

# LOCI: Enabling Shared Memory for Confidential Computing on AMD SEV-SNP

Guillaume Prémel\*  
ETH Zurich

Benedict Schlüter\*  
ETH Zurich

Supraja Sridhara  
ETH Zurich

Jagadish Kotra  
AMD

Shweta Shinde  
ETH Zurich

**Abstract**—Confidential computing has gained widespread industry adoption in recent years. AMD SEV-SNP is a widely used solution for enabling confidential computing in the cloud in the form of confidential virtual machines (CVMs). It strictly isolates CVMs from the hypervisor and each other. However, this strict isolation precludes fundamental shared memory primitives, preventing efficient deployment of data-intensive, multi-tenant workloads. Without shared memory, large datasets must be duplicated across isolated instances, leading to memory bloat.

While recent research proposes firmware extensions to enable shared memory on emerging architectures like Arm CCA, enabling confidential shared memory on widely deployed x86 trusted execution environments (TEEs) like AMD SEV-SNP without hardware modifications has remained an open problem. To solve this, we present LOCI, a system that introduces SubVMs to enable secure, confidential shared memory regions on AMD SEV-SNP. By leveraging a novel observation regarding the virtual machine privilege level (VMPL) permission hierarchy, LOCI allows multiple mutually distrusting SubVMs to map and operate on shared memory securely. Crucially, LOCI extends this shared memory domain beyond CPU cores by integrating with SEV-TIO, securely attaching hardware accelerators to SubVMs natively at the VMPL boundary.

We evaluate LOCI using large language model (LLM) inference as a representative workload that uses our confidential shared memory primitive. Operating on unmodified LLM applications across eight models, LOCI proves highly efficient: it reduces main memory capacity requirements by  $2.77\times$  on average. At the same time, LOCI does not negatively impact performance and may improve it by leveraging shared caching benefits.

**Index Terms**—confidential computing, confidential virtual machines, memory sharing, trusted execution environments, virtualization.

## 1. Introduction

Confidential computing has emerged as a crucial technology for securing cloud deployments, enabling users to protect their sensitive data and proprietary code from untrusted cloud providers and co-tenants. Modern hardware trusted execution environments (TEEs), such as AMD SEV-SNP [1], Intel TDX [2], and Arm CCA [3], facilitate this

by providing hardware-isolated confidential virtual machines (CVMs). Major cloud platforms rapidly adopted these technologies to offer secure VM-based hosting and confidential containers [4]–[7].

However, the strict memory isolation that underpins the security of CVMs introduces fundamental systems challenges. Unlike traditional VMs, which routinely leverage memory deduplication (e.g., Linux kernel samepage merging (KSM)) and shared memory abstractions to reduce capacity requirements, CVMs cannot efficiently share private memory with each other. Each CVM instance must independently load its execution environment and necessary datasets into its isolated memory space. For data-intensive microservices and serverless architectures, where multiple instances often rely on the same underlying read-only state, this lack of shared memory leads to severe memory bloat and high cold-start latencies [8], [9].

Furthermore, modern cloud workloads are highly heterogeneous, increasingly relying on hardware accelerators like GPUs to meet computational demands. New extensions like AMD SEV-TIO aim to securely bind accelerators to CVMs. The absence of secure shared memory primitives exacerbates inefficiencies in these heterogeneous deployments. Data must be redundantly loaded across isolated CVMs before being dispatched to accelerators, bottlenecking both memory capacity and system performance.

Existing approaches to enable shared memory for TEEs fall short for these demanding deployments. Sharing via the untrusted host’s memory violates confidentiality and integrity guarantees. Alternative software-based techniques, such as bounce buffers, preserve security but introduce prohibitive access latencies due to runtime encryption and copying overheads. While secure shared memory views have been explored on academic RISC-V platforms [10], [11], no equivalent capability exists on the commodity x86 or Arm TEEs offered by public clouds today.

We investigate AMD SEV-SNP to determine if we can construct a confidential shared memory primitive without modifying the hardware. SEV-SNP introduces virtual machine privilege levels (VMPLs) to divide a single CVM’s memory into hierarchical trust domains (e.g., VMPL0 for a secure virtual machine service module (SVSM), VMPL1 for the guest OS). Several prior works observed that higher-privilege VMPLs can access the memory of lower-privilege VMPLs [12]–[14], suggesting that VMPL-based memory lacks the strict mutual isolation required to securely share data between distinct, untrusting instances.

\* denotes equal contribution

In this paper, we make a novel observation regarding the VMPL permission model: while a more privileged VMPL can reduce the memory permissions of a less privileged VMPL (e.g., changing `rwX` to `rw-`), it can never escalate them. By carefully managing these one-way permission transitions, we can safely enforce confidential shared memory views across different VMPLs. Even if a privileged VMPL acts maliciously, it can only revoke access but cannot compromise the confidentiality or integrity of the shared data. When the hypervisor forwards the resulting fault, a trusted SVSM can detect the revocation and stop the offending SubVM.

Based on this insight, we present LOCI, a system that introduces the abstraction of SubVMs to enable secure, hardware-isolated shared memory on AMD SEV-SNP. LOCI uses the highest privilege level (VMPL0) to run a trusted SVSM. This SVSM creates and manages up to three mutually distrusting SubVMs in VMPLs 1–3 and enables secure confidential memory sharing between SubVMs through its APIs. Crucially, LOCI extends this shared memory abstraction to heterogeneous devices by integrating with SEV-TIO, allowing hardware accelerators to be securely attached directly to SubVMs and operate on the shared memory regions. While our design supports SEV-TIO, we do not empirically evaluate this support because compatible accelerators are unavailable at the time of writing.

To evaluate LOCI, we use large language model (LLM) inference as a representative heterogeneous, data-intensive workload. LLM inference exemplifies the need for scalable CVMs, massive shared datasets (model weights), and accelerator integration. We implement LOCI with minimal changes to the existing AMD SEV-SNP stack, adding 3717 lines of code (LoC) to the shared trusted computing base (TCB) of Coconut SVSM [15] and  $\approx 2100$  LoC to the Linux kernel for SubVM support. Figure 1 shows an architectural overview of the LOCI case studies, in which we use confidential memory sharing to reduce system memory pressure.

**Results.** We execute unmodified `llama.cpp` workloads across eight popular open LLMs (ranging from 3B to 33B parameters). LOCI achieves inference performance comparable to baseline CVM execution while drastically reducing overheads. By enabling memory sharing, LOCI reduces memory capacity requirements by  $\approx 2.77\times$  on average and enables near-instant spin-up compared to loading duplicate models. Furthermore, by exploiting the shared memory architecture to optimize data locality, LOCI achieves up to a 9.36% performance improvement over standard CVMs.

LOCI makes the following contributions:

- *Observation.* We identify a crucial property of the AMD SEV-SNP VMPL permission hierarchy that allows the creation of fully isolated entities.
- *SubVMs.* Based on the previous observation, we design a novel abstraction we call SubVMs, which allows running up to three mutually distrusting kernels inside a single SEV-SNP CVM.

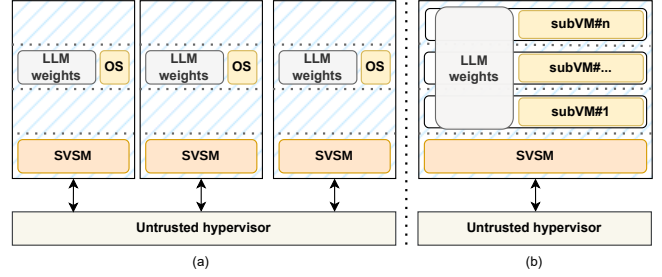


Figure 1: Case study: (a) Memory duplication across three CVMs running the same LLM model. (b) Memory sharing for an LLM model.

- *Memory Sharing.* We enable memory sharing across mutually distrusting SubVMs and securely integrate heterogeneous accelerators via SEV-TIO, reducing memory capacity requirements.
- *Evaluation.* Using LLM inference as a demanding case study, we demonstrate that LOCI scales efficiently across eight diverse models, matching baseline performance, enabling 9.36% locality-driven speedups, and reducing the memory footprint by a factor of 2.77.

## 2. Background

We introduce the relevant background for LOCI.

**AMD SEV-SNP.** AMD SEV-SNP is a hardware feature that enables CVMs. SEV-SNP ensures the confidentiality and integrity of CVM code and data in the presence of an untrusted hypervisor. It adds several new platform features to achieve these security guarantees [1]. With SEV-SNP, the hypervisor continues to manage the VM (e.g., scheduling, memory management) using the virtual machine control block (VMCB). The VMCB contains two distinct areas: a hypervisor-managed control area and a guest-managed encrypted save area (VMSA). The hypervisor control area holds CVM-related data such as the interrupt vectors, nested `cr3` values, instruction intercept bits, and the physical address of the VMSA. The VMSA contains the encrypted guest state and is used by microcode to restore the guest’s context on `vmrun`. In SEV-SNP, the hypervisor controls the second-level translation tables for the CVMs. To isolate the CVMs and ensure integrity, SEV-SNP introduces a reverse map table (RMP). The RMP validates every memory read and write and ensures a binding between guest physical and host physical addresses. Without the RMP, the binding does not exist, and a hypervisor can mount remapping attacks [16].

**Virtual machine privilege levels (VMPLs).** SEV-SNP introduces VMPLs to partition the physical address space of SNP guests, where each VMPL has a dedicated permission view of the memory. Each VMPL implements kernel mode (CPL0) as well as user mode (CPL3). Currently, AMD supports four different VMPLs within one CVM. AMD

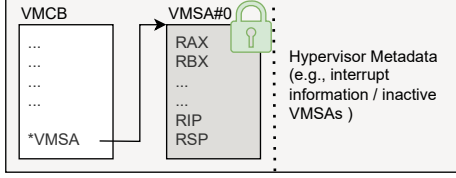


Figure 2: Overview of an SEV-SNP vCPU.

reuses the RMP to isolate VMPLs from each other. The RMP checks memory accesses by VMPLs (read, write, execute) and determines whether the access is allowed or not. The lower the VMPL number, the greater the privilege: VMPL0 is the most privileged, while VMPL3 is the least privileged. A more privileged VMPL can modify the memory permissions of a less privileged VMPL as long as its page permissions are greater than or equal to the newly specified ones for the less privileged VMPL. LOCI’s security model is based on this nuanced but important detail. The VMPLs are defined by a field within the VMSA. The untrusted hypervisor switches VMSAs to run the guest at different VMPLs. Hardware loads and saves the state of each VMPL using its respective VMSA. A numerically lower VMPL can use the RMPADJUST instruction to modify the permissions of a numerically higher VMPL. Notably, a more privileged VMPL can always revoke the permissions of a less privileged VMPL by setting the permission bit to zero (Figure 4(c)). For instance, if VMPL1 has only read permissions for the page at address  $0 \times 1000$ , it can grant numerically higher VMPLs (e.g., VMPL3) only read access to page  $0 \times 1000$ . Using RMPADJUST to grant write access for this page will result in the instruction returning an error code to the caller [17].

**SEV-Trusted I/O.** AMD recently introduced SEV-Trusted I/O, TIO for short, which enables accelerators to be attached to SEV-SNP CVMs. TIO allows attached accelerators to directly access the CVM’s encrypted memory. Furthermore, TIO also enables attaching the accelerators at VMPL granularity within each CVM [18].

**Virtual CPUs (vCPUs).** Hypervisors use vCPUs as an abstraction to manage different types of hardware virtualization implementations. A vCPU provides a high-level interface to one virtual machine control block (VMCB) for AMD SEV-SNP. Each VMCB can be associated with up to four different virtual machine save areas (VMSAs), representing the four VMPLs implemented. The VMSA contains the register state of the respective VMPL while that VMPL is not running. When scheduling the vCPU, the hypervisor decides which VMPL is entered by choosing the corresponding VMSA. Hypervisors use vCPUs to assign stateful information to different VMCBs; this information is not directly needed for hardware execution. The hypervisor allocates one vCPU for each virtual machine core and enters the virtual machine to execute on a per-vCPU basis. All vCPUs belonging to the same virtual machine share the same guest physical address space.

**SVSMs and Paravisors.** A secure virtual machine service

module (SVSM) runs within a CVM at a higher privilege level than the guest OS and provides services through an interface used by an enlightened guest (i.e., a guest aware of that interface) [19]. Coconut SVSM is an independent open-source project that provides such services on AMD SEV-SNP and can be used with Linux as both hypervisor and guest [15]. A paravisor supports unenlightened guests inside a CVM by handling confidential-computing functionality on their behalf [12]. Microsoft’s OpenHCL is an open-source paravisor [20]. LOCI extends its services for enlightened SubVM guests.

**Core compute complex (CCX) blocks.** AMD platforms consist of core compute complex (CCX) blocks. For Zen 4, these blocks include eight physical cores and up to 16 hyperthreads. Up to 12 CCX blocks are connected through an interconnect. Each CCX block has its own L3 cache slice. Within a CCX, all cores access the same physical L3 cache, while inter-CCX cache accesses incur higher latency because the cache is physically farther away. In certain scenarios, filling the cache from another CCX may be more expensive than fetching the data from DRAM [21]. In the default SEV-SNP setting, if the hypervisor schedules two CVMs on the same CCX, they will share the caches, making side-channel attacks possible.

### 3. Overview

We introduce SubVMs for confidential shared memory.

#### 3.1. SubVMs

A SubVM is essentially a guest VM image, comprising the guest OS kernel, drivers, user libraries, application binaries, etc. Figure 1(b) shows how LOCI uses confidential shared memory to share an LLM model among three SubVMs. If three SubVMs map the same 12 GB model, we reduce the memory capacity requirement from 36 GB to 12 GB (i.e., by a factor of three). The runtime performance is not adversely impacted.

The private memory regions of the SubVMs (e.g., code, inputs, outputs, and intermediate data) are fully isolated from one another. Since SubVMs have kernel space, they can run accelerator drivers. In combination with SEV-TIO, SubVMs can attach dedicated GPUs for exclusive use. At the same time, LOCI also enables the sharing of memory by exposing a trusted API to the SubVMs. The SubVMs can use the API to map shared memory regions. LOCI allows the creation of an arbitrary number of secure shared memory regions between SubVMs.

LOCI ensures the separation of SubVMs by using the VMPL feature of SEV-SNP. LOCI runs SubVMs in VMPL1, VMPL2, and VMPL3, while an SVSM runs in VMPL0. Sections 3.2 and 4.2 explain the challenges and the details of how LOCI achieves isolation across VMPLs. Together, these properties make SubVMs functional computing instances capable of running a typical guest workload (OS, apps, etc.) while allowing efficient memory sharing of model weights on AMD SEV-SNP.

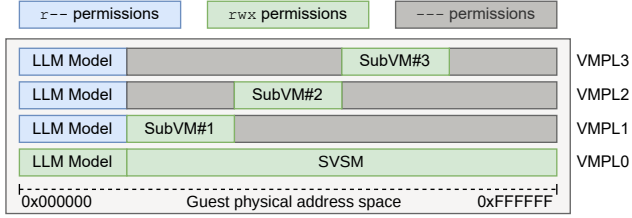


Figure 3: Memory permission views of VMPLs in LOCI.

**Threat Model.** We assume AMD hardware (including microcode and firmware) is trusted. Furthermore, we assume VMPL0 is trusted, and all SubVMs trust VMPL0. The SubVMs establish trust in VMPL0 using SEV hardware-based attestation. SubVMs do not trust each other and might act maliciously to compromise the confidentiality or integrity of co-located SubVMs. SubVMs can run any code in kernel and user mode within their VMPL. The hypervisor is untrusted and might collaborate with SubVMs to mount attacks against VMPL0 or other SubVMs. As in the default SEV-SNP threat model, we consider side channels out of scope [1].

### 3.2. Challenges and Key Observations

We discuss the challenges we face and the key observations we make when developing LOCI.

**Key Insight.** To isolate mutually distrusting SubVMs, we can consider using page-table-based isolation programmed by an SVSM in VMPL0. This design would limit SubVMs to user space, excluding *kernel space*. This limitation is particularly problematic because our goal is to enable confidential shared memory for heterogeneous devices, whose drivers require kernel space. Instead, we use the RMP, which the hypervisor cannot manipulate. We observe that the RMP contains VMPL information, which we can leverage in LOCI. At first, it may seem counterintuitive to use VMPLs to isolate workloads since they are organized hierarchically. A lower VMPL can control the access permissions of a higher VMPL. Intuitively, one might assume a malicious VMPL1 could change the permissions of VMPL2 to gain illegitimate access to VMPL3. However, access permissions of a numerically higher VMPL can only be altered by taking the current VMPL’s permissions as a threshold. This implies that VMPL1 cannot grant VMPL2 permission to access the confidential data of VMPL3 unless VMPL1 already has access. LOCI is the first work that uses the key insight into the non-strict hierarchy of VMPL permissions, which allow us to execute mutually distrusting SubVMs in one CVM. Previous works deemed this impossible or prohibitively burdensome, precluding confidential shared memory on SEV-SNP [13], [22].

In LOCI’s design, we create SubVMs by using VMPL0 to set up the memory regions of VMPL1–VMPL3 such that they do not overlap with each other, giving each SubVM its own private address space. Figure 3 depicts the memory permissions we enforce for SubVMs. To map shared memory,

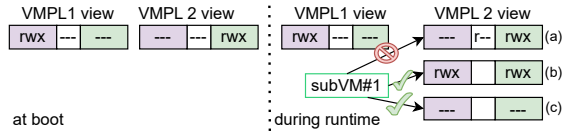


Figure 4: VMPL memory views at different stages of LOCI’s lifecycle. (a) SubVM 1 cannot give a less privileged SubVM executing in VMPL2 access to a region it does not have access to. (b) SubVM 1 gives a less privileged SubVM in VMPL2 access to a memory region that it can access. (c) SubVM 1 removes access from a memory region belonging to a less privileged SubVM in VMPL2.

VMPL0 exposes an API to the VMPLs in a way that the SubVMs can dynamically request shared memory regions. Apart from memory view management, LOCI has to support the execution of multiple SubVMs housed within a single CVM. This deviation from the norm requires adapting the existing software stack, as we summarize next.

**SVSM.** The Coconut SVSM implementation on which we build is designed to provide services to one guest kernel per CVM [15]. Thus, Coconut cannot boot more than one guest OS and lacks the API to enable memory sharing between SubVMs. LOCI, which builds on top of Coconut, must manage up to three kernels and securely assign their private and shared memory.

**Guest OS.** Guest OSes typically use a stub loader to create the memory map and scan the hardware information (e.g., ACPI tables) during boot. UEFI stub loaders such as EDK-II assume they have nearly full control over the physical address space. During boot, EDK-II starts in real mode and requires physical pages below 1 MiB and 4 GiB while transitioning to long mode. While this holds when booting one kernel per CVM, it does not hold when booting multiple SubVMs per CVM as we do in LOCI. Thus, LOCI must meet these requirements and assign each VMPL the pages necessary for system boot. Other techniques, such as virtualization and nested virtualization, offer each instance an isolated virtualized physical address space. In contrast, in LOCI, multiple instances share a physical address space, and isolation happens through the RMP. We must accommodate the shared address space and create a new stub loader compatible with existing operating systems.

**Guest User Space.** Our design ensures that LOCI’s changes are fully transparent to user space. User space is not aware of any changes made at the SVSM or OS level and can leverage shared memory regions as normal files. In other words, user-space stacks and applications do not need to be recompiled or rewritten to benefit from LOCI.

## 4. Design

We explain our design for creating and isolating SubVMs.

## 4.1. LOCI's Lifecycle

We introduce the SVSM and SubVM lifecycles in LOCI.

**SVSM Boot and Attestation.** Typically, to boot an SEV-SNP CVM, the cloud provider obtains the CVM image from the tenant and loads it into memory. Then, it invokes SEV-SNP APIs to encrypt, attest, and boot the CVM. With LOCI, this flow remains intact. The difference is that VMPL0 software is now within the TCB and controlled by a trusted entity (e.g., AMD). This design is similar to Arm CCA and Intel TDX, where the RMM or TDX Module is under the control of the hardware manufacturer. As in the default SEV-SNP startup process, the cloud provider invokes the SEV-SNP firmware to place a VMPL0 image in encrypted memory and boots it.

**Setting Up and Spinning Up SubVMs.** With LOCI, the tenants can have up to three SubVMs within one CVM. The SVSM spins up one SubVM per VMPL, for a total of three SubVMs, each with a custom CVM image that can differ across VMPLs. Because the SubVMs are mutually distrusting, LOCI spins up each SubVM in a dedicated VMPL and restricts its memory accesses to dedicated memory regions (see Section 4.2). Here, we explain how the SVSM sets up and spins up the SubVMs. Before spinning up the SubVMs, the SVSM (a) allocates vCPUs and sets up VMSAs, (b) duplicates and relocates SubVM images, and (c) sets up the per-SubVM memory maps. As we discuss in Section 4.3, for efficient SubVM scheduling, the SVSM allocates multiple vCPUs for each CVM. Once these vCPUs are set up, the SVSM populates the VMSAs with the initial register states for each of the SubVMs. Next, the SVSM needs the SubVM images that it should spin up.

The tenant includes SubVM images in the initial CVM configuration, which is loaded into the CVM memory as part of the boot process. The SVSM duplicates this initial image for each SubVM that it spins up. Alternatively, the SVSM can expose an API to the hypervisor to dynamically load the images.

To ensure mutual isolation between the SubVMs, the SVSM relocates the SubVM images into non-overlapping, dedicated SubVM memory regions. Finally, the SubVM OS needs to know where its dedicated memory region resides. To facilitate this, the SVSM builds a per-SubVM memory map that it provides as part of the boot process. After setup, the SVSM spins up the SubVM using the standard guest OS initialization process.

To establish trust in the SubVM, we rely on the trust that we establish in the SVSM during the initial CVM boot using attestation. Because the SVSM is trusted, we can trust it to correctly set up and spin up the SubVM. Subsequently, the SubVMs use vTPM-based attestation provided by the trusted SVSM to validate their memory and startup configurations.

**What Runs in a SubVM?** SubVMs are like CVMs with one important difference: they share the same guest physical address space. With LOCI, SubVMs can run any modern OS capable of booting in a CVM. For functional correctness, the only requirement is that the OS must adhere to the memory

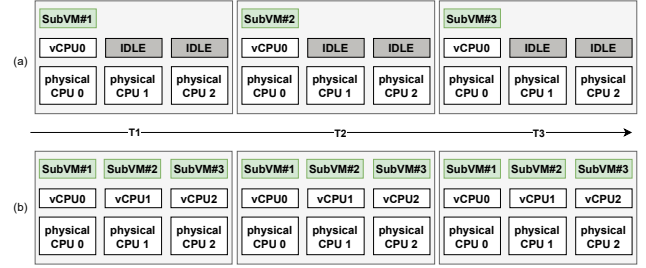


Figure 5: (a) Naive vCPU allocation resulting in sequential SubVM execution. (b) LOCI's vCPU allocation resulting in concurrent SubVM execution.

map given by the SVSM. To enable the execution of three SubVMs in a single CVM, LOCI creates a unique memory map per SubVM. This memory map ensures the correct functioning of benign SubVMs, as it prevents them from accidentally accessing and corrupting each other's memory. We observe that all modern operating systems (e.g., Linux, Windows, OpenBSD) fulfill the requirement and adhere to the SVSM-provided memory map.

## 4.2. Inter-SubVM Isolation

In our threat model, SubVMs in a CVM are mutually distrusting and might try to read from or write to each other's memory as they share the same guest physical address space. To prevent such malicious memory accesses, LOCI enforces the isolation guarantee that a SubVM can never read from or write to the private memory of another SubVM.

To ensure this guarantee, we use our insight that each VMPL has a page-granular view of memory (including `rwX` permission bits). The SVSM programs the VMPL views such that the VMPLs only have access to strictly non-overlapping memory regions. Upon CVM creation, all CVM memory is inaccessible to VMPL1-3. As part of spinning up a SubVM (Section 4.1), the SVSM sets up the SubVM's private memory regions. With this, each SubVM can only access a subset of the SVSM-allocated guest physical memory (Figure 3). The SVSM ensures that the SubVMs' private memory is set up correctly with non-overlapping addresses at spin-up.

## 4.3. SubVM Scheduling

In the default virtualization setting, the hypervisor normally maps one virtual VM core to one vCPU. With SEV-SNP and VMPLs, at any given point in time, each CVM vCPU can only execute in one VMPL (Section 2); that is, the VMPLs temporally share the vCPUs. In the baseline SVSM configuration, one CVM executes an SVSM in VMPL0 and an enlightened guest (e.g., GNU/Linux OS with user-space applications) in another VMPL. When a guest is executing, it can send synchronous requests to the SVSM. To service these requests, the guest instructs the hypervisor to switch

the VMPL on the vCPU to VMPL0 such that the SVSM can execute. Therefore, there is never a case where the guest and the SVSM need to execute in parallel and compete for the vCPU. This default setting is compatible with temporarily sharing the vCPU between the SVSM and the guest.

With LOCI, we run three SubVMs in three different VMPLs. The default mode of temporally sharing vCPUs between the VMPLs now poses a problem because it forces the SubVMs to share the vCPUs. Because the vCPUs can only execute one VMPL at a time, the SubVMs might need to wait to obtain a vCPU to execute on. For example, in the worst case, if the CVM only had one vCPU, then the SubVMs would need to execute sequentially as shown in Figure 5(a). This competition between the SubVMs for the limited vCPU resource leads to performance degradation.

To remedy this issue, we observe that if we assign dedicated vCPUs to the SubVMs, then they can run in parallel on the available physical cores. In LOCI, the SubVMs are identical and execute the same SubVM image. So, to ensure that SubVMs can run on dedicated vCPUs, we use the SVSM to request the hypervisor to triple the number of vCPUs allocated to the CVM. Once the vCPUs are allocated, the SVSM ties each one exclusively to one SubVM. With our solution, the SubVMs can execute in parallel (Figure 5(b)).

Having established how LOCI enables SubVMs to execute concurrently on dedicated vCPUs while maintaining strict isolation, we now detail the mechanism that allows SubVMs to securely establish confidential shared memory.

#### 4.4. Memory Sharing

To support confidential memory sharing, the SVSM maintains an access control list of active shared memory regions. Each SubVM can create a shared memory region and configure the access permissions for other SubVMs for the respective region. Subsequently, the other SubVMs can query the available regions through the SVSM API and request the SVSM to map the regions into their physical address space. To map the respective memory in the SubVM’s address space, the SVSM uses the RMPADJUST instruction to manipulate the RMP such that it reflects the software-defined configuration. If a SubVM wishes to stop having access to the shared memory region, it uses the same interface with the SVSM. Should no SubVM have access to a shared memory region, or in other words, if its reference count drops to zero, the SVSM deletes it and adds the memory to the pool of usable memory or assigns it back to the original SubVM. To facilitate the sharing of read-only memory regions, such as LLM models, the SubVM creating the region can configure the region such that it does not have write access to the memory after creating the shared memory region. To validate the authenticity of the shared memory region content, the mapping SubVM gets an authentic hash of the shared memory region provided by the SVSM. This ensures that the read-only shared memory region content is benign and contains the expected data (e.g., LLM model weights).

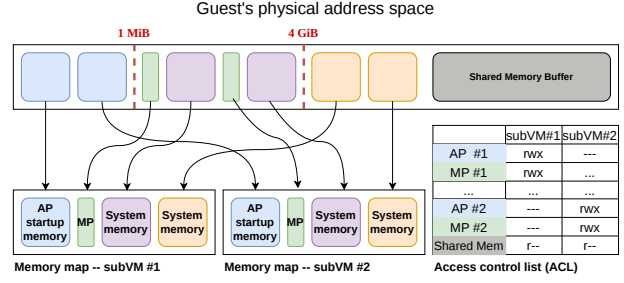


Figure 6: Example of memory partitioning in LOCI.

LOCI does not limit the number of memory regions, meaning it can support an arbitrary number of shared memory regions between SubVMs.

#### 4.5. Attaching Accelerators to SubVMs

Besides having shared memory regions on the processor cores, LOCI can also leverage AMD SEV-TIO to use accelerators with SubVMs [18]. For this, we use the following observations. With TIO, VMPL 0 can configure VMPL 1-3 to be attached to dedicated accelerators. When an accelerator is attached to a VMPL, the accelerator sees the same view (e.g., access permissions) of host memory as its VMPL. More importantly, the accelerator can now directly access encrypted VMPL memory, removing the need for expensive bounce buffers [23], [24]. LOCI’s design, which runs each SubVM in a dedicated VMPL, aligns perfectly with the SEV-TIO model that allows attaching accelerators at VMPL granularity. The SVSM mediates accelerator attachment and vIOMMU paging for the SubVMs. Once the SVSM completes attachment, the SubVMs start executing accelerator drivers in kernel space so that applications in user space can use the accelerator (e.g., for GPU-based LLM inference). LOCI’s choice of using VMPLs satisfies this requirement and provides each SubVM with dedicated kernel and user spaces.

### 5. Implementation

We use COCONUT-SVSM (Coconut)<sup>1</sup> [15], an independent open-source SVSM for AMD SEV-SNP, as the base implementation for LOCI’s trusted SVSM. We add 3717 LoC to Coconut to support SubVM management and confidential memory sharing. Of those 3717 added lines, 35% are comments, headers, and blank lines. We use Linux v6.8.0 as the kernel and BusyBox 1.36.1 for the root filesystem of the SubVM (guest) and hypervisor (host) operating systems. We choose Linux for its extensive feature set, but other lightweight operating systems (e.g., unikernels) can also run in SubVMs.

1. We refer to COCONUT-SVSM as Coconut.

## 5.1. LOCI Setup

Our base implementation, Coconut, runs in VMPL0 and manages one guest running in a numerically higher VMPL; it does not support booting multiple OSes in the CVM. LOCI changes Coconut to support running multiple mutually isolated guests (SubVMs) in dedicated VMPLs (VMPL1–VMPL3). Further, LOCI partitions vCPUs and memory between SubVMs with 3010 added LoC.

**vCPU Allocation.** vCPUs let the hypervisor schedule cores for SubVMs. In LOCI, each SubVM has a dedicated set of vCPUs. At CVM boot, the SVSM runs its initial code in VMPL0 on the only active vCPU. After completing its own initialization, the SVSM requests additional vCPUs for the SubVMs. For example, if each SubVM requires eight cores, the SVSM allocates 23 more vCPUs, bringing the total to 24 vCPUs for the CVM instance. Initially, all vCPUs boot into the SVSM code, meaning the CVM has 24 vCPUs with only VMPL0 initialized. Once all vCPUs are created and initialized, the SVSM partitions memory in preparation for SubVM execution.

**RAM Allocation.** Each SubVM has a dedicated memory region that is isolated from the memory of other SubVMs. The SubVM’s memory region contains the kernel image used during spin-up, the initramfs, and general-purpose memory that the SubVM can use. To function correctly, x86 OSes require access to certain address ranges that the SVSM must partition and allocate to each SubVM.

Since x86 starts executing in real mode, it can only access memory below 1 MiB. During the boot process, x86 transitions to protected mode, where it can access memory up to 4 GiB. Finally, it transitions to 64-bit long mode, where it can access all memory. To run in the real and protected modes during boot, Linux needs at least a few pages of memory below 1 MiB and  $\approx 100$  MiB of memory below 4 GiB. For instance, the application processor (AP) startup memory from Figure 6 must be in memory below 1 MiB to be accessible in real mode during SubVM startup. The SVSM isolates each SubVM’s memory using RMPADJUST. Concretely, for each SubVM’s memory, the SVSM programs the RMP with  $r-w-x$  permissions for VMPL0 (SVSM) and the VMPL that the SubVM runs in. Further, the SVSM configures no access ( $- - -$ ) for other VMPLs (Figure 6). Once the memory isolation is set up, the SVSM builds a memory map for each SubVM in which it marks inaccessible memory (i.e., other SubVMs’ memory) as reserved in the RMP. This ensures that the OS does not perform invalid accesses during benign operation. As a last step, the SVSM passes the memory map to the SubVMs as part of the spin-up protocol by changing the bootloader.

**SubVM Bootloader.** Generally, before an OS boots, a small piece of software executes, acting as a stub to abstract the hardware interface to the OS. Normal SEV-SNP CVMs use EDK-II as the stub for Linux. EDK-II bootstraps the processor core from real mode to protected mode and initializes the firmware structure. It also detects the memory layout and parses it such that the OS can eventually process it. EDK-

II consists of around 1.5 million lines of code to support a variety of use cases and boot flows. However, EDK-II assumes access to the entire guest physical memory and is written using a complex callback structure, which makes it non-trivial to use during SubVM boot.

Thus, we develop a custom bootloader and boot protocol for SubVMs. Our bootloader consists of 1012 LoC (of which 895 are written in assembly) to transition the vCPU core from real mode to protected mode. Our bootloader passes the SVSM-generated memory map and other management data directly to the SubVM OS. The management data includes the kernel command-line arguments, the kernel executable image in memory, and the address of the SubVM’s initramfs. Our stripped-down version narrows the EDK-II interface and mitigates EDK-II-based interface attacks [25].

**SubVM Spin-Up.** Once our bootloader finishes executing, it passes control to the SubVM. We use Linux as the OS for SubVMs and change its initialization process with 1013 LoC to ensure compatibility with LOCI. We change Linux to always use the memory map from the SVSM. However, some legacy features that are not used in a virtualized setting do not respect the memory map. These features cause Linux to probe memory regions (e.g., the Extended BIOS Data Area and legacy DMA buffers) that are not in the memory map, causing the probes to fail. To prevent these failures, we modify Linux to strictly respect the memory map and avoid these probes. Apart from this, we do not change the Linux boot process.

## 5.2. SubVM Memory Management

We implement memory isolation and provide shared memory access for the SubVMs with 707 LoC.

**Memory Isolation.** To isolate SubVM memory and ensure functional correctness, the SVSM tracks the permissions of all guest memory. We use an access control list (ACL) with an entry for each memory region (one or more 4 KiB pages), containing its per-SubVM permission mask (Figure 6) and a reference count. The SVSM keeps the ACL entry of a region in sync with the permission field of the SEV-SNP RMP entries covering it. The RMP enforces the actual hardware protection of the memory region. Consequently, all permission checks besides those involved in changing the permissions of a region are handled directly by the hardware without any involvement of LOCI; this approach has no performance penalties compared to default SEV-SNP CVM operation with only one guest.

**Shared Memory.** For shared memory regions, the SVSM dynamically increments or decrements the reference counts. This ensures that the region cannot be changed and access to it cannot be restricted while a SubVM uses it. For performance, LOCI’s SVSM issues SEV-SNP UMMSMASH requests to the hypervisor so that shared memory regions use 2 MiB RMP mappings instead of 4 KiB ones, speeding up permission changes.

**New SVSM API Calls.** The SVSM specification defines a secure interface for the SVSM to provide services to

TABLE 1: New SVSM API calls for LOCI.

| SVSM API Call          | Description                                                                   |
|------------------------|-------------------------------------------------------------------------------|
| CREATE_SHARED_REGION   | Requests the SVSM to create a new shared memory region                        |
| GET_REGION_INFOS       | Queries the SVSM about shared memory information (e.g., size, memory address) |
| GET_AVAILABLE_REGIONS  | Queries the SVSM about shared memory regions accessible to the calling SubVM  |
| GET_REGION_PERMISSIONS | Queries the SVSM about the permissions of a shared memory region              |
| REGISTER_SHARED_REGION | Informs the SVSM that the shared memory region is now in use by the SubVM     |
| RELEASE_SHARED_REGION  | Informs the SVSM that the shared memory region is no longer used by the SubVM |

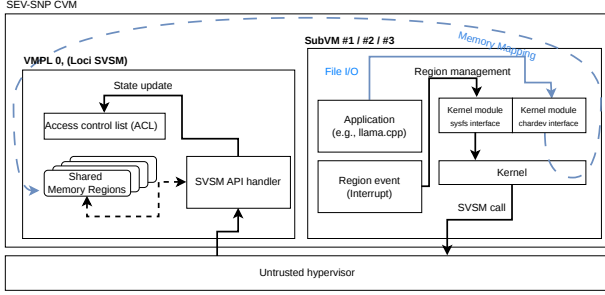


Figure 7: Communication between SubVM user space and VMPL0 in LOCI.

enlightened guests running in other VMPLs [19]. By default, only a single VMPL may run at a time on a given vCPU, and so interface calls in the specification are not bidirectional; the relationship between the guest and the SVSM is similar to a client-server relationship. To issue an SVSM API call, the SubVM sets the CPU’s `rax` register to the identifier corresponding to the call. The SubVM may pass additional parameters in the `rcx`, `rdx`, and `r8` registers depending on the request semantics. Once the registers are set up, the SubVM requests the hypervisor to switch execution to VMPL0 through a Non-Automatic Exit Event [26], transferring control to the SVSM. The SVSM handles the requests, updates the registers with the results and a status code indicating success or failure, and transfers control back to the calling SubVM. During this flow, all the confidential data remains within the SubVM’s address space and thus inaccessible to other SubVMs or the hypervisor.

We implement six new API calls between the SVSM and SubVMs (Table 1). SubVMs use the `GET_*` APIs to fetch information about the shared memory region as part of the discovery process. When the SubVM uses the memory region, it informs the SVSM via `REGISTER_SHARED_REGION` such that the SVSM can increase the reference count in the ACL. When a SubVM wishes to opt out of sharing, it uses `RELEASE_SHARED_REGION`, which results in the SVSM decrementing the reference count. We implement these APIs as extensions to the SVSM interface by changing Coconut’s request-handling routines and Linux in the SubVM [19].

**Exposing Shared Memory to User Space.** Using a kernel

driver (1139 LoC) in the SubVM, we expose shared memory regions as character devices to user space. The kernel drivers use the LOCI API calls from Table 1 to obtain information about the shared memory regions as shown in Figure 7. Further, we implement the Linux file API on top of the shared memory region. This allows processes to use shared memory regions as if they were files. More precisely, the user can map the file with `mmap` or read it with `fread`. This requires no changes to the application logic as long as the application accepts a file as an input argument. When the SVSM notifies the SubVM of a shared memory update, we need a mechanism to check that no operation is currently using the shared memory region before the SubVM can gracefully relinquish it. We implement this mechanism by storing the number of currently open file references to the shared memory region. To do this, our kernel module hooks into each open and close operation of the character device and increments or decrements the count, respectively. The kernel module acknowledges the SVSM’s update notification when the reference count is zero.

## 6. Security Analysis

LOCI defends against adversaries in our threat model. The threat model considers a malicious SubVM and/or a malicious hypervisor. However, we explicitly exclude side channels in line with the SEV-SNP threat model.

**SVSM Boot.** The hypervisor can try to launch a CVM with a compromised SVSM by modifying the boot image or memory configuration. If the hypervisor modifies the boot image or alters the initial memory configuration, the expected and reported attestation measurements will differ, and the SVSM will detect the tampering.

**SubVM Creation.** An attacker can try to use the SVSM to launch compromised SubVMs. However, this is prevented because the SVSM is part of the trusted computing base (TCB) and its integrity is verified through attestation. An attacker may also try to supply the SVSM with malicious SubVM images. Such attacks can be prevented by including the SubVM images in the initial CVM memory measured for the attestation report. Alternatively, the tenant can supply the SVSM with the image, and subsequently, the SubVM can attest itself using SVSM APIs. A tenant whose SubVM runs in VMPL1–VMPL3 can query the SEV firmware running on the security coprocessor to attest VMPL0 (the SVSM) and subsequently use the SVSM API to attest the SubVM. Thus, a malicious hypervisor cannot break this chain of trust.

**Runtime.** A malicious SubVM can try to access the memory of other SubVMs or the SVSM, but our mutual isolation guarantee ensures that this does not happen. The SVSM’s RMP configurations ensure that the hardware blocks such malicious accesses. The SubVM cannot modify its RMP permissions and thus cannot circumvent the protection.

A more privileged SubVM can revoke the memory permissions of a less privileged victim SubVM (see Figure 4(c)). This revocation leads to a nested page fault sent to the hypervisor and stalls the execution of the victim SubVM.

TABLE 2: LLM models used to benchmark LOCI.

| Model Name                          | URL  | Size  | Vendor    |
|-------------------------------------|------|-------|-----------|
| granite-3b-code-base.Q4_K_M [27]    | [28] | 2 GB  | IBM       |
| granite-8b-code-base.Q4_K_M [27]    | [29] | 5 GB  | IBM       |
| granite-20b-code-base.Q4_K_M [27]   | [30] | 12 GB | IBM       |
| gemma-3-27b-it-q4_0 [31]            | [32] | 17 GB | Google    |
| qwen2.5-coder-32b-q4_k_m [33], [34] | [35] | 20 GB | Qwen      |
| mistral-small-3.1-24b-q8_0 [36]     | [37] | 25 GB | Mistral   |
| phi-4 [38]                          | [39] | 29 GB | Microsoft |
| llama-4-Scout-17B-16E-IQ2_XS [40]   | [41] | 33 GB | Meta      |

The hypervisor might forward this page fault to the SVSM. The SVSM detects the malicious permission revocation and identifies the malicious SubVM for shutdown. If the hypervisor does not forward the page fault, we see that this is equivalent to a denial-of-service attack, as the SubVM will never make progress. Denial of service from the hypervisor is outside our threat model. Therefore, malicious permission revocation attacks do not compromise the security of LOCI.

A malicious SubVM can give a less privileged SubVM access to its own memory region (see Figure 4(b)). By granting such access, the malicious SubVM can try to launch code-injection attacks on the victim. However, the injected memory will never be mapped in the victim’s page tables and therefore will never be used by the victim, thwarting such attacks. For this attack to be successful, the malicious SubVM must first modify the victim’s page tables to insert mappings to the injected memory. We observe that an attacker capable of modifying page tables can also perform the same code injection using other methods (e.g., network packets), rendering the baseline vulnerable.

With LOCI, all SubVMs execute in dedicated VMPLs of the same CVM and therefore share the same guest physical address space. A malicious SubVM can try to abuse this to break our memory isolation guarantee by accessing other SubVMs’ memory stored in shared caches. However, SEV-SNP tags all CVM memory with VMPL information in the RMP and TLBs. The SEV-SNP hardware checks all memory accesses against their originating VMPL. So, the hardware will stop accesses from a SubVM executing in its dedicated VMPL when it tries to access a cache line of another SubVM executing in a different VMPL. When an SVSM shrinks shared memory regions and revokes SubVM access permissions, a malicious SubVM can try to use stale TLB entries to access the revoked memory. With LOCI, the SVSM flushes the TLB entries across all SubVMs to stop such attacks.

## 7. Evaluation

We evaluate the memory-sharing mechanism of LOCI. We focus our evaluation on the performance benefits of sharing LLM models in memory. Concretely, our evaluation answers the following questions:

- RQ1: How fast is confidential memory sharing?
- RQ2: Does LOCI reduce memory capacity requirements?

TABLE 3: LLM loading times and memory requirements for LOCI.

| Model       | Time     |            | Memory (MiB) |          |               |
|-------------|----------|------------|--------------|----------|---------------|
|             | Load (s) | Share (ms) | Baseline     | LOCI     | Reduction (×) |
| granite-3b  | 1.75     | 3.80       | 7122.81      | 3058.71  | 2.33          |
| granite-8b  | 4.02     | 8.84       | 14464.08     | 5154.10  | 2.81          |
| granite-20b | 10.72    | 23.22      | 36819.57     | 12370.31 | 2.98          |
| phi-4       | 25.52    | 53.12      | 84594.90     | 28671.72 | 2.95          |

RQ3: Does LOCI impact performance?

RQ4: Can LOCI benefit from the memory hierarchy?

### 7.1. Setup

We run the experiments on a dual-socket system with AMD EPYC 9334 32-core 3.9 GHz Zen 4 processors and 384 GiB of RAM. The processors use microcode version 0x0a101148. The hypervisor boots Linux kernel v6.8.0. For QEMU, we use version 8.2.0 with the patch version provided by the Coconut project (commit 260df2b36be). To assess the performance impact of LOCI’s confidential shared memory, we benchmark the inference speed of various LLMs. We use `llama-bench`, commit 39509fb, as the inference engine [42]. We only use the CPU (i.e., no accelerators and no GPU passthrough) for inference. We select the models based on their size and release dates and evaluate all models shown in Table 2.

We benchmark the baseline (CVM<sub>1</sub>, CVM<sub>3</sub>) and LOCI (SubVM<sub>1</sub>, SubVM<sub>3</sub>) with one and three concurrent VMs and SubVMs, respectively. We execute the benchmarks as follows. First, we load the model into memory: either into a shared memory region when evaluating LOCI or into an empty RAM disk when evaluating the baseline. The memory we use for the model is equal in size in both cases (baseline/LOCI). We execute ten iterations of `llama-bench`, generating 1024 tokens per iteration, to reduce the effects of noise in individual runs. We choose 1024 tokens to ensure that the experiments run long enough to provide meaningful results, irrespective of the model size. To evaluate performance, we measure the inference speed in tokens per second (tok/s). For each run, we read the performance counter values every 20 seconds. We disable preemption on the core allocated to the VMs, ensuring that no other processes pollute our measurements. Additionally, we pin each VM’s vCPUs to different physical cores, ensuring that each vCPU is only ever scheduled on a single physical core. We further disable SMT.

### 7.2. Impact of LOCI

LOCI adds 3717 LoC to VMPL0, bringing the total shared TCB to 24911 LoC, a 17.53% increase over the CVM baseline. We measure the impact of LOCI on SubVM memory usage and LLM inference performance.

**RQ1: Confidential Memory Sharing between SubVMs.** To share the model, we have to map the pages using RM-

PADJUST. Mapping a 2 MiB page and executing RMPADJUST on it takes 9390 cycles on average, corresponding to 3.8  $\mu$ s. The time required to create a shared memory region increases linearly with the size of the region. Most of the cost is due to the RMPADJUST instruction, explaining this linear increase even for regions as small as 4 MiB. Using RMPADJUST on a 30 GiB region (e.g., for phi-4) takes 75 ms. With LOCI, sharing an LLM model is 464 $\times$  faster on average than loading it sequentially from disk for each SubVM. If loading can happen in parallel, our design saves I/O bandwidth.

**RQ2: Memory Reduction.** Next, we evaluate `llama.cpp`'s impact on the overall system memory pressure in the different scenarios. LOCI shares the LLM model across SubVMs and so reduces the LLM memory requirement by a factor of 2.77 on average compared to baseline CVMs (Table 3). The KV cache and the operating system memory, which varies from 48.56 MiB to 342.22 MiB for the benchmarks in Table 3, are still duplicated for different SubVMs. For large LLM models, the LLM model weights dominate the memory footprint as the additional memory used to perform the inference is only a few hundred megabytes. For smaller models, we see that the memory used for inference has a higher impact on the memory footprint, thus slightly reducing the gains that we see for LOCI. However, for smaller LLM models, cache sharing benefits arise, which we discuss later in Section 7.3.

**RQ3: Performance Evaluation.** We benchmark eight popular LLM models on a configuration with eight vCPUs (Table 2). To evaluate the performance of LOCI, we use two baselines: we run one SEV-SNP CVM (CVM<sub>1</sub>), and we run three SEV-SNP CVMs in parallel (CVM<sub>3</sub>). We compare these baselines to two LOCI setups where we run only one SubVM (SubVM<sub>1</sub>) and where we run three SubVMs in parallel (SubVM<sub>3</sub>). Figure 8 shows the inference performance of LOCI for the eight models. Overall, we find that LOCI performs 0.72% better on average than the default CVM baseline.

### 7.3. Detailed Core Performance Analysis

We analyze the impact of LOCI's design on performance. Figure 8 shows that LOCI's performance is, on average, the same as that of the baseline. Table 4 shows the performance counters for the setup with eight vCPUs per instance. Next, we explain how AMD's memory hierarchy impacts LOCI.

**RQ4: Impact of CCX Crossing.** Our benchmark on a configuration with eight vCPUs aligns naturally with the core complex die size of current AMD CPUs [21]. Thus, we run the three VMs on three different CCX blocks in the baseline (Section 2). For LOCI, we execute the three SubVMs on three different CCX blocks. This may cause cross-CCX overhead when accessing shared memory. However, since the model is read-only, we do not observe performance penalties from this aspect. This is because the read-only memory cache lines will always be in a clean state, meaning

they are in sync with DRAM. Thus, a core accessing the LLM model within one CCX does not need to probe all CCX caches for the latest data; it can instead fetch the data from DRAM, which is faster [21].

**RQ4: Impact of CCX Sharing.** To achieve optimal performance, AMD recommends that users avoid crossing CCX boundaries when running memory-intensive workloads [21]. Therefore, we also test the inference performance of LOCI against the baseline when the underlying physical cores running each instance all belong to the same CCX. Specifically, we test the configuration where each VM has two vCPUs, and all three VMs/SubVMs run on the same CCX. We use the Granite 3b model for this experiment because it is performant even with two vCPUs, while the larger models are not. In the benchmarking setup, we repeat the 1024-token generation step 100 times. The rest of the experimental setup is identical to the one in Section 7.1.

Compared to the baseline, LOCI benefits from a lower median L3 miss latency (Table 5). Both the L3 and L1 miss rates are also lower, as indicated by a higher cache hit rate (DC fills from same CCX) of 28.4 per thousand instructions (PTI) compared to the baseline with only 25.8 PTI and L1 DTLB misses of 0.4 vs 0.6 PTI. The higher cache hit rate decreases the memory bandwidth from 68 GiB/s in the baseline to 30.9 GiB/s for LOCI, reducing overall system memory pressure. Additionally, the token generation rate of LOCI instances is higher than that of the baseline. The token generation rate increases by 9.36% in the triple-instance setup due to caching. Concretely, the token generation rate for LOCI SubVM<sub>3</sub> is 10.3 tok/s compared to 9.4 tok/s for the baseline CVM<sub>3</sub>.

**GPU Benchmarking.** AMD announced SEV-TIO; however, no publicly available devices support it. Thus, we are unable to run SEV-TIO GPU benchmarks. However, `llama.cpp` natively supports GPU backends. TIO allows a compatible GPU, when available, to be attached directly to a SubVM [18]. Current GPU benchmarks are imperfect predictors of SEV-TIO performance. Factors such as VMPL I/O TLB usage, I/O port exhaustion, and other macroarchitectural effects cannot be meaningfully benchmarked without TIO hardware. Thus, we intentionally do not benchmark LOCI with non-TIO settings on a GPU running in non-CC mode. LOCI will be effective with GPUs, as the absolute amount of memory shared increases significantly with bigger LLM models. However, we do not evaluate SEV-TIO compatibility because compatible accelerators are unavailable at the time of writing.

As a best-effort analysis, we run LLM inference with shared memory on two GPUs (NVIDIA H100 and NVIDIA RTX PRO 6000). We observe no performance degradation or improvement from memory sharing. The token generation rates on the two GPUs are 22.77 and 26.86 tokens per second without shared memory and 22.73 and 26.86 tokens per second with shared memory, respectively. From these results, we infer that LOCI likely will not induce performance penalties when multiple GPUs share the same model weights but will only reduce the system memory pressure. The

TABLE 4: Median  $\mu$ Prof output for eight vCPUs. Latency is in ns, and bandwidth is in GB/s.

| Instance |             | - | L3 misses |      | DC fills from (pti) |           |           | L2 (pti) |       | L1        | RAM    | - | L3 misses |      | DC fills from (pti) |           |           | L2 (pti) |       | L1        | RAM    |
|----------|-------------|---|-----------|------|---------------------|-----------|-----------|----------|-------|-----------|--------|---|-----------|------|---------------------|-----------|-----------|----------|-------|-----------|--------|
| Type     | Model       | # | Avg. lat. | %    | Same CCX            | Diff. CCX | Local mem | Miss     | Acc.  | DTLB miss | Mem Bw | # | Avg. lat. | %    | Same CCX            | Diff. CCX | Local mem | Miss     | Acc.  | DTLB miss | Mem Bw |
| LOC1     | granite-3b  | 1 | 130.4     | 95.6 | 26.3                | 0.0       | 5.0       | 22.0     | 63.0  | 0.4       | 60.5   | 3 | 141.6     | 95.7 | 25.5                | 0.0       | 4.8       | 22.2     | 58.4  | 0.4       | 177.9  |
| Base     | granite-3b  | 1 | 133.8     | 95.2 | 24.8                | 0.0       | 5.6       | 21.7     | 60.3  | 0.6       | 59.3   | 3 | 142.7     | 95.4 | 24.8                | 0.0       | 5.9       | 21.4     | 59.2  | 0.6       | 175.5  |
| LOC1     | granite-8b  | 1 | 130.4     | 97.1 | 23.7                | 0.0       | 5.9       | 19.7     | 58.6  | 0.4       | 64.8   | 3 | 142.6     | 97.1 | 23.9                | 0.0       | 5.7       | 19.9     | 58.0  | 0.4       | 191.0  |
| Base     | granite-8b  | 1 | 134.3     | 96.8 | 23.2                | 0.0       | 6.5       | 19.3     | 59.8  | 0.4       | 64.4   | 3 | 144.1     | 97.0 | 23.6                | 0.0       | 6.2       | 19.5     | 58.1  | 0.4       | 189.7  |
| LOC1     | granite-20b | 1 | 129.2     | 97.7 | 37.4                | 0.0       | 7.6       | 17.8     | 79.8  | 0.4       | 64.3   | 3 | 138.8     | 97.7 | 37.8                | 0.0       | 7.4       | 17.9     | 79.5  | 0.4       | 188.6  |
| Base     | granite-20b | 1 | 131.0     | 97.7 | 37.4                | 0.0       | 7.7       | 17.7     | 80.2  | 0.4       | 63.6   | 3 | 140.1     | 97.6 | 37.8                | 0.0       | 7.4       | 18.0     | 79.2  | 0.4       | 187.8  |
| LOC1     | gemma-3     | 1 | 131.5     | 98.0 | 30.6                | 0.0       | 3.7       | 22.3     | 75.3  | 0.1       | 69.8   | 3 | 153.8     | 98.3 | 30.2                | 0.0       | 4.1       | 22.1     | 71.3  | 0.1       | 179.6  |
| Base     | gemma-3     | 1 | 133.6     | 97.7 | 29.6                | 0.0       | 4.9       | 21.5     | 62.4  | 0.4       | 67.4   | 3 | 145.5     | 97.8 | 29.6                | 0.0       | 4.9       | 21.5     | 60.2  | 0.4       | 202.6  |
| LOC1     | qwen2.5     | 1 | 132.7     | 98.0 | 27.9                | 0.0       | 4.5       | 20.5     | 60.8  | 0.1       | 69.7   | 3 | 155.0     | 98.0 | 27.8                | 0.0       | 4.6       | 20.5     | 69.8  | 0.1       | 175.2  |
| Base     | qwen2.5     | 1 | 132.4     | 97.8 | 29.1                | 0.0       | 3.5       | 21.6     | 60.8  | 0.4       | 68.5   | 3 | 143.6     | 97.9 | 29.1                | 0.0       | 3.4       | 21.7     | 60.5  | 0.4       | 205.2  |
| LOC1     | mistral-3.1 | 1 | 130.6     | 98.2 | 40.6                | 0.0       | 10.6      | 22.2     | 90.1  | 0.5       | 71.6   | 3 | 138.8     | 98.2 | 41.0                | 0.0       | 10.4      | 22.4     | 90.0  | 0.5       | 211.6  |
| Base     | mistral-3.1 | 1 | 131.9     | 98.1 | 41.2                | 0.0       | 11.0      | 22.0     | 93.4  | 0.5       | 70.6   | 3 | 142.4     | 98.1 | 40.9                | 0.0       | 10.4      | 22.5     | 89.0  | 0.5       | 211.2  |
| LOC1     | phi-4-fp16  | 1 | 131.3     | 98.5 | 122.3               | 0.0       | 23.2      | 95.6     | 278.0 | 1.9       | 72.2   | 3 | 146.8     | 98.7 | 123.4               | 0.0       | 22.5      | 96.4     | 271.2 | 1.9       | 216.7  |
| Base     | phi-4-fp16  | 1 | 131.8     | 98.4 | 122.4               | 0.0       | 22.8      | 95.8     | 278.4 | 1.9       | 72.1   | 3 | 146.0     | 98.5 | 123.2               | 0.0       | 22.9      | 95.6     | 270.4 | 1.9       | 215.7  |
| LOC1     | llama-4     | 1 | 117.5     | 95.9 | 10.5                | 0.0       | 0.9       | 9.6      | 25.3  | 0.1       | 38.5   | 3 | 128.8     | 95.9 | 11.2                | 0.0       | 0.9       | 9.5      | 26.2  | 0.1       | 114.5  |
| Base     | llama-4     | 1 | 119.3     | 95.8 | 11.0                | 0.0       | 0.5       | 9.9      | 22.9  | 0.2       | 38.4   | 3 | 125.2     | 95.8 | 11.0                | 0.0       | 0.5       | 9.9      | 22.8  | 0.2       | 115.3  |

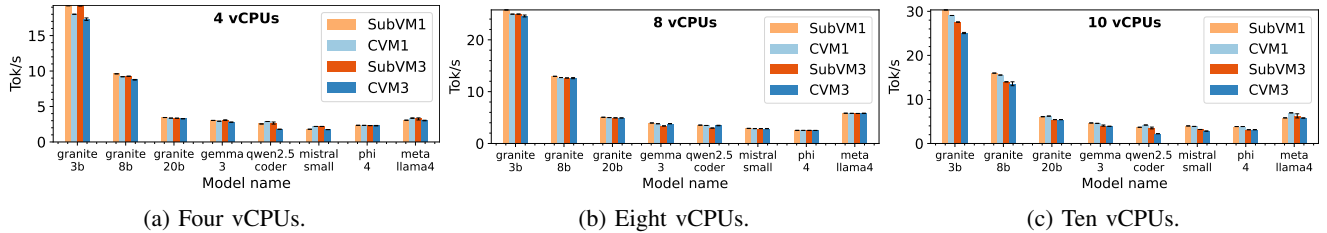


Figure 8: Inference performance for different vCPU configurations.

 TABLE 5: Median  $\mu$ Prof output for granite-3b with two vCPUs in the shared-CCX setup. Inst.: instances; lat.: latency; Acc.: accesses; Mem Bw: memory bandwidth in GB/s.

| Instance |              | L3 misses      |      | DC fills (pti) |            | L2 (pti) |      | L1 (pti)    | RAM    |
|----------|--------------|----------------|------|----------------|------------|----------|------|-------------|--------|
| Type     | No. of inst. | Avg. lat. (ns) | %    | Same CCX       | Local mem. | Miss     | Acc. | DTLB misses | Mem Bw |
| LOC1     | 1            | 113.8          | 95.6 | 26.6           | 3.3        | 23.3     | 62.3 | 0.4         | 25.6   |
| Base     | 1            | 114.4          | 96.2 | 26.7           | 3.2        | 23.4     | 61.5 | 0.6         | 25.3   |
| LOC1     | 3            | 116.4          | 96.1 | 28.4           | 1.7        | 23.9     | 63.3 | 0.4         | 30.9   |
| Base     | 3            | 129.8          | 96.8 | 25.8           | 4.2        | 22.2     | 60.8 | 0.6         | 67.8   |

reduced system memory pressure may improve performance in certain scenarios (e.g., NUMA crossing, CXL memory).

**Summary.** Our evaluation shows that the confidential shared memory primitive in LOCI can efficiently reduce memory pressure from LLM instances (RQ1 and RQ2) without performance penalties (RQ3). In some cases, it improves performance through better cache utilization and reduced pressure on the memory controller (RQ4).

## 8. Discussion

We discuss how LOCI compares to process isolation, how it may be used with other TEE designs, the page-revocation limitation of existing SEV-SNP hardware, and the composition of the shared SVSM TCB.

### 8.1. LOCI vs. Process Isolation

Process isolation naturally scales to more than three instances and uses memory deduplication more effectively than LOCI. However, process isolation does not offer unique and attested accelerator execution. Due to the nature of

processes, they do not run kernel drivers and thus cannot uniquely obtain ownership of an accelerator. While the kernel can pass virtual functions of an accelerator to different processes, these instances have the kernel as their shared TCB. Since the kernel sits between the process and the device, the process cannot leverage hardware attestation.

LOCI trades the more effective memory deduplication of processes for a more compartmentalized architecture and a smaller shared TCB. Each SubVM can leverage SEV-TIO and fully attest to the accelerators it uses. The SVSM shared by SubVMs constitutes a smaller TCB than the OS kernel shared by processes.

### 8.2. Extending LOCI beyond SEV-SNP

LOCI makes use of the VMPL feature of SEV-SNP to enable efficient memory sharing between potentially distrusting guests, as explained in Sections 3.2 and 4.2. Intel TDX supports CVMs in the form of TDs. TDX relies on nested virtualization for its L2 TDs and does not have a concept similar to VMPLs. Thus, LOCI does not directly apply to TDX. Instead of RMP-based isolation across VMPLs in AMD SEV-SNP, TDX isolates L2 VMs using page tables that are programmed by the L1 VM. In TDX, page tables are protected and cannot be modified by the hypervisor, so they are sufficient to isolate distrusting L2 VMs.

Arm CCA supports executing Planes inside a CVM. Each plane may host a SubVM. Therefore, Planes in Arm CCA are similar to SEV-SNP VMPLs, but the number of planes is implementation-defined according to the Arm CCA specification. Thus, LOCI is also applicable to Arm CCA. Alternatively, the RMM could map the same page into different CVMs as shown by prior work [22].

### 8.3. SubVM Page-Revocation Limitation

On existing SEV-SNP hardware, a SubVM in a numerically lower, more privileged VMPL can use RMPADJUST to revoke page permissions of a SubVM in a numerically higher, less privileged VMPL. This can stall the victim SubVM and constitutes a cross-SubVM denial-of-service capability, although it does not grant the attacker access to the victim’s private memory. When the hypervisor forwards the resulting nested page fault to the trusted SVSM, the SVSM can detect the revocation and stop the offending SubVM, as discussed in Section 6. This response does not prevent the initial disruption, and a malicious hypervisor can withhold the fault or refuse to schedule the victim. Consequently, LOCI preserves confidentiality and integrity but does not guarantee availability against such attacks; hypervisor denial of service remains outside our threat model.

### 8.4. Shared TCB

Of the 3717 lines we add to the shared TCB, only 2396 contain code; the remainder are definitions and comments. Of those 2396 LoC, 1913 are necessary to support our boot protocol and boot three distinct SubVMs in one CVM. This code also duplicates data structures already present in Coconut (e.g., boot control blocks and e820 memory maps). Since our SVSM is written in Rust, we can guarantee the absence of typical memory safety errors unless we use the *unsafe* keyword. Throughout the codebase, we use *unsafe* 15 times exclusively for boot code that needs to parse low-level tables (e.g., ACPI) and boot information. The memory-sharing code accounts for only 483 effective LoC and does not use any unsafe keywords. Since the boot code is only active during SubVM initialization, the effective shared TCB increase, reachable by potentially untrusted SubVM programs, is only 483 LoC.

## 9. Related Work

**Secure Virtualization.** The Turtle Project prototypes a system capable of nested virtualization without significant performance degradation [43]. Several prior works have used VMPLs on AMD SEV-SNP. Hecate is the first work to implement an L1 hypervisor (paravisor) in VMPL0 [12]. Hecate uses only two VMPLs, VMPL0 and VMPL1. Schwarz et al. build a framework to re-enable VM introspection on SEV-SNP VMs [14], where VMPL0 acts as a trusted SVSM that the tenant can use to monitor whether VMPL1 is compromised. VeriSMo builds a formally verified SVSM for SEV-SNP [44]. Mei et al. use VMPLs to build a secure KMS in VMPL0 by protecting key material from a compromised higher VMPL [45]. Cabin separates untrusted user-space processes from the guest OS by running them in different VMPLs [46]. Ghosn et al. propose a generic low-level abstraction to create high-level isolation compartments [47]. They could use the insight underlying LOCI to implement their solution on SEV-SNP. Many works aim to provide intra-VM isolation [13], [48]–[51]. While LOCI

uses VMPLs, it is the first work that configures VMPLs to map confidential shared memory regions across mutually distrusting SubVMs in VMPL1–3.

**TEEs & Shared Memory.** A long line of research proposes new TEE designs [11], [52]–[55]. The Keystone framework uses RISC-V PMPs to support enclaves [52]. Timber-V uses memory tagging to create fine-grained enclaves on RISC-V [53]. Cure proposes filtering units on the bus to realize enclaves, requiring hardware changes to RISC-V [11]. Lastly, Sanctum proposes a hardware extension to SGX to partition the caches to enforce stronger isolation [55]. While these platforms propose new TEE designs, they do not address the problem of efficient memory sharing as LOCI does. On the other hand, Cerberus formally models enclave memory sharing to prove correctness [56]. Elasticlave proposes a new TEE memory model on RISC-V that allows secure memory sharing; it can be used to create LOCI’s abstraction as it supports creating read-only views [10]. Recent works on Arm CCA enable efficient memory sharing between devices and CVMs [23], [57], similar to work on RISC-V [11], [58]. Unlike these works, which use memory filters [10], [11], [23], [57], [58] and page tables [23], [57] to enable memory sharing, in LOCI we leverage VMPLs to isolate SubVMs and enable efficient memory sharing between them. Furthermore, our memory-sharing primitive is limited to three SubVMs but supports an arbitrary number of shared memory regions.

## 10. Conclusion

LOCI builds the abstraction of SubVMs on AMD SEV-SNP based on a novel observation. We enable confidential shared memory between SubVMs. Our implementation adds 3717 LoC and increases the shared TCB between SubVMs by 17.53%. We show that this abstraction preserves LLM performance while reducing memory capacity requirements by a factor of up to 2.77. If workloads are co-located within the same CCX, LOCI may improve LLM performance by up to 9.36% depending on the workload. LOCI enables confidential shared memory on existing AMD SEV-SNP hardware without hardware or microcode changes.

## Acknowledgment

We thank all reviewers for their valuable feedback. This work was sponsored by the Funding for Academic Research Program (gift funding) under the AMD University Program. AMD, AMD Instinct, AMD EPYC, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

## References

- [1] AMD, “AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more,” 2020.

- [2] Intel, "Intel Trust Domain Extensions (Intel TDX)," accessed 2023-10-15.
- [3] ARM, "Arm Confidential Compute Architecture (ARM-CCA)," accessed 2023-10-15.
- [4] Google, "Confidential computing — google cloud," <https://cloud.google.com/confidential-computing>.
- [5] Microsoft, "High performance computing - hpc — microsoft azure," <https://azure.microsoft.com/en-us/solutions/high-performance-computing/>, 2021.
- [6] Amazon, "Aws nitro enclaves - create additional isolation to further protect highly sensitive data within ec2 instances," <https://aws.amazon.com/ec2/nitro/nitro-enclaves/>.
- [7] Alibaba, "Build a TDX confidential computing environment," 2024.
- [8] S. Zeng, M. Xie, S. Gao, Y. Chen, and Y. Lu, "Medusa: Accelerating serverless llm inference with materialization," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 653–668.
- [9] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "{ServerlessLLM}:{Low-Latency} serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.
- [10] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4111–4128.
- [11] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapp, "Cure: A security architecture with customizable and resilient enclaves," in *USENIX Security Symposium*, 2021, pp. 1073–1090.
- [12] X. Ge, H.-C. Kuo, and W. Cui, "Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud," in *ACM CCS*, 2022.
- [13] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca, "Veil: A protected services framework for confidential virtual machines," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 378–393. [Online]. Available: <https://doi.org/10.1145/3623278.3624763>
- [14] F. Schwarz and C. Rossow, "00SEVEN – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents," in *USENIX Security*, 2024.
- [15] Coconut SVSM Project, "Coconut SVSM," 2025.
- [16] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions," in *IEEE S&P*, 2020.
- [17] AMD, "AMD64 Architecture Programmer's Manual Volumes 1–5, Rev. 4.07," 2023.
- [18] —, "AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization," 2023.
- [19] —, "Secure vm service module for sev-snp guests," Publication #58019, Revision 1.00, 2023.
- [20] Microsoft, "OpenHCL: the new, open source paravisor," 2024.
- [21] AMD, "AMD RYZEN PROCESSOR SOFTWARE OPTIMIZATION," 2020.
- [22] S. Abdollahi, A. Sadi, M. Kogias, D. Kotz, and H. Haddadi, "Confidential, attestable, and efficient inter-cvm communication with arm cca," 12 2025.
- [23] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Protecting accelerator execution with arm confidential computing architecture," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3423–3440. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/sridhara>
- [24] Y. Yang, M. Sonji, and A. Jog, "Dissecting performance overheads of confidential computing on gpu-based systems," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025, pp. 1–16.
- [25] "QEMU Root Shell Access Vulnerability."
- [26] AMD, "Sev-es guest-hypervisor communication block standardization," Publication #56421, Revision 2.03, 2023.
- [27] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, M. Sethi, X.-H. Dang, P. Li, K.-L. Wu, S. Zawad, A. Coleman, M. White, M. Lewis, R. Pavuluri, Y. Koyfman, B. Lublinsky, M. de Bayser, I. Abdelaziz, K. Basu, M. Agarwal, Y. Zhou, C. Johnson, A. Goyal, H. Patel, Y. Shah, P. Zerfos, H. Ludwig, A. Munawar, M. Crouse, P. Kapanipathi, S. Salaria, B. Calio, S. Wen, S. Seelam, B. Belgodere, C. Fonseca, A. Singhee, N. Desai, D. D. Cox, R. Puri, and R. Panda, "Granite code models: A family of open foundation models for code intelligence," 2024. [Online]. Available: <https://arxiv.org/abs/2405.04324>
- [28] IBM, "Huggingface granite-3b," 2025. [Online]. Available: [https://huggingface.co/ibm-granite/granite-3b-code-base-2k-GGUF/blob/main/granite-3b-code-base.Q4\\_K\\_M.gguf](https://huggingface.co/ibm-granite/granite-3b-code-base-2k-GGUF/blob/main/granite-3b-code-base.Q4_K_M.gguf)
- [29] —, "Huggingface granite-8b," 2025. [Online]. Available: [https://huggingface.co/ibm-granite/granite-8b-code-base-4k-GGUF/blob/main/granite-8b-code-base.Q4\\_K\\_M.gguf](https://huggingface.co/ibm-granite/granite-8b-code-base-4k-GGUF/blob/main/granite-8b-code-base.Q4_K_M.gguf)
- [30] —, "Huggingface granite-20b," 2025. [Online]. Available: [https://huggingface.co/ibm-granite/granite-20b-code-base-8k-GGUF/blob/main/granite-20b-code-base.Q4\\_K\\_M.gguf](https://huggingface.co/ibm-granite/granite-20b-code-base-8k-GGUF/blob/main/granite-20b-code-base.Q4_K_M.gguf)
- [31] G. Team, A. Kamath, J. Ferret, S. Pathak, N. Vieillard, R. Merhej, S. Perrin, T. Matejovicova, A. Ramé, M. Rivière, L. Rouillard, T. Mesnard, G. Cideron, J. bastien Grill, S. Ramos, E. Yvinec, M. Casbon, E. Pot, I. Penchev, G. Liu, F. Visin, K. Kenealy, L. Beyer, X. Zhai, A. Tsitsulin, R. Busa-Fekete, A. Feng, N. Sachdeva, B. Coleman, Y. Gao, B. Mustafa, I. Barr, E. Parisotto, D. Tian, M. Eyal, C. Cherry, J.-T. Peter, D. Sinopalnikov, S. Bhupatiraju, R. Agarwal, M. Kazemi, D. Malkin, R. Kumar, D. Vilar, I. Brusilovsky, J. Luo, A. Steiner, A. Friesen, A. Sharma, A. Sharma, A. M. Gilady, A. Goedeckemeyer, A. Saade, A. Feng, A. Kolesnikov, A. Bendebury, A. Abdagic, A. Vadi, A. György, A. S. Pinto, A. Das, A. Bapna, A. Miech, A. Yang, A. Paterson, A. Shenoy, A. Chakrabarti, B. Piot, B. Wu, B. Shahriari, B. Petrini, C. Chen, C. L. Lan, C. A. Choquette-Choo, C. Carey, C. Brick, D. Deutsch, D. Eisenbud, D. Cattle, D. Cheng, D. Paparas, D. S. Sreepathihalli, D. Reid, D. Tran, D. Zelle, E. Noland, E. Huizenga, E. Kharitonov, F. Liu, G. Amirkhanyan, G. Cameron, H. Hashemi, H. Klimczak-Plucińska, H. Singh, H. Mehta, H. T. Lehri, H. Hazimeh, I. Ballantyne, I. Szpektor, I. Nardini, J. Pouget-Abadie, J. Chan, J. Stanton, J. Wieting, J. Lai, J. Orbay, J. Fernandez, J. Newlan, J. yeong Ji, J. Singh, K. Black, K. Yu, K. Hui, K. Vodrahalli, K. Greff, L. Qiu, M. Valentine, M. Coelho, M. Ritter, M. Hoffman, M. Watson, M. Chaturvedi, M. Moynihan, M. Ma, N. Babar, N. Noy, N. Byrd, N. Roy, N. Momchev, N. Chauhan, N. Sachdeva, O. Bunyan, P. Botarda, P. Caron, P. K. Rubenstein, P. Culliton, P. Schmid, P. G. Sessa, P. Xu, P. Stanczyk, P. Tafti, R. Shivanna, R. Wu, R. Pan, R. Rokni, R. Willoughby, R. Vallu, R. Mullins, S. Jerome, S. Smoot, S. Girgin, S. Iqbal, S. Reddy, S. Sheth, S. Pöder, S. Bhatnagar, S. R. Panyam, S. Eiger, S. Zhang, T. Liu, T. Yacovone, T. Liechty, U. Kalra, U. Evci, V. Misra, V. Roseberry, V. Feinberg, V. Kolesnikov, W. Han, W. Kwon, X. Chen, Y. Chow, Y. Zhu, Z. Wei, Z. Egyed, V. Cotruta, M. Giang, P. Kirk, A. Rao, K. Black, N. Babar, J. Lo, E. Moreira, L. G. Martins, O. Sanseviero, L. Gonzalez, Z. Gleicher, T. Warkentin, V. Mirrokni, E. Senter, E. Collins, J. Barral, Z. Ghahramani, N. Hadsell, Y. Matias, D. Sculley, S. Petrov, N. Fiedel, N. Shazeer, O. Vinyals, J. Dean, D. Hassabis, K. Kavukcuoglu, C. Farabet, E. Buchatskaya, J.-B. Alayrac, R. Anil, Dmitry, Lepikhin, S. Borgeaud, O. Bachem, A. Joulin, A. Andreev, C. Hardin, R. Dadashi, and L. Hussenot, "Gemma 3 technical report," 2025. [Online]. Available: <https://arxiv.org/abs/2503.19786>
- [32] Google, "Huggingface gemma3," 2025. [Online]. Available: [https://huggingface.co/google/gemma-3-27b-it-qat-q4\\_0-gguf/tree/main](https://huggingface.co/google/gemma-3-27b-it-qat-q4_0-gguf/tree/main)

- [33] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, “Qwen2. 5-coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.
- [34] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, G. Dong, H. Wei, H. Lin, J. Tang, J. Wang, J. Yang, J. Tu, J. Zhang, J. Ma, J. Xu, J. Zhou, J. Bai, J. He, J. Lin, K. Dang, K. Lu, K. Chen, K. Yang, M. Li, M. Xue, N. Ni, P. Zhang, P. Wang, R. Peng, R. Men, R. Gao, R. Lin, S. Wang, S. Bai, S. Tan, T. Zhu, T. Li, T. Liu, W. Ge, X. Deng, X. Zhou, X. Ren, X. Zhang, X. Wei, X. Ren, Y. Fan, Y. Yao, Y. Zhang, Y. Wan, Y. Chu, Y. Liu, Z. Cui, Z. Zhang, and Z. Fan, “Qwen2 technical report,” *arXiv preprint arXiv:2407.10671*, 2024.
- [35] Qwen, “Huggingface qwen 2.5,” 2025. [Online]. Available: [https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct-GGUF/blob/main/qwen2.5-coder-32b-instruct-q4\\_k\\_m.gguf](https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct-GGUF/blob/main/qwen2.5-coder-32b-instruct-q4_k_m.gguf)
- [36] “Mistral Small 3.1 .”
- [37] Mistral, “Huggingface mistral-small,” 2025. [Online]. Available: [https://huggingface.co/openfree/Mistral-Small-3.1-24B-Instruct-2503-Q8\\_0-GGUF/tree/main](https://huggingface.co/openfree/Mistral-Small-3.1-24B-Instruct-2503-Q8_0-GGUF/tree/main)
- [38] M. Abidin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett, M. Javaheripi, P. Kauffmann, J. R. Lee, Y. T. Lee, Y. Li, W. Liu, C. C. T. Mendes, A. Nguyen, E. Price, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, X. Wang, R. Ward, Y. Wu, D. Yu, C. Zhang, and Y. Zhang, “Phi-4 technical report,” 2024. [Online]. Available: <https://arxiv.org/abs/2412.08905>
- [39] Microsoft, “Huggingface phi4,” 2025. [Online]. Available: <https://huggingface.co/microsoft/phi-4-gguf/blob/main/phi-4-fp16.gguf>
- [40] “The Llama 4 herd: The beginning of a new era of natively multi-modal AI innovation .”
- [41] Meta, “Huggingface llama4,” 2025. [Online]. Available: [https://huggingface.co/bartowski/meta-llama\\_Llama-4-Scout-17B-16E-Instruct-old-GGUF/blob/main/meta-llama\\_Llama-4-Scout-17B-16E-Instruct-IQ2\\_XS.gguf](https://huggingface.co/bartowski/meta-llama_Llama-4-Scout-17B-16E-Instruct-old-GGUF/blob/main/meta-llama_Llama-4-Scout-17B-16E-Instruct-IQ2_XS.gguf)
- [42] G. Gerganov and the llama.cpp contributors, “The llama.cpp project,” <https://github.com/ggml-org/llama.cpp>.
- [43] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, “The turtles project: Design and implementation of nested virtualization.” *USENIX Association*, 2010.
- [44] Z. Zhou, Anjali, W. Chen, S. Gong, C. Hawblitzel, and W. Cui, “VeriSMo: A verified security module for confidential VMs,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 599–614. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/zhou>
- [45] B. Mei, W. Wang, and D. Lin, “Svsm-kms: Safeguarding keys for cloud services with encrypted virtualization,” in *Science of Cyber Security*, J. Zhao and W. Meng, Eds. Singapore: Springer Nature Singapore, 2025, pp. 217–235.
- [46] B. Mei, S. Xia, W. Wang, and D. Lin, “Cabin: Confining untrusted programs within confidential vms,” in *Information and Communications Security*, S. Katsikas, C. Xenakis, C. Kalloniatis, and C. Lambrinoudakis, Eds. Singapore: Springer Nature Singapore, 2025, pp. 165–184.
- [47] A. Ghosn, C. Castes, N. S. Kalani, Y. Qian, M. Kogias, and E. Bugnion, “Rethinking the confidential cloud through a unified low-level abstraction for composable isolation,” 2025. [Online]. Available: <https://arxiv.org/abs/2507.12364>
- [48] P. Sabanic, M. Misono, T. Bodea, J. Pritzi, M. Hackl, D. Stavroukakis, and P. Bhatotia, “Wallet: Confidential serverless computing,” in *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI ’26)*, 2026, (to appear).
- [49] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, “A Hardware-Software co-design for efficient Intra-Enclave isolation,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3129–3145. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/gu-jinyu>
- [50] D. Kuvaiskii, D. Stavroukakis, K. Qin, C. Xing, P. Bhatotia, and M. Vij, “Gramine-tdx: A lightweight os kernel for confidential vms,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 4598–4612. [Online]. Available: <https://doi.org/10.1145/3658644.3690323>
- [51] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eysers, R. Kapitza, C. Fetzer, and P. Pietzuch, “Glamdring: Automatic application partitioning for intel SGX,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 285–298. [Online]. Available: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind>
- [52] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, “Keystone: An open framework for architecting trusted execution environments,” in *EuroSys*, 2020.
- [53] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, “Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v,” in *NDSS*, 2019.
- [54] S. Arnavutov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eysers, R. Kapitza, P. Pietzuch, and C. Fetzer, “SCONE: Secure Linux Containers with Intel SGX,” in *USENIX OSDI*, 2016.
- [55] V. Costan, I. Lebedev, and S. Devadas, “Sanctum: Minimal hardware extensions for strong software isolation,” in *USENIX Security*, 2016.
- [56] D. Lee, K. Cheang, A. Thomas, C. Lu, P. Gaddamadu, A. Vahldiek-Oberwagner, M. Vij, D. Song, S. A. Seshia, and K. Asanovic, “Cerberus: A formal approach to secure and efficient enclave memory sharing,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’22, 2022.
- [57] F. Sang, J. Lee, X. Zhang, and T. Kim, “PORTAL: Fast and Secure Device Access with Arm CCA for Modern Arm Mobile System-on-Chips (SoCs) ,” in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.
- [58] M. Schneider, A. Dhar, I. Puddu, K. Kostianinen, and S. Čapkun, “Composite enclaves: Towards disaggregated trusted execution,” in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

## LLM Usage Statement

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. Specifically, LLMs were used for code completion during development as well as for grammar and spelling checks.