Enclavisor: A Hardware-Software Co-Design for Enclaves on Untrusted Cloud

Jinyu Gu, Xinyue Wu, Bojun Zhu, Yubin Xia, Binyu Zang, Haibing Guan, and Haibo Chen, *Senior Member, IEEE*

Abstract—The releases of Intel SGX and AMD SEV mark the transition of hardware-based enclaves from research prototypes to mainstream products. These two paradigms of secure enclaves are attractive to both the cloud providers and tenants, since security is one of the key pillars of cloud computing. However, it is found that current hardware-defined enclaves are not flexible and efficient enough for the cloud. For example, although SGX can provide strong memory protection with both confidentiality and integrity, the size of secure memory is tightly restricted. On the contrary, SEV enables enclaves to use more memory but has critical security flaws due to no memory integrity protection. Meanwhile, both types of enclaves have relatively long booting latency, which makes them not suitable for short-term tasks like serverless workloads. After an in-depth analysis, we find that there are some intrinsic tradeoffs between security and performance due to the limitation of architectural designs. In this article, we investigate a novel hardware-software co-design of enclaves to meet the requirements of cloud by placing a part of the logic of the enclave mechanism into a lightweight software layer, named Enclavisor, to achieve a balance between security, performance, and flexibility. Specifically, our implementation is based on AMD's SEV and, Enclavisor is placed in the guest kernel mode of SEV's secure virtual machines. Enclavisor inherently supports memory encryption with no memory limitation and also achieves efficient booting, multiple enclave granularities, and post-launch remote attestation. Meanwhile, we also propose hardware/software solutions to mitigate the security flaws caused by the lack of memory integrity. We implement a prototype of Enclavisor on an AMD SEV server. The experiments on both micro-benchmarks and application benchmarks show that enclaves on Enclavisor can have close-to-native performance.

Index Terms—Enclave, virtulization, AMD SEV

1 Introduction

Since the first day of cloud computing, security has been one of the most concerned issues for end users, who outsource their data and code to cloud servers for efficient computation. However, it is challenging to protect users' outsourced data assets in face of potentially compromised cloud software stack and curious or even malicious cloud operators. There has been a long line of research on constructing isolated execution environments on cloud servers with system software, which are supposed to shield the execution of users' workload from outside attackers. Most of the previous works are based on system software like operating systems and hypervisors [1], [2], [3], which require a relatively large trusted computing base (TCB) and non-trivial modifications to existing cloud software stack. Thus, few of these systems have been deployed by major cloud vendors.

In recent years, hardware support for secure computing is receiving increasing attention in the industry after decades of researching. In 2015, Intel SGX [4] was released, which supports hardware enclave that offers many attractive architectural features, including remote attestation and guarantees of both confidentiality and integrity of data in memory without trusting off-chip DRAM or any peripherals. Thus, many researchers propose to leverage SGX on cloud platforms to protect applications without trusting the cloud [5]. Major cloud vendors have started to explore possible usages of SGX, like Azure's confidential computing and IBM's data-in-use protection.

However, as SGX is getting more widely used and deployed, it is found that many of its architectural limitations hinder its usage. First, SGX only supports a small amount of enclave page cache (EPC) as runtime memory (first 128MB and then 256MB), which significantly limits the performance of memory-intensive applications like in-memory stores and big-data processing which are typical workloads in the cloud. The upcoming SGXv2 will not solve this problem. Second, creating a new enclave involves many steps with non-negligible latency, which makes it not suitable for short-term workloads as in the serverless scenarios (although many of the serverless workloads do have small memory foot-print). Third, the interaction of enclaves (either in-out or between enclaves) causes notorious performance overhead: a cross-boundary function call leads to 7000+ cycles; no memory sharing between enclaves (since they use different memory encryption keys) requires data to be encrypted and decrypted before and after transferring from one enclave to another. One possible reason for these limitations is that SGX is not designed for cloud.[1] Intel also

• *The authors are with the Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {gujinyu, guozhenwuai, tjumongg, xiayubin, byzang, hbguan, haibochen}@sjtu.edu.cn.*

Manuscript received 6 Feb. 2020; revised 12 July 2020; accepted 23 Aug. 2020. Date of publication 26 Aug. 2020; date of current version 8 Sept. 2021. (Corresponding author: Yubin Xia.) Recommended for acceptance by N. Abu-Ghazaleh. Digital Object Identifier no. 10.1109/TC.2020.3019704

1. SGX is first deployed on desktop and notebook processors. Till now, only the Xeon E3 series of server processor supports SGX.

TABLE 1
A Comparison on Intel SGX, AMD SEV, and Enclavisor From Different Dimensions

	Memory encryption	Memory integrity	Memory size limitation	Enclave number	Enclave granularity	Fast boot	Efficient interaction	Remote attestation
SGX	Yes	Yes	All 256MB EPC	Unlimited	Fine-grained	No	No	Pre/post-boot
SEV	Yes	No	All phy-mem	Limited (15)	Coarse-grained	No	No	Pre-boot only
Enclavisor	Yes	Partial [*]	All phy-mem	Unlimited	Multiple	Yes	Yes	Pre/post-boot

Our design can defend against two types of attacks caused by no integrity guarantee (specified in Section 3.6).

suggests that SGX is more suitable to protect only small parts of applications [6].

On the contrary, AMD released its enclave in 2016, with techniques named secure memory encryption (SME) and secure encrypted virtualization (SEV) [7], which is designed for the cloud in the first place. Unlike defining a new abstraction of an enclave as Intel SGX does, AMD SEV reuses a virtual machine as its enclave and tries to make the protection as transparent as possible to existing cloud applications. It also supports memory encryption and has no memory limitation for enclaves to use.

Unfortunately, these benefits come with the cost of security and flexibility: first, the virtual machine granularity is too coarse-grained and heavyweight for enclaves. For cloud users who only want to run an application in the enclave, they have to deploy a guest OS as well. Meanwhile, current SEV can only support a limited number of enclaves running concurrently (15 on AMD EPYC 7281) due to the limitation of ASID. Running more enclaves at the same time requires evicting and reusing of ASID, which will hurt performance. Second, SEV cannot guarantee the integrity of enclave memory. This could be a critical security flaw, and many researchers have reported successful attacks even if the memory of enclaves is encrypted [8], [9]. Third, SEV only supports one-time attestation for each enclave before it is launched, which does not allow multiple users to attest long-running cloud services.

In this paper, we first give a systematic analysis on the mismatching between what hardware enclave offers and what cloud application requires. Then we propose a new enclave system with a novel hardware-software co-design that can well fit the requirements of cloud scenario by embracing the flexibility of software and strong security guarantee of hardware. Specifically, our implementation is mainly based on AMD SEV. But instead of using an entire virtual machine as an enclave, we introduce a small layer of software running in guest supervisor mode, named *Enclavisor*, to manage enclave instances running in guest's user mode. The Enclavisor is responsible for enclaves' life cycle management, including creation, attestation, scheduling, deletion, interaction, etc. As shown in Table 1, our system can achieve the most needed requirements of cloud workloads:

- *High-security insurance*: All the off-chip memory are encrypted by hardware. Meanwhile, we also propose both software solutions and hardware enhancements to mitigate the security problems due to a lack of memory integrity on SEV.
- *Large memory support*: There is no limitation on the memory size of the enclaves. Also, the overhead of secure memory access is usually trivial.
- *High performance*: Enclavisor can create a new enclave instance by *fork* to achieve fast boot for serverless and FaaS workloads [10], [11], [12]. It also supports FlexSC-like [13] fast interaction (see Highway in Section 3.4).
- *Flexible remote attestation*: We propose *two-phase attestation*, which leverages Intel SGX (as a proxy node) to support post-launch remote attestation of Enclavisor, which is not supported by SEV but required by long-running services with multiple users.

Our system is implemented and evaluated on a server with AMD EPYC 7281 processor. The evaluation shows that enclaves on Enclavisor can achieve near-native performance in both micro-benchmarks and real-world benchmarks. In summary, this paper makes the following contributions:

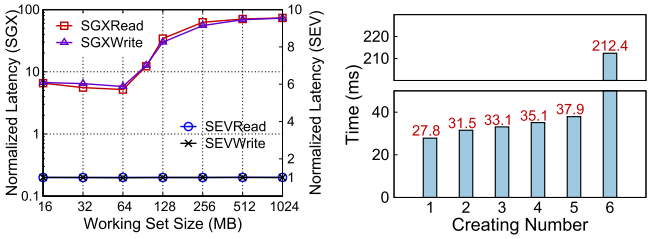
- A systematic analysis on the mismatching between existing architectural features and requirements of cloud applications.
- A practical software-hardware co-design for building secure and efficient enclaves for the cloud.
- An implementation of a prototype and an evaluation on real hardware and applications.

2 Motivation and Background

The security of cloud tenants' data and application highly depends on the infrastructure of the multi-tenant cloud being secure. However, with security threats such as vulnerabilities in hypervisors and malicious cloud operators, it is no surprise to see frequent reports of secret leakages or abuses in clouding computing. To ameliorate this problem, there is a long line of research of attempting to construct enclaves (i.e., trusted execution environment, TEE) for secure computation in commodity clouds. Yet, most prior systems [2], [14], [15] require intrusive software/hardware modifications or bloat the trusted computing base (TCB) of secure applications due to trusting the hypervisor or even the guest OS.

2.1 Intel SGX's Limitations

Intel SGX [4] is one of the most promising solutions. It is the first security extension being deployed on commodity processors that can protect both the integrity and confidentiality of memory, which takes a strong threat model that can even defend against physical attacks. Thus, many researchers try to leverage SGX to provide strong security guarantees for cloud applications with minimal trust on software components. In recent years, SGX is used to protect applications like KMS, databases, big data processing, containers and FaaS [5], [16], [17], [18], [19], [20], [21]. Meanwhile, major



(a) Secure memory access. (b) SGX enclave boot.

Fig. 1. (a) We quantify the performance degradation of secure memory accessing on Intel SGX and AMD SEV machines, separately. The micro-benchmark repeatedly reads/writes a random 4K page within a specified working set, and compares the performance of enclave execution with native execution. SGX experiments are done in a VM with 92 MB EPC while AMD experiments are done in a normal and a secure VM. (b) This micro-benchmark measures the SGX enclave booting performance by continuously creating 6 sample enclaves provided in Intel SGX SDK. The enclave size is 16 MB. If the newly created enclave is accessing memory, the last enclave's boot time increases to 212.4 ms. Even if not, the boot time is still about 110.0 ms. The experiments are done in VM with 64 MB EPC.

cloud vendors like Microsoft Azure and IBM cloud also use SGX for protecting sensitive data.

However, as SGX is getting more used and deployed, it is found that SGX still cannot fulfill some essential requirements of cloud, including the lack of multi-processor support, being vulnerable to various side channel attacks, etc. Among these requirements, the most demanded two are larger memory size and higher performance. As shown in Fig. 1a, the integrity check and the memory swapping caused by limited EPC size can lead to a substantial performance slowdown for EPC access compared with normal memory access. Fig. 1b shows the latency of loading/booting an SGX enclave application, which is much slower than loading a non-SGX application (e.g., launching Memcached only takes 479 us). Note that the presented results do not include remote attestation which can further increase the booting latency significantly. Similar results are reported in many studies [18], [20]. These limitations significantly limit SGX's usage for scenarios like memory-intensive and latency-sensitive applications on the cloud. We analyze these two limitations and find that they are intrinsic issues and the cause is rooted in the design for memory protection.

Why is it Hard to Support Large EPC in SGX?. SGX leverages counter-based encryption for protecting memory confidentiality and Bonsai Merkle Tree (BMT) for ensuring memory integrity. SGX maintains an 8-byte hash for each 64-byte secure memory block (cacheline size) as well as the counters, and checks the hash for each EPC access operation [22], [23]. It stores each hash for the memory block as leaf nodes of a 4-level Merkle hash tree. Specifically, for 128 MB EPC, only 96 MB are used for storing data. Correspondingly, the 96 MB data will have 12 MB counters for encryption and 12 MB MACs for integrity check. The counters will then have a Merkle hash tree with 1.5 MB level-0 nodes, 192 KB level-1 nodes, 24 KB level-2 nodes, and 3 KB level-3 nodes. Only the level-3 nodes are stored in the very limited CPU internal storage.

Therefore, enlarging the EPC size leads to the growth of the size and depth of the hash tree, which will lead to poor cache locality and higher memory bandwidth penalties when navigating the tree, which will in turn cause longer memory access latency.

Why is it Hard to Support Fast Boot of an SGX Enclave?. Boot speed is critical for the emerging short-term cloud tasks, especially in the serverless scenario [10], [11], [12]. The booting cost of SGX enclaves mainly comes from the expensive enclave constructing instructions: during booting, all the memory pages need to be added to an enclave one by one, which leads to data copy from ordinary pages to EPC pages. It is needed by the processor to calculate a hash of the loaded application for remote attestation. Meanwhile, since the total size of EPC is limited, it is likely to trigger EPC swapping during the booting processing, as in the last case of Fig. 1b.

A traditional way to optimize booting latency of cloud applications is using fork-style copy-on-write mechanisms to skip the initialization process of an application [12]. However, since each SGX enclave has its own memory encryption key, the copy-on-write mechanisms are not possible to be applied here.

Besides the two limitations mentioned, the design of SGX is also not friendly to interactions. First, the code within an enclave cannot invoke system calls directly through instructions like *syscall*. Second, as memory sharing is disabled between enclaves, message passing between two enclaves needs at least two memory copies and two encryption/decryption operations. Such overhead suggests to put all services into one enclave, which contradicts the principle to minimize code within each enclave for better security.

2.2 AMD SEV's Limitations

AMD SEV [7] is a new virtualization security paradigm that has been supported by software stacks of modern cloud [24], [25]. SEV integrates memory encryption with AMD-V virtualization architecture and aims to protect virtual machines from the potentially malicious hypervisor. Inside the processor, SEV tags all the data of a VM with its address space ID (ASID), which prevents the data from being used by anyone other than the owner VM. When data leaves/enters the processor, it is automatically encrypted/decrypted by the memory controller with a key bound to its owner VM. The keys are managed by a secure co-processor and will never be exposed to any software. SEV decides whether to encrypt one memory page according to one bit (C-bit) in the corresponding guest page table entry. It is easy for a VM to mark selected memory pages as confidential by setting the C-bits in its own page table, and leave others as plaintext pages for communications with others. An extension of SEV, named SEV-ES (encrypted state), encrypts and protects a secure VM's state as well as its execution contexts. Although SEV has no limitation on enclave's memory size, it brings new limitations for being used to secure critical cloud applications.

First, the isolation granularity in SEV is VM (*Challenge-1*), which means that if a user only needs to protect a small piece of code, she has to deploy a large guest OS in the enclave. Since the size of popular OSes has increased dramatically (e.g., the Linux kernel 4.5 has over 20 million source lines of code), such requirement will excessively enlarge application's software TCB. Moreover, VM-level enclave also leads to long boot time, which mismatches the fast boot requirements in a popular cloud computing paradigm named serverless [10],

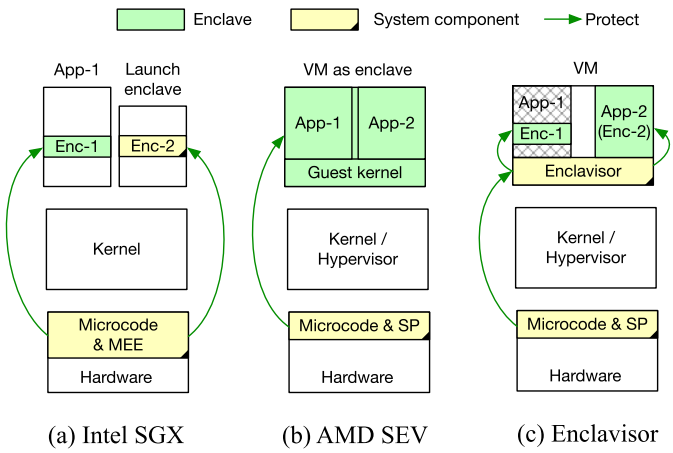


Fig. 2. Enclavisor as the extension of secure hardware. (Enc: Enclave, MEE: Memory Encryption Engine, SP: Secure Processor).

[11], [12] (especially, 1ms in [26]). For example, launching a simple Kata container [25] (the container is actually a micro SEV VM) takes about 2.6s.

Second, current SEV only allows 15 secure enclaves to run concurrently (*Challenge-2*). This is because SEV associates ASIDs with VMs' memory encryption keys and an ASID has only 4-bit. To re-allocate an in-use ASID for a new secure VM (e.g., the 16th secure VM), the hypervisor first needs to flush the cache on all the cores that the old VM runs on to ensure consistency and then deactivates the ASID (i.e., stops the old VM). The process is time consuming and not supported by current software yet. Therefore, in the case of using one application for one enclave, it may not only waste the parallelism of multiple cores, but also incur high booting/switching overhead when overcommitting ASID for more enclaves.

Third, an SEV-secured VM can only be attested for one time at the boot time (*Challenge-3*). So, only one tenant can attest and build a trusted channel with a secure VM, which prevents multiple mutual-distrust tenants from securely sharing the VM.

Last but not least, SEV does not ensure memory data integrity (*Challenge-4*), which has been utilized as a attack vector [8], [9], [27], [28]. It means that a even if the memory of a secure VM is encrypted, a compromised hypervisor is still able to tamper with or even steal VM's private data without knowing the encryption key. There are two categories of attacks, one is *rollback attack* and the other is *NPT mapping attack*. More details are discussed in Section 3.6.

In all, it is challenging to design a secure hardware enclave that can fit different usages in various cloud scenarios. So in this paper, we propose Enclavisor, a hardware-software co-designed system which leverages software as an extension of existing secure hardware to construct enclaves in a more flexible way to meet different requirements of cloud (see Table 1). As shown in Fig. 2, our design and implementation is based on AMD's SEV and Enclavisor is placed in the guest kernel mode of SEV's secure virtual machine. Enclavisor inherently supports memory encryption with no memory limitation, and also achieves multiple enclave granularities, efficient booting and post-launch remote attestation. Meanwhile, we also propose hardware/software solutions to mitigate the security flaws caused by the lack of memory integrity.

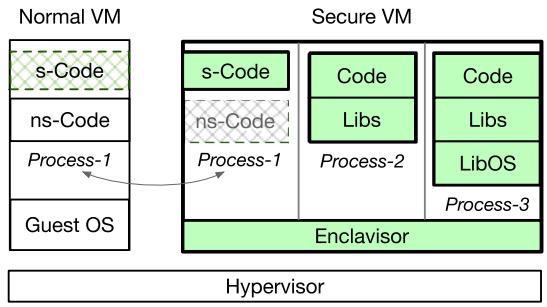


Fig. 3. An overview of our system. An enclave needs to trust Enclavisor besides the SEV hardware. (s-Code: secure code, ns-Code: non-secure code).

3 Design

3.1 Threat Model

We assume that Enclavisor is trusted and cannot be compromised. An attacker can compromise and control system software components, including the guest OS as well as the hypervisor. Enclaves are mutual distrust. We do not consider denial-of-service (DoS), side channel attacks, or hardware bugs (e.g., rowhammer attacks). We also do not consider physical attacks on the secure memory integrity, including *physical* rollback attacks in which an attacker physically overrides a memory page with its previous (encrypted) content (e.g., by injecting memory transactions to the memory bus directly).

3.2 System Overview

SEV technology protects a VM as a whole (*Challenge-1*), which means an application needs to trust the guest OS in the VM. To solve this challenge, our system separates the application and the guest OS into different isolated execution environments. Such design can provide fine-grained isolation for secure applications and can remove the guest OS from TCB.

Nevertheless, current SEV only allows at most 15 secure VMs to run (*Challenge-2*), which makes it impossible to let each secure application to monopolize a secure VM as its isolated execution environment. To solve this challenge, our system uses one secure VM and provides mutually-isolated execution environments within the VM as enclaves for applications to use.

Fig. 3 gives an overview of our system which provides fine-grained enclaves inside one secure VM under the supervision of *Enclavisor*. Enclavisor is a trusted software layer running in the secure VM's kernel level (i.e., ring-0 in guest mode). It is responsible for building and managing secure enclaves: first, it manages the memory for enclaves through page tables including enforcing memory isolation between (mutual-distrusted) enclaves; second, it schedules the enclaves in the secure VM. Different from traditional guest OSes, Enclavisor (9,800 SLOC) is as a flexible extension of the secure hardware to support different enclaves and does not implement most OS functionalities such as POSIX system calls. Logically, an enclave belongs to its host application and just executes in a secure environment, and most of the enclave's system calls are routed to its guest OS on which the host application runs.

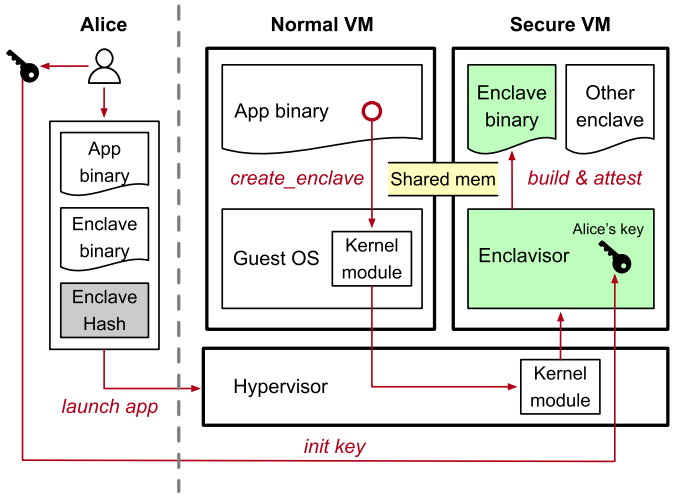


Fig. 4. An application can issue the enclave creation request and Enclavisor is responsible for building the enclave.

```
/** Success: return non-negative enclave id
  * Arg1: enclave image & config & hash
  * Arg2: the enclave owner identifier */
int create_enclave(Buf image, ID userID);

/** Success: return zero
  * Arg1: enclaveID returned by create_enclave
  * Arg2: [1] [input] passing entry_point & arguments
  *       [2] [output] get feedback
  * Arg3: choose normal mode or fast mode */
int enter_enclave(int enclaveID, Buf buffer, int mode);

/* shutdown an enclave */
void destroy_enclave(int enclaveID);
```

Fig. 5. Three simplified interfaces: our SDK provides similar interfaces with those in SGX SDK. As the SGX programming interfaces are easy-to-use and widely adopted, Enclavisor inherits its interfaces for better practicability.

Although Enclavisor can provide mutual-isolated enclaves in a secure VM, cloud tenants still cannot share the VM because SEV remote attestation can only be done for one time at the VM boot time (*Challenge-3*). The probable reason for such attestation design is SEV assumes a secure VM as an enclave and has one owner. To overcome this obstacle, our idea is leveraging an Intel SGX enclave as a secure delegation for establishing trusted communication channels between Enclavisor and different tenants. Details are in Section 3.5.

SEV does not protect memory integrity (*Challenge-4*), which may lead to critical security flaws. First, though it is non-trivial to physically modify the memory in cloud (out of our scope), a compromised hypervisor can easily launch rollback attacks. Second, a malicious hypervisor can steal secrets from a secure VM through manipulating the nested page table (NPT). To defend against these attacks, our basic idea is (i) depriving the untrusted hypervisor of the writing capability to the secure memory, and (ii) reducing the its impact on the NPT mapping. Details are in Section 3.6.

3.3 Enclave Application Launching

Fig. 4 depicts the launching flow of a cloud tenant's enclave application. First, a cloud tenant Alice uploads her application image including the secure enclave part to a VM in the cloud. She also calculates a hash measurement for constructing the secure enclave and encrypts the hash with one secret key (Key_Alice). Second, Alice's application will be started as a normal application in a normal VM, and the application can invoke the *create_enclave* interface for creating an enclave. As shown in Fig. 5, for invoking such interface, the application needs to provide the enclave image, enclave configuration (e.g., legal entry points), and the hash of both. Third, through the added kernel modules, Enclavisor in the secure VM will receive the request of creating an enclave. Then, it will retrieve the requested content (i.e., the arguments of *create_enclave*) in the inter-VM shared memory and construct the secure enclave. The shared memory between the normal and secure VMs is pre-built for transferring data. After that, Enclavisor calculates a measurement on the enclave memory content and its configuration. If the calculated hash measurement matches the decrypted one from the enclave owner, it will establish the inter-process shared memory between the enclave and its host application, approve the enclave to run, and finish the creation process. Otherwise, it will refuse to create the enclave and return an error. For attesting the enclave, Enclavisor needs to share a secret key with enclave owner (e.g., Key_Alice). We will explain when and how Enclavisor gets the key in Section 3.5.

After successfully creating an enclave, the host application can use *enter_enclave* to invoke secure enclave functions. The shared memory established during the process of enclave creation is used for passing arguments and receiving results. Besides, this interface has two modes: the normal mode relies on the privileged software to transfer the control flow; the fast mode is based on *Highway* (see Section 3.4) and does not need the involvement of privileged software.

Besides protecting secure parts of an application, Enclavisor can also protect unmodified applications. Specifically, our SDK provides a common host application which just creates an enclave to hold the unmodified application as well as the relied libraries and invokes its main function. When the application in the enclave invokes system calls, (most of) the corresponding requests will be redirected to the guest OS in the normal VM (see Section 3.4).

3.4 Interaction With Highway

This subsection answers two questions: (i) how an enclave and its host application communicate with each other; (ii) how to redirect the system calls invoked by enclaves to the guest OS.

As depicted in Fig. 6, our system can leverage the functionalities provided by privileged software (i.e., we add kernel modules) to support the interaction between enclaves and their host applications. Nevertheless, this communication way (named *trap-based communication*) is not efficient. Specifically, there are 8 context switches (4 between ring-0 and ring-3 in guest mode, 4 between guest mode and host mode) for an enclave to invoke a host application's function, which involves expensive *VMRUN*/*VMEXIT*. Such a process takes a considerable cost not only because the context switch instructions are expensive but also because the switches can

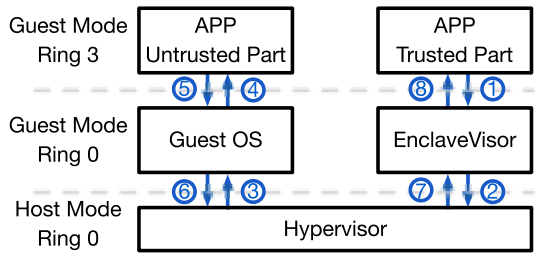


Fig. 6. Trap-based communication. With the help of privileged software, an enclave and its (untrusted) host application can interact with each other.

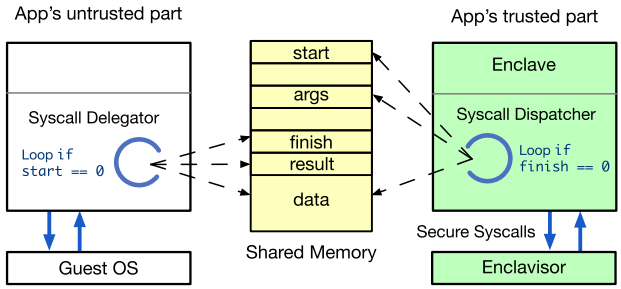


Fig. 7. Highway. A fast communication channel between an enclave and its host application based on shared memory. Dotted arrows represent write accesses.

incur much indirect overhead including cache and TLB pollution and scheduling cost.

For fast communication, our system also supports establishing a shared-memory-based *Highway* between an enclave and its host application. Highway helps them to transfer not only the communication data but also the control flow.

How an enclave redirects system calls: We add a thin layer for dispatching system calls in the standard C library (i.e., glibc) which is transparent to enclave applications. Instead of invoking *syscall* instructions directly, an enclave thread (i.e., syscall dispatcher in Fig. 7) will route system calls to the guest OS through Highway.

Specifically, the dispatcher first writes the arguments as well as the related data into the shared memory, and then sets the *start* field to *one* (indicating a system call request). After that, it waits until the *finish* field becomes *one* and then gets the results of the system call. In the host application, there is a daemon thread (i.e., syscall delegator in Fig. 7) polling on the *start* field for detecting system call requests. The daemon thread actually is the application thread that invokes *enter_enclave*. Recall that *enter_enclave* has a fast mode. When choosing such mode, the application thread loops on the *start* field as the daemon thread for the enclave thread. Once detecting a system call request, it invokes the system call on the guest OS and writes return value in the *result* field. At last, it sets the *finish* field as *one* to inform the dispatcher (the enclave thread) of the finish of the request.

Our system also supports running the daemon thread for system call delegation in the guest OS, which can further remove context switches for invoking system calls. We also carefully design the data layout in the shared buffer for better cache locality. For example, we let the *start* field exclusively take a cacheline (64 bytes) for avoiding false sharing, and insert another 64-byte slot (cacheline) after it to avoid the effects of CPU cacheline prefetching.

Besides system calls, Highway can also be used for other interactions between host applications and the enclaves. Our system supports both the trap-based and the polling-based communication. The latter one is faster and more suitable for the scenario of frequent interactions but burns extra cores, and the former is on the opposite side.

Last but not least, we also consider the security threats of reusing untrustworthy system calls because the guest OS can issue Iago attacks [29]. To mitigate this problem, the system call dispatcher can configure which system calls to redirect. By default, the dispatcher does not route enclave memory-related system calls (e.g., brk) to the guest OS and these system calls are implemented in Enclavisor. Furthermore, Enclavisor also provides an in-memory file system (secure FS) for enclaves to use. If an enclave creates a file with a specific prefix (i.e., *Enclavisor/*), the file will be stored in the secure FS and all the file operations on it are handled by the secure FS. Since cryptographic safeguards such as SSL are usually employed by secure applications to protect network communication, we just reuse the network services of the untrusted guest OS without extra efforts. In brief, Enclavisor can provide some shared services like a secure FS and the system call dispatcher inside each enclave can choose whether to use such service according to different policies. Besides, the dispatcher can embrace different security shields for defending against potential attacks, which can be transparent to enclave applications.

An Optional Optimization. We further give an optimization to improve the performance of the above design. When the syscall dispatcher waits for the finishing of a system call, it occupies the CPU and thus wastes CPU cycles. To avoid the busy wait, we adopt the mechanism of *user-mode scheduling* [13], [18] and then provide a simple scheduler in the LibC which can schedule multiple threads in an application. After issuing a (cost) system call request through Highway, the dispatcher can proactively release the CPU and transfer the control flow to the user-mode scheduler. Then, the scheduler can immediately schedule other ready threads and thus improve the CPU utilization. Note that this optimization is transparent to an application and optional.

3.5 Two-Phase Attestation

This subsection answers two questions: (i) how to let multiple tenants attest the same SEV VM; (ii) how Enclavisor retrieves a tenant's secret key.

SEV only allows a secure VM to be attested for *one time* in the boot process. If some cloud tenants or the cloud provider attests the secure VM, other tenants are hard to trust the VM. An intuitive solution is letting a trusted third party attest the secure VM. However, it takes extra deployment cost.

As shown in Fig. 8, our design allows the cloud provider to attest a secure VM through an Intel SGX enclave. To be specific, when booting an SEV-protected VM, the cloud provider uses an attestation proxy that runs in an SGX enclave to attest the VM. The attestation proxy leverages the AMD Key Distribution Server (KDS) to verify the VM booting proof. If the proof is authenticated, the attestation proxy can know the Enclavisor is securely booted without being tampered. Therefore, the attestation proxy in the SGX enclave can establish a secure channel with Enclavisor in the SEV-protected VM.

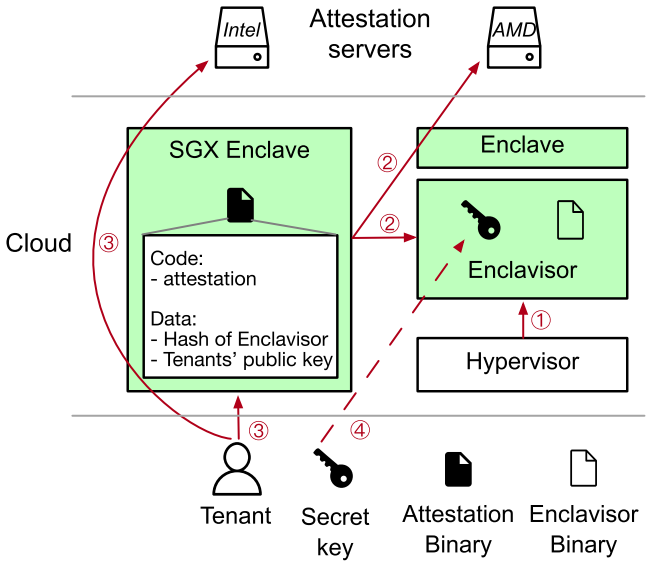


Fig. 8. We utilize an Intel SGX enclave as a secure attestation proxy.

The attestation proxy also contains a number of cloud tenants' public keys which are uploaded when the tenants register in the cloud. A cloud tenant can utilize the Intel Attestation Service (IAS) to verify the attestation proxy and the proxy can use a tenant's public key to authenticate him/her. With such mutual attestation, the cloud tenant can establish a secure channel with the SGX-protected proxy.

Next, the cloud tenant can send one secret key to the proxy and the proxy can forward the key to Enclavisor. Therefore, the cloud tenant and Enclavisor can securely communicate with each other with that key (i.e., a secure communication channel between the tenant and Enclavisor). With SGX for secure connection, different tenants can build private channels with the same Enclavisor and thus can securely share the secure VM. Note that the code of the attestation proxy and Enclavisor should be public and can be investigated by any party.

Our *two-phase attestation* mechanism works as following steps: In *phase-1* which is only needed for one time, a tenant indirectly attests an SEV VM with a trustworthy Enclavisor running inside with SGX and then builds a secure channel with Enclavisor. Afterwards, Enclavisor can directly attest the tenant's secure enclaves without the tenant's involvement (*phase-2*). Our two-phase attestation has two purposes as other remote attestation mechanisms. First, it allows a tenant to ensure his/her initial enclave is correctly running. Second, it helps a tenant to establish a secure communication channel with Enclavisor (and inherently, his/her enclaves). So, a tenant can securely upload his/her private code/data to his/her enclaves and the enclaves can securely send the results back to the tenant.

Our two-phase attestation is a kind of secure boot or chained attestation [1], while its new contribution is making SEV attestation flexible through a practical solution that uses an SGX enclave as a proxy.

3.6 Mitigating Memory Integrity Issues

SEV hardware promises no integrity protection for a secure VM's encrypted (private) memory pages, which, however, can severely degrade the security guarantees. In this subsection, we first describe two potential attacks to an SEV-protected VM and then propose the corresponding mitigations. Although we still cannot provide fully memory integrity guarantee, the mitigations will significantly raise the bar of successful attacking and can meet the requirements of most cloud scenarios.

Rollback Attacks. Although attackers cannot inject arbitrary code/data into a secure VM because they cannot access the memory encryption key,[2] they can replace one memory page with its older version. For instance, a compromised hypervisor can record a snapshot of a secure VM and then guess the password to access the root shell (only allows three trials for entering password) in the VM. Since the hypervisor can rollback the secure VM to an initial state with the legal snapshot, it can use a brute-force attack to guess the password for unlimited times. *The root cause of rollback attacks is that attackers can record secure memory pages and override the new content with the old content by a compromised hypervisor*.

Nested page table (NPT) mapping attacks. In AMD hardware virtualization technology, NPTs map guest physical addresses (GPA) to host physical addresses (HPA) while guest page tables (GPT) in the VM map guest virtual addresses (GVA) to guest physical addresses (GPA). For an SEV-secured VM, the hypervisor cannot modify the GPTs in it but has full control of the NPT. Arbitrarily changing the NPT mapping is meaningless because it (usually) causes DoS attacks only. However, a subtle hypervisor can steal secrets from a secure VM through carefully manipulating the NPT mappings. For example, an enclave on the Enclavisor needs to copy some plaintext content (e.g., a socket configuration) to the shared memory when invoking a system call. It has no problem to expose such insensitive data to attackers. However, a compromised hypervisor can change the NPT mapping for page where the socket configuration resides to any other memory page of the secure VM. Thus, the secure VM will write its secret data into the shared buffer spontaneously. Theoretically, the hypervisor can dump all the secure memory of a VM in such way and thus break the cryptographic (privacy) guarantee. *The root cause of NPT mapping attacks is that the untrusted hypervisor can arbitrarily modify the NPT mappings*.

The basic idea to mitigate these two types of attacks is *depriving/restricting the hypervisor's access to secure memory pages and the related NPT mappings*.

Software Solution. To that end, one possible solution is to introduce another high-privileged layer, e.g., nested virtualization [1], to restrict the capabilities of the hypervisor. However, this solution may incur non-trivial overhead due to frequent cross-layer switches. Therefore, we choose a lightweight approach, sibling protection with non-bypassable memory isolation [30], [31], [32]. Specifically, we can deploy a trusted component such as Fidelius [31] in the same privilege level (i.e., ring 0 in host mode) with the hypervisor, but isolates the component from the untrusted

2. In native SEV, modifying encrypted memory blindly is considered as DoS attacks since the CPU will get meaningless data or code after decryption. Nevertheless, to prevent malicious software from conducting rollback attacks, our designs disallow malicious software including a compromised hypervisor to write secure memory. Therefore, blindly writing secure memory is also forbidden in our system.

hypervisor. The hypervisor must invoke the trusted component to modify the page table mappings. And the latter one is responsible for verifying the mapping requests to enforce the policies as follows:

- *Security Policy 1:* Once a physical page is mapped as a private page to a secure VM, it cannot be mapped as writable to others.
- *Security Policy 2:* When the hypervisor needs to swap out a secure VM's private page, the trusted component needs to calculate and record the hash of the page.
- *Security Policy 3:* When swapping back a private page of some secure VM, the trusted component needs to check the page content against the recorded hash and ensure the page is mapped to the original guest physical addresses.

We let Enclavisor use a fixed GPA region for shared memory and thus the trusted component can easily tell whether a page is private to a secure VM according to the GPA and then enforce the first policy. This policy ensures that any malicious software cannot write (or rollback) a secure VM's mapped private pages.

To preserve the memory oversubscribing functionalities of the hypervisor, we further introduce the other two policies. The trusted component also involves in the process of page swapping and simply uses the hash of page content to prevent possible rollback attacks. Meanwhile, it ensures a swapped-out page can only be mapped back to the original GPA for defending against the NPT mapping attacks.

Hardware Proposal. The above software solution works because there is a trusted software component for maintaining the mapping information for each physical page. Therefore, the new SEV hardware can also implement a similar book-keeping mechanism to defend against those two types of attacks.

Specifically, the SEV hardware reserves a part of physical memory for storing the *Ownership-Table* which records the required metadata of each physical page in the rest of the physical memory. As shown in Table 2, each physical page's metadata contains two fields: *Owner* and *Guest Frame Number (GFN)*. The *Ownership-Table* is checked by MMU at the end of traditional (nested) address translation. After a (guest) virtual address is translated into HPA (physical page), MMU further uses the HPA as index to find the corresponding physical page's metadata in Ownership-Table. If the *Owner* of this page is not the one currently running on the CPU, MMU will deny the memory access and trigger an exception. By default, a physical page has no owner, which means MMU will not trigger any exception when it is accessed. The physical pages owned by some secure VM cannot be accessed by hypervisor and other (secure) VMs. So, any malicious software cannot read/write a secure VM's private memory and thus cannot conduct rollback attacks. Note that memory encryption can be optional (for defending against physical attacks like cold-boot attacks or NVM stealing) in such a new extension because the hypervisor can no longer read the secure memory.

Since MMU checks physical pages' ownership, the untrusted hypervisor cannot map a secure VM's private memory pages (GPA) to physical pages (HPA) not owned by the secure VM (otherwise the VM crashes). However, it still can remap a VM's private memory page to any physical page belonging to the secure VM, i.e., changing the mapping from GPA to HPA for conducting NPT mapping attacks. So, if a GVA is translated (i.e., GVA - GPA - HPA) and the physical page's *Owner* is some secure VM, MMU will also check whether the GPA (the intermediate result of the address translation) matches the physical page's GFN ($GFN = GPA >> 12$) in the *Ownership-Table*. Thus, a malicious hypervisor cannot arbitrarily manipulate the NPT mappings.

TABLE 2
The Hardware-Maintained Ownership-Table (Indexed by Physical Page)

Physical Page	Owner	Guest Frame Number
0	None	–
1	secure VM1	0x2000
2	secure VM1	0x2001
3	secure VM2	0x2000

With the new hardware, the hypervisor needs to explicitly assign physical pages to a secure VM at its boot time or during its runtime. When the hypervisor sends the request of launching a secure VM to the hardware, the request should contain a small physical memory range (i.e., a starting HPA and a size) and the corresponding guest physical memory range (i.e., a starting GPA and a size) of the secure VM. For each physical page (HPA), the hardware checks its *Owner* field: if the field is *None*, the hardware sets it to the secure VM and sets the corresponding GFN (GPA); otherwise, the hardware undoes the changes and returns an error. A small physical memory range is enough for the booting process of a secure VM (e.g., 24M for Enclavisor) and more memory can be added at runtime. Besides, the request information will be also included in the remote attestation, so the secure VM can know which GPA range is already mapped to physical memory.

A secure VM may require more memory and trigger NPT faults during runtime. The hypervisor can assign physical pages (on-demand) to it as follows. The hypervisor allocates a free physical page (HPA) and tells the secure VM the HPA and the faulting GPA (through either a software-based or hardware-provided upcall). The secure VM should first check whether GPA is mapped before and (if not) then execute *accept_page* (a new instruction) to set the corresponding entry in the *Ownership-Table*. After that, the hypervisor can add the new mapping in the NPT. In brief, for adding a new mapping (GPA - HPA), the secure VM (e.g., Enclavisor) ensures its GPA is never double-mapped, which eases the hardware's design and implementation (hardware ensures the HPA has no owner before).

A hypervisor cannot directly revoke physical pages from a secure VM, but it can ask a secure VM (e.g., Enclavisor) to return some physical pages back (like ballooning). A secure VM can use another new instruction named *release_page* to release its not-used pages. Nevertheless, as non-volatile memory (NVM) (i.e., Intel Optane Persistent Memory) is commercially available now, the memory resource pressure will become much smaller since NVM can have larger capacity and lower price than traditional DRAM. Thus, memory ballooning may not frequently happen. Besides, a secure VM

will not exhaust the physical memory resource for two reasons. First, the hypervisor can restrict the maximum amount of the physical memory used by a secure VM. Second, the ballooning code inside the secure VM should be implemented in some trusted module like Enclavisor that is deployed by the cloud provider. Therefore, the hypervisor can still preserve its capability of managing physical memory.

At last, we analyze the above hardware enhancement from two aspects: memory-usage overhead and memory-check latency. First, the *Ownership-Table* occupies less than 0.2 percent physical memory because the size of *owner* (4 bits) and *GFN* (less than 52 bits) for each page (4K) is less than 8 bytes. With hierarchical structures (like page tables), the *Ownership-Table* can be even smaller. Second, MMU has to check the *Ownership-Table* during address translation. But the check results can also be cached in some TLB-like structures of MMU, which can hide the check latency. Since Intel SGX also adds similar checks (i.e., checking EPCM) in its MMU, we believe our proposals are practical.

4 Implementation

We add about 900 SLOC in glibc-2.26 for dispatching the system calls. We implement an Enclavisor prototype by tailoring and retrofitting sv6 [33]. The prototype (6,500 SLOC) has three major components: (i) threads management for creating/scheduling/destroying enclave threads, (ii) memory management for managing both secure private and shared memory, and (iii) an in-memory file system (FS) for temporary secure files. Enclavisor does not contain any device drivers for performing I/O because most system calls including I/O related ones are redirected to the guest OS in the normal VM (see Section 3.4).

Enclavisor implements enclave fork with traditional copy-on-write mechanism. When an enclave invokes fork, our system will not only fork an enclave on Enclavisor but also fork its host application in the normal VM. When an enclave creates a new thread, our system also creates the daemon thread by default. Enclavisor handles enclave exceptions such as page faults by itself and only relies on the timer interrupt (i.e., a virtual interrupt injected by the hypervisor) for scheduling. Since the responsibilities of Enclavisor are clear and limited, it should be feasible to build Enclavisor over a formally verified OS kernel [34] and further verify the entire Enclavisor with more effort. We also add about 500 SLOC in the guest OS kernel module and the hypervisor (KVM module) for inter-VM communication. Referring to [30], [31], We also give a prototype implementation (1,500 SLOC) of the sibling protection mentioned in Section 3.6 in Linux/KVM (hypervisor).

In our system, a secure VM can serve multiple normal VMs and a normal VM can also construct enclaves in different secure VMs. Since the hardware allows 15 secure VMs to run concurrently, it is possible to deploy Enclavisor with different security/functionality configurations in different VMs, such as supporting FS or not. Enclavisor makes the enclave environments highly extensible and thus makes it easier to meet different desired features for different scenarios. Currently, Enclavisor supports enclave forking and shared/grant memory between enclaves, which are hard to achieve in no matter SGX enclaves or SEV VMs.

5 Security Analysis

Enclavisor plays a similar role with hardware firmware (e.g., AMD PSP), which is within the TCB of the whole system. If Enclavisor has bugs and is compromised by an attacker, then the system is compromised, which is the same as the cases if the SGX microcode and AMD PSP have bugs and get compromised. It makes sense to assume Enclavisor is trusted because it only provides simple and clear functionalities and has a small code base (i.e., 6,500 SLOC). Trusting a small software component is also a common assumption (e.g., TrustVisor [15], CloudVisor [1], and Nested Kernel [30]).

Compromised Guest OS. Our system separates an enclave and its guest OS into a secure VM and a normal VM. Even if the guest OS is compromised, it cannot directly access the enclave's secure memory or its execution context which is maintained by Enclavisor. We also consider indirect attacks like *Iago attacks* [29]. To defend against memory-based Iago attacks, our dispatcher routes all the memory-related system calls to the trusted Enclavisor. For other types of Iago attacks, it is convenient to adopt orthogonal solutions [2] into our system.

However, the separation design exposes the system call trace of an enclave, which may be utilized by attackers as a *side channel*. A possible countermeasure is hiding information from the trace, which can be done by either the programmers (manual efforts) or our dispatcher (automatically adding noises).

Compromised Hypervisor. In our design, multiple enclaves share a secure VM, which means that they also *share the same memory encryption key*. Although a malicious enclave cannot directly attack other enclaves since Enclavisor guarantees the isolation, it may collude with a malicious hypervisor to issue new attacks. Specifically, the hypervisor may carefully craft some pages with malicious code or fake data (step-1) and send them to the malicious enclave to get these pages encrypted by the shared memory encryption key (step-2). Then the hypervisor can change the NPT of a victim enclave to remap its virtual pages to these malicious physical pages (step-3) to inject the malicious code and data. In step-2, the hypervisor can easily utilize NPT faults to figure out which physical page maps to which virtual page.

Our system defends against this kind of *colluding attack* by either the software solution or the hardware proposal specified in Section 3.6. In brief, the software mechanism can detect and refuse such malicious mapping operations in step-3. The enhanced hardware encrypts memory with GPA, which makes the above mapping attacks meaningless.

Similarly, recent attacks targeting SEV [8], [9], [27], [28] will also fail on our system since they requires the hypervisor to either replace the physical memory or modify the NPT mappings, which can be classified into *rollback attacks* or *NPT mapping attacks* and are discussed in Section 3.6.

Other Attacks. We do not defend against physical attacks to break the integrity of secure memory. For example, a physical rollback attack requires to record memory transactions first and then replay them directly on the hardware. We argue that such attacks are hard to issue in the cloud environment.

Our design does not target side channel attacks. Nevertheless, it also does not introduce new side channels other than the channel of system call trace, which, however, can

TABLE 3
A Comparison on the Cost of a Round Trip Between an Enclave and its Guest OS

Interaction	Cycles
Native	174
Trap-based	> 5,000
Highway(user)	325
Highway(kernel)	170

TABLE 4
A Comparison on the Booting Performance of a *Hello-World* Program

Boot	Time
Native	398.8 us
Kata SEV	2656 ms
Enclave Creation	1057.9 us
Enclave Fork	365.9 us

be closed by our dispatcher as mentioned above. In recent years, it is a hot topic to propose different solutions to defend against various kinds of side channel attacks, which are orthogonal and can be applied to our work.

Future Commercial Hardware. The design of Enclavisor can make sure different enclaves securely share the same memory encryption key in AMD SEV. Nevertheless, a similar hardware security extension, named Intel Multi-Key Total Memory Encryption (MKTME) [35], can provide multiple (up to 32K) memory encryption keys in a VM. Once MKTME is commercially available, Enclavisor can also be built atop it to provide fine-grained enclaves and leverage the multiple key support to simplify some designs.

We add SGX-assisted attestation as a practical solution to enable post-launch remote attestation which is not supported by SEV hardware yet. This mechanism has little effect on the software TCB since the attestation code is extremely simple. Meanwhile, it is optional and can be easily replaced once the flexible attestation is ready in SEV.

6 Evaluation

Our evaluation of Enclavisor on SEV hardware contains two parts: (i) we present some results from micro-benchmarks regarding the interaction performance and booting performance; (ii) we present real-world application benchmarks to demonstrate the efficiency of Enclavisor. All experiments use an AMD EPYC 7281 CPU with 32 cores and we fix the CPU frequency to 2.1 GHz. The host machine has 96 GB DRAM and runs Linux/KVM 4.20. We assign 4 cores to the secure VM and 24 cores to the normal VM. Each VM has 32GB DRAM and runs Linux 4.18.1.

6.1 Micro-Benchmarks

Interaction Cost. We first measure the interaction cost between an enclave and its underlying guest OS since Enclavisor separates them into different VMs. We choose to test the cost of *getpid*[3] because it is extremely simple and thus can almost reflect the actual cost of a round trip between user-level and kernel-level. It only takes about 174 cycles in a normal VM. Note that the result is the average latency over one million times invocation of *getpid* and KPTI is by default disabled on AMD CPUs which are not affected by Meltdown. In contrast, as shown in Table 3, one round trip of *trap-based* communication takes at least 5,000 cycles even in the ideal case. This cost consists of: one round trip between a normal VM and the hypervisor takes about 1812 cycles; one round trip between a secure VM and the hypervisor takes about 2040 cycles. Actually, the total cost can become larger when considering the uncontrollable scheduling cost.

Nevertheless, Enclavisor also supports polling-based communication as depicted in Fig. 7. When the daemon thread is polling in the host application (i.e., user-level), a *getpid* system call takes about 325 cycles which consists of (i) the "round trip" between the daemon thread and the enclave thread together with (ii) the round trip between the daemon thread and the guest OS (i.e., a native system call). Furthermore, Enclavisor also allows the daemon thread polling in the kernel space, which can eliminate the context switch between the guest OS and the daemon thread.

Booting Performance (shown in Table 4). It takes about 398.8 us to launch a simple *Hello-World* program in the normal VM. However, when using an SEV-secured Kata container [25] to launch the same application, it requires 2656 ms. Since the container is protected in a secure VM, this long boot period can be divided into two stages. The first stage is the VM creation, which requires the involvement of the secure co-processor and takes about 767 ms. The second stage is the container/VM initialization, which is from the VM start point to the main function of the application and costs about 1889 ms. Both of these two stages can bring substantial overhead to a short-term program like a serverless function.

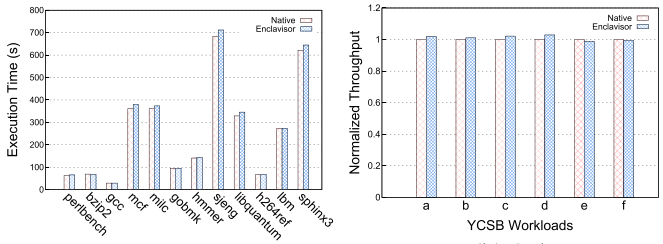
To create an enclave on Enclavisor, the host application first loads the enclave into the shared buffer between the normal VM and the secure VM, and then lets the guest OS send a request of creating enclave to the Enclavisor. When receiving the request, Enclavisor creates the enclave accordingly. We also enable the guest OS to inform the Enclavisor through an inter-VM Highway to avoid the potentially high cost of trap-based communication. It takes about 1 ms (verifying hash takes 587 us) to create a simple *Hello-World* enclave on Enclavisor.

Enclavisor also supports enclave forking which duplicates the host application, makes a copy of the enclave, and establishes the corresponding shared memory. Forking an enclave is usually faster than directly creating an enclave since the former needs not to load and transfer the enclave image and can avoid the initialization overhead, which is a desired feature for serverless computing. Here, we measure the time between the fork point and the point at which forked enclave starts to run as the cost of forking the simple enclave. The cost is about 365.9 us.

6.2 Application Benchmarks

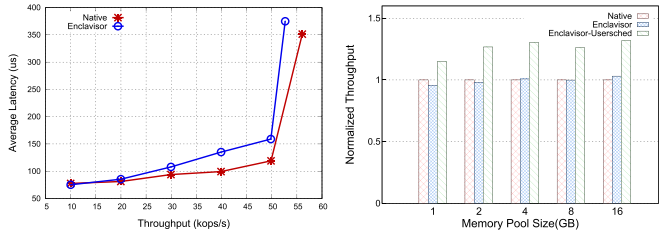
We compare the performance of applications running on Enclavisor with their native variants, i.e., running in a normal VM. Without explicit statement, we use the enclave-to-application Highway (user) in the following evaluations. Note that we do not modify any lines of codes in the evaluated applications and only re-link them with our modified libc.

3. We use glibc-2.28 to evaluate getpid on native Linux. The getpid library call in glibc-2.28 always triggers the corresponding system call.



(a) SPEC CPU 2006. (b) Sqlite3.

Fig. 9. (a) Execution time of 12 applications in SPEC CPU 2006 benchmarks. (b) Sqlite3 is evaluated against 6 YCSB workloads.



(a) Redis. (b) Memcached.

Fig. 10. (a) The performance comparison on Redis. (b) The throughput comparison on Memcached.

SPEC CPU 2006. We measure the execution time of 12 applications in SPEC CPU 2006 benchmarks and present the results in Fig. 9a. For CPU-intensive benchmarks such as *bzip2* and *gobmk*, the performance of enclaves running on Enclavisor is nearly the same as or even better than the native execution performance. Two reasons can explain: first, Enclavisor will not bring overhead to enclave applications when they do not invoke outside-enclave functions (e.g., system calls); second, when executing system calls, the Highway design avoids the context switches for the enclave threads although it requires some extra cycles. Context switches between user-mode (ring 3) and kernel-mode (ring 0) may incur indirect costs like TLB/cache pollution. Other applications show less than 5 percent overhead which mainly comes from the memory copies during system calls. Some of them (*mcf*, *sjeng*, and *sphinx3*) are more system call intensive and most system calls (66 percent-95 percent) are *read* or *write*, which leads to more accesses to the shared memory and more memory copies. The other two (*libquantum* and *milc*) issue more *mmap* system calls on files, which also leads to more memory copies (i.e., copy the file content from the normal VM into the secure VM).

Sqlite3. Fig. 9b shows the normalized throughput of SQLite 3.0 (Sqlite3) under different YCSB workloads. In this experiment, we directly generate the workload in the execution thread and use Highway (kernel) for invoking system calls. Also, we set the object size to 16 bytes to decrease the memory copy overhead. In such setting, Enclavisor can make the enclave application have comparable (or even slightly better) performance with the native one. A breakdown of this evaluation shows that about 35 percent of the total execution time is spent on executing system calls. Thus, the Highway design, which brings better locality for the execution, should be the major reason why Enclavisor shows close-to-native (or even better) performance.

Redis. We run Redis 5.0.4 with persistency disabled and use YCSB workload *a* (50 percent read and 50 percent update operations) to evaluate the performance. Specifically, we set the client number to 20 and run the clients in the same normal VM with the daemon thread and the native application to avoid the interference of going through the real network devices. The Redis server will be loaded 1 million records during the initial phase and the record size is 1KB (1GB in total). Then, we keep increasing the frequency of request sending (i.e., the target parameter in YCSB) for the clients and test the corresponding throughput and latency in the meantime.

Fig. 10a presents the experiment results. When the allowed latency is 135 us, the throughput of enclave Redis is about 80 percent of the native one. The overhead mainly comes from the extra data copy between the enclave thread and the daemon thread. Since the record size is 1KB and each operation reads/updates a record, the memory copy overhead is obvious, especially there is no actual network round trip. For example, Enclavisor can introduce 21 percent overhead for writing 1K data to a socket in such setting. The performance trends in other YCSB workloads are similar.

Memcached. We evaluate Memcached 1.5.16 with YCSB workload *a* (other workloads show similar results). The Memcached server runs with 4 physical CPUs and we use the same client configuration as the that in Redis. We compare the throughput of Memcached on Enclavisor with the native one under different memory pool size. Fig. 10b shows that the enclave application has comparable performance with the native one, which indicates that Enclavisor has the potential to support big memory processing. The memory copy overhead is not obvious as before since we add TLS encryption which is much more expensive than the copy operations in this experiment. When enabling user-mode scheduling in this case (run 8 server threads on the 4 CPUs), Enclavisor-Usersched can further achieve better performance than native no matter the latter one runs 4/8 server threads on the 4 CPUs. This is because user-mode scheduling can effectively improve the CPU utilization. Specifically, Enclavisor-Usersched schedules a physical CPU to execute runnable threads instead of wasting time on waiting for I/O events.

7 Related Work

There are many systems aiming at protecting tenants' private code/data in cloud where many security threats exist. Systems like TrustVisor [15] and CloudVisor [1] leverage trusted hypervisors or nested virtualization to provide secure environment for a whole virtual machine. Though they are efficient in guest VM protection, they cannot defend against intra-domain attacks. Systems like InkTag [2] utilize a trustworthy hypervisor to protect an application from a compromised guest OS. However, they need to modify and trust the hypervisor and it is expensive to frequently intercept transitions between applications and guest OS. Some other systems such as Bastion [14] make a further step to defend against physical attacks but require unavailable hardware. Besides efforts from the research community, Intel published a promising hardware security technology named SGX for enclave computing which is immune to both malicious privileged software and physical attacks. Thus, many studies [5], [16], [17], [18], [19], [20] use Intel SGX for protecting sensitive codes and data. However, the performance problem

impedes the usage of SGX in many scenarios. Although prior work [16], [17], [18], [20], [36] makes efforts to improve the performance of SGX-protected applications, the fundamental problem of limited memory cannot be solved. Especially, the memory problem will become much severer when multiple VMs/enclaves share 128/256 MB memory in a cloud server.

One of our major contributions is the decoupling of security enforcement, which embraces both hardware efficiency (e.g., memory encryption) and software flexibility (e.g., enclave management). In our design, Enclavisor only has limited and fixed functionalities and serves more like a firmware (e.g., PSP in AMD's SEV or Intel's ME), which is much more lightweight than a traditional OS/hypervisor. Our design also achieves close-to-native performance and good compatibility with commercial cloud stack.

Existing attacks [8], [9], [27], [28], [37] targeting SEV secure VMs require either writing the VM's secure memory or manipulating the VM's NPT. However, these two requisites are both impossible when our software or hardware defenses (Section 3.6) are exploited. Recently, AMD proposes new extensions named SEV-SNP (concurrent work) for mitigating integrity problems of SEV but still provides limited secure VMs as its enclaves [38]. Therefore, even for future SEV machines, Enclavisor is still meaningful and actually easier to deploy (without a trusted component in the hypervisor).

The system call dispatch mechanism in our system shares some similarities with prior work [39] and their dispatch policies or tools for generating glue code can be incorporated into our system. Our Highway design is inspired by [13], [18], [36] but is used for accelerating cross-VM communication.

Other Secure Enclaves. Since current SGX-capable CPUs can only be used on single-socket platforms, Intel provides SGX Card [40] to make the SGX technology available on existing dual-socket server platforms. An SGX Card can be attached to one host server over a PCI Express (PCIe) interface. Each card contains three independent Intel Xeon E3 CPUs inside (each CPU provides 128 MB secure memory) and several such cards can be attached to the same host. With Intel SGX Cards, an application can be scaled out by running multiple enclaves: execution and data are partitioned across different enclaves on different cards. Thus, with one card, the total size of the secure memory increases by a factor of 3, with two cards by a factor of 6, and so on. Nevertheless, for a single enclave, the secure memory limitation is not changed. Moreover, SGX Card does not benefit enclave booting or the cross-enclave interaction.

ARM platforms are widely used in mobile and embedded systems. ARM TrustZone technology can divide the whole platform into the normal world and the secure world (as a single TEE for the whole system). There are two kinds of studies on how to multiplex the single secure world: 1) deploying a secure OS in the secure world and letting the secure OS run multiple mutually-isolated secure applications (e.g., OP-TEE [41]); 2) providing multiple virtual secure worlds in a single platform (e.g., our prior work, vTZ [42]). Enclavisor makes an attempt to improve AMD-SEV while it may also be extended to the work on TrustZone.

There are also some open-sourced enclaves (e.g., Sanctum [43], Keystone [44], and our prior work called Penglai [45]) which promise different attracting features. Nevertheless, they are still not available on today's commodity machines.

8 Summary

We propose Enclavisor, a hardware-software co-design, for building enclaves on the untrusted cloud. Through combining the flexibility of software and the efficient security guarantee of hardware, Enclavisor can achieve high-security guarantee, large memory support and high run/boot time performance for typical cloud applications. We implement Enclavisor prototype on AMD SEV and tackle some specific challenges. The evaluation results demonstrate that Enclavisor-protected enclaves can have near-native performance.

Acknowledgments

The authors would like to sincerely thank the anonymous reviewers for their insightful suggestions. This work was supported in part by the National Key Research & Development Program under Grant 2016YFB1000104, in part by the National Natural Science Foundation of China under Grant 61972244, Grand U19A2060, and Grant 61925206, in part by the HighTech Support Program from Shanghai Committee of Science and Technology under Grand 19511121100, and in part by the Program of Shanghai Academic/Technology Research Leader under Grand19XD1401700.

References

[1] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 203–216.

[2] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proc. 18th Int. Conf. Architect. Support Program. Languages Operating Syst.*, 2013, pp. 265–278.

[3] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proc. 19th Int. Conf. Architect. Support Program. Languages Operating Syst.*, 2014, pp. 81–96.

[4] Intel Corporation, "Software guard extensions programming reference, revision 2," 2014.

[5] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, 2015, Art. no. 8.

[6] Intel, Intel Software Guard Extensions Developer Guide, 2016. [Online]. Available: https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf

[7] AMD, AMD Secure Encrypted Virtualization, 2019. [Online]. Available: https://developer.amd.com/sev/

[8] F. HetzeltandR. Buhren,"Security analysis of encrypted virtual machines," *ACM SIGPLAN Notices*, vol. 52, no. 7, pp. 129–142, 2017.

[9] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected I/O operations in AMD's secure encrypted virtualization," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1257–1272.

[10] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 645–650.

[11] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 39.

[12] E. Oakes, *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 57–70.

[13] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 33–46.

[14] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proc. 16th Int. Symp. High-Perform. Comput. Architecture*, 2010, pp. 1–12.

[15] J. M. McCune, *et al.*, "Trustvisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 143–158.

[16] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with SGX," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 14.

[17] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 264–278.
[18] S. Arnautov *et al.*, "Scone: Secure linux containers with intel SGX," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 689–703.
[19] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 1–32.
[20] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS services for SGX enclaves," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 238–253.
[21] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: Towards secure remote execution in FaaS," in *Proc. 12th ACM Int. Conf. Syst. Storage*, 2019, pp. 44–54.
[22] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, 2016, Art. no. 86.
[23] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in SGX with efficient integrity verification structures," in *Proc. 23rd Int. Conf. Architect. Support Program. Languages Operating Syst.*, vol. 53, no. 2, 2018, pp. 665–678.
[24] Linux-KVM, Encrypted virtual machine supports, 2018. [Online]. Available: https://github.com/AMDESE/AMDSEV
[25] Kata-Container, SEV runtime for Kata Containers, 2018. [Online]. Available: https://github.com/AMDESE/AMDSEV/tree/kata
[26] D. Du *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architect. Support Program. Languages Operating Syst.*, 2020, pp. 467–481.
[27] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting AMD's virtual machine encryption," in *Proc. 11th Eur. Workshop Syst. Secur.*, 2018, Art. no. 1.
[28] M. Morbitzer, M. Huber, and J. Horsch, "Extracting secrets from encrypted virtual machines," in *Proc. 9th ACM Conf. Data Appl. Secur. Privacy*, 2019, pp. 221–230.
[29] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proc. Int. Conf. Architect. Support Program. Languages Operating Syst.*, 2013, pp. 253–264.
[30] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve,"Nested kernel: An operating system architecture for intra-kernel privilege separation," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 191–206, 2015.
[31] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan, "Comprehensive VM protection against untrusted hypervisor through retrofitted amd memory encryption," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 441–453.
[32] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 380–395.
[33] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. W. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 1–17.
[34] L. Nelson, *et al.*, "Hyperkernel: Push-button verification of an OS kernel," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 252–269.
[35] "Intel corp, intel memory encryption technologies," 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf
[36] O. Weisse, V. Bertacco, and T. Austin,"Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," *ACM SIGARCH Comput. Architecture News*, vol. 45, no. 2, pp. 81–93, 2017.
[37] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "Sevurity: No security without integrity - breaking integrity-free memory encryption with minimal assumptions," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1746–1759.
[38] "SEV secure nested paging firmware ABI specification," 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/56860.pdf
[39] N. Santos, H. Raj, S. Saroiu, andA. Wolman,"Using ARM trustzone to build a trusted language runtime for mobile applications," *ACM SIGARCH Comput. Architecture News*, vol. 42, no. 1, pp. 67–80, 2014.
[40] S. Chakrabarti, M. Hoekstra, D. Kuvaiskii, and M. Vij, "Scaling intel® software guard extensions applications with intel® SGX card," in *Proc. 8th Int. Workshop Hardware Architect. Support Secur. Privacy*, 2019, pp. 1–9.
[41] "Op-tee," 2019. [Online]. Available: https://optee.readthedocs.io/en/latest/
[42] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "VTZ: Virtualizing ARM trustzone," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 541–556.
[43] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 857–874.
[44] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
[45] "Penglai enclave: Open-sourced secure and scalable TEE system for RISC-V," 2019. [Online]. Available: http://penglai-enclave.systems/

Jinyu Gu received the BS degree in software engineering from Shanghai Jiao Tong University, China, in 2016. He is currently working toward the PhD degree with the School of Software, Shanghai Jiao Tong University. His research interests include operating systems, computer architecture, and security.

Xinyue Wu received the BS degree in software engineering from Shanghai Jiao Tong University, China, in 2019. She is currently working toward the master's degree with the School of Software, Shanghai Jiao Tong University. Her research interests include operating systems and computer architecture.

Bojun Zhu received the BS degree in software engineering from Shanghai Jiao Tong University, China, in 2019. He is currently working toward the master's degree with the School of Software, Shanghai Jiao Tong University. His research interests include computer architecture and security.

Yubin Xia received the diploma degree from Software School, Fudan University, Shanghai, China, in 2004, and the PhD degree in computer science and technology from Peking University, Beijing, China, in 2010. He is currently an associate professor with Shanghai Jiao Tong University. His research interests include computer architecture, operating system, virtualization, and security.

Binyu Zang received the PhD degree in computer science from Fudan University, in 2000. He is currently a tenured full professor with the Shanghai Jiao Tong University, is the director with the School of Software. His research interests include systems software and security.

Haibing Guan received the PhD degree from Tongji University, in 1999. He is a professor with the School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University, and the director with the Shanghai Key Laboratory of Scalable Computing and Systems. His research interests include cloud computing, distributed computing, and virtualization technology.

Haibo Chen (Senior Member, IEEE) received the PhD degree in computer science from Fudan University, in 2009. He is currently a tenured full professor with the School of Software, Shanghai Jiao Tong University. His research interests include operating systems, system security, and computer architecture. He receives the title of ACM distinguished scientist.