Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits

Shih-Wei Li, John S. Koh, and Jason Nieh, *Columbia University*

https://www.usenix.org/conference/usenixsecurity19/presentation/li-shih-wei

This paper is included in the Proceedings of the
28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the
28th USENIX Security Symposium
is sponsored by USENIX.

Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits

Shih-Wei Li John S. Koh Jason Nieh
Department of Computer Science
Columbia University
{shihwei,koh,nieh}@cs.columbia.edu

Abstract

Hypervisors are widely deployed by cloud computing providers to support virtual machines, but their growing complexity poses a security risk as large codebases contain many vulnerabilities. We have created HypSec, a new hypervisor design for retrofitting an existing commodity hypervisor using microkernel principles to reduce its trusted computing base while protecting the confidentiality and integrity of virtual machines. HypSec partitions the hypervisor into an untrusted host that performs most complex hypervisor functionality without access to virtual machine data, and a trusted core that provides access control to virtual machine data and performs basic CPU and memory virtualization. Hardware virtualization support is used to isolate and protect the trusted core and execute it at a higher privilege level so it can mediate virtual machine exceptions and protect VM data in CPU and memory. HypSec takes an end-to-end approach to securing I/O to simplify its design, with applications increasingly using secure network connections in the cloud. We have used HypSec to retrofit KVM, showing how our approach can support a widely-used full-featured hypervisor integrated with a commodity operating system. The implementation has a trusted computing base of only a few thousand lines of code, many orders of magnitude less than KVM. We show that HypSec protects the confidentiality and integrity of virtual machines running unmodified guest operating systems while only incurring modest performance overhead for real application workloads.

1 Introduction

The availability of cost-effective, commodity cloud providers has pushed increasing numbers of companies and users to move their data and computation off site into virtual machines (VMs) running on hosts in the cloud. The hypervisor provides the VM abstraction and has full control of the hardware resources. Modern hypervisors are often integrated with a host operating system (OS) kernel to leverage existing kernel functionality to simplify their implementation and maintenance effort. For example, KVM [44] is integrated with Linux and Hyper-V [56] is integrated with Windows. The result is a huge potential attack surface with access to VM data in CPU registers, memory, I/O data, and boot images. The surge in outsourcing of computational resources to the cloud and away from privately-owned data centers further exacerbates this security risk of relying on the trustworthiness of complex and potentially vulnerable hypervisor and host OS infrastructure. Attackers that successfully exploit hypervisor vulnerabilities can gain unfettered access to VM data, and compromise the privacy and integrity of all VMs—an undesirable outcome for both cloud providers and users.

Recent trends in application design and hardware virtualization support provide an opportunity to revisit hypervisor design requirements to address this crucial security problem. First, modern hardware includes virtualization support to protect and run the hypervisor at a higher privilege level than VMs, potentially providing new opportunities to redesign the hypervisor to improve security. Second, due to greater security awareness because of the Snowden leaks revealing secret surveillance of large portions of the network infrastructure [49], applications are increasingly designed to use end-to-end encryption for I/O channels, including secure network connections [29, 50] and disk encryption [14]. This is decreasing the need for hypervisors to themselves secure I/O channels since applications can do a better job of providing an end-to-end I/O security solution [68].

Based on these trends, we have created HypSec, a new hypervisor design for retrofitting commodity hypervisors to significantly reduce the code size of their trusted computing base (TCB) while maintaining their full functionality. The design employs microkernel principles, but instead of requiring a clean-slate rewrite from scratch—a difficult task that limits both functionality and deployment—applies them to restructure an existing hypervisor with modest modifications. HypSec partitions a monolithic hypervisor into a small trusted core, the *corevisor*, and a large untrusted host, the *hostvisor*. HypSec leverages hardware virtualization support to isolate and protect the corevisor and execute it at a higher privilege

level than the hostvisor. The corevisor enforces access control to protect data in CPU and memory, but relies on VMs or applications to use end-to-end encrypted I/O to protect I/O data, simplifying the corevisor design.

The corevisor has full access to hardware resources, provides basic CPU and memory virtualization, and mediates all exceptions and interrupts, ensuring that only a VM and the corevisor can access the VM's data in CPU and memory. More complex operations including I/O and interrupt virtualization, and resource management such as CPU scheduling, memory management, and device management are delegated to the hostvisor, which can also leverage a host OS. The hostvisor may import or export encrypted VM data from the system to boot VM images or support hypervisor features such as snapshots and migration, but otherwise has no access to VM data. HypSec redesigns the hypervisor to improve security but does not strip it of functionality. We expect that HypSec can be used to restructure existing hypervisors by encapsulating much of their codebase in a hostvisor and augmenting security with a corevisor.

We have implemented a HypSec prototype by retrofitting KVM. Our approach works with existing ARM hardware virtualization extensions to provide VM confidentiality and integrity in a full-featured commodity hypervisor with its own integrated host OS kernel. Our implementation requires only modest modifications to Linux and has a TCB of only a few thousand lines of code (LOC), many orders of magnitude less than KVM and other commodity hypervisors. HypSec significantly reduces the TCB of an existing widely-used hypervisor and improves its security while retaining the same hypervisor functionality, including multiprocessor, full device I/O, multi-VM, VM management, and broad ARM hardware support. We also show that HypSec provides strong security for VMs running unmodified guest operating systems while only incurring modest performance overhead for real application workloads.

2 Assumptions and Threat Model

Assumptions. We assume VMs use end-to-end encrypted channels to protect their I/O data. We assume hardware virtualization support and an IOMMU similar to what is available on x86 and ARM servers in the cloud. We assume a Trusted Execution Environment (TEE) provided by secure mode architectures such as ARM TrustZone [7] or a Trusted Platform Module (TPM) [38] is available for trusted persistent storage. We assume the hardware, including a hardware security module if applicable, is bug-free and trustworthy. We assume the HypSec TCB, the corevisor, does not have any vulnerabilities and can thus be trusted. Given the corevisor's modest size as shown in Section 6.3, it may be possible to formally verify the codebase. We assume it is computationally infeasible to perform brute-force attacks on any encrypted VM data, and any encrypted communication protocols are assumed to be designed to defend against replay attacks. We assume the system is initially benign, allowing signatures and keys to be sealed in the TEE before a compromise of the system.

Threat Model. We consider an attacker with remote access to a hypervisor and its VMs, including administrators without physical access to the machine. The attacker's goal is to compromise the confidentiality and integrity of VM data, which includes: the VM boot image containing the guest kernel binary, data residing in memory addresses belonging to guests, guest memory copied to hardware buffers, data on VM disks or file systems, and data stored in VM CPU registers. VM data does not include generic virtual hardware configuration information, such as the CPU power management status or the interrupt level being raised. An attacker could exploit bugs in the hostvisor or control the VM management interface to access VM data. For example, an attacker could exploit bugs in the hostvisor to execute arbitrary code or access VM memory from the VM or hypervisor host. Attackers may also control peripherals to perform malicious memory access via direct memory access (DMA). We consider it out of scope if the entire cloud provider, who provides the VM infrastructure, is malicious.

A remote attacker does not have physical access to the hardware, so the following attacks are out of scope: physical tampering with the hardware platform, cold boot attacks [31], memory bus snooping, and physical memory access. These threats are better handled with on-site security and tamper-resistant hardware; cloud providers such as Google go to great lengths to ensure the physical security of their data centers and restrict physical access even for administrators [28]. We also do not defend against side-channel attacks in virtualized environments [39, 53, 65, 93, 94], or based on network I/O [10]. This is not unique to HypSec and it is the kernel's responsibility to obfuscate such patterns with defenses orthogonal to HypSec.

We assume a VM does not voluntarily reveal its own sensitive data whether on purpose or by accident. A VM can be compromised by a remote attacker that exploits vulnerabilities in the VM. 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 scope. However, attackers may try to attack other hosted VMs from a compromised VM for which we provide protection.

3 Design

HypSec introduces a new hypervisor design that reduces the TCB necessary to protect VM confidentiality and integrity while retaining full-fledged hypervisor functionality. We observe that many hypervisor functions can be supported without any access to VM data. For example, VM CPU register data is unnecessary for CPU scheduling. Based on this observation, HypSec leverages microkernel design principles to split a monolithic hypervisor into two parts, as depicted in Figure 1: a trusted and privileged corevisor with full access to VM data, and an untrusted and deprivileged hostvisor delegated with most hypervisor functionality. Unlike previous microkernel approaches [1, 13, 51], HypSec is designed

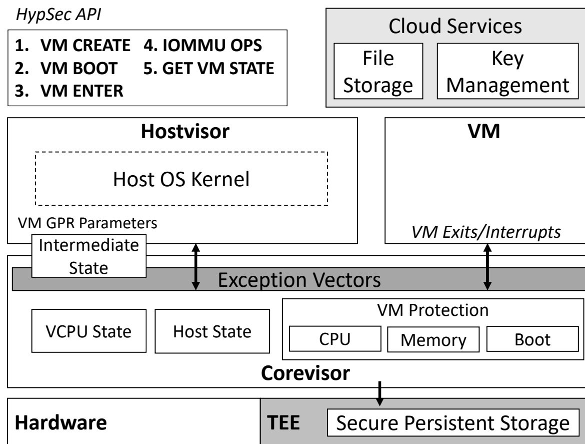

Figure 1: HypSec Architecture

specifically to restructure existing hypervisors with modest modifications as opposed to requiring a clean-slate redesign. Splitting the hypervisor this way results in a significantly smaller TCB that is still flexible enough to implement modern hypervisor features, as discussed in Section 4.

The corevisor is kept small by only performing VM data access control and hypervisor functions that require full access to VM data: secure VM boot, CPU virtualization, and page table management. With applications increasingly using secure communication channels to protect I/O data, HypSec takes an end-to-end approach to simplify its TCB and allows the hostvisor to provide I/O and interrupt virtualization. The hostvisor also handles other complex functions which do not need access to VM data, including resource management such as CPU scheduling and memory allocation. The hostvisor may even incorporate a full existing OS kernel to support its features.

HypSec leverages modern hardware virtualization support in a new way to enforce the hypervisor partitioning. HypSec runs the corevisor in a higher privileged CPU mode designed for running hypervisors, giving it full control of hardware, including virtualization hardware mechanisms such as nested page tables (NPTs).[1] The corevisor deprivileges the hostvisor and VM kernel by running them in a less privileged CPU mode. For example, in HypSec's implementation using ARM Virtualization Extensions (VE) shown in Figure 3, the corevisor runs in hypervisor (EL2) mode while the hostvisor and VM kernel run in a less privileged kernel (EL1) mode. The corevisor interposes on all exceptions and interrupts, enabling it to provide access control mechanisms that prevent the hostvisor from accessing VM CPU and memory data. For example, the corevisor has its own memory and uses NPTs to enforce memory isolation between the hostvisor, VMs, and itself. A compromised hostvisor or VM can neither control hardware virtualization mechanisms nor access corevisor memory and thus cannot disable HypSec.

HypSec Interface. As shown in Figure 1, the corevisor exposes a simple API to the hostvisor and interposes on all hostvisor and VM interactions to ensure secure VM execution throughout the lifecycle of a VM. The life of a VM begins when the hostvisor calls the corevisor's *VM CREATE* and *VM BOOT* calls to safely bootstrap it with a verified VM image. The hostvisor is deprivileged and cannot execute VMs. It must call *VM ENTER* to request the corevisor to execute a VM. When the VM exits execution because an interrupt or exception occurs, it traps to the corevisor, which examines the cause of the exit and if needed, will return to the hostvisor. The corevisor provides the *IOMMU OPS* API to device drivers in the hostvisor for managing the IOMMU, as discussed in Section 3.3. While the hostvisor has no access to VM data in CPU or memory, it may request the corevisor to provide an encrypted copy of VM data via the *GET VM STATE* hypercall API. The hostvisor can use the API to support virtualization features that require exporting VM data to disk or across the network, such as swapping VM memory to disk or VM management functions like VM snapshot and migration. The corevisor only uses encryption to export VM data. It never uses encryption, only access control, to protect VM data in CPU or memory.

[1] Intel's Extended Page Tables or ARM's stage 2 page tables.

3.1 Boot and Initialization

Corevisor Boot. HypSec ensures that the trusted corevisor binary is booted and the bootstrapping code itself is secure. To ensure only the trusted corevisor binary is booted, HypSec relies on Unified Extensible Firmware Interface (UEFI) firmware and its signing infrastructure with a hardware root of trust. The hostvisor and corevisor are linked as a single HypSec binary which is cryptographically ("digitally") signed by the cloud provider, similar to how OS binaries are signed by vendors like Red Hat or Microsoft. The HypSec binary is verified using keys in secure storage provided by the TEE, guaranteeing that only the signed binary can be loaded.

To ensure the bootstrapping code is secure, HypSec could implement it in the trusted corevisor, but does not. Bare-metal hypervisors implement bootstrapping, but this imposes a significant implementation and maintenance burden. The code must be manually ported to each different device, making it more difficult to support a wide range of systems. Instead, HypSec relies on the hostvisor bootstrapping code to install the corevisor securely at boot time since the hostvisor is initially benign. At boot time, the hostvisor initially has full control of the system to initialize hardware. The hostvisor installs the corevisor before entering user space; network and serial input services are not yet available, so remote attackers cannot compromise the corevisor's installation. After its installation, the corevisor gains full control of the hardware and subsequently deprivileges the hostvisor, ensuring the hostvisor can never control the hardware or access the corevisor's memory to disable HypSec. Using information provided at boot time, the corevisor is self-contained and can operate without any external data structures.

VM Boot. HypSec also guarantees the confidentiality and in-

tegrity of VM data during VM boot and initialization. HypSec keeps its TCB small by delegating complicated boot processes to the untrusted hostvisor, and verifying any loaded VM images in the corevsor before they are run. As shown in Figure 1, when a new VM is created, the hostvisor participates with the corevisor in a verified boot process. The hostvisor calls *VM CREATE* to request the corevisor to allocate VM state in corevisor memory, including an NPT and VCPU state, a per virtual CPU (VCPU) data structure. It then calls *VM BOOT* to request the corevisor to authenticate the loaded VM images. If successful, the hostvisor can then call *VM ENTER* to execute the VM. In other words, the hostvisor stores VM images and loads them to memory, avoiding implementing this complex procedure in the corevisor. The corevisor verifies the cryptographic signatures of VM images using public key cryptography, avoiding any shared secret between the user and HypSec.

Both the public keys and VM image signatures are stored in TEE secure storage prior to any attack, as shown in Figure 1. If the VM kernel binary is detached and can be mapped separately to memory, the hostvisor calls the corevisor to verify the image. If the VM kernel binary is in the VM disk image's boot partition, HypSec-aware virtual firmware bootstraps the VM. The firmware is signed and verified like VM boot images. The firmware then loads the signed kernel binary or a signed bootloader such as GRUB from the cleartext VM disk partition. The firmware then calls the corevisor to verify the VM kernel binary or bootloader. In the latter case, the bootloader verifies VM kernel binaries using the signatures on the virtual disk; GRUB already supports this. GRUB can also use public keys in the signed GRUB binary. The corevisor ensures only images it verified, either a kernel binary, virtual firmware, or a bootloader binary, can be mapped to VM memory. Finally, the corevisor sets the VM program counter to the entry point of the VM image to securely boot the VM.

As discussed in Section 3.5, HypSec expects that VM disk images are encrypted as part of an end-to-end encryption approach. HypSec ensures that any password or secret used to decrypt the VM disk is not exposed to the hostvisor. Common encrypted disk formats [6, 57] use user-provided passwords to protect the decryption keys. HypSec can store the encrypted key files locally or remotely using a cloud provider's key management service (KMS) [5, 58]. The KMS maintains a secret key which is preloaded by administrators into hosts' TEE secure storage. The corevisor decrypts the encrypted key file using the secret key, and maps the resulting password to VM memory, allowing VMs to obtain the password without exposing it to the hostvisor. The same key scheme is used for VM migration; HypSec encrypts and decrypts the VM state using the secret key from the KMS.

3.2 CPU

Hypervisors provide CPU virtualization by performing four main functions: handling traps from the VM; emulating privileged CPU instructions executed by the guest OS to ensure the hypervisor retains control of CPU hardware; saving and restoring VM CPU state, including GPRs and system registers such as page table base registers, as needed when switching among VMs and between a VM and the hypervisor; and scheduling VCPUs on physical CPUs. Hypervisors typically have full access to VM CPU state when performing any of these four functions, which can pose a problem for VM security if the hypervisor is compromised.

HypSec protects VM CPU state from the hostvisor while keeping its TCB small by restricting access to VM CPU state to the corevisor while delegating complex CPU functions that can be done without access to VM CPU state to the hostvisor. This is done by having the corevisor handle all traps from the VM, instruction emulation, and world switches between VMs and the hostvisor, all of which require access to VM CPU state. VCPU scheduling is delegated to the hostvisor as it can be done without access to VM CPU state.

The corevisor configures the hardware to route all traps from the VM, as well as interrupts as discussed in Section 3.4, to go to the corevisor, ensuring that it retains full hardware control. It also deprivileges the hostvisor to ensure that the hostvisor has no access to corevisor state. Since all traps from the VM go to the corevisor, the corevisor can trap and emulate CPU instructions on behalf of the VM. The corevisor multiplexes the CPU execution context between the hostvisor and VMs on the hardware. The corevisor maintains VCPU execution context in the VCPU state in-memory data structure allocated on *VM CREATE*, and maintains the hostvisor's CPU context in a similar *Host state* data structure; both states are only accessible to the corevisor. On VM exits, the corevisor first saves the VM execution context from CPU hardware registers to VCPU state, then restores the hostvisor's execution context from Host state to the CPU hardware registers. When the hostvisor calls to the corevisor to re-enter the VM, the corevisor first saves its execution context to Host state, then restores the VM execution context from VCPU state to the hardware. All saving and restoring of VM CPU state is done by the corevisor, and only the corevisor can run a VM.

The hostvisor handles VCPU scheduling, which can involve complex scheduling mechanisms especially for multiprocessors. For example, the Linux scheduler code alone is over 20K LOC, excluding kernel function dependencies and data structures shared with the rest of the kernel. VCPU scheduling requires no access to VM CPU state, as it simply involves mapping VCPUs to physical CPUs. The hostvisor schedules a VCPU to a physical CPU and calls to the corevisor to run the VCPU. The corevisor then loads the VCPU state to the hardware.

HypSec by default ensures that the hostvisor has no access to any VM CPU state, but sometimes a VM may execute instructions that requiring sharing values with the hostvisor that may be stored in general purpose registers (GPRs). For example, if the VM executes a hypercall that includes some parameters and

the hypercall is handled by the hostvisor, it will be necessary to pass the parameters to the hostvisor, and those parameters may be stored in GPRs. In these cases, the instruction will trap to the corevisor. The corevisor will identify the values that need to be passed to the hostvisor, then copy the values from the GPRs to an in-memory per VCPU intermediate VM state structure that is accessible to the hostvisor. Similarly, hostvisor updates to the intermediate VM state structure can be copied back to GPRs by the corevisor to pass values back to the VM. Only values from the GPRs explicitly identified by the corevisor for parameter passing are copied to and from intermediate VM state; values in other CPU registers are not accessible to the hostvisor.

The corevisor determines if and when to copy values from GPRs, and the GPRs from which to copy, based on the specific CPU instructions executed. The set of instructions are those used to execute hypercalls and special instructions provided by the architecture to access virtual hardware via model-specific registers (MSRs), control registers in the x86 instruction set, or memory-mapped I/O (MMIO). There are typically only a few specific CPU instructions that involve parameter passing to the hostvisor via GPRs, but the specific cases are architecture dependent.

For example, on ARM, HypSec copies selected GPRs to and from intermediate VM state for power management hypercalls to the virtual firmware interface and selected MMIO accesses to virtual hardware. For power management hypercalls, the guest kernel passes input parameters in GPRs, and the corevisor copies only those GPRs to intermediate VM state to make the parameters available to the hostvisor. Upon returning to the VM, the hostvisor provides output data as return values to the power management hypercalls, which the corevisor copies from intermediate VM state back to GPRs to make them available to the VM. As discussed in Sections 3.4 and 3.5, values stored and loaded in GPRs on MMIO accesses to the virtual interrupt controller interface or I/O devices are also copied between the selected GPRs and the intermediate VM state to make them available to the hostvisor.

3.3 Memory

Hypervisors provide memory virtualization by performing three main functions: memory protection to ensure VMs cannot access unauthorized physical memory, memory allocation to provide physical memory to VMs, and memory reclamation to reclaim physical memory from VMs. Other advanced memory management features may also be performed that build on these functions. All of these functions rely on NPTs. A guest OS manages the traditional page tables to map guest virtual memory addresses (gVA) to guest physical memory addresses (gPA). The hypervisor manages the NPTs to map from gPAs to host physical memory addresses (hPA) so it can virtualize and restrict a VM's access to physical memory. The hypervisor has full access to physical memory so it can manage VM memory either directly [11] or via a host OS kernel's [23]

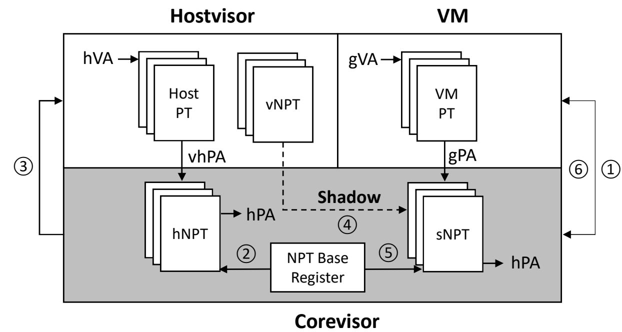

Figure 2: HypSec Memory Virtualization

memory management APIs. A compromised hypervisor or host OS kernel thus has unfettered access to VM memory and can read and write any data stored by VMs in memory.

HypSec protects VM memory from the hostvisor while keeping its TCB small by restricting access to VM memory to the corevisor while delegating complex memory management functions that can be done without access to actual VM data in memory to the hostvisor. The corevisor is responsible for memory protection, including configuring NPT hardware, while memory allocation and reclamation is largely delegated to the hostvisor. HypSec memory protection imposes an additional requirement, which is to also protect corevisor and VM memory from the hostvisor.

Memory Protection. The corevisor uses the NPT hardware in the same way as modern hypervisors to virtualize and restrict a VM's access to physical memory, but in addition leverages NPTs to isolate hostvisor memory access. The corevisor configures NPT hardware as shown in Figure 2. The hostvisor is only allowed to manage its own page tables (Host PT) and can only translate from host virtual memory addresses (hVAs) to what we call virtualized host physical memory addresses (vhPAs). vhPAs are then in turn translated to hPAs by the Host Nested Page Table (hNPT) maintained by the corevisor. The corevisor adopts a flat address space mapping; each vhPA is mapped to an identical hPA. The hostvisor, if granted access, is given essentially the same view of physical memory as the corevisor. The corevisor prevents the hostvisor from accessing corevisor and VM memory by simply unmapping the memory from the hNPT to make the physical memory inaccessible to the hostvisor. Any hostvisor accesses to corevisor or VM memory will trap to the corevisor, enabling the corevisor to intercept unauthorized accesses. Physical memory is statically partitioned between the hostvisor and corevisor, but dynamically allocated between the hostvisor and VMs as discussed below. The corevisor allocates NPTs from its own memory pool which is not accessible to the hostvisor. All VCPU state is also stored in corevisor memory.

The corevisor also protects corevisor and VM memory against DMA attacks [75] by retaining control of the IOMMU. The corevisor allocates IOMMU page tables from its memory and exports the *IOMMU OPS* API to device drivers in the

hostvisor to update page table mappings. The corevisor validates requests and ensures that attackers cannot control the IOMMU to access memory owned by itself or the VMs.

Memory Allocation. Memory allocation for VMs is largely done by the hostvisor, which can reuse memory allocation functions available in an integrated host OS kernel to dynamically allocate memory from its memory pool to VMs. Traditional hypervisors simply manage one NPT per VM. However, HypSec's memory model disallows the hostvisor from managing VM memory and therefore NPTs. The hostvisor instead manages an analogous Virtual NPT (vNPT) for each VM, and HypSec introduces a Shadow Nested Page Table (sNPT) managed by the corevisor for each VM as shown in Figure 2. The sNPT is used to manage the hardware by shadowing the vNPT. The corevisor multiplexes the hardware NPT Base Register between hNPT and sNPT when switching between the hostvisor and a VM.

Figure 2 also depicts the steps in HypSec's memory virtualization strategy. When a guest OS tries to map a gVA to an unmapped gPA, a nested page fault occurs which traps to the corevisor (step 1). If the corevisor finds that the faulted gPA falls within a valid VM memory region, it then points the NPT Base Register to hNPT (step 2) and switches to the hostvisor to allocate a physical page for the gPA (step 3). The hostvisor allocates a virtualized physical page identified by a vhPA and updates the entry in its vNPT corresponding to the faulting gPA with the allocated vhPA. Because the vhPA is mapped to an identical hPA, the hostvisor is able to implicitly manage host physical memory. The hostvisor then traps to the corevisor (step 4), which determines the faulting gPA and identifies the updates made by the hostvisor to the vNPT. The corevisor verifies the resulting vhPA is not owned by itself or other VMs, the latter by tracking ownership of physical memory using a unique VM identifier (VMID), and copies those updates to its sNPT. The corevisor unmaps the vhPA from the hNPT, so that the hostvisor no longer has access to the memory being allocated to the VM. The corevisor updates the NPT Base Register to point to the sNPT (step 5) and returns to the VM (step 6) so that the VM has access to the allocated memory identified by the hPA that is identical to the vhPA. Although possible, HypSec does not scrub pages allocated to VMs by the hostvisor. Guest OSes already scrub memory allocated from their free list before use for security reasons, so the hostvisor cannot allocate pages that contain malicious content to VMs.

HypSec's use of shadow page tables differs significantly from previous applications of it to collapse multi-level page tables down into what is supported by hardware [2, 11, 16, 52, 82]. In contrast, HypSec uses shadowing to protect hardware page tables, not virtualize them. The corevisor does not shadow guest OS updates in its page tables; it only shadows hostvisor updates to the vNPT. HypSec does not introduce additional traps from the VM for page table synchronization. Overshadow [16] maintains multiple shadow page tables for a given VM that provide different views (plaintext/encrypted) of physical memory to protect applications from an untrusted guest OS. In contrast, HypSec manages one shadow page table for each VM that provides a plaintext view of gPA to hPA. The shadowing mechanism in HypSec is also orthogonal to recent work [19] that uses shadow page tables to isolate kernel space memory from user space.

Memory Reclamation. HypSec supports VM memory reclamation in the hostvisor while preserving the privacy and integrity of VM data in memory in the corevisor. When a VM voluntarily releases memory pages, such as on VM termination, the corevisor returns the pages to the hostvisor by first scrubbing them to ensure the reclaimed memory does not leak VM data, then mapping them back to the hNPT so they are accessible to the hostvisor. To allow the hostvisor to reclaim VM memory pages without accessing VM data in memory, HypSec takes advantage of ballooning [82]. Ballooning is widely supported in common hypervisors, so only modest effort is required in HypSec to support this approach. A paravirtual "balloon" device is installed in the VM. When the host is low on free memory, the hostvisor requests the balloon device to inflate. The balloon driver inflates by getting pages from the free list, thereby increasing the VM's memory pressure. The guest OS may therefore start to reclaim pages or swap its pages to the virtual disk. The balloon driver notifies the corevisor about the pages in its balloon that are ready to be reclaimed. The corevisor then unmaps these pages from the VM's sNPT, scrubs the reclaimed pages to ensure they do not leak VM data, and assigns the pages to the hostvisor, which can then treat them as free memory. Deflating the balloon releases memory pressure in the guest, allowing the guest to reclaim pages.

HypSec also safely allows the hostvisor to swap VM memory to disk when it feels memory pressure. The hostvisor uses *GET VM STATE* to get access to the encrypted VM page before swapping it out. Later, when the VM page is swapped in, the corevisor unmaps the swapped-in page from hNPT, decrypts the page, and maps it back to the VM's sNPT.

Advanced VM Memory Management. HypSec by default ensures that the hostvisor has no access to any VM memory, but sometimes a VM may want to share its memory, after encrypting it, with the hostvisor. HypSec provides the `GRANT_MEM` and `REVOKE_MEM` hypercalls which can be explicitly used by a guest OS to share its memory with the hostvisor. As described in Section 3.5, this can be used to support paravirtualized I/O of encrypted data in which a memory region owned by the VM has to be shared between the VM and hostvisor for communication and efficient data copying. The VM passes the start of a guest physical frame number (GFN), the size of the memory region, and the specified access permission to the corevisor via the two hypercalls. The corevisor enforces the access control policy by controlling the memory region's mapping in hNPT. Only VMs can use these two hypercalls, so the hostvisor cannot use it to request access to arbitrary VM pages.

HypSec can support advanced memory virtualization features such as merging similar memory pages, KSM [46]

in Linux, by splitting the work into the simple corevisor functions which require direct access to VM data, and the more complicated hostvisor functions which do not require access to VM data. For example, to support KSM, the hostvisor requests the corevisor for the hash values of a VM's memory pages and maintains the data structure in its address space to support the merging algorithm. The corevisor validates the hostvisor's decision for the pages to be merged, updates the corresponding VM's sNPT, and scrubs the freed page before granting the hostvisor access. While KSM does not provide the hostvisor or other VMs direct access to a VM's memory pages, it can be used to leak some information such as whether the contents of memory pages are the same across different VMs. To avoid this kind of information leakage, HypSec disables KSM support by default.

3.4 Interrupts

Hypervisors trap and handle physical interrupts to retain full control of the hardware while virtualizing interrupts for VMs. Accesses to the interrupt controller interface can by done via MSRs or MMIO. Hypervisors provide a virtual interrupt controller interface and trap and emulate VM access to the interface. Virtual devices in the hypervisors can also raise interrupts to the interface. However, giving hypervisors full control of hardware poses a problem for VM security if the hypervisor is compromised.

To protect against a compromised hostvisor, the corevisor configures the hardware to route all physical interrupts and trap all accesses to the interrupt controller to the corevisor, ensuring that it retains full hardware control. However, to simplify its TCB, HypSec delegates almost all interrupt functionality to the hostvisor, including handling physical interrupts and providing the virtual interrupt controller interface. Before entering the hostvisor to handle interrupts, the corevisor protects all VM CPU and memory state, as discussed in Sections 3.2 and 3.3.

The hostvisor has no access to and requires no VM data to handle physical interrupts. However, VM accesses to the virtual interrupt controller interface involve passing parameters between the VM and the hostvisor since the hostvisor provides the interface. On ARM, this is done using only MMIO via the intermediate state structure discussed in Section 3.2. On an MMIO write to interrupt controller interface, the VM passes the value to be stored in a GPR. The write traps to the corevisor, which identifies the instruction and memory address as corresponding to the interrupt controller interface. The corevisor copies the value to be written from the GPR to the intermediate VM state to make the value available to the hostvisor. For example, when the guest OS in the VM sends an IPI to a destination VCPU by doing an MMIO write to the virtual interrupt controller interface, the identifier of the destination VCPU is passed to the hostvisor by copying the value from the respective GPR to the intermediate VM state. Similarly, on an MMIO read from the interrupt controller interface, the read traps to the corevisor, which identifies the instruction and memory address as corresponding to the interrupt controller interface. The corevisor copies the value from the intermediate VM state updated by the hostvisor to the GPR the VM is using to retrieve the value, updates the PC of the VM to skip the faulting instruction, and returns to the VM.

3.5 Input/Output

To ease the burden of supporting a wide range of virtual devices, modern hypervisors often rely on an OS kernel and its existing device drivers to support I/O virtualization, which significantly increase the hypervisor TCB. Similar to previous work [16,33], HypSec assumes an end-to-end I/O security approach, relying on VMs for I/O protection. VMs can leverage secure communication channels such as TLS/SSL for network communications and full disk encryption for storage. This allows the corevisor to relax its I/O protection requirements, simplifying the TCB. HypSec offloads the support of I/O virtualization to the untrusted hostvisor. Since I/O data is already encrypted by VMs, a compromised hostvisor would at most gain access to encrypted I/O data which would not reveal VM data.

HypSec, like other modern hypervisors, supports all three classes of I/O devices: emulated, paravirtualized, and passthrough devices; the latter two provide better I/O performance. Emulated I/O devices are typically supported by hypervisors using trap-and-emulate to handle both port-mapped I/O (PIO) and MMIO operations. In both cases, HypSec configures the hardware to trap the operations to the corevisor which hides all VM data other than actual I/O data and then allows the hostvisor to emulate the operation. For example, to support MMIO, the corevisor zeroes out the mappings for addresses in the VM's sNPT corresponds to virtual device I/O regions. Any subsequent MMIO accesses from the VM result in a memory access fault that traps to the corevisor. The corevisor then securely supports MMIO accesses as discussed in Section 3.4. We assume security aware users disable the use of emulated devices such as the serial port, keyboard, or mouse to avoid leaking private information to a compromised hostvisor.

Paravirtualized devices require that a front-end driver in the VM coordinate with a back-end driver in the hypervisor; the two drivers communicate through shared memory asynchronously. HypSec allows back-end drivers to be installed as part of the untrusted hostvisor. To support shared memory communication, the front-end driver is modified to use `GRANT_MEM` and `REVOKE_MEM` hypercalls to identify the shared data structure and I/O memory buffers as accessible to the hostvisor back-end driver. Since the I/O data is encrypted, hostvisor access to the I/O memory buffers does not risk VM data.

Passthrough devices are assigned to a VM and managed by the guest OS. To support passthrough I/O, HypSec configures the hardware to trap sensitive operations such as Message Signaled Interrupt (MSI) configuration in BAR to trap to the corevisor for secure emulation, while granting VMs

	Xen	KVM	HypSec
Boot and Initialization			
Secure Boot	○	○	○
Secure VM Boot	⊖	⊖	○
CPU			
VM Symmetric Multiprocessing (SMP)	○	○	○
VCPU Scheduling	○	○	○
Memory			
Dynamic Allocation	○	○	○
Memory Reclamation - Ballooning	○	○	○
Memory Reclamation - Swapping	○	○	⊘
DMA	○	○	○
Same Page Merging	○	○	⊘
Interrupts Virtualization			
Hardware Assisted	○	○	○
I/O			
Device Emulation	○	○	○
Paravirtualized (PV)	○	○	○
Device Passthrough	○	○	○
VM Management			
Multi-VM	○	○	○
VM Snapshot	○	○	○
VM Restore	○	○	○
VM Migration	○	○	○

Table 1: Supported features comparison. (○ = Supported, ⊖ = Not applicable, ⊘ = Not implemented.)

direct access to the non-sensitive device memory region. The corevisor controls the IOMMU to enforce inter-device isolation, and ensures the passthrough device can only access the VM's own I/O buffer. Since we assume the hardware is not malicious, passthrough I/O can be done securely on HypSec.

4 Implementation

We demonstrate how HypSec can improve the security of existing commodity hypervisors by applying our approach to the mainline Linux KVM/ARM [22, 23] hypervisor, given ARM's increasing popularity in server systems [4, 63, 87]. Table 1 compares commodity hypervisors with the current HypSec implementation, showing that this security improvement comes without compromising on hypervisor features. Since KVM is a hosted hypervisor tightly integrated with a host OS kernel, retrofitting KVM also demonstrates the viability of HypSec in supporting an entire OS kernel as part of the hostvisor.

HypSec requires a higher-privileged CPU mode, nested page tables for memory virtualization, and an IOMMU for DMA protection. These requirements are satisfied by the ARM architecture. ARM VE provides Hyp (EL2) mode for hypervisors that is strictly more privileged than user (EL0) and kernel (EL1) modes. EL2 has its own execution context defined by register and control state, and can therefore switch the execution context of both EL0 and EL1 in software. Thus, the hypervisor can run in an address space that is isolated from EL0 and EL1. ARM VE provides stage 2 page tables

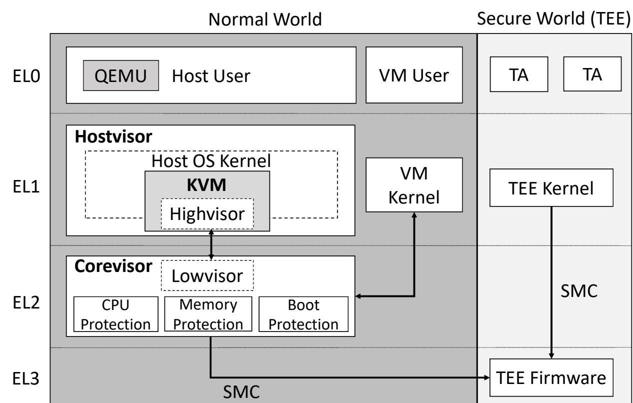

Figure 3: HypSec on KVM/ARM

which are nested level page tables configured in EL2 that affect software in EL0 and EL1. ARM provides the System Memory Management Unit (SMMU) [8] to protect DMA.

HypSec's corevisor is initialized at machine bootup and runs in EL2 to fully control the hardware. HypSec's code is embedded in the Linux kernel binary, which is verified and loaded via UEFI. The kernel boots in EL2 and installs a trap handler to later return to EL2. The kernel then enters EL1 so the hostvisor can bootstrap the machine. The hostvisor allocates resources and configures the hardware for the corevisor. The hostvisor then makes a hypercall to the corevisor in EL2 to enable HypSec.

The HypSec ARM implementation leverages KVM/ARM's split into an EL2 lowvisor and an EL1 highvisor to support the ARM virtualization architecture. This is done because EL2 is necessary for controlling hardware virtualization features, but Linux and KVM are designed to run in kernel mode, EL1. Thus, the lowvisor manages hardware virtualization features and VM-hypervisor switches, while the highvisor contains the rest of the hypervisor and Linux. However, the lowvisor cannot protect VM data if any other part of Linux or KVM are compromised; with KVM/ARM, the Linux host has unfettered access to all VM data.

As shown in Figure 3, the corevisor encapsulates the KVM lowvisor and runs in EL2. The hostvisor, including the KVM highvisor and its integrated Linux OS kernel, runs in EL1. The hostvisor has no access to EL2 registers and cannot compromise the corevisor or disable VM protection. HypSec leverages ARM VE to force VM operations that need hypervisor intervention to trap into EL2. The corevisor either handles the trap directly to protect VM data or world switches the hardware to EL1 to run the hostvisor if more complex handling is necessary. When the hostvisor finishes its work, it makes a hypercall to trap to EL2 so the corevisor can securely restore the VM state to hardware. The corevisor interposes on every switch between the VM and hostvisor, thus protecting the VM's execution context. Our implementation ensures that the hostvisor cannot invoke arbitrary corevisor functions via hypercalls.

HypSec leverages ARM VE's stage 2 memory translation support to virtualize VM memory and prevent accesses to protected physical memory. The corevisor routes stage 2 page faults to EL2 and rejects illegal hostvisor and VM memory accesses. The corevisor allocates hNPTs and VMs' sNPTs from its protected physical memory and manages the page tables.

To secure DMA, the corevisor uses trap-and-emulate on hostvisor accesses to the SMMU. HypSec ensures only the corevisor has access to the SMMU hardware. The corevisor manages the SMMU page tables in its protected memory to ensure hostvisor devices cannot access corevisor or VM memory, and devices assigned to the VM can only access VM memory.

HypSec leverages the hardware features from VGIC and KVM/ARM's existing support to virtualize interrupts. Our implementation supports ARM GIC 2.0. HypSec relies on QEMU and KVM's virtual device support for I/O virtualization. Our implementation supports emulated devices via MMIO, paravirtualized devices via virtio [67], and passthrough devices. For virtio, we modified front-end drivers to use `GRANT/REVOKE_MEM` hypercalls to share memory with the hostvisor back-end drivers. To support passthrough devices, HypSec configures the hardware to grant VMs direct access to them. We modified the front-end virtio-balloon driver to notify the corevisor about the pages allocated for the balloon device. The corevisor scrubs and assigns these pages to the hostvisor, allowing it to safely reclaim memory as needed. Our current implementation does not support page swapping and KSM, which are both left as future work.

HypSec supports secure VM boot using ARM TrustZone-based TEE frameworks such as OP-TEE [61] to store the signatures and keys securely. HypSec tasks QEMU to load the VM boot images to VM memory, but the corevisor requires QEMU to participate with its verified boot process. The corevisor retrieves the VM boot image signatures and the user public key from TrustZone for verifying the VM images remapped to its address space. The corevisor uses Ed25519 [62] to verify the boot images. HypSec builds the VM's stage 2 page table with mappings to the verified VM boot image. If the verification fails, HypSec stops the VM boot process. The same scheme can also verify VM firmware and other binaries. HypSec also retrieves the encrypted password which protects the VM's encrypted disk from either TrustZone or from the cloud provider's key management service. A small AES implementation [45] ported to run in EL2 performs the decryption. We include only two small yet sufficient crypto libraries in EL2 to keep the TCB small. This limits the number of crypto algorithms, but avoids including comprehensive but excessively large crypto libraries such as OpenSSL. HypSec leverages AES to support encrypted VM migration and snapshot, and ensures only encrypted VM data is exposed to the hostvisor.

HypSec's hardware requirements can also be satisfied on Intel's x86 architecture by using Virtual Machine Extensions (VMX) [35] and the IOMMU. Existing x86 hypervisors can be retrofitted to run the corevisor in VMX root operation which allows control of virtualization features for deprivileging the hostvisor. The hostvisor runs in VMX non-root operation to provide resource management and virtual I/O. The corevisor protects VM execution state by managing a Virtual-Machine Control Structure (VMCS) per CPU, and VM memory by using Extended Page Tables (EPT) and controlling the IOMMU.

5 Security Analysis

We present five properties of the HypSec architecture, then discuss how their combination provides a set of security properties regarding HypSec's ability to protect the integrity and confidentiality of VM data.

Property 1. *HypSec's corevisor is trusted during the system's lifetime against remote attackers.*

HypSec leverages hardware secure boot to ensure only the signed and trusted HypSec binary can be booted. This prevents an attacker from trying to boot or reboot the system to force it to load a malicious corevisor. The hostvisor securely installs the corevisor during the boot process before network access and serial input service are available. Thus, remote attackers cannot compromise the hostvisor prior to or during the installation of the corevisor. The corevisor protects itself after initialization. It runs in a privileged CPU mode using a separate address space from the hostvisor and the VMs. The corevisor has full control of the hardware including the virtualization features that prevent attackers from disabling its VM protection. The corevisor also protects its page tables so an attacker cannot map executable memory to the corevisor's address space.

Property 2. *HypSec ensures only trusted VM images can be booted on VMs.*

Based on Property 1, the trusted corevisor verifies the signatures of the VM images loaded to VM memory before they are booted. The public keys and signatures are stored using TEE APIs for persistent secure storage. A compromised hostvisor therefore cannot replace a verified VM with a malicious one.

Property 3. *HypSec isolates a given VM's memory from all other VMs and the hostvisor.*

Based on Property 1, HypSec prevents the hostvisor and a given VM from accessing memory owned by other VMs. The corevisor tracks ownership of physical pages and enforces inter-VM memory isolation using nested paging hardware. A compromised hostvisor could control a DMA capable device to attempt to access VM memory or compromise the corevisor. However, the corevisor controls the IOMMU and its page tables, so the hostvisor cannot access corevisor or VM memory via DMA. VM pages reclaimed by the hostvisor are scrubbed by the corevisor, so they do not leak VM data. HypSec also protects the integrity of VM nested page tables. The corevisor manages shadow page tables for VMs. The MMU can only walk the shadow page tables residing in a protected memory region only accessible to the corevisor. The corevisor manages

and verifies updates to the shadow page tables to protect VM memory mappings.

Property 4. *HypSec protects a given VM's CPU registers from the hostvisor and all other VMs.*

HypSec protects VM CPU registers by only granting the trusted corevisor (Property 1) full access to them. The hostvisor cannot access VM registers without permission. Attackers cannot compromise VM execution flow since only the corevisor can update VM registers including program counter (PC), link register (LR), and TTBR.

Property 5. *HypSec protects the confidentiality of a given VM's I/O data against the hostvisor and all other VMs assuming the VM employs an end-to-end approach to secure I/O.*

Based on Properties 3 and 4, HypSec protects any I/O encryption keys loaded to VM CPU registers or memory, so a compromised hostvisor cannot steal these keys to decrypt encrypted I/O data. The same protection holds against other VMs.

Property 6. *HypSec protects the confidentiality and integrity of a given VM's I/O data against the hostvisor and all other VMs assuming the VM employs an end-to-end approach to secure I/O and the I/O can be verified before it permanently modifies the VM's I/O data.*

Using the reasoning in Property 5 with the additional assumption that I/O can be verified before it permanently modifies I/O data, HypSec also protects the integrity of VM I/O data, as any tampered data will be detected and can be discarded. For example, a network endpoint receiving I/O from a VM over an encrypted channel with authentication can detect modifications of the I/O data by any intermediary such as the hostvisor. If verification is not possible, then HypSec cannot prevent compromises of data availability that result in destruction of I/O data, which can affect data integrity. As an example, HypSec cannot prevent an attacker from arbitrarily destroying a VM's I/O data by blindly overwriting all or parts of a VM's local disk image; both the VM's availability and integrity are compromised since the data is destroyed. Secure disk backups can protect against permanent data loss.

Property 7. *Assuming a VM takes an end-to-end approach for securing its I/O, HypSec protects the confidentiality of all of the VM's data against a remote attacker, including if the attacker compromises any other VMs or the hostvisor itself.*

Based on Properties 1, 3, and 4, a remote attacker cannot compromise the corevisor, and any compromise of the hostvisor or another VM cannot allow the attacker to access VM data stored in CPU registers or memory. This combined with Property 5 allows HypSec to ensure the confidentiality of all of the VM's data.

Property 8. *Under the assumption that a VM takes an end-to-end approach for securing its I/O and I/O can be verified before it permanently modifies any VM data, HypSec protects the integrity of all of the VM's data against a remote attacker, including if the attacker compromises any other VMs or the hostvisor itself.*

Based on Properties 1, 3, and 4, HypSec ensures a remote attacker cannot compromise the corevisor, and that any compromise of the hostvisor or another VM cannot allow the attacker to access VM data stored in CPU registers or memory, thereby preserving VM CPU and memory data integrity. This combined with Property 6 allows HypSec to ensure the integrity of all of the VM's data.

Property 9. *If the hypervisor is benign and responsible for handling I/O, HypSec protects the confidentiality and integrity of all of the VM's data against any compromises of other VMs.*

If both the hostvisor and corevisor are not compromised and the hostvisor is responsible for handling I/O, then the confidentiality and integrity of a VM's I/O data will be protected against other VMs. This combined with Properties 3 and 4 allows HypSec to ensure the confidentiality and integrity of all of the VM's data. This guarantee is equivalent to what is provided by a traditional hypervisor such as KVM.

6 Experimental Results

We quantify the performance and TCB of HypSec compared to other approaches, and demonstrate HypSec's ability to protect VM confidentiality and integrity. All of our experiments were run on ARM server hardware with VE support, specifically 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, and an IOMMU (SMMU-401) to support control over DMA devices and direct device assignment. The hardware did not support ARM Virtualization Host Extensions [20, 21]. For client-server experiments, the clients ran on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The clients and the server communicated via a 10 GbE unsaturated network connection.

To provide comparable measurements across the approaches, we kept the software environments across all platforms as uniform as possible. We compared KVM with our HypSec modifications versus standard KVM, both in Linux 4.18 with QEMU 2.3.50. In both cases, KVM was configured with its standard VHOST virtio network, and with `cache=none` for its virtual block storage devices [30, 47, 77]. All hosts and VMs used Ubuntu 16.04 with the same Linux 4.18 kernel, except for HypSec changes. All VMs used paravirtualized I/O, typical of cloud infrastructure deployments such as Amazon EC2.

We ran benchmarks both natively on the hardware and in VMs. Each physical or VM instance was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. This involved two configurations: (1) native Linux capped at 4 cores and 12 GB RAM, and (2) a VM using KVM with 8 cores and 16 GB RAM with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. We measure multi-core configurations to reflect real-world server deployments. For VMs,

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, and then return to the VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices supported by the hypervisor OS kernel.
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 the hypervisor OS user space.
Virtual IPI	Issue a virtual IPI from a VCPU to another VCPU running on a different PCPU, both PCPUs executing VM code. Measures time between sending the virtual IPI until the receiving VCPU handles it, a frequent operation in multi-core OSes.

Table 2: Microbenchmarks

we pinned each VCPU to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU. All of the host's device interrupts and processes were assigned to run on other PCPUs. For client-server benchmarks, the clients ran natively on Linux and used the full hardware available.

6.1 Microbenchmark Results

We first ran microbenchmarks to quantify the cost of low-level hypervisor operations. We used the KVM unit test framework [48] listed in Table 2 to measure the cost of transitioning between the VM and the hypervisor, initiating a VM-to-hypervisor OS kernel I/O request, emulating user space I/O with QEMU, and sending virtual IPIs. We slightly modified the test framework to measure the cost of virtual IPIs and to obtain cycle counts on ARM to ensure detailed results by configuring the VM with direct access to the cycle counter.

Microbenchmark	KVM	HypSec
Hypercall	2,896	3,202
I/O Kernel	3,831	4,563
I/O User	9,288	10,704
Virtual IPI	8,816	10,047

Table 3: Microbenchmark Measurements (cycles)

Table 3 shows the microbenchmarks measured in cycles for both standard KVM and HypSec. HypSec introduces roughly 5% to 19% overhead over KVM. HypSec does not increase the number of traps in the operations we measured. The corevisor interposes on exisiting traps to add additional logic to protect VM data, so the cost is relatively small. The I/O Kernel, I/O User, and Virtual IPI measurements show relatively higher overhead than Hypercall on HypSec because of the cost involved to secure data transfers between the VM and hostvisor for I/O and interrupt virtualization.

Name	Description
Kernbench	Compilation of the Linux 4.9 kernel using `allnoconfig` for ARM with GCC 5.4.0.
Hackbench	`hackbench` [66] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	`netperf` v2.6.0 [41] running `netserver` on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_-MAERTS (throughput), and TCP_RR (latency).
Apache	`Apache` v2.4.18 Web server running `ApacheBench` [80] v2.3 on the remote client, which measures number of handled requests per second when serving the 41 KB `index.html` file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.25 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.7.24) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 4: Application Benchmarks

6.2 Application Workload Results

Next we ran real application workloads to evaluate HypSec compared to standard KVM. Table 4 lists the workloads which are a mix of widely-used CPU and I/O intensive benchmarks. To evaluate VM performance with end-to-end I/O protection, we used five configurations: (1) Native unmodified Linux host kernel without Full Disk Encryption, (2) Unmodified KVM and guest kernel without FDE (KVM), (3) Unmodified KVM and guest kernel with FDE (KVM-FDE), (4) HypSec and paravirtualized guest kernel without FDE (HypSec), (5) HypSec and paravirtualized guest kernel with FDE (HypSec-FDE). For FDE, we use dm-crypt to create a LUKS-encrypted root partition of the VM filesystem. We measure with and without FDE to separately quantify its extra costs. We leveraged the TLS/SSL support in Apache and MySQL and evaluated VM performance on HypSec with end-to-end network encryption.

Figure 4 shows the relative overhead of executing in a VM in our four VM configurations compared to natively. We normalize the results so that a value of 1.00 means the same performance as native hardware. Lower numbers mean less overhead. The performance on real application workloads shows modest overhead overall for HypSec compared to standard KVM. The overhead for HypSec in many cases is less than 10%, even with FDE enabled.

The worst overhead for HypSec occurs for some of the network workloads. Our current implementation of the front-end network virtio driver applies grant/revoke hypercalls on a per transaction basis to make data available to the back-end driver in the hostvisor. Therefore, HypSec's performance is sub-optimal in workloads where the virtio driver can batch multiple transactions without trapping to the hypervisor, most notably in TCP_MAERTS. TCP_MAERTS measures the bandwidth of a VM sending packets to a client. The virtio driver batches multiple sends to avoid traps to hypervisor, while in the implementation measured in the paper, the driver traps additionally

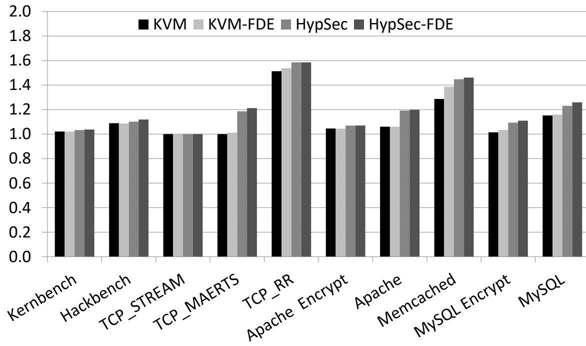

Figure 4: Application Benchmark Performance

on sending network data, resulting in higher overhead. Note that other network workloads such as TCP_STREAM have negligible overhead as the granularity at which the additional traps happen is large enough that the performance impact is negligible. To avoid extra traps to the hypervisor, our implementation can be optimized by batching the effect of the grant/revoke calls at the same level of granularity as used by the virtio driver to batch multiple transactions. This is an area of future work.

6.3 TCB Implementation Complexity

We ran cloc [24] against our implementation's corevisor to measure the TCB, as shown in Table 5. The total is roughly 8.5K LOC of which just under 4.5K LOC is from the Ed25519 and AES crypto libraries. The rest of the HypSec TCB is less than 4.1K LOC, consisting of mostly CPU/memory protection and existing KVM lowvisor code. Overall, we modified or added a total of 8,695 LOC in the mainline Linux kernel v4.18 across both the corevisor and hostvisor. More than 1.3K LOC were in existing Linux files, and around 7.3K LOC were in new files for HypSec, including around 4.5K LOC in the crypto libraries and slightly less than 2.8K LOC for corevisor functions. Finally, less than 70 LOC were added to QEMU to support secure boot and VM migration. These results demonstrate that HypSec can retrofit existing hypervisors with modest implementation effort.

For comparison purposes, we also used cloc to measure KVM's TCB in Linux v4.18 and Xen v4.9 for ARM64 support when running Linux v4.18 on Dom0, shown in Table 6. For KVM, we counted its LOC for the specific Linux v4.18 codebase running on the ARM64 server used in our experiments. KVM's massive TCB with access to VM data consists of more than 1.8M LOC and includes QEMU, the KVM module, core Linux functions such as CPU scheduling, ARM64 architectural support, and the device drivers used on the server.

To provide a fair comparison, we assumed the same threat model for each system and that VMs encrypt their I/O. Even under this assumption, KVM, including its I/O kernel code, must be entirely trusted to protect VM data since a compromised KVM can steal encryption keys from VM CPU and memory

Components	LOC
Ed25519 library	4,074
AES library	403
CPU protection	1,883
Memory protection	1,727
Secure boot	232
Helper	247
HypSec TCB	8,566

Table 5: HypSec TCB

Hypervisor	LOC
HypSec	8,566
KVM	1,857,575
Xen	71,604
Xen + Dom0	2,054,756

Table 6: TCB size comparison with KVM and Xen

state. By retrofitting KVM with HypSec to protect VM CPU and memory against the rest of the KVM codebase, we show that the TCB of KVM can be reduced by more than 200 times.

Using the same assumption, Xen's TCB should include both its hypervisor code and Dom0, a special privileged VM used to reuse existing Linux drivers to support I/O for user VMs. Although Dom0 is not part of the hypervisor, Xen provides it with a management interface that can request the hypervisor to dump entire VM state, thereby giving a compromised Dom0 full access to encryption keys. Xen's resulting TCB including Dom0, which has a full copy of Linux, is therefore larger than KVM and hundreds of times larger than HypSec. If we conservatively assume features in Xen's management stack that expose VM state such as VM dump and migration are disabled, so that we can exclude Dom0 from Xen's TCB and only count Xen ARM hypervisor code in EL2, Xen's TCB is then 71K LOC as listed in Table 6. This is roughly an order of magnitude larger than HypSec because Xen still has to do its own bootstrapping, CPU and memory resource management, and completely support memory and interrupt virtualization.

We estimated HypSec's TCB for an equivalent x86 implementation, assuming HypSec is also applied to KVM Linux v4.18 for x86 hardware with VMX support. We ran cloc against the C files that encapsulate the KVM functions for CPU and memory virtualization to conservatively measure HypSec's TCB size. The total is less than 27K LOC. Although the TCB size for HypSec on x86 would be larger than HypSec on ARM, we believe the resulting TCB on x86 would still result in a substantial reduction as KVM's TCB on x86 is also larger than on ARM, at roughly 10M LOC including x86 device drivers; this is an area of future work.

6.4 Evaluation of Practical Attacks

We evaluated HypSec's effectiveness against a compromised hostvisor by analyzing CVEs and identifying the cases where HypSec protects VM data despite any compromise, assuming an equivalent implementation of HypSec for x86 platforms. We analyzed CVEs related to Linux/KVM, which are listed in Tables 7 and 8. The CVEs consider two cases: a malicious VM who exploits KVM functions supported by the hostvisor, and an unprivileged host user who exploits bugs in Linux/KVM. Among the selected CVEs, 16 of them are x86-specific, one is specific to ARM, while the rest are independent of architecture. An attacker's goal is to exploit these CVEs to obtain

Bug	Description	KVM	HypSec
CVE-2015-4036	Memory Corruption: Array index error in hostvisor.	No	Yes
CVE-2013-0311	Privilege Escalation: Improper handling of descriptors in vhost driver.	No	Yes
CVE-2017-17741	Info Leakage: Stack out-of-bounds read in hostvisor.	No	Yes
CVE-2010-0297	Code Execution: Buffer overflow in I/O virtualization code.	No	Yes
CVE-2014-0049	Code Execution: Buffer overflow in I/O virtualization code.	No	Yes
CVE-2013-1798	Info Leakage: Improper handling of invalid combination of operations for virtual IOAPIC.	No	Yes
CVE-2016-4440	Code Execution: Mishandling of virtual APIC state.	No	Yes
CVE-2016-9777	Privilege Escalation: Out-of-bounds array access using VCPU index in interrupt virtualization code.	No	Yes
CVE-2015-3456	Code Execution: Memory corruption in virtual floppy driver allows VM user to execute arbitrary code in hostvisor.	No	Yes
CVE-2011-2212	Privilege Escalation: Buffer overflow in the virtio subsystem allows guest to gain privileges to the host.	No	Yes
CVE-2011-1750	Privilege Escalation: Buffer overflow in the virtio subsystem allows guest to gain privileges to the host.	No	Yes
CVE-2015-3214	Code Execution: Out-of-bound memory access in QEMU leads to memory corruption.	No	Yes
CVE-2012-0029	Code Execution: Buffer overflow allows VM users to execute arbitrary code in QEMU	No	Yes
CVE-2017-1000407	Denial-of-Service: VMs crash hostvisor by flooding the I/O port with write requests.	No	No
CVE-2017-1000252	Denial-of-Service: Out-of-bounds value causes assertion failure and hypervisor crash.	No	No
CVE-2014-7842	Denial-of-Service: Bug in KVM allows guest users to crash its own OS.	No	No
CVE-2018-1087	Privilege Escalation: Improper handling of exception allows guest users to escalate their privileges to its own OS.	No	No

Table 7: Selected Set of Analyzed CVEs - from VM

Bug	Description	KVM	HypSec
CVE-2009-3234	Privilege Escalation: Kernel stack buffer overflow resulting in ret2usr [43].	No	Yes
CVE-2010-2959	Code Execution: Integer overflow resulting in function pointer overwrite.	No	Yes
CVE-2010-4258	Privilege Escalation: Improper handling of get_fs value resulting in kernel memory overwrite.	No	Yes
CVE-2009-3640	Privilege Escalation: Improper handling of APIC state in hostvisor.	No	Yes
CVE-2009-4004	Privilege Escalation: Buffer overflow in hostvisor.	No	Yes
CVE-2013-1943	Privilege Escalation, Info Leakage: Mishandling of memory slot allocation allows host users to access hostvisor memory.	No	Yes
CVE-2016-10150	Privilege Escalation: Use-after-free in hostvisor.	No	Yes
CVE-2013-4587	Privilege Escalation: Array index error in hostvisor.	No	Yes
CVE-2018-18021	Privilege Escalation: Mishandling of VM register state allows host users to redirect hostvisor execution.	No	Yes
CVE-2016-9756	Info Leakage: Improper initialization in code segment resulting in information leakage in hostvisor stack.	No	Yes
CVE-2013-6368	Privilege Escalation: Mishandling of APIC state in hostvisor.	No	Yes
CVE-2015-4692	Memory Corruption: Mishandling of APIC state in hostvisor.	No	Yes
CVE-2013-4592	Denial-of-Service: Host users cause memory leak in hostvisor.	No	No

Table 8: Selected Set of Analyzed CVEs - from host user

hostvisor privileges and compromise VM data. The CVEs related to our threat model could result in information leakage, privilege escalation, code execution, and memory corruption in Linux/KVM. While KVM does not protect VM data against any of these compromises, HypSec protects against all of them. HypSec does not guarantee availability and cannot protect against CVEs that allow VMs or host users to cause denial of service in the hostvisor. Vulnerabilities that allow unprivileged guest users to attack their own VMs like CVE-2014-7842 and CVE-2018-1087 are unrelated to HypSec's threat model; protection against CVEs of these types is an area of future work.

We also executed attacks representative of information leakage to show that HypSec protects VM data even if an attacker has full control of the hostvisor. First, we simulated an attacker trying to read or modify VMs' memory pages. We added a hook to KVM which modifies a page that a targeted gVA maps to. As expected, the compromised KVM (without HypSec) successfully modified the VM page. Using HypSec, the same attack causes a trap to the corevisor which rejects the invalid memory access.

Second, we simulated a host that tries to tamper with a VM's nested page table by redirecting a gPA's NPT mapping to host-owned pages. This is in contrast to the prior attack of modifying VM pages, but shares the same goal of accessing VM data in memory. We added a hook to the nested page fault handler in KVM; the hook allocates a new zero page in the host OS's address space, which in a real attack could contain arbitrary code data. The hook associates a range of a VM's gPAs with this zero page. As expected, this attack succeeds in KVM but fails in HypSec. First, the attacker has no access to the sNPT walked by the MMU. Second, the corevisor synchronizes the vNPT to sNPT mapping on the gPA's initial fault during VM boot, so malicious vNPT modifications do not propagate to sNPT.

7 Related Work

The idea of retrofitting a commodity hypervisor with a smaller core was inspired by KVM/ARM's split-mode virtualization [22, 23], which introduced a thin software layer to enable Linux KVM to make use of ARM hardware virtualization extensions without significant changes to Linux, but did nothing to reduce the hypervisor TCB. HypSec builds on this work to leverage ARM hardware virtualization support to run the corevisor with special hardware privileges to protect VM

data against a compromised hostvisor. More recently, Nested Kernel [25] used the idea of retrofitting a small TCB into a commodity OS kernel, FreeBSD, to intercept MMU updates to enforce kernel code integrity. Both HypSec and Nested Kernel retrofit commodity system software with a small TCB that mediates accesses to critical hardware resources and strengthens system security guarantees with modest implementation and performance costs. Nested Kernel focuses on a different threat model and does not protect against vulnerabilities in existing kernel code in part because both its TCB and untrusted components run at the highest hardware privilege level. In contrast, HypSec deprivileges the hostvisor and uses its TCB to provide data confidentiality and integrity even in the presence of hypervisor vulnerabilities in the hostvisor.

Bare-metal hypervisors often claim a smaller TCB as an advantage over hosted hypervisors, but in practice, the aggregate TCB of the widely-used Xen [11] bare-metal hypervisor includes Dom0 [18, 92] and therefore can be no smaller than hosted hypervisors like KVM. Some work thus focuses on reducing Xen's attack surface by redesigning Dom0 [15, 18, 59]. Unlike HypSec, these approaches cannot protect a VM against a compromised Xen or Dom0. We believe Xen can be restructured using HypSec by moving resource management, interrupt virtualization, and other hardware-specific dependencies, along with Dom0, into a hostvisor to further reduce Xen's TCB to protect VM data.

Microhypervisors [32, 74] take a microkernel approach to build clean-slate hypervisors from scratch to reduce the hypervisor TCB. For example, NOVA [74] moves various aspects of virtualization such as CPU and I/O virtualization to user space services. The virtualization services are trusted but instantiated per VM so that compromising them only affects the given VM. Others simplify the hypervisor to reduce its TCB by removing [72] or disabling [60] virtual device I/O support in hypervisors, or partitioning VM resources statically [42, 73]. Although a key motivation for both microhypervisors and HypSec is to reduce the size of the TCB, HypSec does not require a clean-slate redesign, and supports existing full-featured commodity hypervisors without removing important hypervisor features such as I/O support and dynamic resource allocation while preserving confidentiality and integrity of VM data even if the hostvisor is compromised.

HyperLock [86], DeHype [88], and Nexen [70] focus on deconstructing existing monolithic hypervisors by segregating hypervisor functions to per VM instances. While this can isolate an exploit of hypervisor functions to a given VM instance, if a vulnerability is exploitable in one VM instance, it is likely to be exploitable in another as well. Nexen builds on Nested Kernel to retrofit Xen in this manner, though it does not protect against vulnerabilities in its shared hypervisor services. In contrast to HypSec, these systems focus on availability and do not fully protect the confidentiality and integrity of VM data against a compromised hypervisor or host OS.

CloudVisor [92] uses a small, specialized host hypervisor to support nested virtualization and protect user VMs against an untrusted Xen guest hypervisor, though Xen modifications are required. CloudVisor encrypts VM I/O and memory but does not fully protect CPU state, contrary to its claims of "providing both secrecy and integrity to a VM's states, including CPU states." For example, the VM program counter is exposed to Xen to support I/O. As with any nested virtualization approach, performance overhead on application workloads is a problem. Furthermore, CloudVisor does not support widely used paravirtual I/O. CloudVisor has a smaller TCB by not supporting public key cryptography, making key management problematic. In contrast, HypSec protects both CPU and memory state via access control, not encryption, making it possible to support full-featured hypervisor functionality such as paravirtual I/O. HypSec also does not require nested virtualization, avoiding its performance overhead.

To protect user data in virtualization systems, others enable and require VM support for specialized hardware such as Intel SGX [36] or ARM TrustZone. Haven [12] and S-NFV [71] use Intel SGX to protect application data but unlike HypSec, cannot protect the whole VM including the guest OS and applications against an untrusted hypervisor. Although HypSec relies on a TEE to support key management, it fundamentally differs from other approaches which extensively use TEEs for much more than storing keys. Others [34, 96] run a security monitor in ARM TrustZone and rely on ARM IP features such as TrustZone Address Space Controller to protect VMs. vTZ [34] virtualizes TrustZone and protects the guest TEE against an untrusted hypervisor, but does not protect the normal world VM. HA-VMSI [96] protects the normal world VM against a compromised hypervisor but supports limited virtualization features. In contrast, HypSec protects the entire normal world VM against an untrusted hypervisor without requiring VMs to use specialized hardware. HypSec leverages ARM VE to trap VM exceptions to EL2 while retaining hypervisor functionality. Others [40, 78, 90] propose hardware-based approaches to protect VM data in CPU and memory against an untrusted hypervisor. However, without actual hardware implementations, these works implement the proposed changes by modifying either Xen [40] or QEMU [90], or on a simulator [78]. Some of them [40, 78] do not support commodity hypervisors. In contrast, HypSec leverages existing hardware features to protect virtual machine data and supports KVM on ARM server hardware.

Recent architectural extensions [3, 37] proposed hardware support on x86 for encrypted virtual machines. Fidelius [89] leverages AMD's SEV (Secure Encrypted Virtualization) [3] to protect VMs. Unlike these encryption-based approaches, HypSec primarily uses access control mechanisms.

Some projects focus on hardening the hypervisor to prevent exploitation. They improve hypervisor security by either enforcing control flow integrity [84] or measuring runtime hypervisor integrity [9, 26]. These approaches can be applied to HypSec to further strengthen VM security. XMHF [81]

verifies the memory integrity of its hypervisor codebase but supports single VM with limited virtualization features. Verification of HypSec's TCB is an area of future work.

Various projects extend a trusted hypervisor to protect software within VMs, including protecting applications running on an untrusted guest OS in the VM [16, 17, 33, 55, 91], ensuring kernel integrity and protecting against rootkits and code injection attacks or to isolate I/O channels [64, 69, 83, 85, 95], and dividing applications and system components in VMs then relying on the hypervisor to safeguard interactions among secure and insecure components [27, 54, 76, 79]. Overshadow [16] and Inktag [33] have some similarities with HypSec in that they use a more trusted hypervisor component to protect against untrusted kernel software. Overshadow and Inktag also assume applications use end-to-end encrypted network I/O, though they protect file I/O by replacing it with memory-mapped I/O to encrypted memory. HypSec has three key differences with these approaches. First, instead of memory encryption, HypSec primarily uses access control, which is more lightweight and avoids the need to emulate functions that are problematic when memory is encrypted. Second, instead of instrumenting or emulating complex system calls, HypSec relies on hardware virtualization mechanisms to interpose on hardware events of interest. Finally, instead of protecting against guest OS exploits, HypSec protects against hypervisor and host OS exploits, which none of the other approaches do.

8 Conclusions

We have created HypSec, a new approach to hypervisor design that reduces the TCB necessary to protect virtual machines. HypSec decomposes a monolithic hypervisor into a small, trusted corevisor and untrusted hostvisor, the latter containing the vast majority of hypervisor functionality including an entire host operating system kernel. The corevisor leverages hardware virtualization support to execute at a higher privilege level and provide access control mechanisms to restrict hostvisor access to VM data. It can be simple because it only needs to perform basic CPU and memory virtualization. When VMs use secure I/O channels, HypSec can protect the confidentiality and integrity of all VM data. We have demonstrated that HypSec can support existing commodity hypervisors by retrofitting KVM/ARM. The resulting TCB is orders of magnitude less than the original KVM/ARM. HypSec provides strong security guarantees to VMs with only modest performance overhead for real application workloads.

Acknowledgments

Steve Bellovin, Christoffer Dall, and Nathan Dautenhahn provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1717801 and CNS-1563555.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference (USENIX Summer 1986)*, pages 93–112, Atlanta, GA, June 1986.

[2] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 2–13, San Jose, CA, Oct. 2006.

[3] Advanced Micro Devices. Secure Encrypted Virtualization API Version 0.16. https://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf, Feb. 2018.

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

[5] Amazon Web Services, Inc. AWS Key Management Service (KMS). https://aws.amazon.com/kms/, May 2019.

[6] ArchWiki. dm-crypt. https://wiki.archlinux.org/index.php/dm-crypt, Apr. 2018.

[7] ARM Ltd. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, Apr. 2009.

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

[9] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 38–49, Chicago, IL, Oct. 2010.

[10] 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.

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

[12] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, pages 267–283, Broomfield, CO, Oct. 2014.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 267–283, Copper

Mountain, CO, Dec. 1995.

[14] Business Wire. Research and Markets: Global Encryption Software Market (Usage, Vertical and Geography) - Size, Global Trends, Company Profiles, Segmentation and Forecast, 2013 - 2020. https://www.businesswire.com/news/home/20150211006369/en/Research-Markets-Global-Encryption-Software-Market-Usage, Feb. 2015.

[15] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service Cloud Computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 253–264, Raleigh, NC, Oct. 2012.

[16] 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)*, pages 2–13, Seattle, WA, Mar. 2008.

[17] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *Proceedings of the 25th International Conference on Supercomputing (ICS 2011)*, pages 108–119, Tucson, AZ, May 2011.

[18] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 189–202, Cascais, Portugal, Oct. 2011.

[19] J. Corbet. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/, Nov. 2017.

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

[21] 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)*, pages 221–234, Santa Clara, CA, July 2017.

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

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

[24] A. Danial. cloc: Count Lines of Code. https://github.com/AlDanial/cloc, May 2019.

[25] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 191–206, Istanbul, Turkey, Mar. 2015.

[26] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng. Dancing with Wolves: Towards Practical Event-driven VMM Monitoring. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 83–96, Xi'an, China, Apr. 2017.

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

[28] Google. Google Cloud Security and Compliance Whitepaper - How Google protects your data. https://static.googleusercontent.com/media/gsuite.google.com/en//files/google-apps-security-and-compliance-whitepaper.pdf, Sept. 2017.

[29] Google. HTTPS encryption on the web – Google Transparency Report. https://transparencyreport.google.com/https/overview, Apr. 2018.

[30] S. Hajnoczi. An Updated Overview of the QEMU Storage Stack. https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_hajnoczi.pdf, June 2011.

[31] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, pages 45–60, San Jose, CA, July 2008.

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

[33] 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)*, pages 265–278, Houston, TX, Mar. 2013.

[34] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and Haibing. vTZ: Virtualizing ARM Trustzone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 2017)*, pages 541–556, Vancouver, BC, Canada, Aug. 2017.

[35] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 325462-044US, Aug. 2012.

[36] Intel Corporation. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.

[37] Intel Corporation. Intel Architecture Memory Encryption Technologies Specification. https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf, Dec. 2017.

[38] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 11889-1:2015 - Information technology – Trusted platform module library. https://www.iso.org/standard/66510.html, Sept. 2016.

[39] 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)*, pages 591–604, San Jose, CA, May 2015.

[40] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural Support for Secure Virtualization Under a Vulnerable Hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, pages 272–283, Porto Alegre, Brazil, Dec. 2011.

[41] R. Jones. Netperf. https://github.com/HewlettPackard/netperf, June 2018.

[42] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA 2010)*, pages 350–361, Saint-Malo, France, June 2010.

[43] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 459–474, Bellevue, WA, Aug. 2012.

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

[45] kokke. kokke/tiny-aes-c: Small portable aes128/192/256 in c. https://github.com/kokke/tiny-AES-c, 2018.

[46] KVM Contributors. Kernel Samepage Merging. https://www.linux-kvm.org/page/KSM, July 2015.

[47] KVM Contributors. Tuning KVM. http://www.linux-kvm.org/page/Tuning_KVM, May 2015.

[48] KVM Contributors. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests, May 2019.

[49] S. Landau. Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations. *IEEE Security and Privacy*, 11(4):54–63, July 2013.

[50] Let's Encrypt. Let's encrypt stats - let's encrypt. https://letsencrypt.org/stats/, Apr. 2018.

[51] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[52] J. 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)*, pages 201–217, Shanghai, China, Oct. 2017.

[53] 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)*, pages 605–622, San Jose, CA, May 2015.

[54] 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 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, pages 1607–1619, Denver, CO, Oct. 2015.

[55] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pages 143–158, Oakland, CA, May 2010.

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

[57] Microsoft. BitLocker. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, Jan. 2018.

[58] Microsoft Azure. Key Vault - Microsoft Azure. https://azure.microsoft.com/en-in/services/key-vault/, May 2019.

[59] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 151–160, Seattle, WA, Mar. 2008.

[60] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Hypervisors Without Massive Re-engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012)*, pages 141–154, Bern, Switzerland, Apr. 2012.

[61] OP-TEE. Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2017.

[62] orlp. Ed25519. https://github.com/orlp/ed25519, 2017.

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

[64] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, pages 1–20, Cambridge, MA, Sept. 2008.

[65] 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 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Chicago, IL, Nov. 2009.

[66] R. Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, Jan. 2008.

[67] R. Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

[68] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, Nov. 1984.

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

[70] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium (NDSS 2017)*, San Diego, CA, Feb. 2017.

[71] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV:

Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security 2016)*, pages 45–48, New Orleans, LA, Mar. 2016.

[72] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, pages 121–130, Washington, DC, Mar. 2009.

[73] Siemens. jailhouse - Linux-based partitioning hypervisor. https://github.com/siemens/jailhouse, May 2019.

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

[75] 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)*, pages 21–41, Heraklion, Crete, Greece, July 2013.

[76] 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)*, pages 2–13, Raleigh, NC, Oct. 2012.

[77] SUSE. Performance Implications of Cache Modes. https://www.suse.com/documentation/sles11/book_kvm/data/sect1_3_chapter_book_kvm.html, Sept. 2016.

[78] J. Szefer and R. B. Lee. Architectural Support for Hypervisor-secure Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 437–450, London, England, UK, Mar. 2012.

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

[80] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html, Apr. 2015.

[81] 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)*, pages 430–444, San Francisco, CA, May 2013.

[82] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 181–194, Boston, MA, Dec. 2002.

[83] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. Secpod: A Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 2015)*, pages 347–360, Santa Clara, CA, July 2015.

[84] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pages 380–395, Oakland, CA, May 2010.

[85] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 545–554, Chicago, IL, Nov. 2009.

[86] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012)*, pages 127–140, Bern, Switzerland, Apr. 2012.

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

[88] C. Wu, Z. Wang, and X. Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, Feb. 2013.

[89] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 441–453, Vienna, Austria, Feb. 2018.

[90] Y. Xia, Y. Liu, and H. Chen. Architecture Support for Guest-transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA 2013)*, pages 246–257, Shenzhen, China, Feb. 2013.

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

[92] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 203–216, Cascais, Portugal, Oct. 2011.

[93] 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)*, pages 305–316, Raleigh, NC, Oct. 2012.

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

[95] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP 2014)*, pages 308–323, San Jose, CA, May 2014.

[96] M. Zhu, B. Tu, W. Wei, and D. Meng. HA-VMSI: A Lightweight Virtual Machine Isolation Approach with Commodity Hardware for ARM. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 242–256, Xi'an, China, Apr. 2017.