VERISMO: A Verified Security Module for Confidential VMs

Ziqiao Zhou* Anjali† Weiteng Chen* Sishuai Gong‡ Chris Hawblitzel* Weidong Cui*
*Microsoft Research †University of Wisconsin-Madison ‡Purdue University

Abstract

Hardware vendors have introduced confidential VM architectures (e.g., AMD SEV-SNP, Intel TDX and Arm CCA) in recent years. They eliminate the trust in the hypervisor and lead to the need for security modules such as AMD Secure VM Service Module (SVSM). These security modules aim to provide a guest with security features that previously were offered by the hypervisor. Since the security of such modules is critical, Rust is used to implement them for its known memory safety features. However, using Rust for implementation does not guarantee correctness, and the use of unsafe Rust compromises the memory safety guarantee.

In this paper, we introduce VERISMO, the first verified security module for confidential VMs on AMD SEV-SNP. VERISMO is fully functional and provides security features such as code integrity, runtime measurement, and secret management. More importantly, as a Rust-based implementation, VERISMO is fully verified for functional correctness, secure information flow, and VM confidentiality and integrity. The key challenge in verifying VERISMO is that the untrusted hypervisor can interrupt VERISMO's execution and modify the hardware state at any time. We address this challenge by dividing verification into two layers. The upper layer handles the concurrent hypervisor execution, while the lower layer handles VERISMO's own concurrent execution. When compared with a C-based implementation, VERISMO achieves similar performance. When verifying VERISMO, we identified a subtle requirement for VM confidentiality and found that it was overlooked by AMD SVSM. This demonstrates the necessity for formal verification.

1 Introduction

Confidential computing has been adopted by major cloud providers with the aim of removing the cloud provider out of the Trusted Computing Base (TCB). This is achieved by leveraging hardware-based Trusted Execution Environments (TEEs), which are encrypted and isolated from the rest of the software stack managed by the cloud provider. In recent years, hardware vendors have introduced confidential VM architectures (e.g., AMD SEV-SNP [1], Intel TDX [19], and Arm CCA [6]) that can run a full VM inside a TEE.

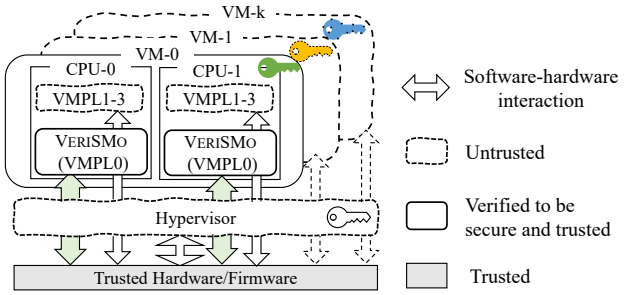

Figure 1: VERISMO in AMD SEV-SNP architecture

While a confidential VM's confidentiality and integrity are protected from the untrusted hypervisor, it also means that a confidential VM cannot use security features that previously were offered by the hypervisor. To fill this gap, security modules such as AMD Secure VM Service Module (SVSM) were introduced to provide the missing security features in a privileged layer inside a confidential VM. Given the importance of the security of such modules [4, 39], Rust is used to implement them for its known memory safety features. However, using Rust for implementation does not guarantee correctness, and the use of unsafe Rust compromises the memory safety guarantee.

In this paper, we present VERISMO[1], the first verified security module for confidential VMs on AMD SEV-SNP. Similar to other security modules like AMD SVSM [2], VERISMO is a privileged software layer that runs inside a confidential VM and provides security features such as code integrity, runtime measurement, and secret management. The isolation between the security module and the guest OS is based on a new privilege dimension called Virtual Machine Privi-

[1] VERISMO is derived from realism in the arts, particularly late 19th-century Italian opera. Its pronunciation reflects its small size.

lege Levels (VMPLs) on AMD SEV-SNP. VERISMO runs at the highest-privileged VMPL. Unlike other security modules, VERISMO is fully verified for functional correctness, secure information flow, and memory safety. Specifically, VERISMO is implemented in Rust and verified using Verus [22], a program verificaton tool designed for Rust.

AMD SEV-SNP provides confidential VMs with confidentiality and integrity. The former is achieved by encrypting the memory of a confidential VM and the latter is achieved by tracking the ownership of memory pages based on a new mechanism called the Reverse Map Table (RMP). While VERISMO can directly control memory encryption, it has to interact with the hypervisor to maintain the integrity of memory pages. This is because the hypervisor controls the nested page table and the owernship of memory pages in the RMP, while VERISMO is responsible for updating the RMP to validate memory pages assigned to a confidential VM.

The key challenge in verifying VERISMO is that the untrusted hypervisor can interrupt VERISMO's execution and modify the hardware state at any time. This concurrent interference makes it unwieldy to use standard Floyd-Hoare reasoning when verifying that VERISMO enforces the confidentiality and integrity of the VMs. To address this challenge, we divide verification into two layers. The upper layer handles the concurrent hypervisor execution, while the lower layer handles the VERISMO implementation, which is itself concurrent. This allows us to reason about these two different forms of concurrency (hypervisor interference and VERISMO's internal concurrency) using two different techniques:

1. For the upper layer, which we call the "machine-model layer", we define an abstract machine model that represents various physical hardware resources and hypervisor operations. We then prove that steps taken by this abstract machine preserve the confidentiality and integrity of the VMs.

2. For the lower layer, which we call the "implementation layer", we use Rust's ownership checking and Verus's permissions to reason about VERISMO's internal resources as the resources are accessed concurrently by different CPUs.

The interaction between the two layers is managed by preconditions that the VERISMO implementation must satisfy when performing hardware operations and postconditions that the VERISMO implementation can assume after hardware operations. For example VERISMO must satisfy a particular precondition when writing to a page table, and VERISMO can assume a postcondition about a memory page after executing the `pvalidate` instruction on the page. The upper layer can assume that the preconditions are satisfied, so that we can use these preconditions to verify that the abstract machine preserves the confidentiality and integrity of the VMs.

To make the implementation layer's verification scalable, especially with concurrent CPU access, we adopt permission-based reasoning, as suggested by previous research [8, 22, 31, 37]. This method combines ideas from Linear Logic [13] and Separation Logic [36], using access permissions as abstract capabilities for operations like reading and writing. Our approach applies these permissions to create type-safe interfaces for hardware resources, ensuring consistent maintenance of correct permissions during software interactions with these resources. Moreover, these interfaces, verified at the machine model level, guarantee memory safety and operational correctness in concurrent environments.

To enforce security information flow, we introduce a security type that carries possible value sets and security labels for each primitive type. The key concept here is to track a security level to each variable at every privilege level and ensure the proper relationship between the security level in value and the proper access permission in memory.

We built VERISMO mostly from the ground up, with the exception of integrating a verified cryptographic library [35], which we trust completely to avoid unnecessary duplication of verification efforts. We compared VERISMO with a C-based implementation and observed similar performance. It takes roughly 6 minutes to verify VERISMO on a 32-core machine, which shows the efficient proof time achieved through our optimized verification design and the use of Verus which is highly optimized for SMT solving.

In summary, our work makes the following contributions:

- VERISMO is the first verified security module operating within a confidential VM.
- We demonstrate how to verify VM integrity and confidentiality in the presence of a potentially malicious concurrent hypervisor, decomposing the verification into two layers to handle two levels of concurrency.
- We utilize the state-of-the-art Rust-based verification framework, showcasing the feasibility of constructing a verified real-world system using permission-based reasoning in Rust.
- We encode security flow policies using a type system and define safe casting to ensure the confidentiality of secret data while allowing all flexible accesses to secrets. (Section 8.4.1).

2 Background

2.1 AMD Confidential VMs

AMD Secure Encrypted Virtualization (SEV) is a confidential VM architecture. The latest version of AMD SEV, known as SEV-SNP, offers enhanced integrity and confidentiality protections for VMs.

Memory Encryption AMD SEV-SNP encrypts memory using a VM-specific encryption key, and secures the virtual

CPU (vCPU) state by encrypting and storing it in a VM Saving Area (VMSA) when the vCPU is trapped into the hypervisor. To support communication with the outside world (hypervisor or traditional IO devices), SEV allows VMs to selectively control encryption for memory pages by either setting an encryption bit in the guest page table or configuring a special MSR called vTOM.

Reverse Map Table AMD SEV-SNP introduces the Reverse Map Table (RMP) for memory integrity. It is located in the reserved system memory and is updated only with special CPU instructions – `rmpupdate` by the hypervisor or `rmpadjust` and `pvalidate` by the VM. The RMP is indexed by System Physical Addresses (SPAs), and each entry includes a Guest Physical Address (GPA) (as the reverse mapping of the nested page table), the assigned security domain (the hypervisor or a VM), as well as a validation bit to indicate whether the VM has accepted the memory assignment via `pvalidate`. To ensure the memory confidentiality and integrity, a confidential VM must correctly manage its page tables and the RMP.

VM Privilege Levels SEV-SNP introduces VM Privilege Levels (VMPLs) to isolate software running within a confidential VM. VERISMO runs in highest-privilege level—VMPL0, and we use VMPL3 to denote the level for running other softwares inside the VM. A vCPU's VMPL is stored in its VMSA. Different VMPLs share the same guest physical memory but have different permissions. By default, only VMPL0 has full permissions enabled to all guest memory pages. A VMPL can grant a subset of its permissions to a lower-privileged VMPL via the `rmpadjust` instruction. Those permissions are stored in the RMP and are part of the RMP check.

VM Platform Communication Key In AMD SEV-SNP, confidential VMs rely on the hypervisor to forward their messages to the Platform Security Processor (PSP) for tasks such as deriving new keys and generating attestation reports. To prevent attacks from a malicious hypervisor, The PSP uses VM Platform Communication Keys (VMPCKs) to establish secure channels with a confidential VM. These keys are passed to a confidential VM at launch time. VMPL0 has access to all keys and can choose to release some keys to other VMPLs.

VM Secure Interrupts A malicious hypervisor may inject arbitrary interrupts to change the data/control flow of the VM. Without secure interrupts, shared memory might be exploited by the hypervisor to leak sensitive data. For example, a recent research [38] demonstrates that #VC interrupts can leak sensitive data via the shared guest-hypervisor communication block (GHCB). To prevent the hypervisor from injecting arbitrary interrupts into a VM, AMD SEV-SNP introduces two secure interrupt injection modes: restricted interrupts and alternative interrupts. Each VMPL can have its own interrupt mode specified in the VMSA. When restricted interrupts are enabled, the hypervisor can only inject one interrupt type introduced by AMD called #HV. When a #HV arrives at a VMPL, the guest code at that VMPL can refer to a shared #HV doorbell page to check the interrupt type instead of directly jumping to an arbitrary interrupt handler. When alternative interrupts are enabled, the hypervisor cannot inject any interrupts into the VMPL, and the interrupts are always controlled by a higher-privileged VMPL. Thus, VMPL0 must use the restricted interrupt mode for security, while other VMPLs can use either the restricted or alternative interrupt mode.

2.2 Rust and Verus

Rust is a modern programming language that offers high performance and memory safety without requiring a garbage collector. Rust's ownership system enforces memory safety in a way conceptually similar to linear logic or separation logic. Rust is safe by default, meaning the compiler enforces memory safety guarantees. However, for scenarios where assembly code or direct control of memory is needed, Rust provides 'unsafe' blocks, which can cause bugs and memory safety issues [28].

Verus [22] is a verification tool designed for Rust. Verus extends Rust with verification features such as preconditions, postconditions, and loop invariants. For specifying and proving properties of Rust programs, Verus allows Rust developers to define three types of variables—executable, ghost, and tracked variables as well as three types of functions—executable, proof, and specification (spec) functions. The non-executable functions and variables are used by Verus during verification but are erased during compilation.

Ghost variables, which are used in proofs to represent mathematical abstractions such as sets or maps, are not checked by Rust's ownership checker. Tracked variables (referred to as "proof variables" in earlier versions of Verus [22]), on the other hand, are used to represent owned resources or permissions, and are checked by Rust's ownership checker. VERISMO uses tracked variables to represent permissions to access hardware resources such as memory and registers, in a style similar to separation logic or linear logic, but checked with Rust's ownership checker rather than with a dedicated separation logic or linear logic checker.

3 System Design

In this section, we present the system design of VERISMO.

3.1 Threat Model

VERISMO follows the threat model assumed by confidential computing. It only trusts the CPU and assumes that every-

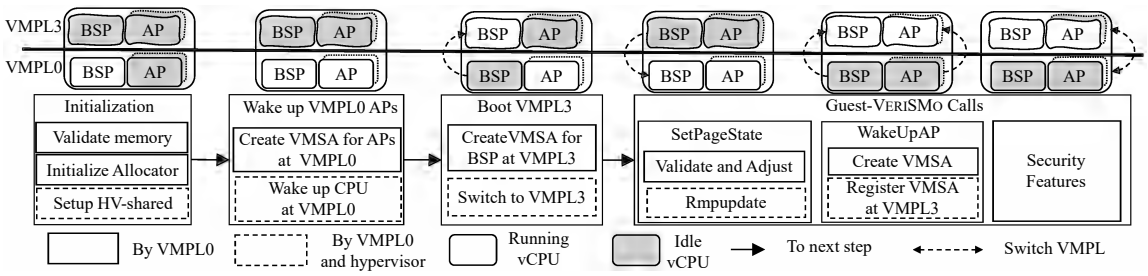

Figure 2: VERISMO work flow

thing outside of a confidential VM is entirely controlled by an adversary, including the hypervisor. Although the hypervisor can interrupt a VM at any time, it can only inject the #HV interrupts because VERISMO uses the restricted interrupt mode to prevent malicious interrupt injections. Furthermore, VERISMO does not trust the guest OS running in the confidential VM. Denial-of-service attacks are possible, as the untrusted hypervisor manages the host and can either shut down the physical machine or opt not to schedule a vCPU for VERISMO to run. Physical attacks are out of scope, as they are orthogonal to our work.

3.2 Architecture

VERISMO runs in VMPL0, while the guest operating system runs in VMPL3 (VMPL1 or VMPL2 could also be used, but we choose VMPL3 in this paper). Both VERISMO and the guest run in the retricted interrupt mode [2], and are thus not vulnerable to the interrupt injection attacks (e.g., [38]).

The work flow of VERISMO is shown in Figure 2. When a confidential VM is launched, VERISMO executes first. It reserves private memory for itself and then launches the guest OS. Afterwards, VERISMO runs in a loop on each processor, waiting for calls from the guest OS.

3.3 Guest-VERISMO Communication

VERISMO and the guest OS running on a processor can transition execution to each other by issuing a hypercall to the hypervisor. Furthermore, a per-CPU memory page is shared between VERISMO and the guest OS so that they can communicate with each other. The hypervisor does not have access to this memory page.

3.4 VERISMO Guest APIs

The guest OS in VMPL3 must rely on VERISMO to wake up its application processors (APs) and to validate memory pages, as it lacks these capabilities. Additionally, the guest OS can use VERISMO-provided security features.

Waking up APs. During the boot time, the guest OS on the bootstrap processor (BSP) calls VERISMO to activate APs. Upon receiving the request, VERISMO's code running on the BSP notifies code running on APs. Once receiving the notification, VERISMO's code running on an AP sets up a per-CPU VMSA page for the guest OS and transitions execution to the guest OS.

Guest Memory Management. While both VERISMO and the guest OS are capable of sharing memory pages with the hypervisor, only VERISMO can make memory pages private by validating them in the RMP. To track the state of memory pages (e.g., private/validated or shared/invalidated), VERISMO requires the guest OS to use VERISMO-provided APIs to share memory pages with the hypervisor. If the guest OS chooses not to follow this requirement, these shared pages will not be validated by VERISMO anymore.

Guest Kernel Code Integrity. To assist the guest OS in preventing unauthorized code execution in kernel mode, VERISMO offers the `LockKernel` API. The guest OS can invoke this API with a list of memory ranges corresponding to its kernel-mode code. VERISMO will then remove from the guest OS the write permission to the kernel code pages and the supervisor-execution permission to other memory pages. VERISMO also ensures that this API can be called only once.

Runtime Measurement. To facilitate runtime measurement for the guest OS, VERISMO provides two APIs, `ExtendPCR` and `Attest`, based on a hash chain. The hash chain's initial value is set to the measurement of the guest OS's starting code and configuration. The guest OS can invoke `ExtendPCR` to extend the hash chain and call `Attest` with a nonce to request an attestation report. VERISMO assembles the attestation report to include a hardware-attested report for VERISMO's identity and a VERISMO-attested report for the hash chain.

[2]The mainstream Linux (v6.8) does not support restricted interrupt injection in either KVM or the guest. We used the Hyper-V hypervisor and our modified guest Linux to enable restricted interrupts with #HV doorbell implementation.

Secret Management. VERISMO provides three APIs to support the guest OS for secret management: `DeriveKey`, `Encrypt`, and `Decrypt`. `DeriveKey` generates an encryption key derived from the current guest runtime measurement. This key is kept in VERISMO and is never discloses to the guest OS. The guest OS can invoke `Encrypt` or `Decrypt` to use this derived key to encrypt or decrypt data.

4 Verification Overview

4.1 Motivation

Traditional software testing can only partially check correctness for certain inputs and cannot formally ensure correctness. Formal verification is the only solution that provides a formal guarantee for the correctness. Below, we demonstrate the need for formally verifying three properties: functional correctness, secure information flow, and VM confidentiality and integrity.

Functional Correctness. Functional correctness defines the desired outcome (i.e., the postcondition) of a function when an input meets certain requirements (i.e., the precondition). In the following code, a key generation function contains a bug that results in a violation of the desired specification.

Listing 1: Incorrect functionality

```
fn GenPrivKey() -> (key: Key)
ensures key.is_random()
{
    return 123; // a constant is not random
}
```

Secure Information Flow. Secure Information Flow defines a safety problem by considering whether the information flow in a system is managed in a way that prevents unauthorized access or leakage of sensitive data. A program is said to be secure if and only if its memory trace and the values of low-security variables are independent of the initial values of its high-security variables. For example, the codes below show the security violation via data flow (left) and control flow (right).

```
low = high % 2
```

```
if high % 2 == 1 {a()} else {b()}
```

VM Confidentiality and Integrity. When a program P operating at a certain privilege level accesses memory M on a CPU, it is possible that M is concurrently accessed by P on a different CPU, or by another program at a different privilege level (e.g., the hypervisor). It is important to note that confidentiality and integrity violations within a program P can be eliminated through verification of P itself. However, the unexpected memory value due to concurrent updates from untrusted programs cannot be prevented. Thus, strict correctness cannot be verified against a specification relying on values from mutable shared memory, which is concurrently accessible by the hypervisor or other VMPLs.

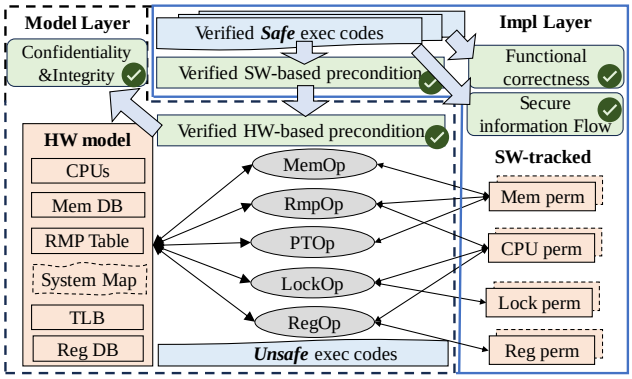

Figure 3: Two layer verification

Listing 2 illustrates an example where an incorrect modification of the page table can lead to the leakage of secrets to the untrusted hypervisor (confidentiality violation) and leave unintended effect in the software (integrity violation). Listing 3 shows a similar violation due to an incorrect RMP change.

Listing 2: Integrity/confidentiality violation via page table change

```
page_table_set_encryption(a_addr, false);
*a = ret_sensitive(); // Leaked result;
do_critical(&a); // Unintended result;
```

Listing 3: Integrity/confidentiality violation via RMP change

```
rmp_adjust(a_addr, READ, VMPL3);
*a = ret_sensitive(); // Leaked result;
rmp_adjust(a_addr, WRITE, VMPL3);
do_critical(&a); // Unintended result;
```

4.2 Verification Design

While we can adapt existing verification techniques to verify functional correctness and secure information flow, verifying VM confidentiality and integrity has its own challenge. The challenge comes from the fact that the untrusted hypervisor can interrupt VERISMO's execution and modify the hardware state at any time. This concurrent interference makes it hard to verify that VERISMO enforces VM confidentiality and integrity. Furthermore, there is another source of concurrency: VERISMO itself is a concurrent program. To handle these two different forms of concurrency separately, we divide verification into two layers: the machine-model layer and the implementation layer (see Figure 3).

In the machine-model layer, we define an abstract machine model that represents various hardware resources. Then we prove that steps taken by this abstract machine preserve VERISMO's confidentiality and integrity. In the implementation layer, we use Rust's ownership checking and Verus's permissions to reason about VERISMO's internal resources

as the resources are accessed concurrently by different CPUs. The interaction between the two layers is managed by preconditions that the VERISMO implementation must satisfy when performing hardware operations and postconditions that it can assume after hardware operations.

5 Machine-Model Layer

In this section, we describe the verification process at the machine-model layer. The goal of this layer is to prove that steps taken by an abstract machine ensures VERISMO's confidentiality and integrity. Specifically, we need to ensure the following security properties required by VERISMO.

(1) Ensuring the integrity and confidentiality of VM data in private memory. The hypervisor is unable to alter or explicitly read VM-private memory in the hardware through any sequence of operations by the hypervisor entity.

(2) Maintaining VMPL isolation. VMPL3 is unable to explicitly read VMPL0-private data, and whenever VMPL0 reads its own private data, it obtains the correct data, not data tampered with by VMPL3.

5.1 Abstract Machine Model

Our abstract machine model, Ψ, represents the contents and attributes of hardware resources such as registers, memory, page tables, and the RMP. This model is updated through some transition operations initiated by different entities. Since the model defines the interactions between these entities, we can formally check the preconditions for each operation required for ensuring the desired security properties.

5.1.1 Entities

Our abstract machine model has three entities.

$\mathtt{E}_0$ represents VERISMO executing at VMPL0.

$\mathtt{E}_3$ represents the guest OS running at VMPL3. $\mathtt{E}_0$ and $\mathtt{E}_3$ share the same memory encryption key.

$\mathtt{E}_{hv}$ represents the hypervisor running in the hypervisor mode. $\mathtt{E}_{hv}$ does not have access to the memory encryption key of a confidential VM. Sibling guest VMs are ignored because the hypervisor's capabilities are their super set.

Both $\mathtt{E}_{hv}$ and $\mathtt{E}_3$ are untrusted and can execute arbitrary code. Therefore, the value read out of the memory shared with $\mathtt{E}_{hv}$ or $\mathtt{E}_3$ is treated as unconstrained.

5.1.2 Primitive Operations

Our abstract machine model defines a set of primitive operations (see Figure 3) that can be initiated by different entities to read or modify hardware resources. Each operation represents a single machine instruction, and its behavior is formally defined based on the AMD manual [3]. For instance, we define how the hardware model returns a memory value after nested page table walks and RMP checks.

For each operation, we define a trusted exec function with a single line of unsafe assembly in Rust with its pre- and post-condition. The postcondition reflects the operations's effect and are fully trusted. For instance, the postcondition for `pvalidate` is that the RMP entry is marked as validated. The preconditions of trusted functions are checked in the verification process to prove that, by enforcing the preconditions, the abstract machine state ensures the security properties when $\mathtt{E}_0$'s operation is constrained by the preconditions. The completeness of the operation model is important to our verification. Since VERISMO is not as large as a guest OS, we currently only model critical memory and cache operations under some assumptions. For example, VERISMO directly uses the guest-hypervisor communication to replace code that may trigger #VC, and always forces a VM termination when a #VC or other unexpected interrupt is triggered. Thus, we do not need to model the potential #VC events when accessing a memory.

5.2 Top-level Security Property Specifications

When proving the security properties (1) and (2), we prove both the confidentiality and integrity theorem outlined in Listing 4 and Listing 5.

Listing 4: VMPL0 confidentiality

```
proof fn proof_confidentiality(Ψ: Machine, e0: Entity,
      e: Entity, va1: Addr, va2: Addr)
requires
  E0.contains(e0), e0 ≠ e,
  m_inv(Ψ),
  m_read(Ψ, va1, e0).is_Ok(),
  m_read_ret(Ψ, va1, e0).is_Secret(),
  m_to_spa(Ψ, va1, e0) ≡ m_to_spa(Ψ, va2, e),
  m_read(Ψ, va2, e).is_Ok(),
ensures
  m_read_ret(Ψ, va2, e).is_Encrypted();
```

Listing 5: VMPL0 integrity

```
proof fn proof_integrity(Ψ: Machine, Ψ′: Machine, e0:
     Entity, va: Addr)
requires
  E0.contains(e0),
  m_inv(Ψ, e0),
  attack_model(Ψ, Ψ′),
  m_read(Ψ, va, E0).is_Ok(),
  m_read(Ψ′, va, E0).is_Ok(),
ensures
  m_read_ret(Ψ, va, e0) ≡ m_read_ret(Ψ′, va, e0);
```

To prove confidentiality, a critical specification is the SNP machine invariant (`m_inv`) representing whether a machine state is valid. For the integrity proof, we additionally rely on a

specification (`model_attack`) that determines whether a state Ψ' is reachable from Ψ under attack. Thus, it is necessary to prove the correctness of both the machine invariant specification (Listing 6) and the attack model specification (Listing 7). This entails consideration of machine model modifications stemming from all possible operations, where only E_0's operation is constrained by preconditions (`Op::sw_requires`).

Listing 6: Correctness of machine state invariant

```
proof fn proof_machine_inv(Ψ: Machine, Ψ′: Machine, op:
      Op, e0: Entity, e: Entity)
requires
  E0.contains(e0),
  (e0 ≡ e) ⟹ Op::sw_requires(e, op),
  Ψ′ ≡ m_op(Ψ, e, op),
  m_inv(Ψ, e0),
ensures
  m_inv(Ψ′, e0);
```

Listing 7: Correctness of attack model

```
proof fn proof_attack_model(Ψ: Machine, Ψ′: Machine,
     Ψ″: Machine, op: Op, e0: Entity, e: Entity)
requires
  E0.contains(e0), e ≠ e0,
  m_inv(Ψ, e0),
ensures
  Ψ′ ≡ Machine:op(Ψ, e, op) ⟹ attack_model(e0, Ψ,
     Ψ′),
  (attack_model(e0, Ψ, Ψ′) && Ψ″ ≡ m_op(Ψ′, e, op))
     ⟹ attack_model(e0, Ψ, Ψ″);
```

5.3 Security Property Proof Sketches

In this section, we describe five critical lemmas and provide a sketch of their proofs, in order to prove the two top theorems and the correctness of critical specifications. It is worth noting that they are fully proved with Verus.

In VERISMO, we classify the guest memory into three sets: VMPL0-Private , VMPL3-Private, and Hypervisor-Shared. The divided memory sets allow us define the security properties for different memory types.

5.3.1 VM-Private Memory

Lemma 1. *Let Ψ represent a machine state in which M is a guest physical memory block that stores value D. Suppose Ψ' is a future state reachable through modifications made by E_{hv}. Then, if M is VM-private in Ψ, VM's read operation on M in Ψ' either fails or returns the original value D.*

A key invariant property of the RMP is that, once a RMP entry is validated by a VM for a VM-private memory page, the guest physical address (GPA) of this memory page will be either bound to the system physical address (SPA) of the RMP entry or nothing at all, regardless of any operations by E_{hv}, as long as E_0 does *not* validate the GPA again. It is straightforward to prove this property. If E_{hv} makes any changes to the RMP entry, the entry will become invalidated, thus the GPA is not bound to any SPA. If E_{hv} does not change the RMP entry, then the GPA remains bound to the same SPA as long as E_0 does not validate the GPA again.

With this property, we can prove the Lemma 1. If M is VM-private and validated in Ψ, then the GPA of M is bound to an SPA. This implies that the read operation on M by either E_0 or E_3 in Ψ' either returns the original value D or fails.

This lemma essentially requires that a valid state of Ψ will always ensure the VM-private M has unique bound from a GPA to an SPA no matter how the hypervisor changes the nested page mapping, as we discussed in Section 2.1. Such invariant property requires that E_0 does not validate a GPA when it is validated and bound to an SPA in the RMP.

Lemma 2. *Given a machine state Ψ, if a guest physical memory block in VM-private is mapped to a system physical memory M that stores a VM's secret S, then in any future hypervisor-reachable machine state Ψ', E_{hv}'s read operation on M will return an encrypted version of S.*

At first glance, one may assume that the VM-private memory page requires both the encryption bit in the guest page table and the validation bit in the RMP. However, our verification indicates that the validation bit in the RMP cannot be reliably guaranteed. When proving the lemma, we confirmed that holding the validation bit in the hardware state is not necessary. This implies that a requirement for a 'C' bit in the page table suffices to prove the lemma.

5.3.2 VMPL0-Private Memory

Lemma 3. *Let Ψ represent a machine state in which M is a VMPL0-private guest physical memory block that stores value D. Suppose Ψ' is a future reachable state through modifications made by E_{hv} and E_3. Then the E_0's read operation on M in Ψ' either fails or returns the original value D, and M cannot be read by E_3.*

Since VMPL0-private memory is a subset of VM-private memory, Lemma 1 and Lemma 2 implies that E_{hv} cannot read it or tamper its value. Here we focus on E_3. An RMP entry contains access permissions for each VMPL. These permissions control whether a VMPL can read, write, and execute on the memory. Furthermore, the hardware restricts E_3 from modifying access permissions for its own VMPL. To ensure that E_3 cannot access VMPL0-private memory, the verification process requires a precondition to `rmpadjust` that E_0 cannot grant access permission to VMPL3 if the memory is in VMPL0-private.

5.3.3 Correct Guest Address Translation

In addition to RMP updates, updating the page table is also critical for safe memory translation. We must ensure the integrity of the memory translation by considering all possible

changes from all entities. We verify it by proving the following lemma. The correctness of our guest address translation helps to prove the top theorem when considering accesses via guest virtual addresses instead of guest physical addresses.

Lemma 4. *Let Ψ represent a machine state in which a guest virtual address GVA is successfully translated to a system physical address SPA by a VMPL0's memory access. Suppose Ψ' is a future state reachable through modifications made by $\mathtt{E}_{\mathtt{hv}}$ and $\mathtt{E}_3$. Then, in Ψ', VMPL0's access to the GVA succeeds with the same translation to SPA or fails.*

Lemma 3 establishes that a GPA for VMPL0-private memory is either bound to a specific SPA or not bound to any SPA. When a GVA is successfully translated to a SPA in Ψ, it implies that the GPA that the GVA is mapped to is bound to the SPA. Therefore, in Ψ', the GPA is either still bound to the same SPA or not bound to any SPA. Since $\mathtt{E}_{\mathtt{hv}}$ and $\mathtt{E}_3$ cannot change $\mathtt{E}_0$'s page table, the translation from the GVA to the GPA remains the same. This guarantees that the GVA is either translated to the same SPA or fails.

To simply the implementation layer verification, we also prove the following lemma to ensure the mapping from GVAs to SPAs is one-to-one.

Lemma 5. *For each reachable Ψ, the mapping from a guest virtual address to a system physical address is a one-to-one mapping.*

Since Lemma 3 implies that the mapping from GPAs to SPAs for VM-private memory is one-to-one, we only need to ensure that the mapping from GVAs to GPAs is one-to-one. We prove it by separating memory writes into two categories: normal memory (`mem_write`) and page table memory (`pt_write`). To simplify the proof, we set aside a set of guest physical pages for $\mathtt{E}_0$'s page table (referred to as the PT memory), and enforce that the PT memory is always VMPL0-Private (by updating the precondition to `rmpadjust`). This allows us to define a precondition for `mem_write` and `pt_write` to check that a memory write falls into their respective categories.

A trusted initial assumption we make is that the initial page table for $\mathtt{E}_0$ at launch time is correct in the sense that the page table pages are in the PT memory and the page table enforces a one-to-one mapping from GVAs to GPAs. Then to prove it is true for any reachable Ψ, we prove that any `pt_write` operation preserves this property by adding a precondition to check that the memory write would keep the page table pages in the PT memory and the one-to-one mapping from GVAs to GPAs.

5.3.4 Connecting Machine Model to Implementation

The confidentiality and integrity of the VM-private and VMPL0-private memory, together with the correct page table translation, ensure that the memory content accessed by $\mathtt{E}_0$ through a guest virtual address remains consistent with the content stored in the hardware state. This consistency allows the implementation layer verification to focus on the software-tracked state, eliminating the complexity of having to worry about the actual hardware state. To convert preconditions for primitive operations from hardware-based to software-based in implementation verification, we prove that if an operation succeeds and the operation's software-based constraint is true, the corresponding hardware-based one must be true.

6 Implementation Verification

In this section, we describe the verification at the implementation layer. For simplicity, software in this section refers to the implementation of VERISMO. We first describe how we use permission-based verification to handle concurrency and scale verification to a large codebase. We then describe how we use information-flow verification to prevent secret leakage.

6.1 Permission-based Verification

In VERISMO, we incorporate the software constraints derived from the machine model verification into "tracked" permissions defined by Verus[22]. Each resource permission includes an identifier and multiple fields that represent the value and attributes of the resource. To ease the proof process across various memory access scenarios, we opt for implementing fine-grained memory permissions. This approach helps avoid the complexities tied to a single large-size global state (e.g., the hardware abstract model used in Section 5), and simplify the concurrency reasoning using ownership. Moreover, to aid in safe memory sharing, we introduce a lock permission for shared memory. To ensure safe register access, we establish register permissions in accordance with their definitions.

6.1.1 Memory Access Permission

Object-based Memory Permission. We extend the definition of a basic memory permission described in Verus to make all memory access safe in the context of AMD SEV-SNP VMs. A memory permission is defined as a `tracked` variable (`SnpPointsTo`) without the ability to be copied or constructed. By incorporating an appropriate initial assumption to ensure the initial uniqueness of all memory permissions, we can guarantee the uniqueness of each permission throughout the program.

As shown in Listing 8, our extended SNP memory permission consists of three elements: the guest virtual address (`addr`) as the permission identifier, the `value` stored at that address by the software, and the memory attributes (`swattr` and `hwattr`) as seen by both software and hardware. These memory attributes include RMP (`rmp`) and page table (`pte`) values tied to the memory. Furthermore, considering the specific use of page tables, we have added an attribute (`is_pt`) to

denote whether the memory serves as a page table. These improvements facilitate efficient management and enforcement of memory safety within the SEV-SNP VM context.

Listing 8: SNP object-based memory permission definition

```
pub ghost struct SnpMemAttr
{ rmp: RmpEntry, pte: PTAttr, is_pt: bool}

pub ghost struct SnpPointsToData<T> {
  addr: int, value: Option<T>,
  swattr: SnpMemAttr, hwattr: SnpMemAttr,
}

pub tracked struct SnpPointsTo<V>
{ _p: marker::PhantomData<V>, _ncopy: NoCopy }

impl<T> SnpPointsTo<T>
{ pub spec fn view(&self) -> SnpPointsToData<T>; }
```

Raw Memory Permission While object-based access permissions provide a user-friendly approach to object-oriented programming, VERISMO operates as a low-level security module, involving a significant number of raw memory operations.

To effectively support raw memory, it is necessary to establish additional foundational information concerning size, value casting, memory splitting, and merging. This essential ground-truth information is not provided by Verus and is defined by VERISMO as trusted specifications or axioms:

Object Size Specification. We assume that an object's size is equivalent to its actual memory usage. The precise size value should only matter when an operation has a specific size requirement (e.g., pvalidate requires page-sized) or when size comparisons are necessary (e.g., memory splitting or joining).

Casting between Objects and Bytes. Our trusted proof for casting aims to enforce unique bindings and ensure consistent sizing between objects and their corresponding byte representations.

Raw Memory Split and Merging. During memory splitting and merging operations, byte values and memory ranges are divided or combined, respectively, while memory attributes remain consistent with the original state.

Examples to Convert Unsafe Rust to Safe Verus. To illustrate how to use memory permission to convert Rust's unsafe memory access into Verus's safe memory access operation, we provide two dummy examples using memory primitive functions defined in Listing 9. The examples demonstrate how unsafe accesses can be identified through either verification (ϟ) or Rust's borrow checker (⊗).

The first example (in Listing 10) takes a memory permission reference pointing to a VMPL0-private memory at 0x1000, and thus it can borrow a value at address 0x1000. However, Line 8 cannot change content, since the permission is borrowed as immutable; Line 9 cannot access raw memory at 0x2000 due to the mismatched memory identifier.

Another example provided in Listing 11 demonstrates how the verification process detects unsafe RMP updates, ensuring the valid memory state. It initializes an non-validated memory permission and then assign it to VMPL1 at the end to render the memory accessible to VMPL0. After `pvalidate`, the memory permission remains not ready for other RMP memory operations until the operation is confirmed and the memory content is cleared. The strict requirement leads to a failed assertion at Line 6. Additionally, Line 11 fails since the pvalidate primitive function requires no double validation.

Listing 9: A selective primitive memory-related functions

```
fn borrow<'a>(vaddr: usize, Tracked(mperm): Tracked<&'a
      SnpPointsTo<V>>) -> (v: &'a V)
requires
  mperm@.wf_borrow(vaddr as int),
ensures
  mperm@.spec_read_rel(*v),
{...}
fn replace(vaddr: usize, in_v: V, Tracked(mperm):
     Tracked<&mut SnpPointsTo<V>>)
requires
  old(mperm)@.wf_replace(vaddr as int, in_v),
ensures
  mperm@.spec_write_rel(old(mperm)@, Some(in_v)),
{...}
fn pvalidate(vaddr: u64, psize: u64, val: bool, rflags:
      &mut u64, Tracked(mperm): Tracked<&mut
     SnpPointsToRaw>) -> (ret: u64)
requires
  spec_pval_requires(vaddr as int, psize, val, old(
     mperm)@),
ensures
  spec_arch_pval(vaddr as int, psize, val, old(mperm)@,
      mperm@, *old(rflags), *rflags, ret),
{...}
fn rmpadjust(vaddr: u64, psize: u64, attr: RmpAttr,
     Tracked(mperm): Tracked<&mut SnpPointsToRaw>) -> (
     ret: u64)
requires
  spec_rmpadjust_requires(vaddr as int, psize as int,
     attr, old(mperm)@),
ensures
  spec_arch_rmpadjust(old(mperm)@, mperm@, vaddr as int
     , psize as int, attr),
{...}
```

Listing 10: Secure access to memory

```
fn access_private(Tracked(mperm): Tracked<&SnpPointsTo<
     u64>>)
requires
  mperm@.wf_not_null_at(0x1000),
  mperm@.is_vmpl0_private() {
✓let val1 = *borrow(0x1000, Tracked(mperm));
✓let val2 = *borrow(0x1000, Tracked(mperm));
✓assert(val2 == val1);
⊗replace(0x1000, 0x1234, Tracked(mperm));
ϟ let _val3 = *borrow(0x2000, Tracked(mperm));
}
```

Listing 11: Safe RMP table updates

```
pub fn init_page(Tracked(mperm): Tracked<SnpPointsToRaw>)
requires
  mperm@.wf_range((0x1000, PAGE_SIZE)) && mperm@.is_init() {
  let mut u64 rflags = 0;
✓let ret = pvalidate(0x1000, 0, 1, &mut rflags, Tracked(&mut mperm));
ϟ assert(mperm@.wf());
  if ret == 0 && rflags != 0 { return false; }
✓mem_set(0x1000, PAGE_SIZE, 0, Tracked(&mut mperm));
✓assert(mperm@.wf());
ϟ pvalidate(0x1000, 0, 1, &mut rflags, Tracked(&mut mperm));
  let rmpattr = RmpAttr::empty().set_vmpl(1).set_read(1).set_write(1);
✓rmpadjust(0x1000, 0, rmpattr, Tracked(&mut mperm));
  if ret  != 0 { return false; }
✓assert(!mperm@.is_vmpl0_private());
  return true;
}
```

6.1.2 Lock Access Permission

The memory permission we previously described does not entirely address the issue of concurrency reasoning. This challenge emerges because memory permissions, in their current form, do not inherently facilitate the shared write permission for concurrent access. For example, when CPU-A moves a memory permission to CPU-B, CPU-A subsequently loses access to that memory. How to retrieve the memory permission back is unclear without introducing locking or atomic permission mechanisms.

To enable safe concurrent memory access in a relaxed memory model, we choose to implement locking permissions since much of our code relies on locks to protect shared resources without directly using atomic operations. Each shared resource starts with a lock permission which stores a memory permission. When the software acquires a lock, the lock permission is converted to a locked state and returns the stored memory permission so the software can use the memory permission to access the shared resource. When the lock is released, the memory permission is returned back to the lock permission whose state is converted back to an unlock state.

Shared objects typically require an **invariant** to constrain their values. Without proving the invariant, for all verifications necessitating such a constraint, programmers may incur extra execution costs to check if the value meets the requirements. To maintain the invariant for values read after acquiring a lock, we have introduced a precondition in the release API. This precondition mandates that the object associated with the memory permission upholds the invariant whenever the release method is invoked. As a result, the value of the global variable read by a CPU is always in compliance with the invariant. The lock mechanism in VERISMO extends beyond shared memory (see Section 7.2). Listing 12 shows an example to use lock to protect a global variable `gvar` while keeping

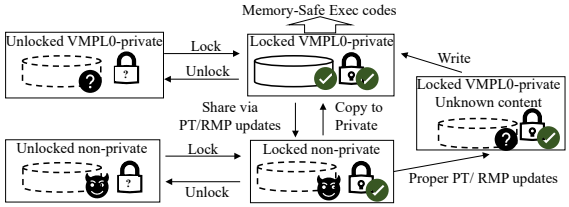

Figure 4: Safe memory access under concurrency

its invariant (`spec_gvar_inv`). The verified code guarantees that the data borrowed satisfies invariant (Line 12) after acquiring the lock, although the data could be different in two lock transactions (Line 13). Our trusted lock APIs can detect the violation of invariant for `gvar` since it fails the invariant at Line 15.

Listing 12: Lock protection

```
spec fn spec_gvar_inv() -> spec_fn(u64) -> bool
{ |v: u64| 0 <= v <= 0xff_ffff}

fn access_global(Tracked(core): Tracked(SnpCore), Tracked(lperm): Tracked(LockPerm<u64>))
requires
  lperm.is_unlocked(spec_gvar) {
✓let (vaddr, Tracked(pt_mperm)) = gvar().acquire(Tracked(&mut lperm), Tracked(&core));
✓let val1 = borrow(vaddr, Tracked(&pt_mperm));
✓gvar().release(Tracked(&mut lperm), Tracked(&core), Tracked(pt_mperm));
✓let (_, Tracked(pt_mperm)) = gvar().acquire(Tracked(&mut lperm), Tracked(&core));
✓let val2 = borrow(vaddr, Tracked(&pt_mperm));
✓assert(spec_gvar_inv()(val1) && spec_gvar_inv()(val2));
ϟ assert(val1 == val2);
✓replace(vaddr, 0x1000_0000, Tracked(&pt_mperm));
ϟ gvar().release(Tracked(&mut lperm), Tracked(&core), Tracked(pt_mperm));
}
```

6.1.3 VMPL0's Memory Safety

By utilizing the memory permission and the lock permission reasoning, we ensure that our implementation always has safe memory access. Here we describe ownership for VERISMO at a high level utilizing the permissions we described in Section 6.1.1 and Section 6.1.2.

Figure 4 illustrates safe operations for handling memory and lock permissions. When VMPL0-private memory is locked by the software on a CPU, no other entity can concurrently modify it. Therefore, the memory can be considered as private to VMPL0, and the acquire operation will safely grant a memory permission for unrestricted memory access. For memory shared with either the hypervisor or the guest OS, the memory permission obtained upon locking only permits copying and writing, but not borrowing references. For unrestricted operations, developers must either copy the mem-

ory to VMPL0-private memory or use rmpadjust/pvalidate operations to transition it to VMPL0-private memory.

6.1.4 Register Access Permission

Similar to memory permissions, we define register permissions for registers. Most registers are privately controlled by the software and thus we usually can maintain a stable invariant property after proper booting steps. Since a CPU has a fixed number of registers, each CPU uses a single permission representing all the CPU's registers together. Some registers need special permission designs.

In the SEV-SNP VM environment, the *GHCB MSR* differs from other registers. While all other registers are owned by the VM, the GHCB can be read and written by the hypervisor. To handle this difference, we introduce a "share" attribute for such registers, treating the values read from these shared registers as unconstrained.

For registers like *IDTR* and *GDTR*, which hold pointers to memory segments, we guarantee that memory ownership is transferred to the hardware upon writing to these registers.

Certain registers, such as the instruction pointer (*RIP*) and stack pointer (*SP*), pose a risk of inadvertently influencing software behavior in dangerous ways. We prevent explicit use of these registers by removing permissions to access them.

CR3 points to the top-level page table. A write operation to *CR3* is required to have tracked mutable permissions for both the page table and all usable virtual memory addresses. This safeguards against incorrect modifications to CR3, as consolidating all memory permissions used by the CPU is complex. VERISMO does not have a user mode, and has a single page table for its kernel mode code. Therefore its CR3 register is only updated once at the boot time.

6.2 Information-Flow Verification

In addition to guaranteeing safe memory access, it is important to prevent secret leakage through explicit or implicit information flows. In VERISMO, we define a precise tracking policy to maintain secure information flow by monitoring potential secret guessing spaces for variables.

Types carrying secret guessing space. We define security types to match Rust's primitive types. Each security type includes a value and a set of possible values (`valset`) that an entity can guess from. If the set is complete, the variable is considered a secret to the entity; if the set is singleton, the variable is public to the entity.

VERISMO only takes three kinds of secrets: the VM communication keys generated by the hardware (Section 2.1), VERISMO's own private/public key pair (Section 3.4), and symmetric encryption keys for the guest OS (Section 3.4). Since these secrets are exclusively used by trusted cryptographic functions in VERISMO, we only encounter a limited number of proofs about the set size when invoking cryptograhic functions. With a trusted and formally verified crypto library, our security checking is highly simplified to check whether a variable is fully public or confidential to an entity, similar to taint tracking without over/under-tainting concerns.

We define a security trait that assigns security levels to different data types based on the size of their possible value sets. Primitive types in Rust have security levels equivalent to constants. To use security types like primitive types, we implement standard operator traits and ensure the correct propagation of the guessing space by applying the relevant set operations to the possible value sets. For example, each binary operation 'op' performed between variables a and b ensures the following constraints for the returned result.

$$(a \text{ op } b).\text{valset} \equiv \left\{ \text{val} \middle| \exists (v,u) : \begin{array}{r} v \in \text{valset}_a \wedge \\ u \in \text{valset}_b \ \wedge \text{val} \equiv a \text{ op } b \end{array} \right\}$$
$$\wedge (a \text{ op } b).\text{val} \equiv (a.\text{val op } b.\text{val})$$

A comparison operation may lead to secret leakage via control flow. Therefore, the comparison operator includes a precondition that requires both variables to be public to all entities. We support the secret downgrade through a trusted function when needed. After a downgrade, a variable's possible value set becomes a singleton and thus can no longer be used as a secret. For instance, if a downgrade operation is applied to a secret key, the key no longer meets a precondition for trusted cryptographic functions, which requires cryptographic key to be a high-security variable, i.e., the possible value set if full.

Re-visit the memory permission for confidentiality. To maintain the confidentiality of secret variables, we impose a requirement that the software must not share them with other entities. To enforce this property, we ensure that every valid memory permission maintains a consistent relationship between the memory's confidentiality attribute (defined by the page table and the RMP) and the security level of the value it carries, as shown below:

$$\forall \text{ entity } \neg \text{memperm.swattr.is\_confidential\_to(entity)}$$
$$\implies \text{memperm.val.is\_constant\_to(entity)}$$

In addition, we cannot use secret data as address for memory access; otherwise, the hypervisor can use control flow to infer the secret. We enforce this precondition for all trusted functions related to memory access.

7 Implementation

We implemented VERISMO in Rust and verified it with Verus, ensuring that the trusted functions are always used safely, and our implementation is correct. Our implementation is available at `https://github.com/microsoft/verismo`.

7.1 TCB in VERISMO

VERISMO fully trusts a small number of primitive functions, ground-truth axioms, the hardware model specification, the model-based specification defining the safety properties, as well as the specification for functional correctness. These trusted functions wrap unsafe code for interacting with the hardware for calling trusted external libraries (e.g., Rust core, HACL[35]). The size of the trusted code will be reported in the evaluation section. Additionally, VERISMO places trust in Verus and the Rust compiler.

7.2 An example with simplified verification

The permission-based verification not only guarantees memory confidentiality and integrity but also ensures correctness for certain functionalities without introducing additional specifications. Here, we use an example in VERISMO to explain how a lock can automatically protect associated resources and how memory permissions automate functional correctness for a Guest-VERISMO call to update a memory page's attributes.

VERISMO assigns certain memory ranges to VMPL3 and stores their *ranges* (for execution) along with a zero-sized tracked permission *map* (for proof) in a global variable (`OSMEM`). The `OSMEM` is shared by multiple cores and is protected by a lock that guards an invariant, ensuring that a memory page has its tracked permission inside the *map* if and only if the page is within the memory *ranges*.

Extended lock protection. When VMPL3 sends a request to VERISMO to grant or revoke permission for a page, VERISMO needs to acquire the lock to access the tracked permission from `OSMEM`. This automatically prevents concurrent changes to the RMP table for VMPL3's memory without introducing new locks. Here, a single lock protects both the `OSMEM` variable and the memory assigned to VMPL3.

Extended Functional Correctness. Additionally, since the tracked map only stores page permissions from the `OSMEM`-defined ranges, VERISMO must check whether the requested page is within the range to obtain the page permission required for updating the memory's attributes. This automatically ensures that a correct handler will conduct the necessary checks before any update happens.

8 Evaluation

Our performance evaluation is based on the Hyper-V hypervisor. Our experimental machine has an AMD EPYC 7543P 32-Core Processor, which supports SEV-SNP features. We allocated 8 dedicated cores to the host domain using minroot Hyper-V, allowing the hypervisor to assign the remaining 24 CPUs to VMs. This setup prevents unpredictable competition for CPU resources between different domains.

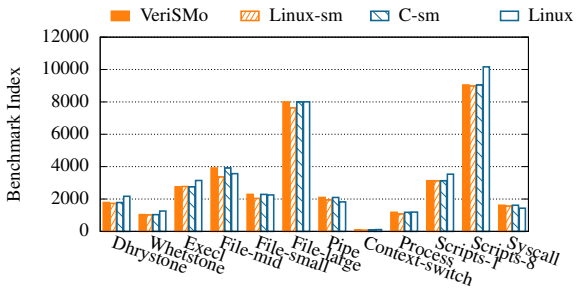

Figure 5: Unix benchmark results

8.1 Performance

To measure the performance of guest-VERISMO APIs, we created a kernel driver that sends these requests to VERISMO. We ran each call 100 times and recorded the average cost per call in Table 1.

Table 1: Microbenchmarking guest-VERISMO APIs

Request	Cycle (*1k)	Time (ms)	STD (ms)
`Switch`	34	0.012	0.002
`ExtendPCR`	37	0.013	0.002
`SetPageShared`	77	0.027	0.002
`SetPagePrivate`	82	0.029	0.002
`AttestPCR`	17633	6.298	4.270
`LockKernel`(8GiB)	4260969	1522	151
`LockKernel`(4GiB)	2201565	786	58

The `Switch` call serves as a test to complete a switch (VMPL3 → hypervisor → VMPL0 → hypervisor → VMPL3), and its cost prevails over some less expensive GVCA calls. The operation to extend a measurement (`ExtendPCR`) is quick, as it only involves adding a value from VMPL3 to the previous one and updating it with a cryptographic hash. `SetPageShared` is relatively more expensive since VERISMO needs to invalidate it and call the hypervisor to make a page shared which triggers additional context switch. `SetPagePrivate` is marginally more expensive than `SetPageShared`, as the former requires an additional rmp change via `rmpadjust`. Attestation is much more resource-intensive than others due to the need for hardware-based cryptograhic signing. Kernel code integrity protection (`LockKernel`), operating on all guest memory, is costly, but it is performed only once per VM session.

8.2 Performance Impact on Guest OS

We used UnixBench to measure the performance of a Linux kernel running with the VERISMO security module at VMPL3. The benchmarking results are presented in Figure 5. The security module introduces almost zero overhead. In contrast, Hecate [12], a compatibility-oriented L1 hypervisor in

Table 2: Lines of code.

	Exec	Spec	Proof	Axiom	Trusted
Verus	906	1361	2388	803	101
Model	0	2194	3188	67	0
HACL	107	7	0	0	90
Macro	2093	-	-	-	-
Common	606	603	2101	55	10
RawMem	640	635	324	105	11
Reg	341	162	22	0	12
Lock	183	145	105	0	7
PageTable	354	261	330	0	1
Alloc	386	164	1003	24	30
Core	3205	1013	3884	14	0

VMPL0, can incur up to a 40% slowdown. This is because our security module does not trap any guest OS's #VC Exception.

In some case, formal verification can be simplified in a less-efficient implementation or with extra runtime checks. Here, we compared VERISMO with two additional prototypes that we implemented, demonstrating no performance trade-off introduced by verification. While those prototypes—'linux-sm' (a modified Linux kernel as security module) and 'c-sm' (C-based security module)—only supports `WakeUpAP`, `SetPageShared`, and `SetPagePrivate` APIs, they serve as baselines for the fully-featured VERISMO. The comparable performance of VERISMO demonstrates that we did not sacrifice performance for the sake of verification.

8.3 Verification Size and Performance

Table 2 shows the number of lines of codes for all used libraries and modules in our code. 'Verus' represents modules VERISMO used from Verus's basic library, 'HACL' wraps 3 functions for encryption and hashing from an external formally verified HACL crate written in C and assembly. 'Core' represents code for VERISMO's core functionalities. The trusted executable code (majorly unsafe code) are from three categories. We only used 31 LOC unsafe Rust for hardware primitive operations. 'RawMem', 'Reg', 'Lock' and 'PageTable' include the trusted primitive operations for memory and registers. 'HACL', 'Common' includes external safe functions with a trusted postcondition. 'Alloc' includes 1 unsafe block to trust the global allocator interface. Although this allocator interface must be trusted, as the Rust compiler depends on it, we have verified the correctness of the implementation of our global allocator. On average, the proof and specification annotations are approximately twice the size of the executable code for the implementation layer. The machine-model layer is reusable if we do not change primitive functions. Verification of both model and implementation layers takes around 6 minutes with a multi-threaded Verus backed by Z3-4.11.2 as the solver running on our test machine with 32 cores. The verification efficiency is due to Verus's optmization for Z3 solver, Rust's ownership checking for automated memory reasoning, and the modularity of our proof code with Verus.

8.4 Security Improvement

Concurrently with our project, AMD SVSM [4] which is now replaced by COCONUT SVSM [39], is an ongoing Rust project to provide a security module in VMPL0 for AMD SEV-SNP VMs. However, they do not apply formal verification and heavily use unsafe Rust features. COCONUT uses 235 unsafe blocks, and AMD SVSM uses 150 unsafe blocks. In contrast, VERISMO uses only 32 unsafe blocks. We did not compare our performance with them since VERISMO runs on a different hypervisor. Since they share the same hardware model as VERISMO, our verification design can potentially be applied to their code.

8.4.1 An Example Bug Detected in VERISMO

Here, we showcase the importance of formal verification, using an unsafe memory update in Listing 13 that we detected via verification. This code handles the `SetPageShared` request from VMPL3. Before modifying the RMP entry for a memory page used by VMPL3, a traditional approach is to check the security of the change by examining whether the content carried in the guest physical page can be released externally. It uses a lock to protect VMPL3's memory to prevent concurrent updates. However, these measures are not sufficient for ensuring VERISMO's confidentiality when taking into account a malicious hypervisor.

Listing 13: Handling a memory state change request

```
fn SetPagePrivate(gpn: u64, attr: RmpAttr, lperm:
     Tracked<LockPerm>) -> Tracked<LockPerm> {
  let gpn = vn_to_pn(gvn);
  let tracked mut lperm = lperm;
✓let osmem = OSMEM.acquire(Tracked(&mut lperm));
✓match osmem_check(osmem, gpn, attr) {
  Ok(i) => {
    let Tracked(mut pperms) = osmem[i].pperms;
  ✓let tracked mut pperm = pperms.tracked_remove(gvn);
  ✓pvalidate(gvn, true, Tracked(&mut pperm), ..);
  ϟ rmpadjust(gvn, 1, attr, Tracked(&mut pperm), ..);
```

Consider two system physical memory pages: SPN_0 and SPN_3. SPN_0 is mapped to GPN_0, while SPN_3 is mapped to GPN_3. The memory at SPN_0 holds VMPL0's secret key, which must remain confidential to VMPL3 and the hypervisor. Meanwhile, the memory at GPN_3 is allocated to VMPL3, and thus its access permissions can be adjusted upon VMPL3's requests. VMPL3 could gain access to the secret key at SPN_0 using the following steps (with the hypervisor's help):

1. VMPL3 requests VERISMO to transition GPN_3 to a shared status. VERISMO fulfills the request by invalidating the target memory and recording that GPN_3 is now invalidated since it is assigned to VMPL3.

2. To perform the attack, a malicious hypervisor binds SPN_0 with GPN_3 by executing `rmpupdate` and updating the nested page table to map GPN_3 to SPN_0.

3. When VMPL3 asks VERISMO to revert GPN_3 back to private, VERISMO validates the memory page at GPN_3 and adjusts permissions to permit VMPL3 access to the memory. This occurs because VERISMO incorrectly assumes that GPN_3 was shared and does not contain VMPL0's secret.

As a result of the attack, VMPL3 obtains access to SPN_0 via GPN_3, which violates VERISMO's confidentiality. The hardware's *failed-to-be-secure* design only detects the attack when there is an attempt to access the memory at GPN_0, as the memory binding for GPN_0 becomes invalid.

Without verification, detecting such a security vulnerability can be challenging. Its solution is straightforward: simply clear a memory page after validating it but before assigning it to VMPL3.

We reported this vulnerability to AMD in early May 2023, which resulted in a security fix [5]. However, COCONUT SVSM [39], which reuses a significant amount of code from AMD SVSM, still had the old buggy code at its early stage.

9 Related Work

Trusted execution environments. Traditional hypervisor-enforced VM isolation no longer meets the minimal trust requirements. This has led to a recent trend of adding an extra software layer as a trusted security monitor to deal with hypervisor-based attacks. Examples include Keystone [23], Komodo [11], and the Intel TDX module [18], which enforce enclave isolation and provide security features. Arm CCA [6] goes a step further by relying on two external layers for VMs: a Realm Management Monitor (RMM) for managing realms and a separate monitor for facilitating interactions between the RMM and the hypervisor. However, the issue of separating security domains within an enclave remains largely unaddressed. Different from these solutions, the security module [2, 4, 39] for AMD SEV VM [1] is isolated from both hypervisor and other in-TEE softwares in a VM.

Model checking. Formal methods can be applied to both model-level (i.e., model checking) and implementation-level verification. Model checking tools (e.g., SPIN[17], Tamarin[30]) formally check some properties based on an abstract model of a system design (e.g., [15, 20]) to prove the correctness. Consequently, the correctness is proved without directly connecting to the implementation itself.

Software verification. Software projects (e.g., [11, 16, 29, 35, 40]) can be implemented in verification-friendly languages, such as Dafny [25] and F*[34], to enable end-to-end verification for both model and implementation. Many OS projects (such as [7, 10, 14, 21, 24, 26, 32]), based on unsafe assembly or C , are verified in proof languages. The implementation-level verification is made possible through a trusted language transformer from unsafe C. Due to the nature of unsafe C, however, their memory safety proofs require more effort than proofs about Rust. A recent study [9] uses verification to check secure information flow for confidential computing but assumes memory safety by requiring no unsafe Rust in code, which is not applicable to OS-level code.

Security model for verified OS. OS-level verification efforts ([7, 11, 14, 21, 26, 33]) typically focus on the traditional security model, either with a trusted hypervisor or without one. Some recent efforts, such as [27], have verified the security of the Arm CCA, ensuring the proper implementation of the RMM firmware for isolating multiple enclaves or VMs. Different from those works, we use the permission-based method to verify proper memory access, encoding both the state of the memory and the security level of its content. This approach simplifies our proof and streamlines our verification process, eliminating the need for additional abstract layers for proving concurrency and information flow. In addition, we verify memory accesses without extra proof efforts for the safe portion of the Rust code.

10 Conclusion

We developed and implemented VERISMO, the first verified security module for confidential VMs enabled by AMD SEV-SNP. Operating at the highest privilege level, the security module provides protection to the guest while maintaining its own confidentiality and integrity, even in the presence of an untrusted concurrent hypervisor. Our verification process validated the security and correctness of the security module software, building upon our specifications of the AMD hardware primitives. Utilizing Rust's ownership and borrowing, the verification showcased the application of concepts from Verus's permission model on a large scale, encompassing thousands of lines of verified concurrent executable code. Our evaluation demonstrated that our security module is efficient and our verification is scalable.

Acknowledgments

We thank Verus team for sharing their insights into permission-based reasoning and providing desired features for verifying VERISMO. The detailed reviews we received from OSDI'24 and the feedback from our shepherd Nickolai Zeldovich helped us greatly improve the paper.

References

[1] AMD. Strengthening VM isolation with integrity protection and more, 2020. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[2] AMD. Secure VM service module for SEV-SNP guests. *White Paper, August*, 2022. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf.

[3] AMD. *AMD64 Architecture Programmer's Manual (v4.07)*. Volume 3: General-purpose and system instructions edition, 2023. https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5.

[4] AMD. Secure VM service module for SEV-SNP guests. 2023. https://github.com/AMDESE/linux-svsm.

[5] AMD. Linux secure VM service module security fix for hypervisor-based attacks, 2023. https://github.com/AMDESE/linux-svsm/commit/0de111e9b85a340203759a3ab217a3e2f2be4b0b.

[6] Arm. Arm confidential compute architecture (Arm CCA), 2021. https://www.arm.com/products/security/arm-confidential-compute-architecture.

[7] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium*, volume 152, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[8] J. Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003. doi: 10.5555/1760267.1760273.

[9] H. Chen, H. H. Chen, M. Sun, K. Li, Z. Chen, and X. Wang. A verified confidential computing as a service framework for privacy preservation. In *32nd USENIX Security Symposium*, pages 4733–4750, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/chen-hongbo.

[10] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics: 22nd International Conference*, pages 23–42. Springer, 2009. doi: 10.1007/978-3-642-03359-9_2.

[11] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *26th ACM Symposium on Operating Systems Principles*, pages 287–305, 2017. doi: 10.1145/3132747.3132782.

[12] X. Ge, H.-C. Kuo, and W. Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *29th ACM Conference on Computer and Communications Security*, pages 1231—-1242, New York, NY, USA, 2022. ISBN 9781450394505. doi: 10.1145/3548606.3560592.

[13] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987. doi: 10.1016/0304-3975(87)90045-4.

[14] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, volume 16, pages 653–669, 2016. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[15] R. Guanciale, M. Balliu, and M. Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *27th ACM Conference on Computer and Communications Security*, pages 1853–1869, 2020. doi: 10.1145/3372297.3417246.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *25th ACM Symposium on Operating Systems Principles*, pages 1–17, 2015. doi: 10.1145/2815400.2815428.

[17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi: 10.1109/32.588521.

[18] Intel. Intel TDX module v1.5 base architecture specification, 2023. https://cdrdv2.intel.com/v1/dl/getContent/733575.

[19] Intel. Intel trust domain extensions (Intel TDX) module TD partitioning architecture specification, 2023. https://cdrdv2.intel.com/v1/dl/getContent/773039.

[20] M. K. Jangid, G. Chen, Y. Zhang, and Z. Lin. Towards formal verification of state continuity for enclave programs. In *30th USENIX Security Symposium*, pages 573–590, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/jangid.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification

of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009. doi: 10.1145/1629575.1629596.

[22] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *38th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2023.

[23] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *15th European Conference on Computer Systems*, pages 1–16, 2020. doi: 10.1145/3342195.3387532.

[24] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Formal Methods: Second World Congress*, pages 806–809. Springer, 2009. doi: 10.1007/978-3-642-05089-3_51.

[25] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: In 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

[26] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A secure and formally verified linux kvm hypervisor. In *42nd IEEE Symposium on Security and Privacy*, pages 1782–1799. IEEE, 2021. doi: 10.1109/SP40001.2021.00049.

[27] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the Arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 465–484, 2022. https://www.usenix.org/conference/osdi22/presentation/li.

[28] Z. Li, J. Wang, M. Sun, and J. C. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *2021 ACM Conference on Computer and Communications Security*, page 2183–2196, New York, NY, USA, 2021. ISBN 9781450384544. doi: 10.1145/3460120.3484541.

[29] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–304, 2013. doi: 10.1145/2451116.2451148.

[30] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *25th International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013. doi: 10.1007/978-3-642-39799-8_48.

[31] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2.

[32] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy*, pages 415–429, 2013. doi: 10.1109/SP.2013.35.

[33] S. Peters, A. Danis, K. Elphinstone, and G. Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–7, 2015. doi: 10.1145/2797022.2797042.

[34] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. doi: 10.1145/3110261.

[35] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *41st IEEE Symposium on Security and Privacy*, pages 983–1002, 2020. doi: 10.1109/SP40000.2020.00114.

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

[37] A. Sadiq, Y.-F. Li, and S. Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, 159:110450, 2020. doi: 10.1016/j.jss.2019.110450.

[38] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde. WeSee: Using malicious #VC interrupts to break AMD SEV-SNP. In *45th IEEE Symposium on Security and Privacy*, 2024. https://ahoi-attacks.github.io/wesee.

[39] SUSE. COCONUT-SVSM, 2023. https://github.com/coconut-svsm/svsm.

[40] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *24th ACM Conference on Computer and Communications Security*, pages 1789–1806, 2017. doi: 10.1145/3133956.3134043.