Comprehensive VM Protection against Untrusted Hypervisor through Retrofitted AMD Memory Encryption

Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen*, Binyu Zang, Haibing Guan
Institute of Parallel and Distributed Systems
Shanghai Key Laboratory for Scalable Computing and Systems
Shanghai Jiao Tong University

ABSTRACT

The confidentiality of tenant's data is confronted with high risk when facing hardware attacks and privileged malicious software. Hardware-based memory encryption is one of the promising means to provide strong guarantees of data security. Recently AMD has proposed its new memory encryption hardware called SME and SEV, which can selectively encrypt memory regions in a fine-grained manner, e.g., by setting the C-bits in the page table entries. More importantly, SEV further supports encrypted virtual machines. This, intuitively, has provided a new opportunity to protect data confidentiality in guest VMs against an untrusted hypervisor in the cloud environment. In this paper, we first provide a security analysis on the (in)security of SEV and uncover a set of security issues of using SEV as a means to defend against an untrusted hypervisor. Based on the study, we then propose a software-based extension to the SEV feature, namely Fidelius, to address those issues while retaining performance efficiency. Fidelius separates the management of critical resources from service provisioning and revokes the permissions of accessing specific resources from the untrusted hypervisor. By adopting a sibling-based protection mechanism with non-bypassable memory isolation, Fidelius embraces both security and efficiency, as it introduces no new layer of abstraction. Meanwhile, Fidelius reuses the SEV API to provide a full VM life-cycle protection, including two sets of para-virtualized I/O interfaces to encode the I/O data, which is not considered in the SEV hardware design. A detailed and quantitative security analysis shows its effectiveness in protecting tenant's data from a variety of attack surfaces, and the performance evaluation confirms the performance efficiency of Fidelius.

1. INTRODUCTION

One of the primary premises of current multi-tenant clouds is to guarantee the confidentiality of the tenant's data, even when facing curious or malicious insiders. It is a well-known fact that purely encrypting private data in the secondary storage is far from enough since such data will be finally loaded into memory in plaintext for processing. Such data will be exposed to the underlying privileged software and hardware attacks like cold boot or bus snooping.

One promising direction of further protecting in-memory processed data is to use relatively efficient hardware-based memory encryption so that even though the memory is somehow accessed by the attackers, its confidentiality is still guaranteed. Recently, Intel and AMD have both proposed hardware supports for memory encryption within their manufactured processors. Intel proposed the Software Guard Extensions (SGX) [1], to protect pieces of application logic inside encrypted enclave memory against malicious OS. However SGX is limited to protect a relatively small portion of memory, and the developers have to mostly reconstruct the protected software or build it from scratch. Thus it is nontrivial for SGX to protect large-scale software like the operating system or even the entire virtual machine. AMD proposed another simpler mechanism called Secure Memory Encryption (SME), with which enabled, the memory can be encrypted in page level granularity by simply setting the C-bit in the page table entry. More importantly, it enables the Secure Encrypted Virtualization (SEV) feature, so that each virtual machine can use its own key to selectively encrypt memory, which provides an opportunity to guarantee the data security of guest VM against untrusted hypervisor.

However, we observed that there are numerous issues when leveraging SEV for VM protection against an untrusted hypervisor. First, a malicious hypervisor can bypass the protection by manipulating some critical resources. For example, the SEV does not encrypt the virtual machine control block (*VMCB*) and general purpose registers, so that hypervisor can easily breach their privacy and integrity, and arbitrarily disable corresponding policies [2]. Though the optional SEV-ES feature can eliminate the above attack surfaces by encrypting these guest states, the hypervisor is still in the position of managing the guest memory mapping and key sharing mechanisms, where replay attacks can be conducted to infer VM's encrypted memory, and the shared keys may be abused by the hypervisor with other collusive VM. On the other hand, we observed that SEV misses some functionalities to provide the full VM protection. For instance, SEV cannot encrypt I/O based (DMA) memory regions, which can be directly accessed by the untrusted driver domain. In addition, given SEV, there is currently no secure way for two cooperative guest VMs to share memory without exposing it to the hypervisor.

In this paper, we propose a software-based extension to AMD SEV, namely Fidelius, to enable a comprehensive VM protection against the untrusted hypervisor (including the management VM). To address the security issue from bypassed protection by the hypervisor, Fidelius separates the critical resource management from the service provisioning, such that the hypervisor is deprived of the permissions to directly operate on specific resources. Fidelius introduces another isolated context to manage those resources with

*Corresponding author: haibochen@sjtu.edu.cn

policy enforcement, and this isolation is guaranteed by the non-bypassable memory protection mechanism. Meanwhile, for performance consideration, Fidelius builds this isolated context in the same privilege level as the hypervisor, and it provides three types of gates to enable lightweight context switch compared with the otherwise cross-world approaches.

To remedy the missed functions of SEV while retaining compatibility, Fidelius provides a novel reuse of the SEV API to enable the full VM life-cycle protection. Specifically, Fidelius retrofits the usage of SEV's *SEND* and *RECEIVE* APIs to enable booting from encrypted kernel image. For I/O protection, Fidelius provides two sets of para-virtualized interfaces for guest VM, so that it can enjoy fast I/O data encoding and decoding with hardware-based cryptographic acceleration while preserving compatibility for systems with or without AES instruction set. Finally, to provide a secure approach for memory sharing between cooperative guest VMs, Fidelius extends the hypercall interfaces to verify the sharing contexts between VMs.

Our contributions: To summarize, this paper makes the following contributions:

- A comprehensive security analysis and discussion of leveraging AMD SEV for VM protection against untrusted host system (Section 2).
- Fidelius, a software extension to address the issues of leveraging SEV for full VM protection by separating critical resource management from service provisioning with sibling based protection, and a novel reuse of the SEV API for VM life-cycle protection. (Section 3 and 4).
- A working prototype implemented on Xen, and policy enforcement based on it (Section 5), as well as its security (Section 6) and performance (Section 7) evaluations that confirm its effectiveness and efficiency, and some hardware suggestions for a better security solution (Section 8).

2. BACKGROUND AND MOTIVATION

The idea of protecting virtual machine under untrusted hypervisor with cryptographic support is not new, some previous wisdom (e.g., HyperCoffer [3]) has been proposed to involve in such scenario. However, compared with the secure processors used in those systems, the AMD SEV feature has some appealing characteristics. Firstly, SEV is implemented in the mainstream processors instead of the academic simulators. Thus it can be widely supported in reality. Meanwhile, other than encrypting the whole memory of the entire guest virtual machine, SEV chooses to grant guests the capabilities to control the encryption in a manner of page granularity. Therefore it is more flexible and can be adapted to more complicated scenarios. In this section, we will first introduce the background of SME and SEV proposed in the AMD processors, as well as its potential problems when enabling guest VM running on the untrusted hypervisor with SEV support to motivate our work. Then we will briefly show some background of Xen virtualization techniques.

2.1 AMD Memory Encryption

Recently AMD proposed its memory encryption solutions called SME and SEV [4] and integrated them into the AMD-V techniques. As indicated in Figure 1, a new SoC firmware is brought in, and it cooperates with the memory controller accommodated in CPU to serve both SME and SEV. All the

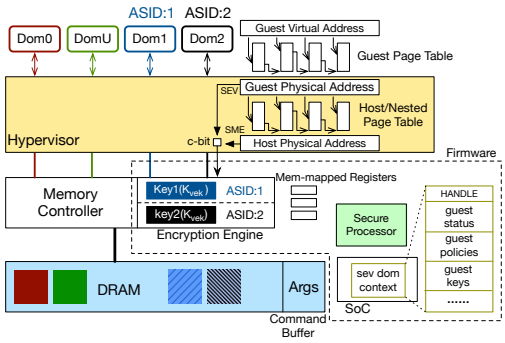

Figure 1: AMD memory encryption solutions

keys are managed by a secure processor and installed into the memory controller to support on-die AES encryption engine. The keys will be regenerated every time the host system resets in SME or guest reboots in SEV.

SME is used to defend against physical attacks. The kernel can decide which pages to encrypt by setting the C-bits in the page table entries. While SEV can further protect guest memory in a similar manner with the help of additional guest key management instructions. When a guest VM is launched in SEV mode, the firmware initializes the guest SEV context in SoC and returns a *handle* to the hypervisor. Each *handle* corresponds to one SEV context including the VM encryption key (K_{vek}). Since then when the guest VM is about to be scheduled, the hypervisor will first issue the *ACTIVATE* instruction, with parameters of *handle* and *ASID* of the guest VM, so that the processor will install the corresponding K_{vek} into the memory controller together with *ASID* tagged. AMD also proposed the SEV-ES [5] in its future version of SEV so that the *VMCB* and general purpose registers can be encrypted when trapped into the hypervisor, to prevent attackers from tampering them.

As shown in Figure 1, guest dom1 and dom2 can encrypt their memory with different K_{vek}, by setting the C-bits in the guest page table entries. It is important to note that SEV can be used together with SME, but C-bits in the guest page tables take priority.

2.2 (In)securities of AMD Memory Encryption

It is noted that though the protected guest VM has been endowed with the capability of sheltering its memory from an untrusted host system, the management role of hypervisor weakens this power for following potential reasons.

Problems existing in SEV. The *VMCB* contains critical information (e.g., instruction pointer, control registers), and it controls the execution, as well as exit and entry conditions of the guest VM through specific control vectors. However, the *VMCB* is not encrypted, and its integrity is not protected in the current SEV. Besides, the general purpose registers are also exposed when control is trapped. Once context switch happens from guest mode to host mode, the hypervisor is capable of stealing confidential information from the unencrypted registers and tampering critical fields in *VMCB*. This can lead to arbitrary guest memory reads and writes or even disable SEV protection completely [2].

Remaining problems even with SEV-ES enabled. Although SEV-ES can disallow the above-mentioned attack surfaces, there are still at least two potential weaknesses.

First, the second level memory mapping is still controlled by the hypervisor to map from guest physical address to host physical address. This is shown to be vulnerable to replay attacks for bypassing password validation [2] even with SEV-ES enabled. Second, a key-sharing enabled guest can still not prevent hypervisor from exposing its K_{vek} to a collusive guest VM since *handle* and *ASID* are managed by the hypervisor, and this *handle-ASID* relationship is not protected by SEV-ES.

Security issues not considered by AMD memory encryption. It is noted that DMA is not allowed to operate on encrypted guest memory by the SEV hardware for security reasons. Thus guest should allocate shared memory pages unencrypted for DMA, which may break the confidentiality of I/O data. Also, since guest VMs have their own cryptographic keys, the shared memory between two guests has to be in plaintext, which is exposed to the underlying hypervisor. Meanwhile, since the grant table is maintained by the hypervisor, it can intentionally manipulate the grant references (including the access permissions), and map the shared memory to its conspirator VM, or abuse the permission systems. For example, the hypervisor can tamper the permission to writable, while the origin VM shares its memory with only read permission. It is noted that the I/O data transferring highly relies on the memory sharing mechanism. Thus it reveals high risk for the I/O guest VM.

2.3 Xen Virtualization

Xen [6] was famous for its para-virtualized interfaces that guest VM should be explicitly ported to communicate with virtualization layer during privileged operations, memory shadowing and I/O transfer. As Intel VT-x and AMD-V virtualization hardware extensions came into the market. Currently, CPU and memory virtualizations are widely and efficiently supported by hardware, while for I/O virtualization, the para-virtualized approaches with front-end and back-end drivers are still popular for its efficiency. We take AMD-V as an example to explain CPU, memory and I/O virtualization implemented in Xen environment, as well as its memory sharing mechanisms.

CPU virtualization. Guest VM normally runs in the guest mode, while the hypervisor and management VM run in the host mode. Both of the two modes have ring 0 and ring 3 privilege levels for kernel and user processes to run respectively. Before guest VM bootup, hypervisor initializes data structures called virtual machine control block (*VMCB*) for every virtual core, and each *VMCB* contains the runtime states of the corresponding guest, as well as the control bits for VM entry and exit. Once the *VMCB* and the general purpose registers are initialized, the hypervisor can issue the *VMRUN* instruction, taking the address of the *VMCB* as its argument, then the CPU is switched to the guest mode, and its states are synchronized with the *VMCB* fields. During running cycles of guest mode, the privileged instructions and events pre-configured in the *VMCB* will trap the control flow into the host mode, and then the hypervisor can handle them according to the vmexit reasons.

Nested paging. AMD introduces a nested layer for memory virtualization. For each guest VM, the hypervisor maintains one nested page table (NPT) referenced by a specific hardware register, to translate from guest physical address (GPA) to host physical address (HPA), while the guest governs its own page tables referenced by CR3 registers, to map from guest virtual address (GVA) to GPA. One complete memory read involves two steps of hardware-based addressing, firstly, the GPA is retrieved from the guest page tables, then the real HPA is further obtained through the nested page tables walking. When the nested page faults happen, the hypervisor is responsible for allocating physical pages and fill the addresses to the corresponding NPT entries.

Para-virtualized I/O. In Xen the para-virtualized I/O interface is widely used, it can outperform the emulated I/O interface as the transferred data are batched instead of trapping every I/O operation. Specifically, the whole I/O process is divided into front-end and back-end, initially the front-end driver in the guest VM first establishes a shared memory buffer with the back-end driver through the persistent grant table mechanism. During I/O write, the front-end driver copies data to this memory buffer, and informs the back-end driver through the event channel mechanism; afterward, the latter does the actual I/O write. The reverse path is similar to the I/O read.

Memory sharing in Xen. Memory sharing is common in Xen, it is realized by the underlying grant table mechanism. Specifically, once a VM is about to share memory regions, it first offers the memory pages by creating a grant through related hypercalls, with the information of the domain id, flags, and the number of page frames. Hypervisor fills these pieces of information in the grant table. For the other end of the guest VM, it takes the grant reference from the *XenStore* and maps the shared pages to its own address space, which needs to be first validated by the hypervisor.

3. OVERVIEW

3.1 Approaches Overview

Efficiently separating resource management from service provision. We observed that the main obstacle to providing sufficient VM protection under untrusted hypervisor is the mixing of resource management and service provision in the hypervisor. Normally hypervisor plays the role who not only takes over control when specific events occur to provide services for guest VM but also manages nearly all resources including critical ones. The basic idea is to separate these two functionalities, that hypervisor is still responsible for serving guest VM like interrupt handling, scheduling, etc., while the permissions of accessing specific resources are revoked from it.

Meanwhile, to enable a secure but also efficient approach, instead of completely restricting hypervisor from accessing critical resources, we adopt another idea of separating resource accessing from policy enforcement. Specifically, in our scenario, Fidelius represents a trusted context controlling the exit and entry boundaries between the guest and host modes. For resources that are compactly organized and may frequently be accessed by the hypervisor (e.g., *VMCB*), whenever guest VM exits to host mode, Fidelius first shadows those resources, and hides the confidential fields before passing the control to the hypervisor. Thus during the hypervisor running, it is permitted to read or write those resources, while the privacy is still reserved and malicious tampering can be detected during the policy enforcement phase before entering back to the guest mode in the trusted context. For sparsely organized resources (e.g., page tables), they are mapped as read-only in the hypervisor, and any

modification to them requires the trusted context to involve in for policy enforcement. For other resources having nothing to do with the service provision (e.g., SEV metadata), they are simply unmapped in the hypervisor's address space.

Sibling protection with non-bypassable memory isolation. Since hypervisor lies in the most privileged software layer, theoretically it can break any software based isolation. One possible solution is to introduce another lower layer of abstraction, e.g., nested virtualization [7] to enforce the isolation. However, this approach incurs much overhead due to frequent cross-layer switches. Therefore, we opt for the sibling protection mechanisms [8, 9], that the hypervisor and the Fidelius lie in the same privilege level, but remains separated through the non-bypassable memory isolation mechanism [10]. Specifically, the capabilities of arbitrarily mapping memory are revoked from the hypervisor by mapping its page-table-pages as read-only, and eliminating all related privileged instructions (e.g., modify *CR0* to disable paging, or replace *CR3* to reuse other page tables) out of the hypervisor's code. Therefore Fidelius needs to involve in for the normal page table manipulation and validate its updating with page information table (PIT) based policy enforcement.

Novel reuse of SEV API for full VM life-cycle protection. Currently, SEV does not support full VM protection due to the missing of some functionalities, including not supporting I/O encryption. By retrofitting the usage of SEV's *SEND* and *RECEIVE* APIs, Fidelius enables securely booting guest VM from encrypted kernel image. Meanwhile, it provides two sets of para-virtualized interfaces with hardware-based cryptographic acceleration for guest VM to protect its own disk I/O data. Specifically, for processors with AES-NI hardware support, guest VM can encrypt its block I/O data by directly executing the AES instruction set. Otherwise, Fidelius provides another mechanism that innovatively reuses the SEV API to enable hardware-based I/O encryption and decryption. Besides, Fidelius ensures that the VM migration and memory sharing are all protected during its life-cycle.

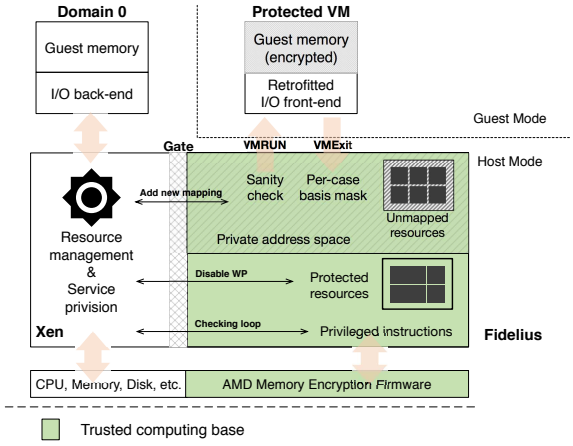

Figure 2: Architecture overview of Fidelius

Architecture overview. Figure 2 presents the architecture of Fidelius. Its trust computing base includes the AMD memory encryption engines and the Fidelius context lying in parallel with the hypervisor, but they are isolated from each other through non-bypassable memory isolation mechanism. Meanwhile, to separate resource accessing from policy enforcement, and retain most of the compatibilities, the hypervisor is deprived of the permissions of accessing critical resources through three categories of approaches: shadowing (*VMCB* and registers), write-protecting (memory mapping structures and grant tables), and self-maintaining (SEV metadata and specific instructions). The accessing of them should go through three types of gates (disable WP, checking loop, add new mappings), where corresponding policies are enforced. It is noted that shadowing *VMCB* and registers can be regarded as a software version of SEV-ES, while others will solve the remaining issues. Finally, Fidelius controls the boundaries between guest and host modes and reuses the interfaces of AMD memory encryption engines and AES instruction sets to provide the full VM life-cycle protection.

3.2 Threat Model and Assumptions

Fidelius aims at protecting privacy and integrity of guest VM from untrusted host including hypervisor and management VM, as well as physical attacks like cold-boot or bus snooping. A benign hypervisor may be compromised and become untrusted. The untrusted hypervisor still handles service requests from the guest VM and is able to access any data it has permissions. The backend I/O paths are controlled by the management VM that any I/O data transferred to it are exposed. However, Fidelius makes no attempt to prevent following types of attacks. First, guest VMs may be compromised through the security flaws inside them, this kind of protection is out of the scope of this paper. Second, DoS attack and availability guarantee of the guest VM is not under consideration since they are not aimed at disclosing privacy data. Finally, Fidelius does not prevent against side-channel attacks, we believe that both of the encryption mechanism and the trusted software extension are irrelevant to data access pattern, and Fidelius will not bring in new side-channel attack surfaces.

4. Fidelius DESIGN

4.1 Non-bypassable Memory Isolation

Theoretically, kernel code can do anything, including arbitrarily mapping memory, enabling or disabling protection by configuring specific registers, etc. However, these capabilities can be restricted if the kernel page tables are controlled, and the protection mechanism is ensured to not be disabled. The most key premise of revoking permissions of accessing critical resources from the privileged hypervisor is to create an isolated context out of it. To avoid an additional layer of abstraction, Fidelius puts this isolated context in the same privileged layer as the hypervisor, i.e., the host kernel mode.

Table 1: Permissions and policies resources

Resources	Xen	Fidelius	Policies
Page tables (Xen)	Read-only	Writable	PIT based policy
NPT (guest VM)			
Grant tables			GIT based policy
Page info table			Xen not writable
Grant info table			
Guest states	Writable		Exit reasons based
Shadow states	No access		Xen not accessible
SEV metadata			

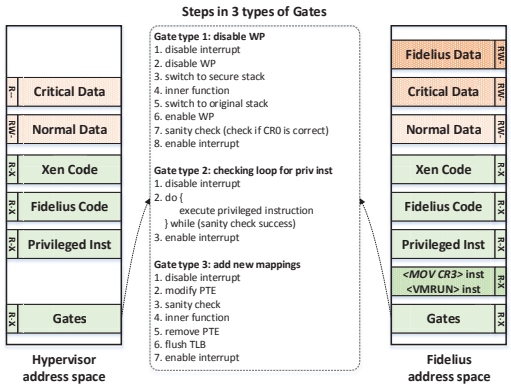

Figure 3: Memory layout of hypervisor and Fidelius.

4.1.1 Controlling memory mapping structures

Fidelius is in control of the memory mapping structures by write-protecting the page-table-pages of the hypervisor, so that every attempt to update the memory mapping will be forwarded to the Fidelius's handler, and pre-defined policies will be enforced. The overall memory layout of Xen and Fidelius is shown in Figure 3. Table 1 details the permissions mapped in Xen and Fidelius for different resources, as well as the enforced policies when hypervisor accesses them. For example, the *Fidelius data* representing the shadow guest states and the SEV metadata are unmapped from the Xen's context. While the *critical data* including the page-table-pages of the hypervisor and NPT of the protected guest VM, the grant tables, as well as PIT and GIT, are all mapped as read-only in Xen. On the other hand, for performance consideration, we map most of the code as executable in both Xen and Fidelius except some specific privileged instructions.

4.1.2 Restricting privileged instructions

In addition to restricting hypervisor from directly tampering its page-table-pages, another key point is to forbid hypervisor to execute specific privileged instructions which may hijack the control flow, switch the context, or even disable the protection mechanisms. Accordingly, Fidelius provides two options to restrict the execution of these instructions. For instructions that may disable protection mechanisms, we simply monopolize them, which means for each of these instructions, binary scanning is used to ensure that no such instruction, no matter aligned to instruction boundaries or not, exists in the code region, excepting the only one copy existing in the Fidelius's code. In this case, these instructions are still mapped as executable in the Xen's address space. To avoid attackers from directly executing the instruction through control flow hijacking attacks, we add checking loop logics (type 2 gate in Figure 3) around the instruction for sanity check with policies detailed in Table 2,

For instructions that may hijack the control flow (e.g., *VMRUN*), or directly switch the address space (e.g., *mov CR3*), Fidelius unmaps them from the Xen's address space and remaps them as executable before their execution. In this case, the sanity check is located between the remapping and the execution, to prevent attackers from directly jumping to these instructions. Moreover, we should give special treatments to the *mov CR3* instruction, since the execution of this instruction will directly switch to a new address space, where this instruction should not be mapped so that the next instruction cannot be executed. To resolve this problem, we elaborately place the *mov CR3* instruction in the last few bytes of the page and make sure the next page containing the subsequent instructions are mapped in all valid address spaces.

4.1.3 Securing transition between isolated contexts

When hypervisor needs to normally update its page tables, modify protected resources, or execute privileged instructions, it should first switch to the Fidelius's context. Generally, there are several approaches. The most common one is to change the *CR3* during the transition, which switches to a completely separated address space. However, this approach introduces significant overhead as it involves full TLB flush in the AMD architecture. The second approach is to temporarily add a pre-allocated address space during context transition, and withdraw them in the reverse path. This approach only involves one single memory write to the page-table-page, as well as one TLB entry flush. Therefore, we leverage it for those unmapped privileged instructions (*VMRUN* and *mov CR3*) and resources (the shadow guest states and SEV metadata). To further reduce the overhead, we adopt the third approach for the most common cases, that no additional address space is brought in. Instead, only the permissions of current address space mapping are modified during the transition. In this respect, we only need to clear the WP bit in *CR0*, so that the original read-only resources in Xen's space becomes writable within Fidelius's space.

As shown in Figure 3, we provide three types of gates to achieve the context transition. The first one is to disable the WP bit in *CR0*. Besides, it is also needed to disable interrupts, switch stacks, and do sanity checks after setting the WP bit. The second and third types of gates are used for privileged instructions and unmapped resources, as elaborated previously. Their main difference is the placement of the sanity check logic, and for the type 3 gate, specific TLB entries should be flushed for mapping freshness.

4.2 Resource Management

In Fidelius, we apply three categories of protection for the critical resources.

4.2.1 Shadowing guest runtime states

The current version of AMD SEV can only encrypt guest VM memory, while the release date of SEV-ES that can encrypt runtime states (e.g., registers and *VMCB*) is unknown. As demonstrated in [2], it leads to arbitrary memory reads and writes, as well as protection disabling. To protect the guest runtime states, one intuitive way is to hide the register values, and completely restrict hypervisor from accessing the *VMCB*. However, in most scenarios, hypervisor requires to read or even update them. Thus we adopt the shadowing approach, that Fidelius guards the boundaries between guest and host modes, upon exiting from the guest VM, Fidelius first shadows the registers and *VMCB* by copying their contents to a private memory region which is unmapped from the hypervisor's context, and masks the original contents except for some selective fields according to the exit reasons. Once the hypervisor finishes handling the VMExit and is about to return to the guest mode, Fidelius checks the integrity of the *VMCB* using the shadowed *VMCB*, and the registers are directly overwritten by the shadowed registers.

Table 2: Description of privileged instructions protected in Fidelius

Instruction	Description	Perm-Xen	Perm-Fidelius	Gate type	Policies
MOV CR0	May disable PG and WP	Executable	Executable	Type 2: checking loop	PG and WP bits cannot be cleared
MOV CR4	May disable SMEP				SMEP bit cannot be cleared
WRMSR	May disable NX				NXE bit in EFER cannot be cleared
VMRUN	May change the control flow	Inaccessible		Type 3: add new mapping	Specific *VMCB* fields cannot be tampered
MOV CR3	May switch address space				The target CR3 must be valid

4.2.2 Write-protecting memory mapping data

Since the NPT of the guest VM is maintained by the hypervisor, a malicious hypervisor can conduct replay attacks [2]. Thus in Fidelius, the NPT is mapped as read-only in the hypervisor's address space, any attempt of tampering will be trapped and detected in the Fidelius's fault handler. When hypervisor requires updating the NPT entries routinely, it has to go through the type 1 gate, where the subsequent updates of the NPT entries are checked by enforcing the PIT based policies (Section 5).

Also, the encryption of guest memory poses a great challenge of sharing its memory with others. Since each guest VM has its own encryption key, normally the shared memory should be in plaintext. Meanwhile, currently, the hypervisor is in the core path of managing the sharing information (grant table). Therefore, even if the hypervisor is forbidden to access the shared memory, it can still deceive the guest VM to share its memory with the accomplice VM with incorrect permissions. To secure the memory sharing, we map the grant table as read-only in the hypervisor, similarly, when the hypervisor is about to update it in the normal process, it first goes through the type 1 gate, then the grant table update is checked by enforcing the GIT based policies.

4.2.3 Self-maintaining SEV metadata

The final type of critical resources is the SEV metadata, including the ASID of the guest VM, the handle returned from the firmware, as well as that cryptographic metadata (guest public keys and nonce) used for guest launching. The SEV metadata is used in Fidelius, and it has nothing to do with the hypervisor's service provision, thus they are simply unmapped from the hypervisor, the only way to access them is through the type 3 gate.

4.3 Full VM Life-cycle Protection

4.3.1 System initialization

To reduce complexity, Fidelius adopts the late launching approach [7]. Xen remains booting up itself as usual until it boots Fidelius and leverages existing hardware support to issue a measurement on its integrity, which can be used in remote attestation to verify its validity. During the booting process of Fidelius, it measures the integrity of the hypervisor's code. Then it remaps the hypervisor's page tables as read-only, and allocates memory regions for PIT and GIT. It also updates the PIT to track the used physical pages, e.g., whether they are used as page-table-pages, Xen pages, or Fidelius pages. Next, it executes the *INIT* API provided by SEV, so that the whole system turns to the initialized state. Finally, the control is transferred back to the hypervisor for subsequent initialization.

4.3.2 VM preparing

Since SEV does not consider about I/O encryption, it is unable to directly load the encrypted data from the disk image to memory. To resolve this problem, Fidelius combines the *SEND* and *RECEIVE* API used for migration, as well as the para-virtualized API used for I/O encryption, to enable a secure VM booting from encrypted disk and memory images. Before booting, the owner of guest VM should first provide following components:

- The encrypted kernel image of guest VM, which is generated by the *SEND* APIs: the *SEND_START* API begins the sending process, and returns wrapped keys for encryption and integrity check, the *SEND_UPDATE* API encrypts memory regions using the wrapped encryption key. Finally, the *SEND_FINISH* API finishes the sending process, which results in an encrypted kernel image of and the measurement M_{vm} of it.
- K_{wrap}, the wrapped keys generated by the *SEND_START* API as shown above. It includes the wrapped encryption key K_{tek} and integrity key K_{tik}, and it should be sent to the Fidelius offline for later use.
- K_{blk}, the encryption key of the disk image, which is predefined by the guest owner. It is embedded in the encrypted kernel image and will be used by the front-end driver during block I/O.
- The disk image of the guest VM, which is encrypted by the K_{blk} and mounted during VM bootup.
- The SEV metadata, which is used to generate the master secret S_m by the firmware. It includes a guest provided nonce N_{vm} and the origin's public ECDH key.

We assume that the environment to generate the kernel and disk images is trusted, the K_{blk} is hidden in the encrypted kernel image and is not exposed to the hypervisor. While the other keys like K_{wrap} or nonce like N_{vm} are in public, since they are used for the ECDH key agreement algorithm, that only the guest owner and the firmware can agree on the master secret using their private key, while the hypervisor in the middle cannot guess them.

4.3.3 VM bootup

The guest VM bootup includes following steps:

1. Fidelius invokes the *RECEIVE_START* API, taking the parameters like K_{wrap}, N_{vm} and origin's public ECDH key. The firmware then unwraps the K_{wrap} to get K_{tek} and K_{tik}. Meanwhile, the firmware generates the guest VM's encryption key (K_{vek}) for memory encryption. The successful execution of this command returns a guest handle (H_{vm}) which refers to the internal structure in the firmware representing the SEV states of the guest.
2. The hypervisor loads the encrypted kernel image to the memory, and Fidelius uses the *RECEIVE_UPDATE* API to re-encrypt the loaded pages. Specifically, the firmware decrypts the pages with the K_{tek} and re-encrypts them using the K_{vek} in place.

3. After the whole kernel image is re-encrypted, Fidelius invokes the *RECEIVE_FINISH* API to finish the kernel memory re-encryption process and verifies its integrity using the K_{tik} by comparing it with M_{vm}.

4. Upon the re-encrypted kernel is ready, Fidelius prepares the *VMCB* of the guest VM and executes the *VMRUN* instruction, to run the guest kernel. The init phase of the guest VM is as normal, except for disk initialization, the front-end driver decrypts and encrypts the I/O data using the K_{blk} embedded in the kernel image.

4.3.4 Runtime memory protection

During runtime, the firmware uses K_{vek} to encrypt and decrypt memory. These encrypted memory pages are protected from malicious hypervisor and hardware attacks. To further prevent hypervisor from mapping guest memory and tampering it, the NPT of the protected guest is write-protected from the hypervisor, and the memory pages allocated for the guest are also unmapped from the hypervisor. This is guaranteed by the non-bypassable memory isolation described in Section 4.1.

When NPT violation happens, it first traps into the Fidelius's context, which then forwards it to the NPT violation handler of the hypervisor. The handler checks the exact reason, and decides to allocate new physical pages or simply update the permission bits. Immediately when the handler requires to fill a new NPT entry or update an old one, it calls the type 1 gate clearing the WP bit in *CR0*, and the PIT based policies will be enforced. This transition between two contexts complicates the process of updating NPT, which may potentially introduce performance slowdown. However, we found that instead of lazily updating the NPT, Xen will first allocate most of the physical memory regions for the guest by default, and update the NPT respectively. This means the operations of NPT updates happen in a batched manner during its bootup, while for normal run, there is rare NPT violation happening.

4.3.5 Runtime I/O protection

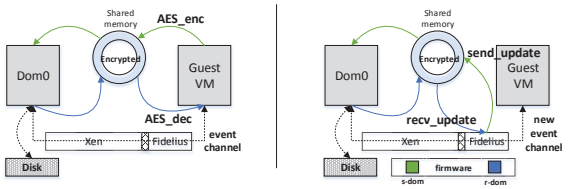

Figure 4: Two approaches for I/O protection.

Since network I/O data has been protected by the SSL protocol, we only consider about the disk I/O. By design of the SEV, the I/O (DMA) data cannot be encrypted by the encryption engine, thus normally the data has to be first copied in plaintext to the shared memory with the driver domain, which is untrusted. According to the implementation of para-virtualized I/O, it is reasonable for the guest front-end driver to utilize the pre-allocated persistent shared pages to reduce redundant data copy and protect its data before copying. Fidelius provides two interfaces for guest VM to protect its I/O data.

AES-NI based I/O protection. For processors with AES-NI hardware support, guest VM can encrypt its block I/O data by directly using the AES instruction set, as shown in the left part of Figure 4. With this approach, the guest owner should first provide the disk image encrypted by K_{blk}. During disk read, the back-end driver first copies the disk data to the shared memory, at this point, the data are encrypted and privacy preserved. Afterwards, the front-end driver decrypts the data in the shared memory using K_{blk}, and re-encrypt them with hardware-based encryption with K_{vek}. The write process is similar in the reverse path, except that it will batch the I/O write requests and process in sector granularity.

SEV based I/O protection For processors without AES-NI support, Fidelius can still provide another efficient mechanism which novelly reuses the SEV interfaces, as shown in the right part of Figure 4. During the initialization, Fidelius executes the *LAUNCH* serial API with the parameters of guest VM's handle, so that firmware will create the SEV context for the *s-dom*, which shares the same K_{vek} with the guest VM. Then Fidelius executes the *SEND_START* API to transfer the state of *s-dom* into the *sending* state. Afterwards, Fidelius executes *RECEIVE_START* to create the SEV context for the *r-dom*, which shares the same K_{vek} and K_{tek} with the guest VM and *s-dom* respectively, and turns its state to the *receiving* state. During runtime for I/O write, instead of directly using AES-NI instruction to encrypt data to the shared I/O buffer, the front-end driver first copies the memory to another dedicated buffer M_d, and leverages the retrofitted event channel mechanism, in which Fidelius uses *SEND_UPDATE* to do the data encryption using the SEV context of the *s-dom*. Specifically, the firmware first decrypts the data in the M_d with K_{vek} and then re-encrypt them using K_{tek} to the shared I/O buffer. I/O read is similar using the *RECEIVE_UPDATE* API with the SEV context of the *r-dom*. Here the *s-dom* and *r-dom* are the helper domains stored in the firmware for the protected guest VM, they are required since the *SEND_UPDATE* and *RECEIVE_UPDATE* can only be executed when the SEV context in the firmware is in *sending* or *receiving* state, while it is in *running* state for the guest VM.

4.3.6 VM migration

SEV has already provided interfaces for VM snapshot, restore, as well as migration. These are the exact APIs that Fidelius used to make the kernel image and load it into memory during boot time. Specifically, when a guest starts to initiate the migration process, it first invokes the *SEND* serial APIs to decrypt current memory pages with K_{vek}, and re-encrypt them with the transport encryption key (TEK) to generate the memory snapshot, as well as calculate the measurement of this snapshot with the transport integrity key (TIK). The target machine does the receiving process by executing the *RECEIVE* APIs step by step. The TEK and TIK are wrapped by the origin and sent to the target, and the key to unwrapping them is negotiated by two parties through the key agreement algorithm, so that the target can decrypt the memory snapshot, then re-encrypt it for the SEV-enabled guest with integrity verified.

It is noted that these operations are all done by the SEV hardware. We assume that the origin and target machines are all SEV supported, otherwise the privacy of memory content cannot be guaranteed. Meanwhile, currently, Fidelius does not support live migration, since the *SEND_START* API transfers the running state of guest VM into sending

state, which first stops the VM execution.

4.3.7 VM memory sharing

Originally before sharing the memory with other guests, the initiator should first offer the memory pages by creating the grant table entry through *grant_table_op* hypercall with *setup_table* command, then the other guest takes the grant reference and maps it by the same *grant_table_op* hypercall with *map_grant_ref* command. Since the shared memory is in plaintext, even though Fidelius revokes the permissions of accessing them from the hypervisor, the latter can still deceive the guest by manipulating the grant table. To enable a secure memory sharing mechanism, Fidelius provides an addition hypercall *pre_sharing_op* that is called by the initiator guest before creating the grant table entry, with the parameters of the target domain ID, the shared memory address and the number of shared frames. Fidelius directly handles this hypercall and records the information in the GIT structure. Afterwards, when the hypervisor handles the *grant_table_op* hypercalls to create or update grant table entries since the grant table is write-protected, it calls through the type 1 gate into Fidelius's context, and the GIT policies are enforced on all grant table operations.

4.3.8 VM shutdown

The termination of guest VM involves the Fidelius to first invoke the *DEACTIVATE* API to disengage the guest VM from the corresponding ASID, and uninstall the guest's key from the memory controller, then invoke the *DECOMMISSION* API to erase the guest context in the firmware. Besides, Fidelius revokes the memory pages of this guest VM, and modifies the PIT entries for each of these pages, as well as GIT entries if this guest VM ever shared memory with others, and finally deletes the SEV metadata.

5. POLICY ENFORCEMENT

Fidelius is in the position of updating critical resources, that proper policies are required to enforce the critical resources protection. In this section, we explore several sample policies used in Fidelius, which demonstrate good examples of protecting critical resources in a predictably secure manner.

5.1 Classified Policies with Exit Reasons

Some representative resources (e.g., *VMCB*) are compactly organized and may be frequently accessed by the hypervisor. If we strictly write protect them, there may be extensive context switches incurring large overhead. Instead, Fidelius shadows these resources. When hypervisor finishes its task and is about to re-execute the guest VM, Fidelius verifies the modification basing on different exit reasons.

Taking *VMCB* as an example. When vmexit happens, Fidelius first copies the entire *VMCB* to a pre-allocated private memory region, and checks the exit reason, according to which it decides which fields in *VMCB* should be masked. For instance, if it is due to a nested page fault, Fidelius will mask all guest states since the fault address used by hypervisor is in the *exitinfo* field. Similarly, if the exit reason is CPUID, then all states are masked except for specific four registers. Before executing *VMRUN*, Fidelius compares the previously shadowed version with the resources passed to the hypervisor and verifies the updates to check if the fields in two versions are different according to the exit reason. Taking the example of CPUID exit again, the corresponding policy should guarantee that only those four registers can be updated by the hypervisor.

5.2 Information Table Based Enforcement

For memory mapping data (page tables of the hypervisor, and NPT of guest VM), as well as grant tables used in VM memory sharing, Fidelius adopts the write-protecting mechanism. Therefore Fidelius is involved in the updating process of them. To ensure the updates are valid, Fidelius maintains one page information table (PIT) and one grant information table (GIT), through which Fidelius can enforce correct updates.

For memory mapping data update, when the hypervisor calls through the type 1 gate to the Fidelius's context, Fidelius queries on the PIT using the physical frame number (PFN). PIT is a three-level radix tree similar to normal page table, however, instead of storing PFN of next level page, PIT uses virtual frame number to accelerate page walking. The last level PIT page (4KB) holds page information of 1024 PFNs each of which is a 32-bit entry indicating the owner, usage, ASID and validity of the corresponding PFN. For example, when the hypervisor requires updating a page table entry (PTE), Fidelius first queries on the PIT using the PFN of this page-table-page and makes sure that its owner is the hypervisor, it is used as a last level page-table-page, and it is valid. Then Fidelius queries on the PIT using the PFN of the new mapped page, to check whether this page can be mapped by the hypervisor. During this process, any illegal mapping update will be aborted.

On the other hand, the GIT consists of an array of grant information entries, each of which stores the information of one grant, including the initiator guest domain ID as index, the target domain ID, as well as the shared memory address and the number of page frames. Before the initiator guest creating the grant table entry, it first invokes the *pre_sharing_op* hypercall to create the corresponding GIT entry with the grant information. Afterwards, when the hypervisor requires to create or update the grant table entries, it goes through the type 1 gate to the Fidelius's context. The latter queries on the GIT using the ID to ensure the new entries are consistent with those saved in GIT.

5.3 Other Policies

Similar to PerspicuOS [8], Fidelius also defines other policies to enforce more fine-grained and flexible protections. For example, the *write-once* policy is enforced for start info page and shared info pages; the *execute-once* policy is enforced for some privileged instructions (e.g., lgdt, lidt), that the hypervisor can only write these pages or execute these instructions once during its initialization, and latter modification and execution are not allowed. This policy is enforced by using a bit-vector to record specific memory regions with one bit per byte. Since the *write-once* and *execute-once* policies will not frequently be used, Fidelius ensures that these data and instructions pages are allocated from predefined memory regions, which are mapped as read-only or non-executable in the hypervisor. The modification or execution of them are mediated by Fidelius through page fault handler, where the bit-vector is checked to make sure there is no previous write or execution to them, and the corresponding bit is set to forbid further operation.

Meanwhile, Fidelius also enforces the *write-forbidding* pol-

icy for the code pages of the hypervisor. Fidelius tracks the code pages at the very beginning by recording the address and size of the text section. Since the code pages are marked as read-only in the hypervisor's page tables, any attempt of modification to them either goes through type 1 gate or results in a page fault, then Fidelius is involved in to simply impede the write operation, and log this operation for further auditing.

6. SECURITY ANALYSIS

6.1 Defending against Hardware Attacks

The intrinsic features of memory encryption benefited from AMD SEV can be directly leveraged to protect software from hardware-based attacks, including cold-boot and bus snooping, and so on. Suppose an attacker is able to directly dump the memory, what she can see are all encrypted data, and the cryptographic keys only reside in the secure processor, which cannot be obtained by the attacker. Meanwhile, for disk data, the disk I/O is encrypted through the key K_{blk}, which resides in the encrypted kernel memory of the guest VM, and is concealed from hardware-based stealing.

6.2 Preventing against Malicious Host System

The most important security problems resolved by Fidelius is to protect guest VMs from the malicious host system. In this section, we first explain how Fidelius protect guest VM from following potential attacks, and then give a quantitative analysis using the XSA statistics.

Breaking memory privacy. Except for side-channel attacks, which are out of scope, the only way for the hypervisor to read memory is to have permissions to access them. There are two possible approaches: directly mapping the physical memory of guest VM to its page tables, or using inter-VM remapping attacks by mapping the guest memory to its conspirator VM's NPT. In the latter approach, since CPU caches are in plaintext, when the conspirator VM tries to access the memory, a cache-hit may happen in a high probability to leak privacy. These attacks can be defeated using Fidelius. Since the page tables of hypervisor and guest VM's NPT are all write protected, PIT based policies are enforced on their updates. Fidelius will find that the memory pages belong to the protected guest VM, it simply forbids these mappings, thus prevents such attacks.

Violating memory integrity. Malicious hypervisors can violate the integrity protection of guest VM's memory through direct or indirect tampering. Fidelius strictly prevents direct memory tampering through memory permission control, that hypervisor has no permission to modify it arbitrarily. However, hypervisor needs to load the kernel image into the memory during guest VM creating, at that point, it temporarily obtains write permissions. Fortunately, the integrity can be guaranteed by the measurement verification with K_{tik} in the *RECEIVE* process, thus preventing tampering at that phase. For indirect approaches, Rowhammer attacks [11] can flip specific memory bits by consecutively accessing the memory in its adjacent rows. Generally, Fidelius cannot strictly eradicate this malevolent bit flipping, however as the memory is encrypted, it is terribly difficult for attackers to exploit this for further destruction.

Disabling protection. The memory protection can be disabled by executing specific privileged instructions (Table 2). Our binary scanner ensures that these instructions only exist in pre-defined locations. Normally the hypervisor can execute them through either the type 2 gate or even ROP based attacks, however, since the checking loop are placed right after the instruction with policy enforcement, the invalid operations will be detected and prevented. For other instructions that may immediately alter the control flow (*VMRUN*) or switch the context (*mov CR3*), Fidelius unmaps them in the hypervisor's address space, and the sanity check is placed right after the instruction to remap them with policy enforcement.

I/O data stealing and tampering. Disk I/O is another attack surface for the malicious driver domain since it is in the middle of the I/O path. However since the data sent to the shared buffer or received from the disk are all encrypted, no privacy will be leaked.

Other issues. Fidelius is designed to prevent other possible attacks. For instance, the replay attacks [2] can be defeated since the hypervisor cannot directly manipulate the NPT. The Iago attacks [12] can be avoided since Fidelius lies in the middle of hypervisor and guest, appropriate policies can be defined to check the values returned by the hypervisor before *VMRUN*.

Quantitative analysis. To quantitatively present how Fidelius can benefit guest VM, we analyzed 235 Xen vulnerabilities from Xen Security Advisories (XSA). We found that among the 235 XSAs, 177 are related to the hypervisor (while others are related to the Qemu and are out of scope). For these 177 vulnerabilities, Fidelius can thwart 31(17.5%) which is used to escalate privilege and 22(12.4%) which lead to information leakage. While others are not considered by Fidelius (e.g., 14(7.9%) of them are due to flaws inside the guest VM, and the remaining are all related to DoS attacks.).

6.3 Security Analysis of Fidelius Extension

Fidelius introduces a small portion of context lying in the same privileged level with the hypervisor. Therefore the security of this extension must be ensured. It is noted that different with some other approaches [8, 9], Fidelius shares most of its context with the hypervisor, e.g., most of the Fidelius's code is also executable in the hypervisor, and the data are also mapped in the hypervisor's address space except some special ones. To guarantee that the Fidelius's code will not be exploited by the malicious hypervisor, and the critical data are prevented from being arbitrarily manipulated, the Fidelius is required to satisfy following rules:

- The critical data are either unmapped or mapped as read-only in the hypervisor's address space, that only through type 1 or type 3 gates, they are turned to writable.
- The privileged instructions listed in Table 2 are ensured to be monopolized (only has one instance) in the Fidelius's code section, and their executions are sanity checked with policies enforcement.
- The TCB of the Fidelius's code should be small enough so that verification can be leveraged to prove its correctness.
- The integrity of the code itself is guaranteed by the PIT based policy enforcement, that no code injection is allowed, and the system is armed with data execution prevention mechanism.

Totally Fidelius introduces 1780 lines of C code (LoCs) as the TCB. Though in the Fidelius's context, the hypervisor's

code is also executable, we can remove them out of the TCB, as far as the Fidelius's code is verified to be secure enough, that no control will flow to other code, and it includes no vulnerability for ROP-like exploits. Since currently verification can be applied to more than ten thousand LoCs [13], it is reasonable to verify the correctness of Fidelius, which is left as our future work.

7. PERFORMANCE EVALUATION

To evaluate the efficiency of Fidelius, we measured the performance slowdown for applications under protection by Fidelius, as well as some micro benchmarks to shed light on the factors introducing this overhead. All experiments are done on a machine with 8 AMD Ryzen cores (16 hardware threads) running at 3.4 GHz and with 8 GB memory. The hypervisor Xen is 4.5.1. Both of management VM and guest VM are with Linux kernel 4.10.2. Each guest VM is configured with 2 virtual cores and 2 GB memory.

7.1 Macro Benchmarks

Currently, there is no motherboard supporting AMD SEV feature, thus we use the SME to simulate the performance overhead. Since the two features share the same hardware encryption engine, we believe this simulated overhead is convincing. Specifically, when the guest finishes booting up, it invokes a hypercall to the Fidelius, which set the C-bits in the nested page tables for all of the free pages, so that the subsequently allocated memory pages are encrypted by the SME hardware.

SPECCPU 2006. We measured all C programs in the SPECCPU 2006 benchmarks in guest VM running upon original Xen, Fidelius (without SME enabled) and Fidelius-enc (with SME enabled). Figure 5 shows the normalized overhead of Fidelius and Fidelius-enc compared with original Xen. The average overhead caused by Fidelius is less than 1%, and for Fidelius-enc the average overhead is 5.38%. Particularly, for CPU-intensive benchmarks like bzip2, hmmer, and h264ref, there is nearly no overhead. While for benchmarks with large memory access, like mcf and omnetpp, Fidelius-enc shows relatively high slowdown (17.3% and 16.3%) as a result of memory encryption.

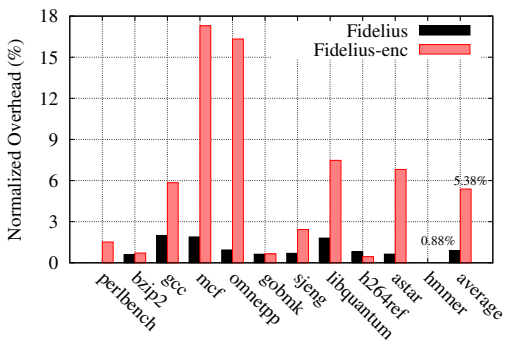

Figure 5: Performance evaluation: SPECCPU 2006

PARSEC. We evaluated the PARSEC benchmark suites. The results are shown in Figure 6. It implies that Fidelius introduces negligible overhead (0.43%) compared to original Xen. When SME is enabled, only *canneal* benchmark shows relatively large overhead (14.27%) since it has an unstructured data model and will access a huge amount of memory. For other benchmarks, the average overhead of Fidelius-enc is (0.95%), which is imperceptible.

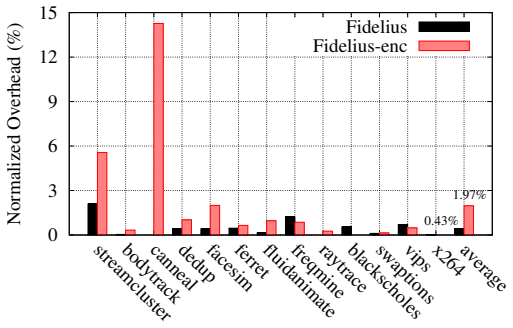

Figure 6: Performance evaluation: PARSEC

Fio. To test the overhead caused by the I/O protection, we selected the fio benchmark with sequential and random read/write configurations. The results are presented in Table 3. It is noted that decryption incurs more overhead in seq-read for two reasons: 1) write requests are encrypted in a batched manner, while read requests are not, and the decryption may be duplicated due to sector level granularity; 2) encryption is apart from the critical path of the writing process, while the driver has to wait for decrypted data for subsequent processing.

Table 3: Performance evalutaion: FIO

Operation	Xen	Fidelius AES-NI	Slowdown
rand-read (KB/s)	1506.8	1486	1.38%
seq-read (MB/s)	1196.8	922.6	22.91%
rand-write (KB/s)	21066.8	20920	0.70%
seq-write (MB/s)	152.7	147.2	3.61%

7.2 Micro Benchmarks

To better understand the factors causing the performance overhead, we conduct some micro benchmarks to answer following three questions. These tests are all run for 100,000 times to get the average values.

What is the overhead of runtime transition between Xen and Fidelius? We measure the additional CPU cycles of different gates. The results show that the overhead of type 1 gate (disable WP) is 306 cycles, type 2 gate (checking loop) is 16 cycles, and type 3 gate (add new mappings) is 339 cycles. For the type 3 gate, flushing TLB uses 128 cycles and writing data into cache uses less than 2 cycles.

What is the overhead of shadowing critical resources? We write a kernel module in the guest to invoke a void hypercall and calculate the round-trip cycles. The overhead shadowing and checking are averagely 661 cycles.

What is the overhead of I/O protection using AES-NI, SEV API and software emulated encryption? We evaluate this overhead by copying 512 MB memory in the guest kernel using the three encryption techniques, then compare the time cost with normal memory copy. The results show that the slowdown of AES-NI is 11.49%, while SME is 8.69%. And both of them perform much better than software emulated encryption which incurs more than 20X overhead. This implies that the SEV based I/O protection is more attractive considering its efficiency.

8. REFLECTION ON HARDWARE

Fidelius demonstrated one direction of providing encrypted VM protection using SEV. Nonetheless, our experience of working on SME and SEV suggests the following possible hardware suggestions for a more secure and lightweight design: 1). *Hardware-based integrity checking.* Currently, the integrity of Fidelius is not guaranteed if the memory is tampered with by hardware-based attacks (e.g., RowHammer), or the I/O data is maliciously manipulated. This can be addressed by integrating a Bonsai Merkle Tree (BMT) [3] to enable hardware-based integrity in the secure processor. 2). *Customized keys.* Currently, it takes a plenty of effort for the guest owner to deploy its VM to the Fidelius environment. The guest owner should first create an encrypted kernel image using the *SEND* APIs in a trusted environment. Meanwhile, the key agreement protocol between the guest owner and the target machine requires them to preidentify each other, which means, the encrypted kernel image can only be loaded into one pre-defined machine. On the other hand, currently we can only use the *SEND* API to do the SEV-based I/O protection, which is restricted by its hardware design. To address such issues, we find a better solution is to add a series of instructions which are similar to *SEND* and *RECEIVE* APIs except that they allow customized keys. Specifically, we can use a *SETENC_GEK* instruction to generate a customized guest encryption key (GEK), which is then used to encrypt and decrypt specified memory range through the *ENC* and *DEC* series of APIs.

9. RELATED WORK

9.1 Hardware-based Memory Encryption

Secure processor has been extensively studied during the last decade [14, 15, 16, 17, 18, 19, 20, 21, 22, 3]. AEGIS [20] was an architecture for a single-chip processor that provides users with tamper-evident, authenticated environments. XOMOS [15] leveraged its secure processor architecture XOM [14] to support normal mechanisms in existing operating systems like shared libraries, IPC, etc. Researchers continued the lines of techniques to improve memory encryption [23, 18, 19] and integrity verification [16, 24, 19]. E.g., Brian et al. [19] proposed the address independent seed encryption (AISE) to further optimize the counter-mode based memory encryption scheme, as well as bonsai merkle tree (BMT) to optimize the security and performance of merkle tree based memory integrity verification. Based on these optimizations, Bastion [21] and SecureME [22] were proposed to protect security-critical applications from physical attacks and untrusted OS. Similar with Fidelius, HyperCoffer [3, 25] was the first to consider about running VM on untrusted hypervisor through the help of secure processor, it was based on AISE and BMT, and introduced a shim layer to enable a transparent approach on commercial off-the-shelf virtualization stack.

Besides these academic prototypes, mainstream processor manufacturers also proposed products with memory encryption support. Intel proposed SGX, a hardware extension to the processor, provides user-level software a set of new instructions for allocating encrypted memory regions in the so-called enclaves, that even higher privileged software can not access enclave memory. Extensive studies have attempted to leverage SGX to shield software [26, 27, 28, 29, 30, 31] or to strengthen the security of SGX itself [32, 33]. E.g., Haven [26] introduced the idea of shielded execution, which the code and data of the unmodified application can be placed in the SGX enclaves thus are shielded from compromised OS. SCONE [27] leveraged SGX to provide a secure container mechanism. Ryoan [28] was proposed to provide a distributed sandbox using SGX for preserving privacy in the data processing services, in the presence of multiple untrusted service providers. M2R [29] and VC3 [29] both utilized SGX to secure the MapReduce framework. In the direction of hardening security inside SGX enclaves, SGX-Shiled [32] was proposed to bring address space randomization feature to the programs running within the enclaves, and T-SGX [32] utilized hardware transactional memory to eradicate page fault based side channel attacks targeted on critical applications executing in the enclaves.

As explained in Section 2, AMD proposed another processor solutions called SME and SEV to do memory encryption, that privileged software can selectively encrypt memory by simply setting the C-bits in the page table entries. One recent work [2] tried to analyze the security of SEV, which shows that there are at least three possible shortcomings of current SEV design, which can be abused to conduct various attacks. Fidelius is the first to leverage SEV for comprehensive guest VM protection, and to evaluate its efficiency using the real hardware-based memory encryption.

9.2 Defending Against Untrusted Hypervisor

Besides HyperCoffer [3], some other approaches also consider removing hypervisor out of the TCB [34, 35, 7, 36, 37, 38, 39, 9, 40]. For architectural support, H-SVM [35] extended the hardware of memory virtualization, that every NPT mapping should be verified by the secure hardware extension instead of directly modified by the hypervisor. Similarly, HyperWall [36] made minor modifications to the microprocessor and MMU, to construct and maintain a CIP table preventing untrusted hypervisor and DMA mechanism from arbitrarily accessing memory of guest VM. For software-based methods, CloudVisor [7] proposed nested virtualization to intercept communications between hypervisor and guest VM. Analogously, Dichotomy [39] presented the ephemeral virtualization, to divide hypervisor into a hyperplexor running in the nested mode and a featurevisor executing in the guest mode. The featurevisor is designed to voluntarily relinquish control to the hyperplexor to reduce the overhead of nested virtualization. In the other direction, NOVA [34], HyperLock [37] and DeHype [38] shared the similar ideas of splitting the monolithic hypervisor into pieces, that only one concise and verifiable core component remains in the privileged mode, while others functional components are executed in either user mode [34, 38] or non-root environment [37] for each VM, so that least privilege principle is reserved. Similar to Fidelius, Nexen [9] and Liang et al. [40] both leveraged the same privilege level protection for an untrusted hypervisor. Nexen [9] deconstructed the xen hypervisor into one privileged security monitor, one component for shared service, and duplicated xen code and data called xen-slice for each VM, to thwart a large number of known Xen vulnerabilities. Liang et al. [40] further eliminated the relatively expensive cross-boundary instructions like *mov cr0* through privilege instruction interception and address space randomization. On the other hand, researchers also considered about extracting the control VM (domain-0 in Xen) out of the TCB [41, 42, 43].

Pure software based extensions like CloudVisor, Nexen, Nova cannot defend against hardware attacks in nature. Solutions like H-SVM, HyperWall mainly aim to isolate privileged operations from hypervisor instead of encrypting memory. Therefore, all of these work trust the whole hardware stack (including memory) and excluding physical attacks like cold-boot or bus snooping attacks in their threat models. Meanwhile, some of them incur large performance overhead due to extra layer indirections. Fidelius, by contrast, is designed to defend against physical attacks with acceptable overhead, facing threat model of untrusted host systems.

10. CONCLUSION

In this paper, we presented Fidelius, which is the first to leverage the AMD's SEV hardware feature to protect guest VMs from hardware-based stealing and to provide a software-based extension to enable a comprehensive VM protection without trusting the underlying host systems. Fidelius revokes the permissions of accessing critical resources from the privileged hypervisor by non-bypassable memory isolation and provides efficient and effective gate mechanisms to secure the transition between isolated address spaces. It also adopts the para-virtualized approaches for a secure and efficient disk I/O protection. The life-cycle protection of guest VM proves the effectiveness of Fidelius. And the performance evaluation shows its efficiency by introducing negligible overhead during system runtime.

Acknowledgment

We thank the anonymous reviewers and shepherd for their constructive comments. This work is supported in part by National Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314, 61525204), a research grant from Huawei Technologies, Inc., National Top-notch Youth Talents Program of China, and Singapore NRF (CREATE E2S2).

11. REFERENCES

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[2] F. Hetzelt and R. Buhren, "Security analysis of encrypted virtual machines," in *Proc. VEE*, 2017.

[3] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 246–257, IEEE, 2013.

[4] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper, Apr*, 2016.

[5] D. Kaplan, "Protecting vm register state with sev-es," *White paper, Feb*, 2017.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, pp. 164–177, ACM, 2003.

[7] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, ACM, 2011.

[8] 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.

[9] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, H. Guan, and J. Li, "Deconstructing xen," 2017.

[10] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 380–395, IEEE, 2010.

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 361–372, IEEE Press, 2014.

[12] S. Checkoway and H. Shacham, *Iago attacks: Why the system call api is a bad untrusted rpc interface*, vol. 41. ACM, 2013.

[13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: an extensible architecture for building certified concurrent os kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, 2016.

[14] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[15] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 178–192, 2003.

[16] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 339, IEEE Computer Society, 2003.

[17] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 103–112, IEEE Computer Society, 2006.

[18] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 179–190, IEEE Computer Society, 2006.

[19] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 183–196, IEEE, 2007.

[20] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*, pp. 160–171, ACM, 2003.

[21] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, IEEE, 2010.

[22] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Secureme: a hardware-software approach to full system security," in *Proceedings of the international conference on Supercomputing*, pp. 108–119, ACM, 2011.

[23] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 84–94, ACM, 2006.

[24] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pp. 295–306, IEEE, 2003.

[25] Y. Xia, Y. Liu, H. Guan, Y. Chen, T. Chen, B. Zang, and H. Chen, "Secure outsourcing of virtual appliance," *IEEE Transactions on Cloud Computing*, 2015.

[26] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[27] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OâĂŹKeeffe, M. L. Stillwell, *et al.*, "Scone: Secure linux containers with intel sgx," in *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[28] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: a distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 533–549, USENIX Association, 2016.

[29] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M2r: Enabling stronger privacy in mapreduce computation," in *USENIX Security*, vol. 15, pp. 447–462, 2015.

[30] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 38–54, IEEE, 2015.

[31] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-nfv: securing nfv states by using sgx," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 45–48, ACM, 2016.

[32] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[33] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[34] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, pp. 209–222, ACM, 2010.

[35] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 272–283, ACM, 2011.

[36] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," in *ACM SIGPLAN Notices*, vol. 47, pp. 437–450, ACM, 2012.

[37] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with hyperlock," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 127–140, ACM, 2012.

[38] C. Wu, Z. Wang, and X. Jiang, "Taming hosted hypervisors with (mostly) deprivileged execution.," in *NDSS*, Citeseer, 2013.

[39] D. Williams, Y. Hu, U. Deshpande, P. K. Sinha, N. Bila, K. Gopalan, and H. Jamjoom, "Enabling efficient hypervisor-as-a-service clouds with eemeral virtualization," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 79–92, ACM, 2016.

[40] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng, "Dancing with wolves: Towards practical event-driven vmm monitoring," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 83–96, ACM, 2017.

[41] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 189–202, ACM, 2011.

[42] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 253–264, ACM, 2012.

[43] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman, "Delusional boot: securing hypervisors without massive re-engineering," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 141–154, ACM, 2012.