $M_{\texttt{set\_npt}}$(`E0`)@`NPTWalk` $\sqsubseteq_R$ `set_npt_spec`(`E0'`)

This says that, for any valid event oracle `E0` of `NPTWalk`, there exists a valid event oracle `E0'` for `NPTOps` such that, starting from two logs related $R$, the logs generated by the implementation and specification, respectively, are still related by $R$. Here, $R$ is the refinement relation, which maps the `SET_NPT` event to the following four consecutive events and maps other events to themselves:

`ACQ_NPT` `P_LD` `P_ST` `REL_NPT`

To prove this refinement, MicroV first uses CompCert's ClightGen to parse the C implementation to its Clight representation $M_{\texttt{set\_npt}}$, a C abstract syntax tree defined in Coq. Based on the semantics of Clight, we can show that, for event oracle `E0` and log $l$, the updated log after executing $M_{\texttt{set\_npt}}$ is:

[$l$, `E01`, `ACQ_NPT`, `E02`, `P_LD`, `E03`, `P_ST`, `E04`, `REL_NPT`, `E05`]

where `E0n` represents other CPUs' events, from the event oracle.

Since the `lock_npt` spinlock enforces WDRF over VM `vmid`'s shared page table pool $\texttt{pt\_pool}_{\texttt{vmid}}$, events generated by other CPUs can be safely shuffled across events in the same *observer group* over $\texttt{pt\_pool}_{\texttt{vmid}}$ using transparent trace refinement. For example, `E03` can be shuffled to the left of `P_LD` since the replayed state before and after `P_LD` share the same $\texttt{pt\_pool}_{\texttt{vmid}}$, but cannot be shuffled to the right of `P_ST`, which belongs to a different observer group. Thus, we can shuffle the above log to the following:

[$l$, `E01`, `E02`, `E03`, `ACQ_NPT`, `P_LD`, `P_ST`, `REL_NPT`, `E04`, `E05`]

We can then prove that this log transparently refines the log generated by the specification:

[$l'$, `E01'`, `SET_NPT`, `E02'`]

when the input logs and oracle `E0'` satisfy $R$, i.e., $R\ l\ l'$, $R$ [`E01'`] [`E01`, `E02`, `E03`], and $R$ [`E02'`] [`E04`, `E05`].

We can also show that the following log generated by the insecure implementation shown in Section III-A cannot be transparently refined to the specification's log, because `E04` cannot be shuffled across observer groups:

[$l$, `E01`, `ACQ_NPT`, `E02`, `P_LD`, `E03`, `P_ST`, `E04`, `P_ST`, `E05`, ...]

**Layer 3: *`MemOps`*.** This layer introduces the `assign_to_vm` primitive to transfer a page from KServ to a VM, and hides `NPTOps` primitives:

```
Definition MemOps: Layer AbsSt :=
  assign_to_vm      ↦ csem assign_to_vm_spec.
```

`assign_to_vm`'s specification has a precondition that it must be invoked by KServ and the `vmid` must be valid:

```
Definition assign_to_vm_spec
    (o: E0) (st: AbsSt) (vmid gfn pfn: Z) :=
  if vid st =? KSERV && vmid != KSERV
  then
    let l0 := o (log st) ++ log st in
    let l1 := (ASG_TO_VM vmid gfn pfn, cid st) :: l0 in
    match replay l1 with
    | Some (_, Some res)) => (* res is the return value *)
      let st' := st {log: l1} in (* update the log *)
      if res =? 1 (* if pfn is owned by KSERV *)
      then mask_with_doracle st' vmid pfn
      else Some st' (* return without masking the page *)
    | _ => None
    end
  (*get stuck if it's not transferred from KServ to a VM*)
  else None.
```

It transfers a page from KServ to a VM via `set_s2pg_owner`, so the contents are declassified and must be masked using the data oracle.

**Layer 4: *`TrapHandler`*.** This top layer interface introduces `run_vcpu`, which invokes `assign_to_vm` and context switches from KServ to the VM. We first prove that `run_vcpu` does not violate the precondition of `assign_to_vm`. We then prove noninterference as discussed in Section V-B. We can see here why the page content will be masked with the same data oracle query results in the proof of Lemma 3 for `run_vcpu` in Section V-B. Two indistinguishable states will have the same VM local states, and therefore the same local data oracle and counter. Thus, the data oracle query results must be the same.