KVSEV: A Secure In-Memory Key-Value Store with Secure Encrypted Virtualization

Junseung You
ECE and ISRC,
Seoul National University
Republic of Korea
jsyou@sor.snu.ac.kr

Kyeongryong Lee
ECE and ISRC,
Seoul National University
Republic of Korea
krlee@sor.snu.ac.kr

Hyungon Moon*
UNIST
Republic of Korea
hyungon@unist.ac.kr

Yeongpil Cho
Hanyang University
Republic of Korea
ypcho@hanyang.ac.kr

Yunheung Paek*
ECE and ISRC,
Seoul National University
Republic of Korea
ypaek@snu.ac.kr

ABSTRACT

AMD's Secure Encrypted Virtualization (SEV) is a hardware-based Trusted Execution Environment (TEE) designed to secure tenants' data on the cloud, even against insider threats. The latest version of SEV, SEV-Secure Nested Paging (SEV-SNP), offers protection against most well-known attacks such as cold boot and hypervisor-based attacks. However, it remains susceptible to a specific type of attack known as Active DRAM Corruption (ADC), where attackers manipulate memory content using specially crafted memory devices. The in-memory key-value store (KVS) on SEV is a prime target for ADC attacks due to its critical role in cloud infrastructure and the predictability of its data structures. To counter this threat, we propose KVSEV, an in-memory KVS resilient to ADC attacks. KVSEV leverages SNP's Virtual Machine Management (VMM) and attestation mechanism to protect the integrity of key-value pairs, thereby securing the KVS from ADC attacks. Our evaluation shows that KVSEV secures in-memory KVSs on SEV with a performance overhead comparable to other secure in-memory KVS solutions.

*Corresponding authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624658

CCS CONCEPTS

• **Security and privacy → Database and storage security**; **Software and application security**; **Trusted computing**.

KEYWORDS

Trusted execution environments, Secure Encrypted Virtualization, Key-value store, Confidential computing

ACM Reference Format:
Junseung You, Kyeongryong Lee, Hyungon Moon, Yeongpil Cho, and Yunheung Paek. 2023. KVSEV: A Secure In-Memory Key-Value Store with Secure Encrypted Virtualization. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624658

1 INTRODUCTION

These days, many applications are deployed as cloud services to take various advantages of cloud computing, such as elasticity, cost-effectiveness, and high availability. Cloud services are often built and enacted by stitching together many individual components or services. In-memory *key-value stores* (KVSs), such as Redis [45] are indispensable components that almost every cloud service commonly employs as its temporal data storage. The benefit of cloud computing comes at a cost, particularly in terms of security, primarily because cloud services execute remotely and are not under the complete control of their clients. The machines operate under the control of privileged software owned by cloud platform providers. Therefore, this remote execution in the cloud implies that applications for cloud services have no choice but to trust the platform providers and execution environments for the integrity of their code and data. This blind trust in the cloud environments presents a potential attack vector to *adversarial insiders* (*e.g.*, malicious administrators

and co-located tenants in the cloud) with capabilities that the insiders have. To expose or corrupt sensitive contents managed by cloud applications, such adversarial insiders could either physically replace hardware components with a maliciously crafted one or exploit the privileged software layers like OS and hypervisor.

Concerning these ever-escalating *insider threats* in the era of cloud computing, multiple processor vendors such as Intel, ARM, and AMD have introduced commodity trusted execution environments (TEE) to provide a hardware-assisted secure computing platform. These TEEs, despite their implementation differences, share a common attribute: they protect user code and data from any privileged software layers in cloud infrastructure using hardware-enforced mechanisms. Such an appealing security promise offered by TEEs and the prevalence of KVSs in the cloud have naturally brought attention from many engineers to leveraging TEEs for developing *secure KVSs* [4–6, 29]. These secure KVSs ensure the confidentiality and integrity of their key-value pairs to protect the services' data. However, to the best of our knowledge, no published work has proposed a secure KVS using a commodity TEE on x86/x64, known as *Secure Encrypted Virtualization* (SEV) [1], which has been continuously gaining traction since its initial release in 2016. Considering the growing popularity of AMD processors in many cloud platforms today, there is undoubtedly a strong demand for providing a secure KVS for remote users of AMD systems.

Our work aims to address the demand for protecting a KVS on SEV from viable attacks that untermine the security advantages of using SEV. The arms race surrounding SEV has led to the development of its latest version, SEV-SNP (Secure Nested Paging) [2]. This latest version of SEV thwarts most existing attacks (*e.g.*, hypervisor-based attacks, cold boot attacks[17], and DMA attacks [12]), but remains vulnerable to a specific class of attacks, referred to as *active DRAM corruption (ADC)*, because SEV does not cryptographically authenticate the contents of off-chip memory. In particular, attackers with tools like DDR bus interposers [15, 49] can dynamically modify or replace off-chip memory content. Through ADC, an adversary can either corrupt or replay the key-value pairs, necessitating additional security mechanism to ensure the integrity of key-value pairs. Unfortunately, software-only measures such as Merkle tree [48] that stores cryptographic hashes of the target data cannot be adopted as a drop-in solution, due to the fact that such software-protected hashes are also vulnerable to ADC, allowing the attackers to replay hash and key-value pair to pass the integrity validation with outdated values.

This paper presents our ADC-safe in-memory KVS, named KVSEV, on an AMD system. The key idea behind KVSEV is that a sequence of virtual machines (VMs) is used to provide trusted storage for the cryptographic hashes of key-value pairs. Different encryption keys assigned for respective VMs prevent the hashes from being rolled back as outdated hash will be decrypted to invalid value. VM encryption keys are secured by on-chip AMD Secure Processor (SP), safe from ADCs. For the safe deployment and termination of the hash-holding VMs against possible attacks, KVSEV makes us of several new features of SEV-SNP. Particularly as useful primitives for our work, we utilize two protocols that SEV-SNP implements. One is for *virtual machine management* (VMM), which empowers remote clients of a cloud to directly communicate with the firmware in trusted hardware. The other is for *enhanced attestation* that allows a flexible request for the report at runtime. Further, we make three optimizations to reduce the latency stemming from continuous VM creation and VM verification process; we *eagerly create VMs* that KVSEV uses to handle requests in advance, *debloat* the VMs to quickly boot new VMs, and perform the verification *asynchronously*.

We have designed and implemented KVSEV on an off-the-shelf AMD machine with SEV and, as presented in §5, evaluated it using two workload distributions. Our experimental results show that KVSEV incurs 13.38× slowdown compared to an insecure KVS on SEV when handling workload with 90% reads. Experiments with large KVSs also demonstrate that the performance of KVSEV is comparable to or better than existing secure in-memory KVSs [29].

2 AMD SEV

SEV is an extension to the AMD Virtualization [10] architecture that is built with Secure Memory Encryption (SME) [26]. SME offers protection against snooping or probing of DRAM contents by the adversaries. With a hardware-accelerated Advanced Encryption Standard (AES) engine embedded in the on-die memory controllers, memory contents are encrypted and decrypted on arrival and departure from the CPU package. While Secure Memory Encryption (SME) uses just one encryption key for the whole machine, SEV provides distinct ephemeral keys, called *VM Encryption Keys* (VEKs), to encrypt each VM data when stored outside the CPU package. This encryption with the per-VM key prevents the malicious host hypervisor or physical attackers from obtaining the VM's data The encryption keys are securely held by the SEV firmware running on an isolated system on the same package, called the AMD Secure Processor (SP). This isolation renders any attempts to access the VEKs fruitless.

The SEV memory controller encrypts the VM memory using 128-bit AES symmetric encryption. In the first generation of SEV released in 2016, the AES encryption used the XOR-Encrypt (XE) mode [11], but this was soon replaced by

Attack precision	Enabling mechanisms	Applicability on SEV-SNP	Protection
high (single instruction)	Interrupt injection [33, 52] (e.g., APIC controller)	✘	Hypervisor is restricted from injecting interrupts and exceptions into the VM by introducing a single injection vector (#HV) to act as a doorbell for VM to acknowledge [2, 31].
medium (few instructions)	fault injection [27, 40, 42] (e.g., manipulating DVFS)	✘	Software-based fault injections through undervolting were not able to generate effective faults on SEV machines [43].
low	data access monitoring	✔	**KVSEV**

Table 1: Mechanisms that ADC attackers can use to improve attack precision, and the corresponding protection measures.

the XOR-Encrypt-XOR (XEX) mode in the following generations, SEV-ES [25] and SEV-SNP [2]. The SEV's encryption mode encrypts each aligned 16-byte memory block. In order to prevent the attacker from directly inferring a plaintext by comparing the ciphertext blocks stored in different locations, SEV uses a physical address-based tweak function T [26]. With this tweak function, a ciphertext c is calculated by running the tweak function with the system physical address during the encryption process, expressed as $c = Enc_K(p \bigoplus T(P_m)) \bigoplus T(P_m)$ where p denotes the plaintext and K denotes the VEK generated from a secure entropy source by the firmware.

The second generation of SEV, called SEV-ES, comes with an additional capability of protecting the register state integrity during the context switches. The register states that are stored in an in-memory data structure called *Virtual Machine Save Area (VMSA)* are encrypted for this purpose. The latest one, called SEV-SNP, also protects the memory integrity from software attackers who control the hypervisor. The ownership of each physical page is managed with the newly added *Reverse Map Table (RMP)*, which is used to prevent the hypervisor from writing to VM's physical pages.

SEV enables its clients to obtain an attestation report as proof of the integrity of the launched VM. As the first step of this attestation procedure, the client obtains from the SEV-enabled server the version information (e.g., Chip-ID and TCB version). Subsequently, when the VM is loaded, the server provides the client with the attestation report that the server signed using a set of private keys. The client verifies this report using the public keys that it obtains from the AMD's key server using the previously obtained Chip-ID and TCB version.

3 MOTIVATION AND THREAT MODEL

The threat model and design decisions behind the AMD SEV and its latest variant, SEV-SNP, leave SEV VMs vulnerable to some physical attacks corrupting the external memory content. SEV primarily aims to protect memory confidentiality through encryption and prevents malicious memory corruption from software-oriented attacks in its latest variant,

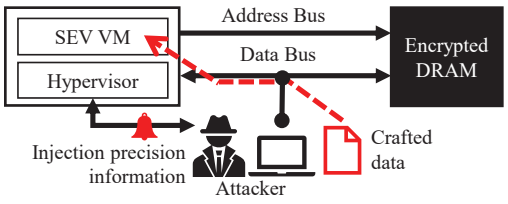

Figure 1: Illustration of ADC using DDR bus interposer to inject data or code into SEV-protected VM. The precision of the injection depends on the information from the other mechanisms (see Table 1 that the attackers can combine, as will be explained in §3.2.

SEV-SNP. Specifically, SEV encrypts memory contents for a VM, performs access control on the memory, and prohibits using Direct Memory Access (DMA) to leak or corrupt the protected memory content. These mechanisms are known to be enough to prevent many physical attacks leaking the memory content (*e.g.*, the cold boot attack [17]) or software-oriented attacks corrupting the memory content (*e.g.*, DMA attacks [12]).

However, the mechanisms cannot prohibit the attacks directly corrupting the external memory content, namely *active DRAM corruption (ADC)*, leaving it as a valid threat against SEV. An attacker can perform ADC by incorporating various tools (*e.g.*, DDR bus interposer [15, 49] or specially manufactured physical devices like custom DIMM) to not only observe DRAM contents but also corrupt or inject data/code into buses or directly into the DRAM [56], as depicted in Figure 1

AMD also explicitly acknowledges this limitation in their document [2]. As discussed earlier, SEV-SNP does not cryptographically authenticate the external memory content, making SEV-SNP incapable of protecting the memory integrity against the intended or unintended faults to memory content. With ADC, an attacker can make, for example, a random corruption at their disposal without being noticed [8] or replay older memory content [18]. The attacker can even forge the encrypted memory content using a strategy similar to the one used for software attacks against older editions of SEV [53] if the victim VM handles external data that they can control. In

the worst case, ADC using a maliciously crafted buffer chip can arbitrarily manipulate all data [50]. Not only because realizing such attacks is challenging but also because SEV-SNP was released just a year ago and deployed only recently in commodity systems, there have been yet no real-world attack examples reported in the literature. However, we argue that we must proactively provide countermeasures against such probable attacks before adversaries inflict actual damage on commodity KVSs relying on SEV-SNP.

3.1 Threat Model

We consider the adversarial insiders who have full privilege over the machine that we aim to protect. The full privilege includes physical access to the machines and the control of privileged software such as hypervisor. With their physical access, these attackers may attach maliciously crafted hardware to corrupt the content of a victim KVS in the granularity of their choice (*e.g.*, byte or page) by putting the specially crafted physical devices into the machine, as pointed out in previous work [16]. However, they are limited to target key-value pairs with ADCs, unable to utilize KVS program code or other data as a medium for successful attack, as will be explained in §3.2. Specifically, they are unable to freely perform injection to a victim KVS to conduct attacks at a precise timing, restricted to be only capable of low-precision ADC, thus leaving key-value pairs as a sole viable target of attack. They can take control of privileged software by either running a maliciously crafted hypervisor or exploiting the running hypervisor's vulnerability. The only system component such an adversary cannot alter is the CPU package, including the AMD SP. We also assume that SP's security protocol to manage the encrypted VMs does not have vulnerability and achieves its stated goal. That is, an attacker can neither take a recoverable snapshot of a VM without the help of the VM itself nor obtain the cryptographic key stored within SP for each VM. The attacker can only take a snapshot by obtaining the data directly from the physical memory and can replay the snapshot to the VM running on the same machine. Like other confidential computing mechanisms, we only aim to protect the confidentiality and integrity of an in-memory KVS and rule out upholding availability from our security goal.

3.2 Feasibility of ADC under SEV-SNP

Though it appears omnipotent by itself, manipulation such as *bus injection* [13, 14, 56] merely enables ADC. To perform fruitful ADC corrupting valid targets in an attacker-beneficial way, an attacker must combine the manipulation with other capabilities. For example, an attacker must at least have a means to locate the target of corruption to construct a fully-fledged ADC on top of bus injection because blind random corruption is likely to cause unexpected results (*e.g.*, victim crash). To control the exact moment of corruption with respect to the victim program execution, the attacker must be capable of controlling the victim's execution as well (*e.g.*, single-stepping).

We find that SEV-SNP limits the attacker's capability, which they need for ADC, of precisely controlling the location and moment of memory corruption. In the rest of this section, we present the kind and the level of control the attackers need when they corrupt the victim's code or data and what they can do with the capabilities they still have under SEV-SNP.

3.2.1 ADC against the program code. Program code is a juicy target for ADC in that attackers can alter the program behavior by manipulating the code. To mount a successful attack, an adversary must incorporate means to accurately encrypt the target code before injection, as SEV decrypts the code within the CPU package when it receives the injected code. However, to the best of our knowledge, there are no known solutions to build an encryption oracle that works on SEV-SNP (further elaborated in §6.1), thus rendering based ADC against the program code infeasible.

3.2.2 ADC against data. An attacker often aims to corrupt the victim's data that resides in the external DRAM while not cached. For example, an attacker may directly inject stack variables to alter the victim VM's execution context or a heap object pointer to lead digest computation into using incorrect data. Depending on the goal, the attacker needs different levels of control over the victim's execution. An attacker intending to change the victim's run-time contexts often needs control, at least at the granularity of an instruction or basic block. Specifically, the attacker must be capable of performing a corruption exactly in between the execution of two specific instructions, not to mention that the target must be evicted from the cache in between. This level of preciseness requires the capability of single-stepping the victim VM execution [30].

We found that an attacker does not have the capability of achieving a high (single-instruction) or medium (few-instructions) level of control over the victim VM execution, as we detail in Table 1. A known way to precisely control a VM's execution is the repeated injection of interrupts and exceptions into the VM, as previous works [33, 52] demonstrate. They trigger the interrupts using APIC controller. SEV-SNP restricts such injections with its new interface, Restricted Injection [2], which prohibits interrupt injection through any vector other than the one for hypervisor (#HV). The medium precision control only requires the injection of faults by manipulating the Dynamic Voltage and Frequency Scaling (DVFS) interface as demonstrated in multiple works [27, 30, 40, 41], and the DVFS-based fault injection was

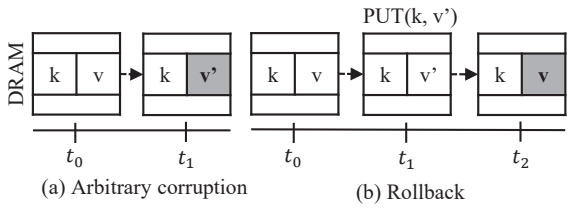

Figure 2: Possible attacks targeting key-value pairs on SEV-protected KVS through low-precision ADC. (a) arbitrary corruption, and (b) rollback

proven effective when targeting Intel SGX. However, a recent study shows that the attacker cannot generate such faults on SEV machines [43]. For these reasons, the *low-precision ADC* is the only viable means left for an attacker to corrupt the victim VM's memory content. With the limited control over the victim VM under SEV-SNP, the only remaining target viable to attackers are the ones that do not need such precise control. One of the examples that an attacker could corrupt is the key-value pairs in-memory KVS, which motivates us to devise a means to mitigate physical attacks that do not require precise control over the victim machine against the in-memory KVSs running on SEV-SNP.

3.3 ADC against KVSs

The regular data structure of in-memory KVS makes it remain susceptible to the low-precision ADC. An unpredictable change to arbitrary data, which is the best that an attacker can do with low-precision ADC, is likely to become a semantic error making the victim crash, but this is not the case for KVSs. An attacker could locate the key-value pairs that the victim KVSs contain by analyzing memory access patterns, and even an unpredictable and random change to them may not crash the KVS. The victim KVS is not likely to recognize this corruption because the corruption will only change the set of key-value pairs or the value bound to a particular key. Specifically, our work aims to defeat two possible attacks targeting the key-value pairs through low-precision ADC (as depicted in Figure 2) by assuring the integrity of key-value pairs in KVSEV running on a SEV-encrypted VM.

- **Arbitrary Corruption** An attacker corrupts a key-value pair (k, v) to arbitrary contents, *e.g.*, to (k, v'), (k', v), or (k', v'). Figure 2a illustrates one of these cases where the attacker corrupts only the value to change (k, v) into (k, v'). Note that they can not alter (k, v) to a *specific* pair of their choice because they have no access to the encryption keys.
- **Rollback** An attacker replaces (k, v') to an older version of the pair (k, v) as depicted in Figure 2b. For this, they maintain a copy of (k, v) and replay it after an operation updating v to v'. After the attack, the client receives an outdated value as the response to a request with the key k.

A straw-man solution to this limited integrity against the physical attacks is to protect the KVS integrity with software-based MAC by constructing a Merkle tree spanning the entire key-value pairs. Hoping to detect the corruption attacks, the KVS could update this Merkle tree on each write (*i.e.*, PUT) and verify its content on each read (*i.e.*, GET). However, this Merkle tree itself is susceptible to the low-precision ADC because the Merkle root is inevitably stored in external memory that the attackers can corrupt, as demonstrated in the next section §3.4.

3.4 Proof-of-Concept Attack to the Straw-man

To demonstrate the insecurity of the straw-man solution against the physical attacks, we implement a Proof-of-Concept (PoC) attack to the KVS that protects its key-value pairs with MAC using Merkle tree. Our PoC assumes that the adversary has the ability to manipulate both the KVS memory and the Merkle tree. Using this capability, we copy the original KVS data and Merkle tree on the initial PUT request from the client and use them to replay the outdated key-value pair on the occurrence of a GET request. To demonstrate the feasibility of our replay attack, we use the `DBG_ENCRYPT` and `DBG_DECRYPT` API provided by SEV to move SEV-protected data to a controlled destination, replacing KVS data and Merkle tree to outdated ones. Our PoC confirmed that the attack successfully bypasses the KVS integrity protection expected to be enforced through Merkle tree without being detected, as we could forge a valid Merkle tree and pass the MAC verification step. Although the `DBG` APIs can be disabled on SEV initialization, we believe that an adversary capable of performing ADCs can craft a similar means to obtain and corrupt the victim VM data as discussed earlier in §3.2.2. Note that, the attack mechanism is also valid on SEV-SNP machines as an adversary can bypass the RMP protection which is incapable of detecting data corruption through ADCs as discussed in §3.

3.5 Lack of Software Control on Cache

Due to the lack of built-in memory integrity protection in SEV, we must have a separate means to authenticate external memory content. A straightforward approach is to follow what SGX does for memory integrity. SGX computes a *Message Authentication Code* (MAC) of the protected memory content on each external memory write and uses the MAC for verifying the reads. To this end, SGX incorporates an integrity tree based on Bonsai Merkle Tree (BMT) [48] to compute the MAC efficiently [16]. This authentication process is believed to be crucial for protecting the integrity of data stored in an untrusted medium, such as the external memory in this case. Underneath the integrity protection

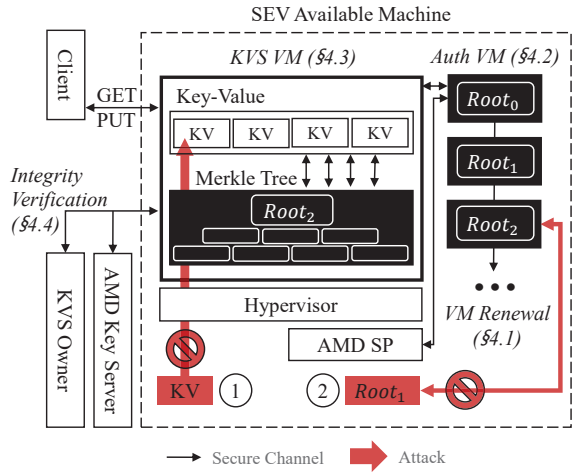

Figure 3: KVSEV Design Overview. KVSEV is protected from attacks that either ① corrupt key-value pairs or ② replace MT root to outdated value. New components introduced by KVSEV are colored in black.

with this procedure by SGX are two hardware components, the memory or register in which the MAC is kept and the hardware module authenticating each memory transaction.

This observation motivated us to find the hardware features of SEV-SNP that we can repurpose to play the roles of the two components, but we found that no hardware feature can directly fulfill the requirements. Modern CPUs do not provide a means to explicitly control the cached content with software, making it impossible to intercept and handle all external memory reads and writes. Instead, we devise a mechanism that emulates the MAC computation and verification procedure for each write (*i.e.*, PUT) to and each read (*i.e.*, GET) from a KVS by utilizing SEV-SNP's VM management protocol, as detailed in §4.

Note that SEV is not the only one that follows the design principle in which it does not authenticate all external memory reads and writes. For example, Intel announced the deprecation of SGX on the 11th and 12th generation of Core processors [23, 24] and started to ship machines that support a SEV-like feature called Total Memory Encryption (TME) [22]. We believe that KVSs running on such systems will also need a mechanism like KVSEV to become resilient against the low-precision ADCs.

4 DESIGN AND IMPLEMENTATION

Figure 3 gives an overview of KVSEV, which is composed of two VMs. At the heart of KVSEV is the *Authentication VM (Auth VM)* that provides the other, the *KVS VM*, with the integrity-protected store for the hash (*i.e.*, the Merkle Root) of the KVS. The Auth VM securely stores a small amount of data and raises the alarm for any data corruption by sending a report to the KVS owner and the clients. KVSEV prevents the attackers from modifying the data in the Auth VM into an attacker-controlled value by making each instance of Auth VM short-lived (§4.1) and free from external data (§4.2), which an attacker could control. The short lifetime of each Auth VM prevents the attacker from replaying older Merkle roots. The absence of external data limits the chance for an attacker to implant the target value into the Auth VM in an attempt to exploit the Auth VM as an encryption oracle. The KVS VM handles requests from clients by executing typical in-memory KVS operations and verifying the integrity of key-value pairs using its Merkle tree with the help of the Auth VM's secure data store. The verification by KVS VM detects any corruption of the key-value pairs because the Merkle root computed from the corrupted KVS will not match the one contained in the Auth VM (§5.4.1). To avoid this, the attacker must also corrupt the Merkle root in the Auth VM. However, the protection scheme of KVSEV using a series of fresh Auth VMs prohibits such an attack (§5.4.2).

The KVS VM in KVSEV handles a GET request by first obtaining the key-value pair and then verifying its integrity using the genuine Merkle root protected by the Auth VM (§4.4). For each PUT request that it handles, KVSEV creates a new VM to store the new Merkle root reflecting the inserted or updated key-value pair and disposes of the older one (§4.5).

4.1 Renewing the Auth VM

The Auth VM securely renews itself whenever needed (*e.g.*, upon each Merkle root update) to make each instance short-lived. This renewing procedure fulfills the desired property in which an attacker becomes incapable of using encrypted values collected from an older Auth VM against the new one. On each renewal, the current Auth VM (VM_{curr}) creates the next Auth VM (VM_{next}), which accommodates the next Merkle root as depicted in Figure 4. This renewal prevents the attackers from using the data from VM_{curr} against the new Auth VM, VM_{next} because the two VMs use different encryption keys.

KVSEV can securely perform this renewal procedure without being hindered by privileged attackers, utilizing the SEV-SNP's VM management protocol and AMD SP. To renew itself, VM_{curr} invokes a sequence of calls directly to SP following the protocol that the SEV API defines. Upon its successful creation, VM_{curr} is given a secure channel to VM_{next} by SP and uses the channel to help the KVS VM establish a secure channel with VM_{next}. At the same time, VM_{curr} obtains the attestation report for the authenticity of VM_{next} and reports to the KVS owner for any anomaly. Upon receiving a valid report, VM_{curr} notifies the KVS VM of the success so that the KVS VM sends the new Merkle root to VM_{next}. VM_{curr} also sends the address given by SP on launch time

for its identification to VM_{next} for the verification of the termination. After the successful transmission of the Merkle root, VM_{curr} terminates, making itself not runnable on the platform, and requests the hypervisor to shut it down. To serve this request, the hypervisor issues a decommission request to SP, which deletes the memory encryption key and all other internal states that are associated with the VM, VM_{curr} here. VM_{next} verifies that VM_{curr} is decommissioned by sending a guest message directly to the SP with a VM_{curr}-specific address received with the Merkle root as one of the message fields. The attempt to send the message is expected to fail with `INVALID_PAGE_STATE` status code from SP, if VM_{curr}-specific key and context have been successfully deleted through the decommission. After verifying the proper decommission of VM_{curr}, VM_{next} notifies the KVS VM that it has successfully received the Merkle root.

Reducing Auth VM Size. The latency in creating a new Auth VM should be reduced to a minimum as it significantly impacts the latency of PUT requests and becomes a performance bottleneck, as shown in §5.2. KVSEV reduces the size of Auth VM based on an observation that Auth VM does not perform complex operations or operating system services, and booting and initializing the operating system remains a significant bottleneck despite the rich line of work on reducing VM launching latency [34, 35, 46]. The only functionality that Auth VM needs is a minimal cryptographic library to establish and communicate over the secure channel because Auth VM primarily serves as a storage for the Merkle root. Accordingly, we built the Auth VM by directly using a minimal KVM-based VM example [54], reducing the VM renewal latency significantly.

Limiting the Hypervisor's Privilege to Auth VMs. SEV's VM management protocol enables VM_{curr} to specify how the SP should initialize VM_{next} when VM_{curr} requests AMD SP to create a new Auth VM (*i.e.*, VM_{next}), to limit the set of actions that the hypervisor can take on VM_{next}. Specifically, KVSEV disables two policy bits, `DEBUG` and `MIGRATE_MA`, during the renewal of the Auth VM to protect VM_{next} from potential malicious activity by the hypervisor. Disabling the `DEBUG` policy bit prevents the hypervisor from accessing a set of debug APIs and thereby secures the contents of the Auth VM, such as the Merkle root, from being decrypted. Turning off the `MIGRATION_MA` policy restricts the hypervisor from initiating unintended migration of the Auth VM, which if replayed by an attacker, could result in a valid attacker-chosen Auth VM. If `MIGRATION_MA` bit is disabled, the hypervisor cannot trigger migration as the policy enforces Auth VM to initialize migration on its own directly to AMD SP.

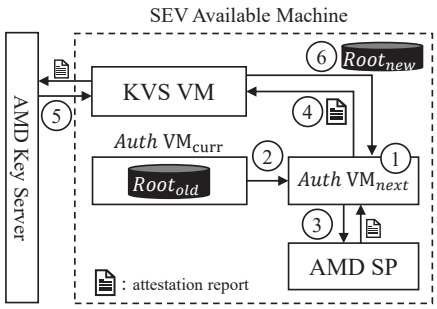

Figure 4: VM renewal procedure. ① KVS VM requests a new Auth VM, ② *VM_curr* identification address is sent, ③ *VM_next* requests attestation report and verifies termination of *VM_curr*, ④ attestation report is sent, ⑤ report is verified, and ⑥ updated MT root is exchanged.

4.2 Limiting the Auth VM Interfaces

KVSEV hinders attackers from leveraging the Auth VM as an encryption oracle for controlled data injection by carefully limiting the Auth VM's communication interfaces. First, the Auth VM stores the incoming data from outside (*e.g.*, from the KVS VM) on shared memory that is not encrypted using the Auth VM's memory encryption key. This data placement prevents an attacker from sending a controlled packet to the Auth VM's external interface to harvest the plaintext-ciphertext pairs for later data injection. This design choice does not sacrifice communication security because the packets on the shared memory are already encrypted and authenticated using the channel keys. Second, the Auth VM also randomizes the location where it decrypts the incoming packets. An adversary must pinpoint the location of decrypted packets to harvest the result of encrypting a desired value by the Auth VM, but the randomization prevents the adversary from reliably locating the decrypted packets during the Auth VM's short life time.

4.3 KVS VM

The second VM that KVSEV composes, the KVS VM, is a regular VM that runs a modified in-memory KVS. The KVS it runs is modified to compute hash using Merkle tree from its content on each request as detailed in the following sections (§4.4 and §4.5). When KVSEV starts up, the KVS owner authenticates the KVS VM using the remote attestation protocol for an encrypted VM on SEV-SNP. This protocol ensures that the genuine KVS VM boots up on the remote machine. Subsequently, the KVS VM creates the first Auth VM, receives the Transport Layer Security (TLS) protocol certificate, and waits for the client's requests. The clients use TLS protocol to authenticate and protect the requests and responses.

4.4 Handling GET Requests

The KVS VM handles a GET request by responding with the requested key-value pair after verifying the KVS integrity. Inside the KVS VM is a modified in-memory KVS that performs the integrity verification using a software-implemented Merkle tree and its usual GET processing. The modified KVS computes the Merkle root from its content at the moment and sends it to the Auth VM. The Auth VM responds to the KVS VM with the signed attestation only if the computed Merkle root matches the correct one found inside the Auth VM. This signed attestation is sent back to the client who requested the key-value pair so that the client can verify the authenticity of the response. This integrity verification with software Merkle tree backed by Auth VM's Merkle root protection prevents the attackers from corrupting key-value pairs in KVSEV into random attacker-chosen ones, as we further analyze in §5.4.2. The drawback of this verification step, which is essential for integrity guarantee, is the additional delay it causes in the critical path of the request handling. To overcome this, we make the integrity verification become asynchronous, similar to the deferred verification proposed earlier in [5].

Asynchronous Integrity Verification. KVSEV hides the additional delay from integrity verification by removing the verification from the critical path, using the KVS VM as a relay. A client placing a PUT or GET request does not wait for the proof for each request. Instead, the KVS owner receives the proof for each transaction and only forwards it to the client when requested. Specifically, upon receiving the proof (*i.e.*, attestation report), the KVS VM first sends the PUT/GET requests' results to the client rather than waiting for the response of the proof's verification from the AMD key-server. Upon receiving the verification of the proof, the KVS VM forwards the results to the client posterior to sending the PUT/GET requests' results.

Associating Attestation with Request. KVSEV associates the attestation report from Auth VM with the `PUT/GET` operation to track (proof, operation) pair even when the proofs are requested asynchronously. To track the `PUT/GET` operation that requested the proof, KVSEV leverages `REPORT_DATA` feature in SEV's attestation report generation. Specifically, `REPORT_DATA` feature enables Auth VM to provide certain data along with the attestation report when requesting AMD SP for the report. When Auth VM receives a request for the attestation report, Auth VM provides the operation (*i.e.*, PUT/GET) as input data for report generation, which KVSEV can use to find the operation corresponding to the proof.

4.5 Handling PUT Requests

The KVS VM starts to handle an incoming PUT request by first locating where it must place the new key-value pair.

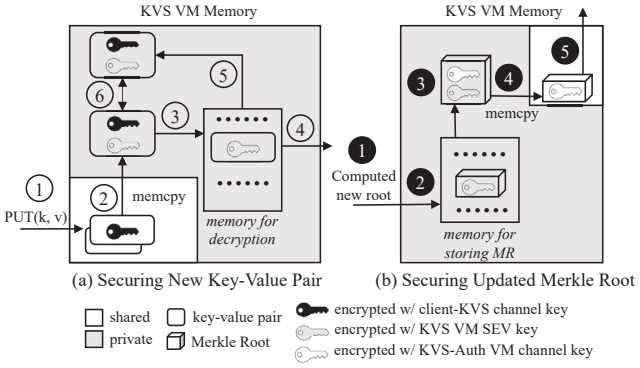

Figure 5: (a) Ensuring the integrity of the new key-value (KV) pair. ① New KV is placed into the unencrypted shared memory of KVS. ② KV is copied into KVS private memory. ③ KV is decrypted at a random location with KVS-client channel key. ④ KV is accessed to update MT. ⑤ KV is re-encrypted with KVS-client channel key. ⑥ Re-encrypted KV is compared with the original KV. (b) Protecting new Merkle root (MR). ❶ KVS computed new MR. ❷ MR is stored at a random location. ❸ MR is encrypted with KVS-Auth VM channel key. ❹ Encrypted MR is copied to KVS shared memory. ❺ New MR is sent to Auth VM.

Subsequently, the KVS VM renews the Auth VM (§4.1) by requesting the hypervisor to create a new one that will inherit the current Auth VM. While the Auth VM renewal is in progress, the KVS VM computes the new Merkle root reflecting the new key-value pair and saves it in the Auth VM when the new one becomes ready. Underneath these steps is the process of authenticating the new Auth VM from the KVS VM with the help of the VM management protocol and SP. This provisioning delay potentially increases the PUT latency significantly.

Eager Provisioning. KVSEV eagerly creates new Auth VMs to hide the cost of VM creation and initialization when handling a PUT request. The observation behind the design is that the replay resistance of Auth VMs is maintained independently from the VM creation, as long as KVSEV disposes of old ones during the renewal process before storing the new Merkle root in the new Auth VM. An adversary cannot leverage an empty Auth VM, which does not contain any data useful to the adversary in any way to perform an attack. On top of this, KVSEV creates a separate thread that continuously requests to launch a new Auth VM until a certain number (*e.g.*, 500) of Auth VMs are prepared. We used 500 as the limit because the maximum number of SEV-protected VM that a processor can run is 512 in our prototype, and §5.2 presents the impact of choosing different bounds.

Integrity of New Key-Value Pair. KVSEV ensures the integrity of the new key-value pair sent by PUT request

after Merkle tree root is sent to the Auth VM. However, the decrypted pair at rest *awaiting* for Merkle tree computation is still susceptible to corruption. KVSEV secures such an attack window by incorporating the following mechanisms. KVSEV prevents an adversary from changing the key-value pair into an uncontrolled one before Merkle tree computation by comparing the re-encrypted pair with the original pair in the PUT request right after accessing the pair for computation, as depicted in Figure 5a-⑥. Any arbitrary corruption will result in a mismatch because the attacker cannot craft a correctly re-encrypted pair without the secure channel key between the client and the KVS VM. Note that even the physical attack does not enable to the attacker to obtain the channel key in the KVS VM in plaintext. KVSEV also prevents the adversary from replacing a key-value pair into its older version before the Merkle tree computation by randomizing the location of request decryption for every PUT request (Figure 5a-③), similar to the strategy taken by Auth VM discussed in §4.2. Under SEV's encryption mechanism, replaying the encrypted pair at a different location will cause differently decrypted contents. As a result, the replayed key-value pair will be decrypted into a different, arbitrary pair after the decryption by the KVS VM. The KVS VM recognizes this by comparing the re-encrypted pair as explained previously.

Protecting New Merkle Root. KVSEV prevents the attacker from corrupting the newly computed Merkle root *before* the KVS VM sending it over to renewed Auth VM (*i.e.*, while the updated root resides in KVS VM). Note that corrupting the Merkle root into an arbitrary value is counterproductive for the attacker because it results in a root mismatch on the next GET request. Therefore, it is enough for KVSEV to prevent the attacker from replaying previously valid Merkle root only. KVSEV does this by randomizing the location of the root on every calculation as depicted in Figure 5b. Replaying the root at a changed location will cause it to be decrypted into a different, unknown root, resulting in the same consequence as arbitrary corruption.

Protecting Merkle Root Calculation. KVSEV ensures the integrity of Merkle root by safekeeping the data in Auth VM, but the *remainder* of Merkle tree (*i.e.*, intermediate nodes) residing on KVS VM still remains susceptible to replay *during* new Merkle root computation. Specifically, replacing intermediate nodes during the calculation of the new root will result in a successful update of the root to a valid value corresponding to the replayed Merkle nodes, while replacement during an idle state of the KVS will result in a root mismatch, as the attacker lacks the means to alter the root stored in the Auth VM. KVSEV prevents an attacker from replaying Merkle nodes during new root computation by reserving one register to hold the node value before spilling it onto the memory. When the node is read back to register from memory to resume calculation, KVSEV compares the value with the one stored in the reserved register, thereby detecting any modification or tampering of intermediate Merkle tree nodes during the calculation. Note that most intermediate results of calculation (*i.e.*, hash) stays in registers, safe from replay, and only a small subset of nodes are spilled onto the memory, vulnerable to replacement.

Components	Modified	Added	Total
Memcached 1.6.12	21	1027	1048
Linux Kernel (Ubuntu)	-	219	219
Minimal KVM VM	-	348	348
Total	21	1594	1615

Table 2: Components of KVSEV and their complexities in terms of their lines of code (LoC). All three components are written in C.

4.6 Implementation

KVSEV is composed of an in-memory KVS running on one Linux-based VM (KVS VM) and a small bare metal program running on another VM (the Auth VM). Table 2 shows the number of lines of code (LoC) that we added or modified to implement KVSEV. KVSEV's KVS is an extension of Memcached [36] version 1.6.12. We modified Memcached to execute Merkle tree updates and Merkle tree verification when handling `PUT` and `GET` requests, respectively. Specifically, we extend Memcached's data structure to hold hash and insert custom Merkle tree APIs (*i.e.*, adding and verifying hash entries) at the start of request handling. The Linux kernel for KVS VM and the hypervisor are modified to emulate SEV-SNP machine because we evaluate KVSEV on a SEV-ES machine. As the baseline, we use the kernel obtained from the AMD's official repository for Linux kernel [3]. The Auth VM consists of a small codebase to support basic functionalities that store and compare the Merkle Root. As discussed earlier, we build the Auth VM on top of a minimal KVM VM example [54] that executes a small piece of assembly code in a VM. On top of this, Auth VM contains a small cryptographic library (TinyCrypt [9]) to provide an essential set of primitives to build secure channels with external entities.

5 EVALUATION

Experimental Setup. We run KVSEV on an AMD system with SEV, which has AMD EPYC™ 7262 as its CPU and 64GB of main memory for performance measurement. As its network interface card (NIC), this system uses Intel I350 Gigabit Ethernet Controller (rev 01). The machine runs Ubuntu 20.04.3 LTS with Linux kernel 5.13.0-21 as the host (hypervisor) and Ubuntu 18.04.6 LTS 64bit with Linux kernel 4.15.0-163 as the KVS VM. As discussed earlier in §4.1, the Auth VMs do not run an operating system. For the comparison with ShieldStore [29], which is the state-of-the-art in-memory

KVS on confidential computing, we use a machine with Intel Core™ i9-10900K CPU and 128GB memory. We inevitably use different machines for the comparison study because KVSEV requires AMD's hardware extension and ShieldStore requires the Intel's. For the same reason, we use normalized throughput for the performance comparison, using the throughput of Memcached [36] on each system as the baseline.

Client. We measure the performance of KVSEV in two configurations depending on how the client sends requests to KVSEV. The first is the standalone setup in which the client runs on the same machine with KVSEV. The client generates the workload within the KVS VM and sends the requests over the KVS VM's network stack. We measure the performance impact of KVSEV mostly on this setup to rule out the network communication overheads and thus stress-test the direct impact of using KVSEV on performance. We additionally evaluate the networked setting with a separate client machine connected through networks via 1Gb Ethernet.

Workloads. We use two workload patterns (uniform and zipfian) that the prior work [29] also used to measure the performance. The keys of requests in the uniform workload follow the uniform distribution, and those of zipfian follow the zipf distribution of skewness 0.99, as in YCSB [55]. The key size is fixed to 16B, while the value size, initial KVS size, and the ratio of reads vary. The KVS that is initialized to have 10–160 million key-value pairs before the measurement, and the client sends 1 million requests.

5.1 Throughput

Figure 6 shows the absolute throughput of KVSEV for various workload distributions (*i.e.*, read ratio) when accommodating 10 million key-value pairs. The main trend the results reveal is that the throughput of KVSEV increases as the ratio of writes (PUTs) in the workload decreases. This is because KVSEV needs to conduct more VM renewals when handling an increased number of PUTs. Compared to KVS on native SEV VM, KVSEV demonstrates up to a 64.23x slowdown when handling a workload with 10% reads (R10), which decreases to 13.38x when handling R90. Later in this section (§5.3), we show that KVSEV is practical despite its performance overheads when compared to other state-of-the-art secure KVSs [29].

Value Size. The value size does not affect the performance of KVSEV as shown in Figure 7. To observe the impact of value size, we use two different workload patterns (uniform and zipfian). We also present if the impact of value size varies as the ratio of writes in the request increases. KVSEV runs with four threads, and the KVSs initially have 10 million key-value pairs. As the result shows, the value size does not significantly affect the performance of KVSEV, while the slowdown becomes higher as the value size increases for the

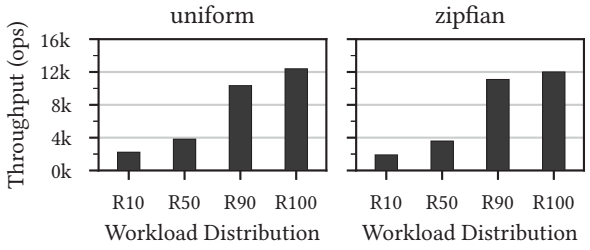

Figure 6: Throughput of KVSEV in operations per second (ops) over various workload distributions.

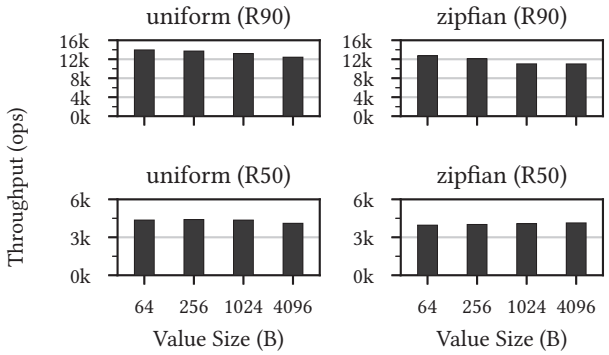

Figure 7: Throughput of KVSEV under different workload distribution with varying value sizes in kops.

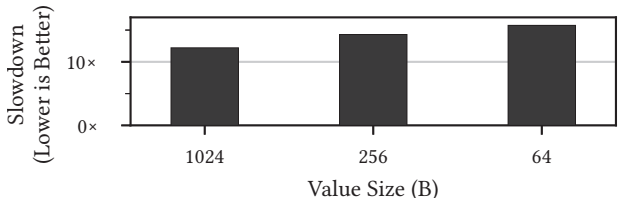

Figure 8: Normalized throughput of KVSEV with fixed KVS size and different (value size, number of entries) pairs.

workload with 90% reads (R90). The higher overhead when accommodating larger values is primarily due to the larger total size of key-value pairs. As the value size increases, the total size of KVS also increases in this measurement because we fixed the number of entries the KVS initially has. This increase in total KVS size negatively affects the throughput in this measurement. To back this analysis, we also measure the throughput when the value size varies but the size of KVS does not. To fix the KVS size to about 10GB, we set the pair of the number of entries and value size to (160M, 64B); (40M, 256B); and (10M, 1024B). Figure 8 shows the result when tested with R90 workload. As anticipated earlier, the throughput of KVSEV does not decrease as the value size increases.

Multi-thread Scalability. Figure 9 shows the throughput of KVSEV when running with a varying number of threads. The throughput of KVSEV does not scale with the number of

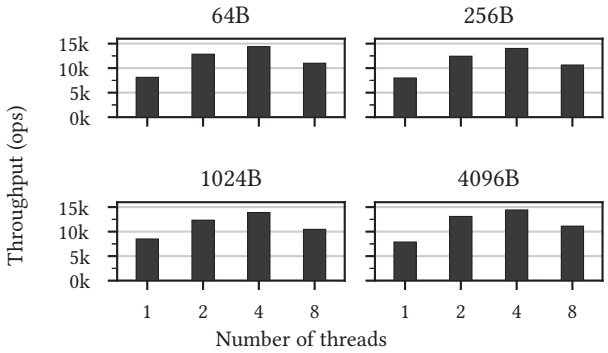

Figure 9: Throughput of KVSEV with varying number of threads with R50 workload for different value sizes.

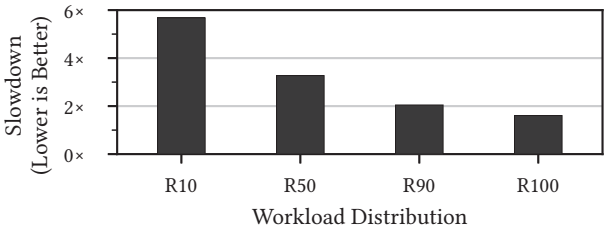

Figure 10: Normalized throughput of KVSEV when receiving requests over the network in ops.

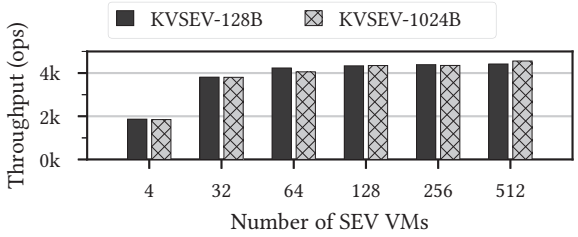

Figure 11: Performance of KVSEV with different number of eagerly created Auth VMs under R50 workload using 4 threads.

threads due to the limited throughput of Auth VM creation and the synchronous Merkle tree handling. As §5.2 shows, KVSEV can create about 2160 Auth VMs every second, limiting the number of PUT requests that KVSEV can handle every second to 2160. This makes 20.21kops as the theoretical upper bound of throughput when running a workload with 90% reads.

Networked Throughput. Figure 10 shows the normalized throughput of KVSEV running the uniform workload when it runs with four threads. As expected, the network communication overheads amortize the overheads incurred by KVSEV. For example, KVSEV only shows 2.0x slowdown when running the workload of 90% reads with 128B values, However, as observed in the standalone evaluations, the slowdown increases along with the ratio of writes in workloads as VM-renewal overheads become the bottleneck, showing a 3.27x slowdown on the workload with 50% reads.

Table 3: The breakdown Auth VM renewal latency in Figure 4 without asynchronous verification.

VM Renewal Step	Time (us)	Proportion
New Auth VM request	12	2.2%
VM_curr identification	12	2.2%
VM_next attestation report request	61	11.2%
Attestation report & Merkle root	10	1.9%
Report verification	448	82.5
Total	543	100.0%

5.2 Impact of Design Decisions

Impact of Debloating. To show the benefit of debloating the Auth VM in VM renewal throughput, we measure the VM creation time taken from `LAUNCH_START` command, which initializes the launch process, to `LAUNCH_FINISH` command, which finalizes the process before `VMRUN` for the Auth VM and Linux VM, respectively. The time it takes to bootstrap and set up the Auth VM after `VMRUN` is not included. We use the same version that KVS VM runs with as the Linux VM. Our effort to debloat the Auth VM improved the VM creation throughput from 151.1VMs/s to 2156.3VMs/s, by about 14×. As discussed earlier, this significantly contributed to the throughput of KVSEV in that the VM renewal throughput bounds the number of PUT requests that KVSEV can handle every second.

Eager VM Creation. Figure 11 shows the throughput of KVSEV with a varying number of pre-created VMs running workloads of two different value sizes. This result shows that eager VM creation is indispensable to achieving the desired performance. Compared to the minimal setup, which is four VMs for four threads, our optimization creating VMs eagerly improves the throughput by up to 2.3×. The result also suggests that KVSEV does not need to maintain too many VMs. The performance quickly saturates between 64 and 128 VMs, suggesting that having around 100 VMs is enough in our system. Note that the performance increases slightly as the number of eagerly created VMs increase beyond 128 due to the less number of executions of `WBINVD` and `DF_FLUSH` instructions which are required when all available SEV VMs are in use.

Impact of Asynchronous Verification. Figure 12 shows the throughput of KVSEV and its variant that does not use asynchronous verification. The variant not using asynchronous verification completes handling a request only after the VM attestation, suffering from even longer verification latency. The asynchronous verification improves the throughput of KVSEV by 5.7x by hiding the long latency of the VM verification procedure. Table 3 shows the latency of each step in the Auth VM renewal procedure depicted in Figure 4 and explained in §4.1. The table shows that the report verification overhead consists of over 80% of the total time taken for VM

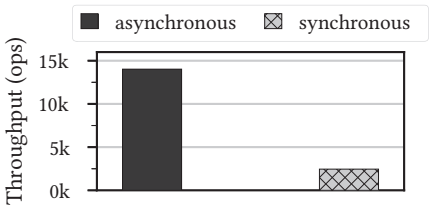

Figure 12: Performance of KVSEV with and without asynchronous verification with four threads under R50 workload and 128B value size.

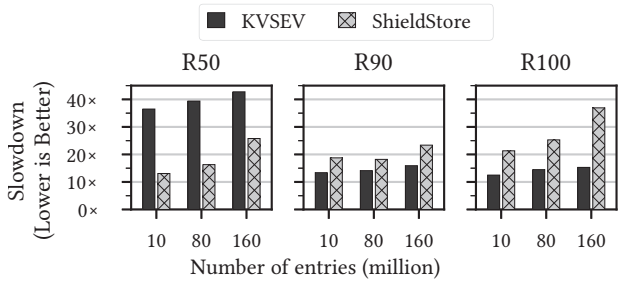

Figure 13: Normalized throughput of KVSEV and ShieldStore with different KVS size and read ratio. For fair comparison, we use the same distributions for workload, R50, R90 and R100, as ShieldStore [29].

renewal when asynchronous verification is not used. The results suggest that the report verification overhead is the primary bottleneck when conducted synchronously, and the asynchronous verification is another essential optimization.

5.3 Comparison to an Existing Secure KVS

Figure 13 shows the slowdown of KVSEV and ShieldStore when running with three workload distributions, three initial KVS sizes, and 128B values. As discussed earlier, we present the normalized performance using the insecure MemCached as the baseline due to the inevitable difference in machine configurations. The results show two trends in the slowdown by KVSEV and ShieldStore. First, the overhead of KVSEV does not increase as the KVS size increases, while ShieldStore slows down as the KVS size increases. This widens the performance gap between KVSEV and ShieldStore when running with a large KVS. When holding 160 million key-value pairs, whose size is about 21GB, the slowdown of KVSEV is about 1.47x lower than that of ShieldStore when running the workload with 90% reads. Second, KVSEV's overhead is lower for workloads with more reads (i.e., R90) due to the decreased number of required VM renewals. These observations suggest that AMD-based KVSEV would be more advantageous than SGX-based ShieldStore when an in-memory KVS should hold large key-value pairs.

5.4 Security Analysis

This section discusses how KVSEV detects or prevents the attempts to corrupt its key-value pairs. We defer the discussion about the attacks on the other memory content, such as KVS VM code, to the next section (§6).

5.4.1 Uncontrolled Corruption against the KVS VM. An attacker who has physical access to the machine and controls the hypervisor may attempt to corrupt the key-value pairs by overwriting a memory location containing a key-value pair with an arbitrary value. The attacker is assumed to be capable of locating the target by observing the memory access pattern and the network transactions together. Physical access enables such an attacker to watch the external memory transactions, and the control of the hypervisor gives the capability of learning the network usage pattern. Without the protection by KVSEV, such an attacker can mislead the KVS to associate an incorrect value with a key.

KVSEV detects such an attack while comparing the hash of the key-value pair and the one stored in another location in the KVS VM, as an internal node of the Merkle tree. Even if the attacker correctly corrupts the internal node as well, KVSEV continues toward the Merkle root and detects the attack from the mismatch in the Merkle root. The attacker cannot manipulate the KVS VM beyond this and trick the KVS VM either into skipping the Merkle root comparison or using an incorrect one for comparison because the attacker cannot modify the KVS VM memory to a value of the attacker's choice. We discuss why the previously reported exploits modifying the memory with the controlled values do not work in §6.

5.4.2 Integrity of Merkle Root in the Auth VM. Another chance for an attacker is to modify the Merkle root in the Auth VM in an attempt to bypass the protection of KVSEV. To such an attacker, the corruption of the Merkle root to an arbitrary value is not an option. Doing so causes the KVS VM to detect the mismatch because the attacker cannot control which value the Merkle root will be modified to and cannot calculate the desired Merkle root value. Therefore, such an attacker has no choice but to perform the replay attack in which it retains a previous Merkle root, replay the KVS VM using its snapshot, and reuse the retained Merkle root so that the KVS VM does not recognize that it is being replayed maliciously. KVSEV prevents such an attempt with its VM renewal, as described in §4.1. Any old Merkle root value, in an encrypted form, will be of no use because, after one or more PUT requests, KVSEV disposes of the VM that contained the old Merkle root. If the attacker replaces the Merkle root of the current Auth VM with the retained value, the current Auth VM will decrypt the attacker-injected value

into an unpredictable one, which is highly unlikely to match the desired Merkle root.

5.4.3 Intervening with Auth VM Renewal. Another option for an attacker is to use its capability to control the hypervisor to intervene with the VM renewal of KVSEV and enforce KVSEV to use one Auth VM repeatedly. This repeated use of one Auth VM will enable the attacker to replay the Merkle root, allowing the replaying of the KVS contents undetected as the root value will match the replayed data. An adversary can make this happen in one of the following two ways. First, an attacker may migrate the Auth VM to have two identical instances simultaneously. While the standard renewal procedure continues on one instance, the other can stay idle to retain the previous Merkle root. On a request for verification from the KVS VM, the attacker will relay the request to the Auth VM holding the previous Merkle root. KVSEV prevents this attack by disabling migration as a policy during launch as explained in §4.1. The VM policy is included in the generated attestation report, which is later verified to detect any misconfiguration that may lead up to the attack mentioned above. Second, an attacker may intercept a disposal request from the Auth VM to subvert disposal and preserve the outdated Merkle root stored in the Auth VM. An adversary will try to perform a replay attack using this preserved root on a verification request. KVSEV prevents such an attack by validating the correct disposal of old Auth VM during the VM renewal process as explained in §4.1. The mechanism is immune to hypervisor intervention because KVSEV performs the validation through a secure channel between VM and AMD SP provided by guest message protocol.

6 DISCUSSION

6.1 KVS VM Code Integrity

The security guarantee that KVSEV brings relies on the code integrity of the KVS VM. If an attacker successfully manipulates the KVS VM code, they can skip the integrity check that the KVS VM performs before it responds to the requester with the obtained key-value pair. To manipulate the KVS VM code in a controlled way, the attacker must be capable of modifying a SEV VM memory to a controlled value. The lack of integrity protection against physical attackers leaves a memory corruption attack undetected, but the attacker cannot control which value the memory content will be decrypted. Only an attacker who can forge a correct ciphertext for the desired plaintext can predictably control the result of memory corruption. The KVS VM is susceptible to such an attack that manipulates SEV VM memory with chosen values but only from an attacker who exploits a vulnerability in the design of SEV. Recent studies have reported a couple of such vulnerabilities, but the latest release of SEV is known to be appropriately patched, and the demonstrated attacks do not work anymore. Consequently, we can assume that an attacker cannot modify KVS VM memory in a controlled manner and thus cannot predictably modify the KVS VM code to nullify the integrity protection of KVSEV.

6.2 Attacks on the Other Data in KVS VM

While KVSEV verifies the integrity of key-value pairs, other data that the KVS uses are not protected. For example, an adversary might modify a condition variable to a specific value to subvert the security checks or modify the pointer variables to change the KVS execution flow. To perform a meaningful attack, an attacker must modify the data to a controlled value at a specific time, as random corruption will only cause unexpected behavior. An attacker can make such a controlled modification by changing data stored in memory or registers. An attack changing the data stored in memory was demonstrated by Radev et al. [44], and the attack exploits the `Reserved` bit in NPT entry during the MMIO or nested page fault exception to forge the MMIO region and exfiltrate data such as AES key. However, this is no longer possible as an immediate patch [47] that implements `#VC` handler to check whether the accessed page is encrypted or not when receiving an MMIO or nested page fault exception was applied after the disclosure of the attack. Additionally, data corruption using the same steps to inject arbitrary code proposed by [39] is also not applicable for the reasons that we described in the previous section(§6.1).

6.3 Possible Alternatives for Auth VMs

Reserved Registers. The first option is to reserve one or more general-purpose registers to store the Merkle root. KVSEV would update the register while handling `PUT` requests (*i.e.*, re-calculating Merkle root) and read it when handling `GET` requests. The content remains out of the reach of attackers because all programs running inside the KVS VM are recompiled not to use the reserved register as a general-purpose register. What makes this approach less desirable than KVSEV's Auth VMs is the need for thorough recompilation. Not only the KVS, but also all the software components that may ever run within the KVS VM must be recompiled to reserve the particular register.

Randomizing the Location of Merkle Root. Another potential method is to randomize the address of the Merkle root whenever it is updated. Randomization cannot prevent the privileged attackers to locate where Merkle root is, due to the fact that malicious hypervisor can infer information about guest VM's memory assignment by monitoring NPT entries through manipulating permission bits and causing `#NPF`s, which was proven effective to gather location of VM's

critical data in multiple attacks [19, 32, 37, 38]. Instead, randomization causes the new Merkle root to be encrypted with different tweak, thwarting the replay of old root to the new location. The challenge of taking this approach is in the fact that KVS VM can only randomize the location at the guest physical address space while the actual tweak is determined by the host physical address, or machine address. As a result, an attacker may repeatedly map the guest physical page containing the Merkle root to the same host physical page, regardless of the actual guest physical page base address. This still allows the KVS VM to randomize the Merkle root location within the page, but the number of slots inside a page is severely limited, and the KVS VM will inevitably place the Merkle root at a conflicting location from which the attacker has previously collected a Merkle root.

Periodic Re-encryption. The third of the potential alternatives is to store encrypted root (on top of transparent encryption provided by SEV) inside KVS VM and periodically re-encrypt the root with different keys, thus having similar effect as Auth VM renewal. The limitation of this approach lies in the possibility of replaying the cryptographic key, which resides in the KVS VM's memory. While encryption keys for Auth VMs are safekept via inherent mechanism of SEV and AMD SP, re-encryption requires keys to be stored in KVS VM, thereby allowing an adversary to replay the encryption key and Merkle root at the same time.

Monotonic Counter. Utilizing secure monotonic counters [20, 21, 51] is another option. Secure monotonic counters provide tamper-resistant counter values that cannot be reverted to previous values once incremented. Such counter values can be used as a salt for encrypting the Merkle root and incremented after encryption, thus preventing the root rollback as decryption with different counter value will result in an invalid Merkle root. Unfortunately, the resolution of existing secure monotonic counters is not high enough. The synchronous trusted counter of SGX can increment only once every 60ms [7], thus limiting the rate of Merkle root updates to once per 60ms.

7 RELATED WORK

Secure In-memory KVSs. Existing secure in-memory KVSs are built on Intel SGX to utilize its security guarantee that ensures the KVS integrity against both software and physical attacks. ShieldStore [29] alleviates the overheads of securing all key-value pairs inside an enclave by only maintaining the hash of the pairs inside it and placing the encrypted data outside (i.e., unprotected memory). Avocado [6] shows that a secure in-memory KVS can be accelerated and expanded to the distributed environment with direct I/O networking, well-established protocols, and optimized data structures. Concerto [5] introduced a verification method called *deferred verification* to reduce the overheads from using Merkle Tree-like data structures, and FastVer [4] extended the idea to make it faster by combining advantages of both Merkle Trees and deferred verification. KVSEV is similar to these studies in that they all aim to provide secure in-memory KVS by utilizing hardware-assisted TEEs. However, KVSEV differs from the other approaches in that it is the first to leverage SEV as a TEE. KVSEV focuses on providing the missing security guarantees rather than performance optimizations because SEV inherently provides different security guarantees than SGX. Speicher [7] and Tweezer [28] are another secure KVSs protected with SGX, but it aims to adapt a log-structured merge tree-based, persistent KVS, unlike KVSEV designed as an in-memory KVS.

8 CONCLUSION

To the best of our knowledge, this study is the first attempt to design an AMD SEV-based in-memory KVS that is resistant to the types of ADC that are capable of corrupting and/or replaying key-value pairs. Under the observation that software-only measures cannot be adopted as a drop-in solution as software-protected components are also vulnerable to ADCs, KVSEV leverages a sequence of VMs to provide trusted storage for the cryptographic hashes of key-value pairs. Combining VMM and attestation protocol provided by SEV-SNP for safe deployment and termination of hash-holding VMs, KVSEV protects the integrity of key-value pairs against ADCs. Our evaluation shows that KVSEV secures in-memory KVSs on SEV with a performance overhead comparable to or better than existing secure in-memory KVS solutions when running large KVSs.

ACKNOWLEDGMENTS

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government, Ministry of Science and ICT (MSIT) (NRF-2022R1F1A1076100 and NRF-2023-00277326), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2023, and Inter-University Semiconductor Research Center (ISRC); in part by Institute of Information & Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-01840, Analysis on Technique of Accessing and Acquiring User Data in Smartphone, and No.2021-0-00528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box); and in part by the MSIT under the ITRC (Information Technology Research Center) support program (IITP-2023-2021-0-01817) supervised by the IITP, and Samsung Electronics Co., Ltd.

REFERENCES

[1] AMD. 2016. AMD Secure Encrypted Virtualization (SEV) - AMD. https://developer.amd.com/sev/. Accessed: 2022-03.

[2] AMD. 2020. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[3] AMDESE. 2016. AMDESE_Linux. https://github.com/AMDESE/linux. [Online; accessed 2022-03].

[4] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, et al. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data*. 89–101.

[5] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 251–266.

[6] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, Pramod Bhatotia, et al. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 65–79.

[7] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 173–190.

[8] Rodrigo Branco and Shay Gueron. 2016. Blinded random corruption attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 85–90.

[9] Intel Corporation. 2017. TinyCrypt - TinyCrypt Cryptographic Library. https://github.com/tinycrypt. [Online; accessed 2022-03].

[10] A Micro Devices. 2006. AMD64 architecture programmer's manual volume 2: System programming.

[11] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. 2017. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090* (2017).

[12] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. 2010. Can you still trust your network card. *CanSecWest/core10* (2010), 24–26.

[13] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. 2009. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science IV* (2009), 1–22.

[14] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. 2006. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *Proceedings of the 43rd annual Design Automation Conference*. 506–509.

[15] NCC Group. 2018. TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus. https://www.nccgroup.com/globalassets/about-us/us/documents/tpm-genie.pdf. Accessed: 2022-11.

[16] Shay Gueron. 2016. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.* 2016 (2016), 204.

[17] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[18] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. *SIGPLAN Not.* 52, 7 (apr 2017), 129–142. https://doi.org/10.1145/3140607.3050763

[19] Felicitas Hetzelt and Robert Buhren. 2017. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices* 52, 7 (2017), 129–142.

[20] IBM. 2019. IBM's TPM 2.0 TSS. https://sourceforge.net/projects/ibmtpm20tss. Accessed: 2023-09.

[21] Intel. 2016. SGX documentation: sgx_create_monotonic_counter. https://software.intel.com/en-us/node/696638. Accessed: 2023-09.

[22] Intel. 2019. Intel Hardware Shield - Intel Total Memory Encryption. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf. Accessed: 2022-04.

[23] Intel. 2021. 11th Generation Intel Core Processor Desktop. https://cdrdv2.intel.com/v1/dl/getContent/634648. Accessed: 2022-04.

[24] Intel. 2022. 12th Generation Intel Core Processors. https://cdrdv2.intel.com/v1/dl/getContent/655258. Accessed: 2022-04.

[25] David Kaplan. 2017. Protecting vm register state with sev-es. *White paper, Feb* (2017).

[26] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[27] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*. 1445–1461.

[28] Igjae Kim, J. Hyun Kim, Minu Chung, HyunGon Moon, and Sam H. Noh. 2022. A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 363–380. https://www.usenix.org/conference/fast22/presentation/kim-igjae

[29] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[30] Andreas Kogler, Daniel Gruss, and Michael Schwarz. 2022. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In *USENIX Security Symposium*.

[31] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1541–1541.

[32] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. 1257–1272.

[33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMDSEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.

[34] Ashish Lingayat, Ranjana R Badre, and Anil Kumar Gupta. 2018. Performance evaluation for deploying docker containers on baremetal and virtual machine. In *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 1019–1023.

[35] Ming Mao and Marty Humphrey. 2012. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 423–430.

[36] MemCached. 2014. memcached - a distributed memory object caching system. http://memcached.org. Accessed: 2022-03.

[37] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting secrets from encrypted virtual machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 221–230.

[38] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*. 1–6.

[39] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. *arXiv preprint arXiv:2105.13824* (2021).

[40] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[41] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 195–209.

[42] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. 2020. VoltJockey: A New Dynamic Voltage Scaling-Based Fault Injection Attack on Intel SGX. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 6 (2020), 1130–1143.

[43] Anja Rabich, Thomas Eisenbarth, and Luca Wilke. 2020. Software-based Undervolting Faults in AMD Zen Processors. (2020).

[44] Martin Radev and Mathias Morbitzer. 2020. Exploiting Interfaces of Secure Encrypted Virtual Machines. In *Reversing and Offensive-oriented Trends Symposium*. 1–12.

[45] Redis. 2015. Redis. https://redis.io. Accessed: 2022-03.

[46] Albert Reuther, Peter Michaleas, Andrew Prout, and Jeremy Kepner. 2012. HPC-VMs: Virtual machines in high performance computing systems. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–6.

[47] Joerg Roedel. 2020. Joerg Roedel. 2020. x86/sev-es: Do not support MMIO to/from encrypted memory. https://git.kernel.org/pub/scm/linux/kernel/git/joro/linux.git/commit/?h=sev-es-tip-updates&id=5282faf01e085d57658a39494ea760c2b7309f3d. Accessed: 2022-01.

[48] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 183–196.

[49] FuturePlus System. 2016. DDR2 800 Bus Analysis Probe. http://www.futureplus.com/News-Releases/fs2332_pr_010604.pdf. Accessed: 2022-11.

[50] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 665–678.

[51] T.C.Group. 2019. TPM Library Part 1: Architecture, Family "2.0", Level 00, Revision 01.38. http://www.trustedcomputinggroup.org/resources/tpm_library_specification. Accessed: 2023-09.

[52] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[53] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1483–1496.

[54] David Wragg. 2016. KVM hello world. https://github.com/dpw/kvm-hello-world. [Online; accessed 2022-03].

[55] ycsb. 2017. YCSB. https://github.com/brianfrankcooper/YCSB. Accessed: 2022-03.

[56] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. 2019. Minimal kernel: an operating system architecture for TEE to resist board level physical attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 105–120.