Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud

Xinyang Ge*
Microsoft Research
and Databricks
xinyang.ge@databricks.com

Hsuan-Chi Kuo
University of Illinois at
Urbana-Champaign
hckuo2@illinois.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

ABSTRACT

Despite the recent exponential growth in cloud adoption, businesses that handle sensitive data (e.g., health and financial sectors) are hesitant to migrate their on-premises IT infrastructure to the public cloud due to the lack of trust on the cloud provider. Confidential computing aims to move the cloud provider out of the trusted computing base. New hardware features such as AMD's SEV-SNP can run a full virtual machine (VM) with confidentiality and integrity protection against the cloud. However, there exist challenges in supporting legacy operating systems and enforcing security policies (e.g., firewalls) in confidential VMs.

In this paper, we present Hecate[1], an L1 hypervisor that runs *inside* a confidential VM enabled by SEV-SNP. Hecate can support legacy operating systems by running them in a nested VM and enforce various security policies on the nested VM based on the virtualization boundary. The key challenge in designing Hecate is that it cannot rely on the untrusted L0 hypervisor for nested virtualization. To solve it, we repurpose SEV-SNP's newly added privilege dimension called Virtual Machine Privilege Levels (VMPLs) to enable virtualization for a single nested VM.

We have built a prototype of Hecate based on the Linux KVM virtualization stack. Our prototype is capable of running MS-DOS, FreeBSD and vanilla Linux without any modification. It also supports security checks on the nested VM such as network firewalls and kernel integrity. When compared with a regular, non-confidential VM, the nested VM enabled by Hecate can achieve a throughput between 57% and 85% for real-world applications such as the Nginx web server and the MySQL database.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Virtualization and security**.

KEYWORDS

confidential computing; virtualization; AMD SEV-SNP

*This work was done while the author worked at Microsoft Research.
[1]Hecate is a Greek goddess of the mist that hides the mythical world from mortals.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '22, November 7–11, 2022, Los Angeles, CA, USA.
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560592

ACM Reference Format:
Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. 2022. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3548606.3560592

1 INTRODUCTION

Despite the recent exponential growth of the cloud adoption, businesses that handle sensitive data, such as those in the financial and health sectors, are hesitant to migrate their on-premises IT infrastructure to the public cloud due to the lack of the trust on the cloud provider. Confidential computing aims to move the cloud provider out of the trusted computing base (TCB) to enable these businesses to migrate their on-premises workloads to the cloud. It achieves the security goal by creating a hardware-based trusted execution environment (TEE) that is encrypted and isolated from the rest of the software stack managed by the cloud provider. In particular, the recently launched 3rd-generation AMD EPYC CPUs are capable of running a full virtual machine (VM) inside a TEE and protecting its confidentiality and integrity against a potentially malicious hypervisor owned by an untrusted cloud provider.

While a confidential VM is a promising primitive to host sensitive on-premises workloads, it has two major limitations. First, a confidential VM does not provide the VM-level backward compatibility. It requires non-trivial changes in the guest OS kernel to accommodate architectural differences and run securely in the new environment. For example, the guest OS in a confidential VM has to manage the state of its physical address space (e.g., private pages for security-sensitive computation and shared pages for I/O), and it also has to be implemented defensively whenever interacting with the untrusted hypervisor (e.g., hypervisor-injected interrupts). While some confidential workload only requires user-mode compatibility (e.g., containers), many on-premises workloads have strict VM-level compatibility requirements that must be supported in order to migrate to confidential VMs. For example, Microsoft SQL Server 2014 can only run on legacy Windows versions up to Windows Server 2019 [10], and it is unlikely for either Microsoft or a third party to make the required kernel changes to support running a legacy Windows in a confidential VM. Second, an on-premises IT infrastructure usually enforces a variety of security policies on its workloads. For instance, network firewalls are usually deployed to stop incoming attacks and outgoing leaks. It is challenging to enforce security policies on confidential VMs because anything outside of a confidential VM is not trusted.

To overcome these limitations of confidential VMs, our key idea is to run an L1 hypervisor *inside* a confidential VM, and then run an unmodified, regular VM image in a nested manner. Such an L1

hypervisor solves both limitations simultaneously. For compatibility, the L1 hypervisor can hide the architectural differences of a confidential VM and present the nested VM with a regular machine abstraction. For security, the L1 hypervisor can use the virtualization boundary *within* the confidential VM to enforce security policies on the nested VM. In addition, the L1 hypervisor can shield the nested VM from the untrusted L0 hypervisor since the legacy OS in the nested VM is not designed to run defensively against a malicious hypervisor. A confidential VM customer can trust the L1 hypervisor because it is under the customer's control, while the L0 hypervisor is not trusted because it is owned by the untrusted cloud provider.

A key challenge in implementing an L1 hypervisor inside a confidential VM is that we cannot rely on the untrusted L0 hypervisor to virtualize the hardware support (e.g., AMD SVM and Intel VMX) required by nested virtualization [14]. In this paper, we present Hecate, the first L1 hypervisor that can run inside a confidential VM. The enabling technology of Hecate is a new privilege dimension added to confidential VMs on AMD CPUs called Virtual Machine Privilege Levels (VMPLs). Every VMPL has its own user and kernel spaces (ring 0-3), and different VMPLs share the same guest physical address space with different permission views.

Intuitively, VMPLs allow us to isolate Hecate's state from the nested VM which is capable of running an entire OS. Unfortunately, VMPLs are not designed for running a full-fledged hypervisor as they lack key mechanisms required by a typical virtualization stack. For example, different VMPLs share the same guest-physical-to-host-physical address mappings configured by the L0 hypervisor (i.e., nested page tables), and there is no additional level of address translation between VMPLs. Shadow page tables are commonly used before the introduction of hardware-assisted nested paging, but they are not applicable here because a high-privileged VMPL cannot intercept a low-privileged VMPL's access to the `cr3` register which stores the physical address of the page table root.

Our key observation is that the primary purpose of building an L1 hypervisor inside a confidential VM is for compatibility and security rather than resource multiplexing. Therefore, we can simplify the problem by supporting only a *single* nested VM. Specifically, we reserve the lower address range for the nested VM so that it has an illusion of owning the entire memory starting from (guest) physical address 0. We run Hecate at a higher address range, and isolate it from the nested VM by configuring VMPL permissions. We enable "trap-and-emulate" in Hecate based on new hardware features introduced in SEV-SNP that allow code running in VMPL0 to receive exceptions from other VMPLs and access their CPU contexts directly. We further protect the nested VM which is not designed with an untrusted L0 hypervisor in mind by fully mediating the interactions between them.

We have built a prototype of Hecate based on the Linux KVM virtualization stack. By construction, the Hecate hypervisor is capable of running any OS that runs in a vanilla KVM VM, and we show that Hecate can run MS-DOS 6.22, FreeBSD 12 and Linux 5.11 without requiring any guest modification. We evaluate the performance of real-world server workload (e.g., MySQL, memcached and Nginx) running in the nested VM on Hecate, and show that it can achieve a throughput between 57% and 85% of the same software stack running in a regular, non-confidential VM.

2 AMD SECURE ENCRYPTED VIRTUALIZATION

In 2016, AMD introduced Secure Encrypted Virtualization (SEV) [1, Chapter 15.34] to isolate a virtual machine from an untrusted hypervisor. SEV achieves the isolation by automatically encrypting a VM's memory with a per-VM key inaccessible to the hypervisor. AMD extended such protection to the VM's register state in SEV-ES [1, Chapter 15.35] (released in 2017), and added the integrity protection of the VM's memory in SEV-SNP [1, Chapter 15.36] (released in 2021). In the rest of this section, we discuss key architectural features in SEV that are relevant to this work. We omit the exact versions (SEV/SEV-ES/SEV-SNP) when the features were added for brevity.

2.1 VM Save Area

A traditional hypervisor implements the hardware abstraction for VMs by *trapping* critical instructions of a VM (e.g., accessing a system-wide control register) and *emulating* the operation properly. To protect a VM's state, SEV adds a VM Save Area (VMSA) that is encrypted from the hypervisor, and automatically saves/restores the VM's state when the VM's execution is trapped/resumed.

However, the hypervisor needs to access a VM's state to perform any emulation. Therefore, SEV adds a new exception vector called VMM Communication Exception (#VC). The #VC exception occurs when the executed guest instruction would have caused a VM exit so that the VM has a chance to selectively expose its state to the hypervisor for emulation. A VM can expose its selected state via the guest-hypervisor communication block (GHCB) that is shared between the VM and the hypervisor. For example, when a VM executes `WRMSR` and triggers a #VC, it only needs to expose `EAX`, `ECX` and `EDX` registers to the hypervisor because these three registers are operands of that instruction and sufficient for emulation. AMD standardizes the GHCB format to allow a confidential VM to interoperate with any supporting hypervisor [6].

2.2 Virtual Machine Privilege Levels

AMD SEV-SNP adds a new privilege dimension called Virtual Machine Privilege Levels (VMPLs). There are a total of four VMPLs (0-3) where a lower number indicates a higher privilege. Each VMPL has its own ring 0-3 and hence its own user and kernel modes. AMD introduces VMPLs to "provide hardware isolated abstraction layers within a VM for additional security controls, as well as assistance with managing communication with the hypervisor" [8].

Different VMPLs share the same guest physical memory but have different permission views. VMPL0 has full permissions enabled over the guest physical memory—readable, writable, user executable and supervisor executable. It can grant a subset of its permissions to lower-privileged VMPLs, and a higher-privileged VMPL always has a more permissible view of the guest physical memory.

Each VMPL has its own VMSA, and the CPU state associated with each VMPL is automatically saved and restored by the hardware when switching VMPLs. A higher-privileged VMPL can access the VMSA of a lower-privileged VMPL. This enables a higher-privileged VMPL to control certain behaviors of a lower-privileged VMPL. We give two examples that are relevant to this work. First, it allows a higher-privileged VMPL to inject interrupts and exceptions to a

lower-privileged VMPL. Second, it allows a higher-privileged VMPL to enable *Reflect #VC* for a lower-privileged VMPL. When Reflect #VC is enabled for a VMPL, its execution no longer triggers a #VC exception, but instead causes a special VM exit to the hypervisor which can further reflect such exit to the higher-privileged VMPL for handling. Given that a higher-privileged VMPL can access the state of a lower-privileged VMPL, it does not require explicit state sharing as it would for the hypervisor.

VMPLs are switched by the hypervisor. The untrusted hypervisor can switch VMPLs at arbitrary time, or refuse to switch to a VMPL at all, or even attempt to run two different VMPLs concurrently.

2.3 Virtual Top-of-Memory

The initial SEV implementation designates one bit (called C-bit) on the guest page table entry to determine if a guest physical page should be encrypted. This requires non-trivial changes to the MMU code of the guest kernel.

SEV-SNP adds an alternative mechanism called Virtual Top-of-Memory (vTOM). vTOM allows a confidential VM to specify a guest physical address below which all the guest memory is encrypted, and it can be configured per VMPL. vTOM is meant to ease the engineering efforts when porting a new kernel to SEV-enabled confidential VMs.

3 THREAT MODEL

We aim to build an L1 hypervisor to run inside a confidential VM enabled by AMD's SEV-SNP technology. Our threat model is in line with the threat model assumed by confidential computing: we assume the attacker has the full control over everything outside of confidential VMs, including the L0 hypervisor. Specifically, the attacker is able to schedule virtual CPUs arbitrarily, inject spurious interrupts, swap memory pages of a confidential VM, lie about the current time, report conflicting information about virtual CPUs, and randomly drop VMEXIT events.

We assume the legacy VM we intend to run on top of the L1 hypervisor is buggy and can become hostile after being compromised. The compromised nested VM can attempt to escape its virtualization boundary and further compromise the L1 hypervisor. It is worth noting that, without proper protections, the nested VM does not necessarily have to have a traditional bug before it can be compromised. Given no legacy OS is written with a malicious hypervisor in mind, the untrusted hypervisor can simply inject an interrupt while the legacy OS has interrupt disabled to trigger race conditions which should have never occurred on a correctly-behaving (virtual) hardware.

We leave hardware attacks and side-channel attacks out of scope. We assume the SEV-SNP hardware will perform correctly according to its architectural specification including the memory encryption and integrity protection of confidential VMs. We recognize that the L0 hypervisor can launch various side-channel attacks [31, 43] against confidential VMs to infer secrets. We consider defenses against side-channel attacks as orthogonal to our work. The focus of our work is on providing compatibility and security policy enforcement to confidential VMs.

We assume denial-of-service attacks are possible with the untrusted L0 hypervisor controlling the host. For instance, the hypervisor can do so by simply shutting down the physical machine.

In summary, we trust the L1 hypervisor, but not the L0 hypervisor or the nested VM. This is not because the L1 hypervisor has fewer bugs than the L0 hypervisor, but because the L1 hypervisor is controlled by a confidential VM customer while the L0 hypervisor is controlled by an untrusted cloud provider.

4 DESIGN

4.1 Challenges

We present two key challenges in building an L1 hypervisor to run inside a confidential VM.

4.1.1 *Untrusted L0 Hypervisor.*

A traditional nested virtualization stack relies on the L0 hypervisor to virtualize the hardware support, such as Intel VMX and AMD SVM, to the L1 hypervisor. This requires the L0 hypervisor to correctly forward nested VMEXITs, shadow key data structures (e.g., VMCB on AMD SVM), and track updates to the nested page tables used by the L1 hypervisor. However, given that the L0 hypervisor is no longer trusted in our setup, the L1 hypervisor cannot rely on the L0 hypervisor to provide the needed nested virtualization support.

In addition, the traditional nested virtualization stack does not need to isolate the nested VM from the L0 hypervisor because the L0 hypervisor is assumed trusted. In Hecate, the L1 hypervisor running inside a confidential VM must protect the nested VM from the untrusted L0 hypervisor. This is particularly challenging because no legacy OS is designed with a malicious (virtual) hardware in mind, and the untrusted L0 hypervisor running at the highest privilege can interfere with its execution in various ways. To protect the nested VM from the L0 hypervisor, the L1 hypervisor must ensure that the untrusted L0 hypervisor cannot interfere with the execution of the nested VM except for denial-of-service, which our threat model explicitly permits. It must also protect the nested VM from accidentally leaking secret information to the untrusted L0 hypervisor despite the presence of latent bugs.

4.1.2 *Lack of Virtualization Support.*

Modern hypervisors rely on a set of hardware virtualization features to implement the hardware abstraction for VMs securely and efficiently. For example, both Intel VMX and AMD SVM add a per-vCPU control structure to manage the virtual CPU state, and a nested paging mechanism to translate guest physical addresses (GPA) to host physical addresses (HPA) for memory isolation.

While the VMPLs added by SEV-SNP is a promising feature for state isolation as each VMPL has its own separate CPU state, it is not designed for running a full-fledged L1 hypervisor. In particular, modern hypervisors rely on nested paging to achieve isolation and give each VM an illusion that it owns the entire memory space, but there is no additional address translation between VMPLs. In fact, all VMPLs see the same guest-physical-to-host-physical address translation configured by the L0 hypervisor. In addition, there is no hardware mechanism for a high-privileged VMPL to intercept selected events from a low-privileged VMPL. Therefore, the L1 hypervisor cannot use the traditional shadow page tables to virtualize the memory management unit (MMU) because the L1 hypervisor

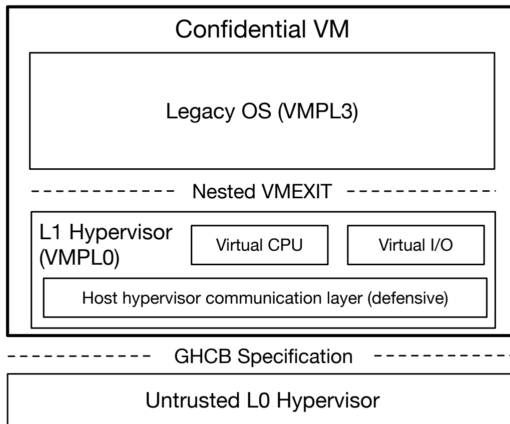

Figure 1: The architecture of Hecate.

cannot force a VMEXIT when the nested VM updates its page table root (i.e., `cr3`).

4.2 Architecture

The architecture of Hecate is shown in Figure 1. Hecate runs the trusted L1 hypervisor in the most-privileged VMPL0 and supports a single nested VM running in the least-privileged VMPL3. To protect the L1 hypervisor from the nested VM, Hecate configures the memory permission in such a way that VMPL0 has access to the entire address space while VMPL3 is only allowed to access the nested VM's own memory range. To protect the nested VM from the untrusted L0 hypervisor, the L1 hypervisor fully mediates the interactions between them.

To support nested VMExit, the L1 hypervisor enables *Reflect #VC* for VMPL3. Whenever the nested VM executes an instruction that would have required the hypervisor interaction, instead of generating a #VC exception, it would immediately cause a VMExit to the L0 hypervisor which then forwards the event to the L1 hypervisor. The L1 hypervisor has access to both the CPU and memory state of the nested VM, so it can perform the emulation properly.

Hecate relies on the L1 hypervisor to provide system services such as timer interrupts. Since the L0 hypervisor is untrusted, the L1 hypervisor must be defensive in its host hypervisor communication layer. Hecate combines both hardware and software restrictions to achieve this goal. For example, SEV-SNP adds an interrupt restriction mechanism to prohibit the hypervisor from injecting interrupts other than a newly-added vector (#HV). Hecate enables the restriction so that it can ensure that any arisen exception (e.g., page fault) is genuine and originated from its own execution rather than spuriously generated by an untrusted L0 hypervisor. However, Hecate still needs to handle the situation in which an injected interrupt occurs while the interrupt is disabled (`EFLAGS.IF=0`). We discuss interfaces that require careful engineering to defend against an untrusted L0 hypervisor in §5.1. For the rest of the section, we assume the L1 hypervisor in Hecate is self-defensive and focus on remaining challenges.

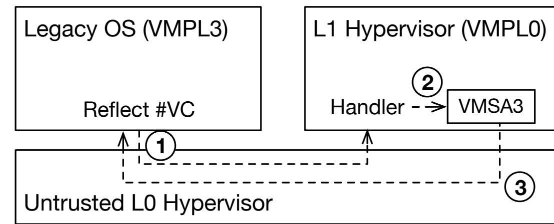

Figure 2: Nested VMExit flow in Hecate.

4.3 CPU: Nested VMExit

Hecate virtualizes the CPU for the nested VM via trap and emulate based on nested VMExits. An example VMExit flow is shown in Figure 2. A nested VMExit occurs when the nested VM executes an instruction that requires emulations from the hypervisor, such as the `CPUID` instruction that probes the available features on the current virtual CPU. To implement nested VMExits, the L1 hypervisor enables Reflect #VC for VMPL3 so that a VMExit in the nested VM would cause a trap to the L0 hypervisor instead of raising a #VC exception as it normally would in an SEV-SNP VM. When this happens, the SEV-SNP hardware will save VMPL3's CPU state to its VMSA that is encrypted and integrity-protected before passing the control to the L0 hypervisor. Next, the L0 hypervisor forwards the VMExit event to the L1 hypervisor running at VMPL0. The L1 hypervisor examines the VMExit reason (a field in the VMSA of VMPL3), performs the emulation, and then updates the relevant states in the VMSA (e.g., general-purpose registers) before resuming VMPL3.

This flow is similar to the nested VMExit implementation in the traditional nested virtualization. However, one important difference here is that the L0 hypervisor is no longer trusted. We identify three possible attacks the untrusted L0 hypervisor can launch. First, it can lie about a non-existent VMExit to VMPL0 to trick the L1 hypervisor to handle the same VMExit more than once. To defend against this attack, the L1 hypervisor resets the VMExit reason field in the hardware-protected VMSA to an invalid value every time it handles a VMExit from VMPL3. Given only the SEV-SNP hardware can change it back to a valid value on the next VMExit, the L1 hypervisor detects this attack by simply checking if the VMExit reason is legitimate before it starts to perform the emulation.

Second, the L0 hypervisor can swallow a legitimate VMExit and refuse to forward it to VMPL0. Since the L0 hypervisor cannot update the VMSA which is integrity-protected by the hardware, resuming VMPL3 without handling its VMExit will cause it to trigger the same VMExit repeatedly given its instruction pointer (`RIP`) still points to the same VMExit-inducing instruction. We are not concerned with this attack since it will only cause denial-of-service to the nested VM, which our threat model permits anyway.

Third, the L0 hypervisor can pause VMPL0 in the middle of handling a nested VMExit and resume VMPL3 prematurely. This can cause the nested VM to misbehave since it may see partial updates from the L1 hypervisor. Hecate defends against this attack by locking the VMSA throughout the VMExit handling. Specifically, the L1 hypervisor sets the busy bit in the VMSA before handling

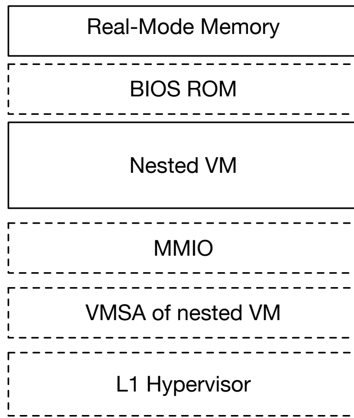

Figure 3: The guest physical memory layout of Hecate (from low to high). Dashed boxes represent the memory regions inaccessible to the nested VM running at VMPL3.

the VMExit, and clears the bit afterwards. The busy bit prevents the L0 hypervisor from resuming the execution of VMPL3.

4.4 MMU: Uni-VM Design

MMU virtualization creates an isolated guest physical address space to give an VM the illusion that it owns the entire physical memory. The challenge in enabling MMU virtualization in Hecate is that there lacks an address translation layer between VMPLs. Early hypervisors without nested paging use a more expensive shadow paging mechanism to support MMU virtualization. It combines the guest's virtual MMU configuration with the hypervisor's own MMU configuration into a unified physical configuration, and monitors the changes on the guest's MMU configuration to update the physical MMU accordingly. Unfortunately, shadow paging cannot be used in Hecate because the L1 hypervisor running at VMPL0 cannot intercept the nested VM's updates to its own page table root (i.e., the `cr3` register). This is different from the traditional nested virtualization where the L0 hypervisor can be trusted to intercept events of a nested VM. Hecate cannot rely on the untrusted L0 hypervisor for critical operations such as intercepting the nested VM's writes to control registers.

Given the primary goal of Hecate is for compatibility and security rather than for resource multiplexing, we make a unique design choice to support running only a *single* nested VM. We show the memory layout of the guest physical address space in Figure 3. At a high level, the lower memory range is reserved for the nested VM, and the trusted L1 hypervisor runs at a higher address range. This preserves the compatibility for legacy OSes that assume the physical address space spans a contiguous range from address 0 to the physical memory limit. The L1 hypervisor further sets up the VMPL permissions so that the nested VM can only access its own memory while the L1 hypervisor itself has full access to the whole address space. By supporting only a single VM and restricting its memory accesses using VMPL permissions, Hecate enables MMU virtualization without paying the cost of an additional level of nested paging or shadow paging.

One caveat of this design is that Hecate cannot support mapping two guest physical pages in the nested VM to the same host physical page (e.g., double mapping or aliased mapping) because this mapping is directly controlled by the L0 hypervisor. However, double mapping is actually a legitimate requirement for x86 because x86 maps the BIOS ROM at two aliased physical addresses `0xF0000` and `0xFFFF0000`, and the legacy ROM makes this assumption when booting a guest OS. To handle aliased guest physical addresses, the L1 hypervisor marks them as inaccessible to the nested VM, and traps and emulates all accesses to these addresses to realize the aliased effect. This does not cause any performance issue because there are only a small number of BIOS operations on the aliased guest physical pages.

4.5 I/O: Two-Layer Communication

The nested VM sends and receives information via virtualized I/O. The L1 hypervisor implements virtual devices for the nested VM, and then relays its communication to the outside world through virtual devices exposed by the L0 hypervisor. This requires I/O operations of the nested VM to go through two layers of virtual devices. But it also allows the L1 hypervisor to enforce firewall policies over the nested VM.

4.6 Shielding the Nested VM

The legacy OS running in the nested VM may not be written with a malicious hypervisor in mind, so the trusted L1 hypervisor is responsible for protecting it from the untrusted L0 hypervisor. Hecate achieves this goal by isolating the CPU execution and the memory state of the nested VM from the L0 hypervisor.

First, although the CPU state of the nested VM stored in the VMSA is encrypted and integrity-protected from the L0 hypervisor on SEV-SNP, the L0 hypervisor can interfere with its execution by injecting spurious interrupts. For example, the L0 hypervisor can lie to the nested VM that an inter-processor interrupt (IPI) was sent from a different virtual CPU. Hecate protects the nested VM from such attacks by preventing the L0 hypervisor from injecting any interrupt to VMPL3 including the #HV vector mentioned in §4.2. This can be done on the SEV-SNP hardware by enabling *Alternative Injection* for VMPL3. Alternative Injection allows only a higher-privileged VMPL (the L1 hypervisor) to inject interrupts to a lower-privileged VMPL (the nested VM). Hecate enables it and virtualizes the interrupt controller for the nested VM.

Second, the nested VM may accidentally leak sensitive information to the untrusted L0 hypervisor via shared, unencrypted memory. The technical challenge here is that shared memory pages can be established between the L0 hypervisor and the nested VM *without* involving the L1 hypervisor. Moreover, the L1 hypervisor cannot restrict the nested VM's memory access to shared pages using VMPL permissions because these permissions can only be applied to private memory pages. We solve this problem by repurposing the Virtual Top-of-Memory (vTOM) feature on SEV-SNP. Basically, vTOM allows the L1 hypervisor to specify a guest physical address boundary below which memory accesses are unconditionally deemed encrypted. In Hecate, the L1 hypervisor sets the boundary to the end of the address space for VMPL3 to force all memory accesses of the nested VM to be encrypted. By doing so,

any accidental write to the shared memory from the nested VM will fail and trigger a special VMExit. Note that the vTOM restriction is only enabled for VMPL3, so the L1 hypervisor can still communicate with the L0 hypervisor via shared memory.

4.7 Nested Virtualization Comparison

We compare the nested virtualization in Hecate with the traditional nested virtualization in this section. The traditional nested virtualization emulates the hardware virtualization extension such as Intel VMX and AMD SVM to an L1 guest VM, so that it can launch and manage its own nested VMs [14]. At a high level, it requires the L0 hypervisor to emulate the following functionalities, and we compare its implementation with the L1 hypervisor in Hecate.

Nested VMExits allow the L1 hypervisor to handle VMExits from nested VMs. They are implemented by trapping and forwarding hardware-generated VMExits to an L1 guest in a similar manner shown in Figure 2. However, the nested VMExits are more expensive in Hecate because the SEV-SNP hardware has to flush more CPU states (e.g., TLB caches) due to confidentiality requirements. We measure the cost of nested VMExits of the Hecate VMM in §7.1.

Per-vCPU control structure is a hardware-defined data structure (e.g., VMCS on Intel and VMCB on AMD) to assist the hypervisor to manage the virtual CPU state. The L0 hypervisor exposes a shadow copy of the per-vCPU control structure of nested VMs to the L1 hypervisor and synchronizes its content to the real copy (used by the hardware) at different points in time. On the contrary, the L1 hypervisor in Hecate has a direct access to the VMSA of the nested VM, so it avoids the performance cost of shadowing the vCPU control structures.

Nested paging virtualization allows the L1 hypervisor to manage the physical address space of nested VMs using nested page tables. It is implemented via shadow page tables that combine the two levels of nested page tables into one, and the L0 hypervisor has to track the L1 hypervisor's page table updates for synchronization. On the contrary, the L1 hypervisor in Hecate does not use nested paging at all by only supporting a single nested VM (see §4.4 for details). Therefore, it avoids the performance cost of extra VMExits required for shadow page table synchronization.

5 IMPLEMENTATION

We implement the L1 hypervisor in Hecate based on Linux KVM [26]. At a high level, the implementation has two parts. First, we enlighten the Linux kernel (which KVM depends upon) so that it can function correctly and securely on top of the hardware features of SEV-SNP for confidential VMs. Second, we enlighten the KVM module to enable CPU and MMU virtualization for the nested VM running at VMPL3. Note that KVM relies on QEMU (or other KVM-compatible I/O virtualization providers) to provide I/O virtualization. Since Hecate does not change KVM's APIs, no changes are required for QEMU.

Our implementation is based off Linux 5.11 which includes the architectural support for running as an SEV-ES guest. We added 4000 lines of code to support SEV-SNP (§5.1), and another 2400 lines of code to enlighten KVM for nested virtualization (§5.2).

5.1 Enlightening Linux Kernel

The L1 hypervisor in Hecate runs a customized yet complete Linux kernel at VMPL0. To make it functional, we follow the GHCB specification of SEV-SNP [6] and adapt the Linux kernel to the new architectural environment of a confidential VM. This includes an implementation of the interrupt handler for the new #HV vector and the management of the guest page state (private vs. shared).

Given that the kernel is specialized to run the KVM virtualization stack rather than arbitrary workloads, we strip down unused kernel functionalities to minimize the attack surface from the untrusted L0 hypervisor. For example, we disable the legacy MMIO-based APIC and port I/O-based 8259A PIC, and only enable the modern Model Specific Register (MSR)-based x2APIC interface for managing interrupts. We then add checks on any information returned by the untrusted L0 hypervisor to ensure that it is valid and consistent. For example, Hecate checks if the hypervisor returns the same APIC ID as the virtual CPU. These defensive checks require careful engineering, but fortunately it only needs to be done correctly once to run the nested VM securely. Next, we discuss two interesting cases.

5.1.1 CPUID Security. The enlightened kernel executes the `CPUID` instruction to probe available features on a virtual CPU. Traditionally, `CPUID` unconditionally causes a VMExit and the hypervisor has to emulate the instruction accordingly. However, the untrusted L0 hypervisor may report inconsistent values and cause confusion to the VM. Even worse, the CPUID leaf `0xD` reports the buffer size needed for saving the extended CPU state, and an incorrectly reported size can cause memory corruptions in the enlightened kernel.

To secure the `CPUID` instruction, we follow the recommendation from AMD to require the untrusted L0 hypervisor to store all `CPUID` results on a special guest page when initializing a confidential VM, and have the SEV-SNP hardware check them for inconsistency. When the enlightened kernel executes `CPUID`, it will trigger the #VC exception, and the exception handler in the guest can emulate the instruction based on the pre-populated results within the VM.

5.1.2 Interrupt Security. The enlightened kernel relies on external interrupt sources emulated by the untrusted L0 hypervisor to function (e.g., timer and I/O). The SEV-SNP hardware prohibits the L0 hypervisor from injecting arbitrary exceptions, such as spurious page faults, to a confidential VM. Instead, it only allows the L0 hypervisor to inject the newly-added #HV vector and informs a confidential VM about the actual vector number via a separate channel defined in the GHCB specification. The challenge is that the hypervisor can inject the #HV exception at *arbitrary* times, e.g., when the enlightened kernel is holding a spinlock and does not expect any external interrupt.

To avoid potential race conditions and/or deadlocks, an invariant that must hold is that the enlightened kernel can only process an external interrupt when the interrupt is enabled. Note that the enlightened kernel cannot simply panic if an #HV interrupt arrives when the interrupt is disabled because it is a legitimate scenario given that the L0 hypervisor does not know if the interrupt is enabled in a confidential VM.

Our approach is to delay the interrupt processing until the interrupt is re-enabled in the enlightened kernel. On x86, there are three instructions that can change the interrupt enablement: `STI`, `POPF` and `IRET`. For `STI` and `POPF`, we append a check for pending interrupt and process it as if the interrupt occurs right after the interrupt enablement. We cannot use the same trick for `IRET` because `IRET` is used for interrupt return and the next instruction may even be in the user mode. Instead, we check whether `IRET` is about to enable interrupts before the interrupt handler restores the general-purpose registers. If so, we enable interrupts prematurely and process the pending interrupt as if the interrupt occurs right after `IRET`.

5.2 Enlightening KVM

We enlighten KVM to enable CPU and MMU virtualization for the nested VM at VMPL3. KVM contains both x86 common code and platform-dependent code (e.g., Intel VMX, AMD SVM), and the platform-dependent code is implemented as a set of callbacks so that the common code can use it without concerning the actual platform. We implement the required callbacks for SEV-SNP to enable KVM to run at VMPL0 and operate on the VMSAs of the nested VM. Note that our code changes to KVM are different from the changes required for making KVM to support SEV-SNP as an L0 hypervisor.

5.2.1 Memory Management. The L1 hypervisor in Hecate shares the guest physical address space with the nested VM due to the lack of nested paging support between VMPLs. Therefore, the first thing the L1 hypervisor needs to do is to reserve a configurable amount of memory starting from address 0 for the nested VM during booting. This ensures that the lower memory range is always available to the nested VM.

When QEMU launches the nested VM and maps its physical memory to the user mode, the L1 hypervisor fills the mapped region with the page frame numbers (PFNs) of the reserved memory in a one-to-one fashion. This allows KVM, the QEMU process and the nested VM to share a consistent view of its physical address space. In addition, QEMU may create special mappings (e.g., BIOS ROM) or even remove mappings (e.g., MMIO) for some physical addresses of the nested VM. For these special physical addresses, the L1 hypervisor makes the corresponding pages inaccessible to the nested VM with VMPL permissions, and emulates all its memory accesses to these pages accordingly.

5.2.2 Interceptions. The L1 hypervisor in Hecate can only intercept events that cause an unconditional VMExit in the nested VM. The untrusted L0 hypervisor can configure conditional intercepts, but the L1 hypervisor cannot count on it since it is untrusted. Meanwhile, existing KVM relies on conditional intercepts to function properly. For example, KVM intercepts its VM's writes to its control registers (e.g., `CR0`) to track the guest CPU mode, such as when the VM is transitioning from the real mode to the 32-bit/64-bit protected mode. KVM tracks the guest CPU mode for the instruction emulation whose behavior depends on various architectural states of the virtual CPU (e.g., whether paging is enabled, etc.). We modify KVM so that it does not rely on these conditional intercepts. For the above example that tracks the virtual CPU mode, we simply re-evaluate the CPU mode every time when KVM handles a VMExit.

6 SECURITY POLICIES

In this section, we present two security policies enabled by Hecate. By design, Hecate can support any security policy that can be enforced at the virtualization boundary. In fact, there is a long history of research that builds security policies on top of virtualization, and we use two existing policies for demonstration purposes.

6.1 Network Filtering

Since the L1 hypervisor in Hecate runs in a full-fledged Linux kernel (referred to as the Hecate kernel), it can enable network filtering on the nested VM in more than one way. Next, we describe the approach we take in our Hecate prototype.

First, we use MacVTap [7] to connect a virtual network interface card (NIC) of the nested VM (exposed by QEMU via the L1 hypervisor) to a virtual NIC of the confidential VM (exposed by the L0 hypervisor). We refer to the former as the *inner* NIC and the latter as the *outer* NIC. Then, we enable eBPF/XDP-based network filters [16] on the outer NIC inside the Hecate kernel. To filter network traffic of the nested VM, we use the MAC address of the inner NIC to identify packets to/from the nested VM. The Cilium open source project [18] has shown that flexible network policies [17] can be enforced with low performance impact based on eBPF/XDP.

6.2 Kernel Code Integrity

Kernel code integrity is an effective mitigation that prevents an adversary from executing foreign code with the kernel privilege despite the presence of kernel vulnerabilities. It has many different implementations [4, 22], and next we describe our approach based on VMPL permissions.

On the L1 hypervisor in Hecate, we added a pair of synthetic model-specific registers (MSR) that hold the nested kernel's base physical address and size. Both MSRs have an initial value of zero upon reset. The guest OS in the nested VM must be modified to enable the code integrity protection. Specifically, when the guest OS finishes booting, it should configure the two MSRs based on the kernel code range, and the L1 hypervisor in Hecate would then adjust the page permissions for VMPL3 in the following way. For pages within the specified range, the L1 hypervisor strips the write permission to ensure the kernel code cannot be modified afterwards. For pages outside of the range, it strips the supervisor-execute permission to prevent them from being abused as kernel code pages. Finally, the L1 hypervisor enforces that the two synthetic MSRs can only be written once and then remain locked until subsequent reset. This approach leverages existing VMPL permission checks to enforce lifetime kernel code integrity for the nested VM without adding extra performance overhead.

Note that the current implementation does not support loadable kernel modules given the protected kernel code must span a contiguous range of memory. Protecting loadable kernel modules requires code signing support, and we leave the engineering effort to future work. We also recognize that leveraging kernel code integrity protection requires minor modifications of the guest OS in the nested VM. While the goal of Hecate is to support unmodified

Benchmark	Baseline	Linux0	Linux3
VMExit	1120	5200	18096
SendIPI	10474	23099	41395

Table 1: Microbenchmark performance in CPU cycles.

```c
unsigned long vmexit_cost(void)
{
    unsigned long now = rdtsc();
    native_read_msr(MAGIC_MSR_INDEX); // VMExit
    return rdtsc() - now;
}
```

Figure 4: Code snippet to measure the CPU cycles of a VMExit.

guest OS, we show that minor kernel changes can be done to further strengthen the security.

7 EVALUATION

In this section, we evaluate the compatibility and performance of Hecate. We run all experiments on a dual-socket 3rd Gen AMD EPYC processor (codenamed Milan) with 256 logical cores and 512GB RAM. The Milan processor is the first generation of CPUs that support AMD's SEV-SNP confidential VM technology. We choose Microsoft Hyper-V as the L0 hypervisor because of its support of the GHCB specification [6].

By construction, the L1 hypervisor in Hecate is capable of running any OS that runs in a vanilla VM supported by KVM, and we do the compatibility evaluation as a sanity check. We are able to run multiple unmodified OSes, including MS-DOS 6.22 (which was released in 1994), FreeBSD 12 (which was released in 2018) and Linux 5.11 (which was released in 2021 and used for performance evaluation below), in the nested VM.

We measure the performance by running both micro- and macrobenchmarks. To provide comparable measurements, we keep the guest software stack used in each experiment as identical as possible. Specifically, we run Linux 5.11 compiled with the same configuration and the same virtual disk image in three types of VMs: (1) a regular VM with SEV-SNP disabled (baseline), (2) a confidential VM without nested virtualization (the enlightened Linux running at VMPL0, or simply Linux0), and (3) a nested confidential VM enabled by Hecate (Linux3). Each VM has 4 virtual CPUs and 8GB RAM.

7.1 Microbenchmarks

For microbenchmarks, we measure the CPU cycles taken for handling a VMExit event and delivering an inter-processor interrupt (IPI). The results are shown in Table 1.

To measure the cost of a VMExit, we execute the code snippet shown in Figure 4 in the guest kernel. We use the `rdtsc` instruction to retrieve the current timestamp directly from the CPU, and trigger a VMExit via the `rdmsr` instruction. We modify the L0 hypervisor's binary in such a way that it immediately resumes the guest upon seeing the magic MSR index. Finally, we take another timestamp to measure the elapsed cycles for the round trip.

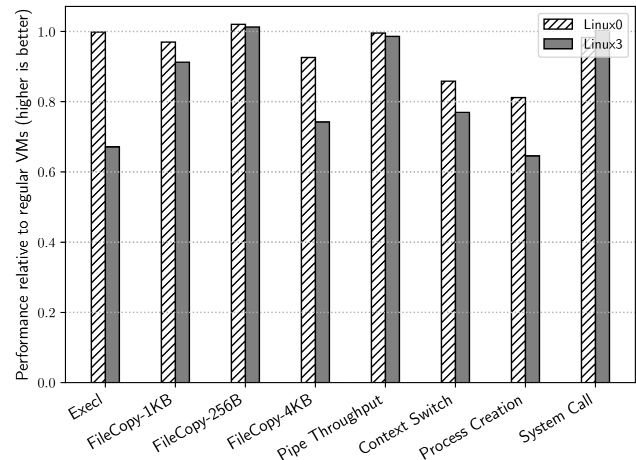

Figure 5: UnixBench performance.

The VMExit incurs an increasingly higher CPU cycles from the baseline all the way to the nested confidential VM enabled by Hecate. For both baseline and Linux0, `rdmsr` is handled by the L0 hypervisor. However, Linux0 has a more expensive world switch because the hardware needs to flush more CPU states (e.g., caches, TLBs) before transferring the control to the untrusted L0 hypervisor. A VMExit in Linux3 involves more world switches because it first traps to the L0 hypervisor which then forwards the event to the L1 hypervisor running in VMPL0. And the execution has to go through the L0 hypervisor again when returning to Linux3 after the L1 hypervisor finishes handling the VMExit event. As a result, it incurs the highest number of CPU cycles in this experiment.

We also evaluate the cost of sending an IPI, which the guest kernel uses to coordinate different tasks on a multi-core system. We measure the total cycles elapsed between when a virtual CPU sends an IPI and when the destination virtual CPU receives the hypervisor-injected interrupt. The results show a similar trend to the VMExit cost because a virtual CPU needs to interact with the hypervisor to send and receive an IPI. In the case of Linux3, both L0 and L1 hypervisors are involved when the guest kernel in the nested VM sends an IPI.

7.2 Macrobenchmarks

For macrobenchmarks, we run UnixBench and a set of popular server programs. We use UnixBench to measure various aspects of a Unix-like system such as file systems and system calls, and use server programs to measure the performance impact on throughput. We follow the experiment setup in [32] when evaluating the performance impacts on server programs.

7.2.1 UnixBench. We show the results of UnixBench in Figure 5. In general, benchmarks that perform in-memory computations inside a VM do not have noticeable overhead (e.g., system calls). File system benchmarks perform I/O operations that require device emulation from Hecate. However, FileCopy-1KB and FileCopy-256B do not have noticeable overhead because most of their operations are serviced by in-memory file caches. In addition, context switch

Program	Description
Netperf	Measures the latency (TCP_RR) and throughput (TCP_STREAM, TCP_MAERTS) with default parameters.
Nginx	Measures the throughput (number of requests per second) by retrieving the GCC reference manual (368KB) using ApacheBench v2.3 with 32 concurrent requests.
Memcached	Measures the throughput (number of requests per second) using `memtier` benchmark v1.3.0 with binary data.
MySQL	Measures the throughput (number of queries per second) using `sysbench v1.0.18` with command `oltp_read_write.lua`, `-tables=10` and `-table-size=100000`.

Table 2: Network server benchmarks. We use default values for parameters that are not explicitly specified in the description. All experiments require a client-server setup, and we run the client in a separate, regular VM.

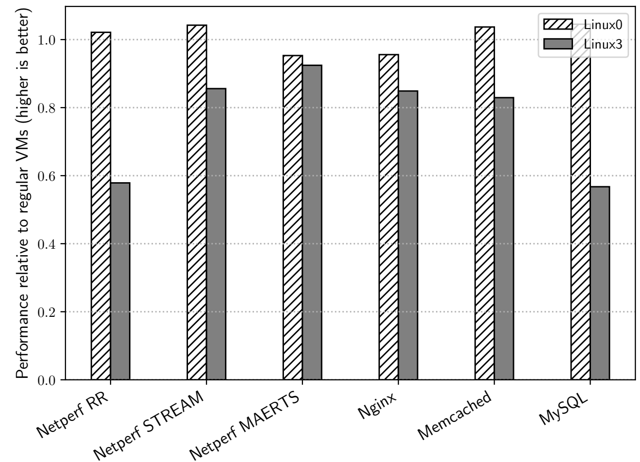

Figure 6: Network server performance.

and process creation also have non-trivial performance overhead. This is because both benchmarks require frequent coordinations between two processes potentially running on different virtual CPUs, and this will lead to frequent IPIs. We have shown that IPIs are expensive in confidential VMs (whether nested or not) in Table 1.

Execl has the biggest performance difference between Linux0 and Linux3. We first describe what Execl does and then explain why. Execl repeatedly invokes the `execve` system call to (re-)execute itself and measures the number of iterations it can complete within a fixed period of time. The `execve` system call wipes out the entire user-mode address space, loads the exactly same program binary, and starts execution from its entry point as if it were a newly created process. At a first glance, Execl does not involve any I/O operation and should not incur any noticeable overhead. However, it turns out that the startup code (which is executed before the `main` function) needs to probe the available CPU features for which the `CPUID` instruction is executed multiple times. The `CPUID` instruction in Linux3 causes a VMExit to the L1 hypervisor in Hecate unconditionally, which incurs non-trivial performance overhead as shown in Table 1. On the other hand, executing `CPUID` in Linux0 does not cause a VMExit to the L0 hypervisor (see §5.1.1 for details), so it does not show any noticeable overhead.

7.2.2 Server Programs. We run benchmarks on a set of popular server programs listed in Table 2. For each benchmark, we use default parameters except for those specified in the description. We show the results in Figure 6. All of the evaluated server programs have disk and/or network I/O operations. I/O operations are costly in Linux3 because they need to go through two layers of virtualized I/O devices: (1) the KVM virtio layer exposed by the L1 hypervisor in Hecate, and (2) the Hyper-V vmbus layer used between Hecate and the L0 hypervisor. MySQL has over 40% reduction in throughput because of its intensive I/O operations of both disk and network.

8 SECURITY ANALYSIS

In this section, we perform a systematic security analysis of Hecate. Specifically, we enumerate the possible attack vectors and describe how Hecate is designed to defend against them.

8.1 From L0 Hypervisor to L1 Hypervisor

There are two ways for the L0 hypervisor to attack against Hecate. First, the L0 hypervisor can attack Hecate as if it were attacking a regular confidential VM that runs the guest OS natively. In addition to the attacks described in §5.1, the L0 hypervisor may swap two guest physical pages at runtime. SEV-SNP requires that a swapped page will need to be validated again by the confidential VM before it can be reused. We defend against such attacks by ensuring the L1 hypervisor never accidentally validates the same guest physical page more than once. Alternatively, the L0 hypervisor can also lie about the current time to trick the confidential VM as if the time were going back. This attack is being addressed by a recently added SEV-SNP CPU feature called SecureTSC [11]. The attack vectors of this category are not unique to Hecate. Instead, it must be handled by any OS that runs natively inside a confidential VM. The nice thing about the nested VM design of Hecate is that we only need to do it correctly in the L1 hypervisor *once*, and all the unmodified nested VMs will be protected against these attacks despite their unawareness of a potentially malicious L0 hypervisor sitting underneath.

Second, the L0 hypervisor can interfere with the communication between the nested VM and the L1 hypervisor. We described the three attacks that the L0 hypervisor can mount in §4.3. To briefly recap, to defend against a non-existent VMEXIT lied by the L0 hypervisor, the L1 hypervisor resets the VMEXIT reason before it handles the event from the nested VM. To defend against the L0 hypervisor preempting the L1 hypervisor's handling of the nested VMEXIT event, the L1 hypervisor locks the VMSA throughout the VMEXIT handling to ensure it never *partially* handles the event. We are not concerned that the L0 hypervisor swallows a VMEXIT event from the nested VM because resuming the nested VM without

involving the L1 hypervisor will cause the same VMEXIT to trigger over and over again.

8.2 From L0 Hypervisor to Nested VM

It is one of the key challenges to protect the nested VM from the L0 hypervisor when we design Hecate. Putting aside latent software bugs, the difficult part is that no legacy OS is written with a malicious (virtual) hardware in mind. For example, the untrusted L0 hypervisor can inject a spurious interrupt when the guest OS has the interrupt disabled (e.g., holding a lock) and introduce an artificial race condition. Alternatively, the untrusted L0 hypervisor can corrupt any shared memory while the guest OS is reading/writing it simultaneously.

We describe how we shield the nested VM from the L0 hypervisor in §4.6. At a high level, we isolate the CPU and memory state of the nested VM from the L0 hypervisor. We leverage the Alternative Injection feature to prevent the L0 hypervisor from injecting any interrupt to the nested VM directly, and repurpose the Virtual Top-of-Memory (vTOM) feature to disable any page sharing even the nested VM wanted to. By doing so, we achieve the complete mediation and ensure the nested VM can interact with the external world only via the L1 hypervisor.

8.3 From Nested VM to L1 Hypervisor

In our threat model, we assume the nested VM is buggy and can become hostile after being compromised (see §3). Although Hecate only supports a single VM, we need to protect the L1 hypervisor to ensure the correct enforcement of security policies such as those described in §6. While the KVM stack we adapted for Hecate has addressed the attacks between the nested VM and the L1 hypervisor via a regular VM boundary, we would like to emphasize the fundamental isolation mechanisms we use in the Hecate design. To isolate the memory state, we setup the VMPL permissions in a one-way view so that the L1 hypervisor at VMPL0 can access the nested VM's memory but not vice versa. To prevent the nested VM from accessing shared memory (which is not protected by the VMPL permissions), we repurpose the vTOM feature so that all memory accesses from the nested VM is always treated as encrypted, which will fault when accessing the shared memory. To isolate the CPU state, we ensure the VMSA page that stores the CPU state of the L1 hypervisor is inaccessible to the nested VM.

9 RELATED WORK

In this section, we discuss related work in three areas: secure enclaves, nested virtualization, and virtualization-based security.

9.1 Secure Enclaves

Hardware vendors have created an array of trusted execution environments (TEEs or enclaves) for hosting confidential workloads. Intel SGX is a user-mode enclave that runs (a subset of) an application with confidentiality and integrity protection despite a malicious OS or hypervisor. It imposes many restrictions to the application (e.g., cannot make system calls), and prior works have been able to run unmodified applications inside an SGX enclave using library OSes [13, 38, 40]. The Intel IceLake processors launched in 2021 extend the 128MB enclave memory limit to 1TB and make SGX more appealing to server-class applications. However, Intel SGX does not run an unmodified VM image due to its user-mode nature, and thus cannot support the lift-and-shift scenario Hecate enables in this work.

AMD SEV and Intel TDX [5] are virtual-machine-based hardware enclaves. AMD SEV relies on a set of hardware mechanisms to protect a confidential VM from the untrusted hypervisor. For example, AMD SEV relies on the platform security processor (PSP) to enable remote attestation and protect the guest pages from being tampered with. On the other hand, Intel TDX manages confidential VMs using a special TDX module, and the untrusted hypervisor can allocate resources for confidential VMs via the interface defined by the TDX module. We implement Hecate based on the VMPL privilege separation on AMD SEV-SNP, and leave the support for other platforms as future work.

ARM TrustZone partitions system resources into a normal world (which runs the traditional software stack) and a secure world (which runs the security-sensitive applications in isolation). It has been used to host security-sensitive computation [24, 25] as well as security monitors [12, 22]. TwinVisor [28] takes advantage of the ARM TrustZone and runs two isolated hypervisors to support confidential VMs.

ARMv9 introduces a new form of secure enclaves called Realm [2]. A Realm is an attestable environment which is isolated from the existing normal and secure worlds on today's TrustZone architecture. Each Realm can have its own applications and OS kernel. Despite ARM's dominance in mobile computing, the majority of today's on-premises workloads run on the x86 architecture.

Hardware enclaves can have firmware bugs [9] and are subject to side-channel attacks [27, 29, 41–43]. An adversary can leverage these attacks to extract sensitive information protected by the enclave at different granularities. It is an arms race between exploiting new side channels versus defending against them. We deem this line of research orthogonal to this work.

Amazon Nitro Enclave [3] relies on the host hypervisor to create an isolated execution environment from a regular VM. It targets a different threat model where the host hypervisor is assumed trusted.

9.2 Nested Virtualization

The Turtles project [14] demonstrates a practical implementation of nested virtualization on the commodity x86 hardware. It virtualizes the hardware-assisted virtualization extension such as Intel VMX and exposes it to a guest VM for nested virtualization.

DVH [32] optimizes the nested virtualization performance for I/O with direct device assignments. DVH avoids the performance cost of I/O operations involving multiple levels of virtualizations by having the L0 hypervisor provide virtual hardware to nested VMs directly. The same techniques do not apply to Hecate because the host hypervisor is untrusted and prevented from accessing the nested VM.

CloudVisor [44] repurposes nested virtualization to protect VMs in a multi-tenant cloud. It limits the damage of a compromised VMM by running it on top of a security monitor using nested virtualization.

Hecate builds a nested virtualization stack using limited hardware support for compatibility and security purposes. It only supports a single nested VM which simplifies the design and improves the performance.

9.3 Virtualization-Based Security

Hardware-assisted virtualization has been extensively utilized for security purposes. Virtual machine introspection [15, 19–21, 23, 33, 35, 37, 39, 45] is an effective approach to address the growing security concerns within a virtual machine. SIM [37] implements a kernel security monitor within a guest VM's address space, and protects it from a potentially compromised guest kernel using nested page tables. It achieves the isolation of security monitoring tools while minimizing the performance overhead involved in world switches. ImEE [45] further protects the in-VM monitor from being deceived by a spurious virtual address mapping and ensures that the guest kernel and the security monitor have a consistent memory view.

SecVisor [36] and HVCI [4] enforce code integrity for the guest kernel running in a VM. They leverage the hypervisor's control of the MMU (e.g., nested page tables or shadow page tables) to run only approved code in the guest kernel space even when it is compromised. Hecate leverages the VMPL permissions to achieve the same code integrity goal.

These virtualization-based security mechanisms are complementary to Hecate. We have demonstrated two concrete security applications in §6.

10 DISCUSSION

Compared with previous research on OS architectures, our work embraces a new challenge and opportunity: how to protect computations inside a confidential VM while the L0 hypervisor is not trusted anymore. Before confidential VMs were introduced, many security schemes were provided by the L0 hypervisor. A key research question in this new era of confidential VMs is how to enable these security schemes inside confidential VMs. The new architecture, particularly the virtual machine privilege levels (VMPLs), introduced by AMD's SEV-SNP, creates new opportunities for OS security researchers to leverage the isolations inside a confidential VM to develop new security schemes. However, other new confidential VM technologies such as Intel TDX [5] and ARM Realm[2] lack a VMPL-like isolation inside their confidential VMs. This introduces two new challenges: (1) how to enable security schemes based on a security monitor (e.g., TDX Module or Realm Management Monitor) running outside a confidential VM; (2) how to develop a framework that is agnostic to confidential VM technologies. Our work is the first attempt in the direction of protecting computations inside confidential VMs. We hope it will lead to more impactful research from the community.

A lesson we learned from our work is that we need more software-hardware co-design in confidential VMs for better security and performance. For instance, SEV-SNP relies on the L0 hypervisor to perform VMPL switches. Our work shows that this hardware design choice complicates the security design as the L1 hypervisor has to defend itself from various ways the L0 hypervisor can interfere with the switch process (§8.1). Meanwhile, it also introduces unnecessary performance slowdown (§7.1). Since we expect more innovations to happen inside confidential VMs, it is important for the OS security community to work with hardware vendors to design the right hardware primitives to enable new performant security schemes inside confidential VMs.

As shown in our work, AMD SEV-SNP still allows the untrusted hypervisor to manage the computing resources for confidential VMs. This opens up opportunities for the untrusted hypervisor to launch powerful side-channel attacks against a confidential VM [30, 31, 34]. We leave defenses of side-channel attacks to future research.

11 CONCLUSION

In this paper, we present Hecate, the first L1 hypervisor that runs inside a confidential VM. Hecate supports lifting and shifting on-premises workloads by enabling compatibility with unmodified VM images and enforcement of security policies such as network firewalls. The design challenge behind Hecate is that it cannot trust the L0 hypervisor to provide nested virtualization, so we repurpose a new privilege dimension available on AMD's SEV-SNP technology to enable virtualization for a single nested VM. We compare the design of Hecate with traditional nested virtualization, and evaluate its performance with real-world server workload. Our evaluation shows that Hecate offers a practical approach for enterprise customers to migrate their on-premises, security-sensitive workloads to the public cloud.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments and suggestions. We benefited tremendously from the discussions we had at the early stage of this work with Andrew Baumann, Donald Kossmann, Sibin Mohan and Marcus Peinado. We are grateful for the help and feedback from Aditya Bhandari, Pushkar Chitnis, Dexuan Cui, Mike Ebersol, Alexander Grest, David Hepkin, Michael Kelley, Roman Kiselev, Tianyu Lan, Jon Lange, Li Li, Cheng-mean Liu, Chris Oo, and KY Srinivasan at Microsoft. We also want to thank David Kaplan, Thomas Lendacky, and Brijesh Singh from AMD for answering numerous questions we had about AMD SEV-SNP.

REFERENCES

[1] AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://developer.amd.com/resources/developer-guides-manuals.
[2] ARM Confidential Compute Architecture. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.
[3] AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.
[4] Hypervisor-Protected Code Integrity (HVCI). https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard.
[5] Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.
[6] SEV-ES Guest-Hypervisor Communication Block Standardization. https://developer.amd.com/wp-content/resources/56421.pdf.
[7] MacVTap. https://virt.kernelnewbies.org/MacVTap.
[8] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.
[9] AMD Server Vulnerabilities. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1021.
[10] Using SQL Server in Windows. https://docs.microsoft.com/en-us/troubleshoot/sql/general/use-sql-server-in-windows.
[11] SEV Secure Nested Paging Firmware ABI Specification. https://www.amd.com/system/files/TechDocs/56860.pdf.

[12] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*.

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

[14] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[15] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. 2012. Secure and Robust Monitoring of Virtual Machines Through Guest-Assisted Introspection. In *Proceedings of the 15th International Workshop on Recent Advances in Intrusion Detection (RAID)*.

[16] Cilium. BPF and XDP Reference Guide. https://docs.cilium.io/en/latest/bpf/.

[17] Cilium. Cilium Network Policies. https://docs.cilium.io/en/stable/policy/.

[18] Cilium. eBPF-based Networking Observability and Security. https://cilium.io.

[19] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*.

[20] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*.

[21] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*.

[22] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of the 3rd Mobile Security Technology Workshop (MOST)*.

[23] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. 2011. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS)*.

[24] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[25] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*.

[26] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*.

[27] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium*.

[28] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*.

[29] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proceedings of the 28th USENIX Security Symposium* (Santa Clara, CA). 1257–1272. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[31] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *Proceedings of the 30th USENIX Security Symposium*.

[32] Jin Tack Lim and Jason Nieh. 2020. Optimizing Nested Virtualization Performance using Direct Virtual Hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. 2014. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*.

[34] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. 221–230. https://doi.org/10.1145/3292006.3300022

[35] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-Bridge: Efficiently Bridging the Semantic Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*.

[36] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*.

[37] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*.

[38] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[39] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. 2011. Process Out-Grafting: an Efficient "Out-of-VM" Approach for Fine-Grained Process Execution Monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*.

[40] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*.

[41] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*.

[42] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium*.

[43] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[44] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. 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)*.

[45] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. 2017. Seeing Through the Same Lens: Introspecting Guest Address Space at Native Speed. In *Proceedings of the 26th USENIX Security Symposium*.