VirTEE: A Full Backward-Compatible TEE with Native Live Migration and Secure I/O

Jianqiang Wang, Pouya Mahmoody,Ferdinand Brasser,Patrick Jauernig,Ahmad-Reza Sadeghi
Technische Universität of Darmstadt
{firstname.lastname}@trust.tu-darmstadt.de

Donghui Yu,Dahan Pan,Yuanyuan Zhang
Shanghai Jiao Tong University
{yudhui,dhpan98,yyjess}@sjtu.edu.cn

ABSTRACT

Modern security architectures provide Trusted Execution Environments (TEEs) to protect critical data and applications against malicious privileged software in so-called enclaves. However, the seamless integration of existing TEEs into the cloud is hindered, as they require substantial adaptation of the software executing inside an enclave as well as the cloud management software to handle enclaved workloads. We tackle these challenges by presenting VirTEE, the first TEE architecture that allows strongly isolated execution of unmodified virtual machines (VMs) in enclaves, as well as secure live migration of VM enclaves between VirTEE-enabled servers. Combined with its secure I/O capabilities, VirTEE enables the integration of enclaved computing in today's complex cloud infrastructure. We thoroughly evaluate our RISC-V-based prototype, and show its effectiveness and efficiency.

KEYWORDS

RISC-V, TEE, Virtualization, Virtual Machine Migration

1 INTRODUCTION

Trusted Execution Environment (TEE) was proposed to safeguard the confidentiality and integrity of the run-time data in a strong adversary model, i.e., in the presence of malicious software including the operating system. Traditional software sandboxing [34] [33] [18] fails when privileged software such as OS is under the adversary's control. Hence, TEEs enforce memory access control mechanisms to the protected ranges of memory by means of the so-called enclaves that are inaccessible even to the high privilege software. The major platform vendors provide their proprietary enclave security architectures such as Intel SGX [8], Intel TDX [9], AMD SEV [26], ARM TrustZone [2] and ARM CCA [6]. Similarly, academic research has also proposed a variety of enclave architectures using CPU features or customized hardware, such as Keystone [25], Penglai [19], Sanctum [17], CURE [12] and Komodo [20]. However, these solutions have several shortcomings such as lack of full backward-compatibility, native live migration and secure I/O.

In Intel SGX [8], system calls are not allowed to be directly used inside the enclave. Therefore, programmers cannot develop SGX applications by using normal tool chain. Although the provided SDK can facilitate the coding process, the cost of manual development for the specific CPU feature from scratch is still high. Recent works have attempted to port unmodified legacy applications into SGX such as Panoply [31], Haven [13], Scone [11] and Graphene-SGX [32]. However, these solutions suffer from scalability and compatibility problems. Keystone [25], Penglai [19], Komodo [20], and Sanctum [17] also provide their SDK for developers to develop enclaved applications and, as a result, slow down the acceptance of the TEE solutions by the industry.

To provide application level backward-compatibility, virtual machine-based TEEs, e.g., Intel TDX [9], AMD SEV [26] and ARM CCA [6] have been developed by the major industry players. However, the hypervisor is assumed to be untrusted in their threat model. Consequently, the hypervisor is deployed outside the enclave memory. The virtual machine OS kernel needs to be modified to adapt to the untrusted hypervisor. For instance, when it traps into the hypervisor, the virtual machine OS kernel is responsible for cleaning the secret data in the general purpose registers. Since the hypervisor and the virtual machine are not in the same enclave memory, the virtual machine is supposed to implement its own secure I/O which requires extra developing effort for the programmer. ARM TrustZone [2] is another virtual machine based TEE, however, it does not support multiple enclaves so it cannot be deployed on the cloud server. Hence, currently virtual machine-based TEEs do not provide full backward-compatibility.

In order to take full advantage of hardware resources, the cloud server commonly needs to migrate virtual machines to other platforms. However, existing virtual machine based TEEs hardly provide simple and native migration feature as the enclave memory is not accessible to other software even for the hypervisor in the platform. Previous works such as [28] [21] [27] [22] aimed to provide third party migration support to those TEEs. However, they either require additional hardware extension or understanding of the enclave applications.

To tackle the problems of the existing solutions, we propose VirTEE, a full backward-compatible TEE on RISC-V architecture enabling native live migration and secure I/O by utilizing the RISC-V hypervisor extension and VirTEE hardware. VirTEE hardware allows both strong enclave memory isolation and large size enclave while incurring small performance overhead. Facilitated by large-size enclave support, VirTEE runs unmodified kernel and applications on top of an enclave monitor in one enclave. The enclave monitor is located in the same enclave as the virtual machine and consequently is able to provide transparent native migration and secure I/O for the virtual machines. The evaluation results show

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530436

that VirTEE only imposes moderate overhead on standard benchmarks such as rv8, CoreMark, as well as on real-world software such as SQLite and OpenSSL,

In summary, we make the following contribution:

- We designed a novel TEE solution on RISC-V architecture which provides full backward-compatibility, native live migration and secure I/O.
- We implemented VirTEE prototype based on QEMU simulator and CURE RISC-V security architecture [12], and will be open-sourced on GitHub.
- We thoroughly evaluated VirTEE prototype performance on standard benchmarks and real-world software to show its effectiveness and efficiency.

2 BACKGROUND

In this section we give an overview of the aspects that are helpful to understand the remainder of the paper. Specifically, we elaborate on the RISC-V privilege levels and the RISC-V hypervisor extension.

RISC-V Architecture Privilege Levels. The RISC-V architecture basically defines four privilege levels. The firmware runs in machine mode, the most privileged mode (PL0), its memory integrity is protected by secure boot and the Physical Memory Protection (PMP) unit. The operating system kernel runs in PL1, and user-space applications run in PL2. The PL3 privilege level is introduced by the RISC-V hypervisor extension which be elaborated next.

RISC-V Hypervisor Extension. RISC-V introduced a hypervisor extension for virtualization. Instead of the operating system kernel, the hypervisor runs in PL1 and the virtual machine that contains the virtual machine kernel and virtual machine applications run on PL2 and PL3. Any memory access from the virtual machine is further translated by a second level page table to form the real physical address. The hypervisor can decide which physical memory page is mapped to the virtual machine by manipulating the second level page table. Since RISC-V uses memory mapped I/O (MMIO) to access device registers, the hypervisor is also able to intervene the virtual machine I/O process. Similar to system call, the firmware which is running in PL0, provides low level functionality interfaces called environment call (ECALL) for the operating system. Any ECALL from the virtual machine is handled by the hypervisor first. The hypervisor determines whether to forwards the ECALL requests to the firmware or returns with fake values.

3 VIRTEE

3.1 Adversary Model

We consider the adversary model along the line of related works [2, 12, 14, 17]. The Trusted Computing Base (TCB) consists of three components, i.e., 1) the underlying VirTEE hardware, 2)the security monitor, a privileged component in PL0 that is able to configure the VirTEE hardware and 3) the enclave monitor running in the enclave. We assume that the adversary controls the whole OS. The adversary can leak secret data from the enclave by means of cache side-channel attacks, forge enclaves, etc. However, VirTEE, as for other TEE architectures, does not protect enclaves against memory-corruption attacks. We also assume that the peripherals such as hard drive are also accessible to the adversary. DoS attacks are orthogonal to the scope of this paper, as most TEEs do not give guarantees on availability.

3.2 Design Overview

VirTEE is a novel security architecture that allows execution of unmodified virtual machines (VMs) in strongly isolated enclaves. Further, VirTEE completes this design with secure live migration and secure I/O. VirTEE design is shown in Figure 1. The VirTEE hardware provides strong physical enclave memory isolation and cache side channel attack resilience. Based on the VirTEE hardware memory access control mechanism, the security monitor provides enclave memory management (e.g., creating new enclave memory, increasing and shirking enclave size) as well as enclave metadata management (e.g., measurement, header address and size), attestation primitives and context switching for the host and the enclaves. Inside the enclave, the unmodified VMs run on top of the enclave monitor. Since they are in the same enclave, the enclave monitor can directly access the VM memory. The enclave monitor provides live migration and secure I/O support for the VM that facilitates the enclave user a lot. In section 3.6, we will elaborate the main workflow of VirTEE.

In the following, we will present the respective components in more detail.

3.3 VirTEE Hardware

The VirTEE hardware is the secure platform infrastructure of VirTEE. It divides the physical memory into enclaves (different colors in Figure 1). Several pairs of registers which are only accessible by the security monitor are used to record the enclave header address and size, and are further used by the filter engine to grant access permissions. If and only if the instruction and target data are located in the same enclave, permission will be granted. The VirTEE hardware uses registers instead of the page table to manage enclave memory so that it can support large size enclave memory. To provide cache side-channel attack resilience, when a CPU core is executing one enclave, it has its own last-level cache partition which is not shared by other cores. The CPU core will clean its cache information before exiting the enclave. In this way, cache information is locked into the specific core, preventing cache side-channel attacks.

3.4 Security Monitor

The security monitor is a part of firmware and runs in PL0. It manages the enclave memory by manipulating the registers provided by the VirTEE hardware. Since it can change the whole enclave layout, it has full access to the physical memory. During platform boot, the integrity of the security monitor is verified by secure boot [5] so that we assume that the security monitor is not compromised at load time. In summary, the security monitor provides the following functionalities.

Enclave Metadata Management. During enclave initialization, the security monitor generates an enclave instance which contains the enclave id, header address, size, derived local attestation key, and its measurement report (i.e., enclave fingerprint), etc. The instance is stored in a list protected by the security monitor.

Enclave Memory Management. The security monitor can modify the registers to change the enclave memory layout. At

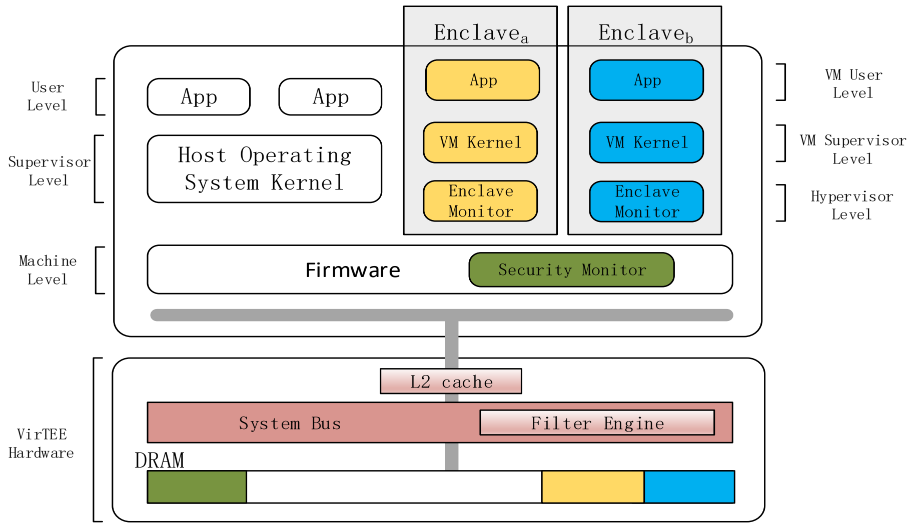

Figure 1: Design overview of VirTEE

enclave initialization, the security monitor allocates new enclave memory according to the request arguments. When it receives a request from the enclave to modify the enclave size, the security monitor first checks if the requested memory will overlap with other enclave memory. If so, the security monitor rejects the request. If not, the security monitor changes the corresponding registers and notifies the enclave.

Attestation Primitives. Attestation is used by the enclave application to prove that it is the genuine entity assumed by the verifier. Based on the prover's and the verifier's identity, the security monitor generates cryptographic report and quota for local attestation and remote attestation respectively. A device key is hardcoded in the security monitor and is used to generate attestation keys. Since the security monitor has full access to the physical memory, it can implement a zero copy (in place) report-generating mechanism which significantly reduces the overhead.

Context Switching. To enter an enclave, the security monitor first checks if the target enclave is ready to be executed. If the check passes, the security monitor prepares the context for the enclave such as arguments and sets up the control registers. Then, it jumps to the enclave. When the enclave terminates, the security monitor updates the enclave status and switches back to the host kernel.

3.5 Enclave Monitor

The enclave monitor running in the hypervisor privilege level is the core of VirTEE. We regard the enclave monitor as part of the TCB, therefore, the VM kernel is not required to clear sensitive data before trapping into the enclave monitor. The RISC-V hypervisor extension guarantees that every memory access including I/O registers access is automatically handled by the enclave monitor first. By intercepting the I/O process, the enclave monitor provides transparent secure I/O for the VM. Since the enclave monitor is located in the same enclave memory as the VM, it can directly read the VM memory and migrate the VM to another platform without third-party support. By using the enclave monitor, VirTEE achieves full backward-compatibility, native live migration and secure I/O. However, note that the enclave monitor is small so that adds only little attack surface to the enclave. We summarize the enclave monitor's main functionalities as following.

Enclave Memory Management. In principle, without the enclave monitor, a kernel running in PL1 can access any physical address including the non-enclave memory even though the access will be blocked by the VirTEE hardware. However, we prevent such unintentional memory access by using the enclave monitor memory protection scheme. Every physical address that the VM accesses will be further translated by a second-level page table to the real physical address. We initialize a second-level page table at the enclave monitor initialization process. At run time, the enclave monitor communicates with the security monitor to allocate new enclave pages and maps the pages for the VM. The enclave monitor can prevent the VM from accessing non-enclave memory by mapping the enclave memory pages to an out-of-enclave VM address. RISC-V architecture uses memory-mapped I/O (MMIO) to access device registers. If the physical address is located in the MMIO memory range, it means that the VM kernel is accessing the device registers. We handle MMIO access by simulating the specific devices. The device virtualization details will be discussed next.

Device Virtualization and Secure I/O. For device-registers accesses, we parse the register values to get the I/O-request arguments such as the buffer address, size, and hard driver sector number. After extracting the arguments, the enclave monitor forwards the I/O requests to the real devices. During the forwarding process, the enclave monitor can encrypt and decrypt the I/O data in a transparent way so that it achieves I/O data confidentiality. In VirTEE, we implemented a serial port as the console and a block device as the hard drive for their limited registers. Note that we can virtualize any device as long as the register accesses are properly handled.

In our threat model, the peripherals such as hard drive can be readable for the attacker. For example, the VM kernel commonly uses a partition in the hard drive as swap space and writes the memory into the partition. In this way, the secret memory will be leaked to the attacker. Without the enclave monitor, the VM kernel needs to be modified to encrypt the memory before writing them to the hard drive. With the enclave monitor, the I/O requests are first handled by the enclave monitor. After receiving an I/O request, the enclave monitor lookup the device tree to infer which device register the VM accesses. If it is the hard drive, the enclave monitor parses the arguments to get the sector number to see if it is in the swap partition, then the enclave monitor decides whether to encrypt or decrypt the data. In this way, VirTEE achieves transparent secure I/O.

Attestation Service. Attestation implementations are commonly deployed as separate enclaves by other TEEs. However, we encapsulate the attestation implementation in the enclave monitor, and the enclave monitor provides interfaces to the VM. For example, a VM, say the verifier, starts to attest another enclave VM, the prover, on the same platform. It then calls the local attestation interface exposed by the the enclave monitor of the corresponding enclave. Then, the enclave monitor creates an attestation report of the enclave. The enclave monitor uses attestation primitives provided by the security monitor to complete the attestation process with the verifier. The Diffie–Hellman key exchange data are encapsulated in the report so that the secure channel is established and returned to the VM.

Live Migration Service. VirTEE provides a live migration service for the source enclave VM to migrate itself to a target trusted platform. VirTEE presents a stub enclave as the migration target. In the stub enclave, the enclave monitor keeps listening on remote migration request and continues the VM execution after finishing the migration process. Similar to [16], VirTEE takes three steps to finish the migration process. First, the source enclave uses remote attestation to verify the target and establishes a secure channel with the target. Secondly, the source enclave monitor keeps the VM running, clears the dirty bit in the whole second-level page table and transfers the VM memory to the target. Finally, the source enclave monitor stops the VM and checks the page-table dirty bit. Dirty pages are transferred again to target platform as well as the virtual devices status and pending I/O requests. In this way, we keep the VM downtime small.

3.6 Enclave Setup

VirTEE works mainly as the following steps:

(1) The host allocates contiguous physical memory, fills the memory with the enclave monitor and the virtual machine binary file and other necessary metadata in a predefined memory layout.
(2) The host notifies the security monitor about the creation of one enclave with its physical address and initial size. The security monitor assigns a unique id to this enclave and binds the id to its header address and size.
(3) The host finds an available CPU core and assigns the core to one enclave, then switches to the core, invokes the security monitor's entering enclave function with the specific enclave id as argument.
(4) The security monitor finds the corresponding enclave via its id and assigns the enclave memory by manipulating the VirTEE hardware registers. Afterwards, the security monitor verifies the enclave signature by using a private key. If it fails, the security monitor refuses to launch the enclave, otherwise it performs context switching and hands over the control flow to the enclave monitor's entry point.
(5) The enclave monitor receives three arguments from the security monitor: CPU core id, device tree which contains all devices information and the enclave memory header address.
(6) The enclave monitor initializes the necessary virtual machine environment, e.g., second-level page table and virtual devices. Peripherals are allocated and bind to the virtual machine.
(7) The enclave monitor hands over the control flow to the virtual machine's entry point,i.e., the virtual machine kernel entry point.
(8) The virtual machine runs under the control of the enclave monitor. The virtual machine can use the migration or attestation services by calling the interfaces exposed by the enclave monitor.

4 EVALUATION

This section presents our thorough evaluation of VirTEE's effectiveness and efficiency regarding 1) Benchmark run-time performance overhead, and 2) VirTEE features' overheads on microbenchmarks including enclave initialization, attestation and live migration. For the run-time performance overhead, we selected standard benchmarks (rv8 [3] and CoreMark [1]), which have been used by CURE [12], the security architecture we build on, and real-world software (OpenSSL 3.0.0 [10], Binutils [7] and SQLite [23]). For VirTEE's features, we evaluated enclave initialization, attestation and migration overhead by measuring the average running time. We did our experiments on a Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz laptop with 16 GB RAM. The QEMU simulator version is 6.0.50.

Implementation We implemented the security monitor on top of OpenSBI. The hsm[1] handlers are extended to support security monitor environment call interfaces. Fourteen Maros are defined in the security monitor to perform enclave creation, enclave memory management and attestation primitives. We emulated a UART8250 serial port as console and a virtio block device as the hard drive in the enclave monitor. In summary, enclave monitor, the security monitor, and the host kernel required approximately 6700, 720, and 250 lines of C code respectively.

4.1 Run-time Performance Overhead

The run-time performance overhead mainly comes from RISC-V second-level page-table address translation, device virtualization (secure I/O) and VirTEE hardware. We selected standard benchmarks and real-world software to test VirTEE's performance overhead. The selected benchmarks includes I/O intensive workloads (Binutils, SQLite) and CPU intensive workloads (rv8, CoreMark, OpenSSL). The hardware performance overhead has already been discussed in CURE [12], therefore we treat performance overhead induced by the CURE hardware as orthogonal. For each program,

[1]Hart State Management SBI extension

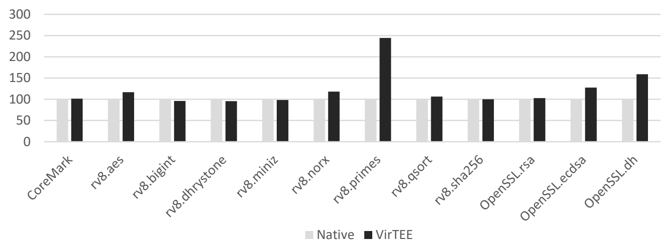

Figure 2: VirTEE’s run-time performance overhead relative to native process

we run it for five times with and without VirTEE in the same simulator and calculate the final average overhead.

I/O Intensive Workloads. We selected Binutils-*strings* and Binutils-*readelf*, two popular binary analysis tools, to extract information from an unstripped binary. The main performance overhead comes from the console virtualization. For SQLite, we used the SQLite *kvtest* program to generate a 1 GB test database file. Then we ran *kvtest* to read blobs from the database file. The main performance overhead comes from the hard-drive virtualization and secure I/O. The experiment results show that VirTEE incurs 51% and 52% overhead in *strings* and *readelf* respectively. For SQLite, *kvtest* can read the database file in 54.7 MB/s and 87.7 MB/s with and without VirTEE respectively. That means VirTEE incurs 61% hard drive I/O overhead induced by hard-drive virtualization and secure I/O.

We conducted the same experiments in an AMD SEV-enabled platform with an AMD EPYC 7262 8-Core 3.2GHz Processor and 32GB RAM. The results show that AMD SEV incurs 532% and 214% overhead for *strings* and *readelf* respectively. For SQLite, AMD SEV incurs 25% hard-drive virtualization overhead. The reason for VirTEE’s better performance for *strings* and *readelf* is that, for console virtualization, VirTEE does not need to intercept the I/O process and directly writes the data to output buffer. However, for hard drive virtualization, VirTEE needs to intercept every I/O process, parse the I/O arguments and performance encryption or decryption while AMD SEV does not support secure I/O. Therefore, VirTEE has a better virtual console performance and a worse virtual hard-drive performance than AMD SEV.

CPU Intensive Workloads. We selected three asymmetric crypto algorithm tests (dh, ecdsa and rsa) in OpenSSL’s test vectors. CoreMark and rv8 use internal test data, therefore, we ran those binaries directly. As shown in Figure 2, for the majority of the benchmarks, VirTEE only incurs less than 15% overhead. An exception is rv8-primes. The reason for that is that VirTEE does not support floating-point registers yet, and rv8-primes performs a lot of division operations. VirTEE has to use soft-floating point to simulate the calculation which is much slower. This also happens in OpenSSL tests as well. We did the same experiment in AMD SEV, the results show that the SEV incurs an average of 15% overhead on the whole test programs as SEV support real floating-point registers.

4.2 VirTEE Feature Overhead

Enclave Initialization. The enclave initialization process mainly includes continuous physical memory allocation, VM and enclave monitor binary-files loading, metadata filling and integrity verification. In our experiment, the initial enclave size is 500MB, consisting of 100MB enclave monitor heap memory, 10MB enclave monitor and 490MB VM. The enclave monitor binary file size is 59KB and the VM kernel is 17MB. We filled the binary files to each part of the memory head and zeroed the remainder. We run an enclave VM for five times and calculate the average time. The result shows that the enclave VM initialization approximately takes less than 150ms.

Location Attestation. We lunched two enclave VMs running in parallel in a simulator. They run the same VM and enclave monitor binary. Starting from the verifier’s first attestation call to the final secure channel setup, we counted the time elapsed and repeated the process for five times. The result shows that the location attestation only takes 144 ms.

Remote Attestation. We simulated a remote attestation server by directly feeding the verification result back to the platform. We left the complete remote attestation system as future work. Starting from receiving the verifier’s request to the final secure channel setup, we counted the time elapsed and repeated the process for five times. The result shows that the remote attestation only takes less than 50ms.

Live Migration. We lunched two QEMU simulators in parallel. The two simulators are bridged in one network card in the laptop. Now the two simulators are running in the same LAN. One is the target which contains a stub enclave and the other one is the migration source which contains a normal enclave VM. We set the bandwidth to 5MB/s. It takes 100 seconds to finish the second migration step. For the third step, it depends on how many dirty pages in the VM. In our experiment, we took two Binutils programs as example, *readelf* and *objdump* contain maximum 1441 and 1652 dirty pages respectively which means they take approximate 1s and 1.2s to finish the final migration step.

5 RELATED WORK

Existing non-virtual machine based TEEs all require non-trivial porting effort to support legacy applications. Intel SGX [8] provides a SDK for programmers to develop SGX applications from scratch. Programmers are supposed to manually define the trusted part, untrusted part and their calling interfaces in a so called EDL file. Although the SDK facilitates the development process, however, complex commercial software are still hard to adapt themselves to SGX. LibOS based solutions such as Graphene [32], Occlum [30], SGX-LKL-OE [29], Fortanix [4], SCONE [11] try to port unmodified legacy applications to SGX. However, they only support limited system call interfaces, thus suffer from compatibility problems. Shim library based solutions such as Haven [13] and Panoply [31] forward system call requests to the operating system kernel by shielding the enclave applications. Previous works show that they are prone attacks through system call return values [15] [24]. Other TEEs such as Penglai [19] and Keystone [25] provide their development kit like SGX. The porting effort hinders the TEEs from being widely adopted. In contrast, VirTEE can run unmodified applications in a virtual machine and does not lead to extra attack surface.

Virtual machine based TEEs run unmodified applications in an enclave. Compared with VirTEE, Intel TDX [9], AMD SEV [26] and ARM CCA [6] isolate the virtual machine from untrusted part including the hypervisor. In this design, the hypervisor has no access to the virtual machine which means the virtual machine kernel has to be modified to support native live migration and secure I/O. In addition, the virtual machine kernel is required to clear the sensitive data (e.g., general registers) before exiting the virtual environment. As this design does not support native migration and secure I/O, several works such as [28] [21] [27] [22] presented third party solutions by using new instructions or security hardware modules. VirTEE deploys a enclave monitor inside the enclave. The enclave monitor is assumed to be a trusted component. It encapsulates the secure I/O and live migration functionalities for the virtual machine so that VirTEE can seamlessly support unmodified kernel.

We implemented VirTEE hardware based on CURE [12] which provides a strong physical enclave memory isolation. However, since there is no enclave monitor in CURE's design, it does not support unmodified kernel, native live migration and secure I/O.

6 CONCLUSION

In this paper, we proposes VirTEE. Based on RISC-V hypervisor extension and VirTEE hardware, we overcomes the disadvantages that most of state-of-the-art TEE solutions have. VirTEE is able to run unmodified kernel and applications in a virtual machine in enclave memory, providing full backward-compatibility. With its novel design architecture, VirTEE supports native live migration and secure I/O. The evaluation indicates that VirTEE only incurs moderate performance overhead.

7 ACKNOWLEDGEMENT

This project was funded by the Deutsche Forschungsgemeinschaft (DFG) SFB-1119 CROSSING/236615297. Moreover, this project has received funding from Huawei within the OpenS3 lab.

REFERENCES

[1] 2009. EMBC. Coremark. https://www.eembc.org/coremark/.

[2] 2015. Security technology: building a secure system using TrustZone technology. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/TrustZone-and-FIDO-white-paper.pdf?revision=98e6ae26-92ca-4ffd-ac4e-3329b7f8a23e.

[3] 2018. rv8-bench. https://github.com/michaeljclark/rv8-bench.

[4] 2018. Side Channels and Runtime Encryption Solutions with Intel® SGX. https://www.fortanix.com/assets/Fortanix_Side_Channel_Whitepaper.pdf.

[5] 2019. RISC-V Secure Bootloader. https://riscv.org/wp-content/uploads/2019/06/13.55-RISC-V-Workshop-Secure-Bootloader.pdf.

[6] 2021. Arm Confidential Compute Architecture. https://documentation-service.arm.com/static/61825631f45f0b1fbf3a7a7d?token=.

[7] 2021. GNU Binutils. https://www.gnu.org/software/binutils/.

[8] 2021. Intel® Software Guard Extensions. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[9] 2021. Intel® Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1.5-base-spec-348549001.pdf.

[10] 2021. OpenSSL. https://www.openssl.org/.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 689–703.

[12] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. {CURE}: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.

[14] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[15] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[16] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.

[17] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 857–874.

[18] Liang Deng, Qingkai Zeng, and Yao Liu. 2015. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security and Privacy Conference*. Springer, 386–400.

[19] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 275–294. https://www.usenix.org/conference/osdi21/presentation/feng

[20] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 287–305.

[21] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Secure live migration of SGX enclaves on untrusted cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 225–236.

[22] Joao Guerreiro, Rui Moura, and Joao Nuno Silva. 2020. TEEnder: SGX enclave migration using HSMs. *Computers & Security* 96 (2020), 101874.

[23] Richard D Hipp. 2020. SQLite.

[24] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN attacks: On insecurity of enclave untrusted interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.

[25] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[26] Ingo Lütkebohle. 2016. AMD Secure Encrypted Virtualization. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[27] Jaemin Park, Sungjin Park, Brent Byunghoon Kang, and Kwangjo Kim. 2019. eMotion: An SGX extension for migrating enclaves. *Computers & Security* 80 (2019), 173–185.

[28] Jaemin Park, Sungjin Park, Jisoo Oh, and Jong-Jin Won. 2016. Toward live migration of SGX-enabled virtual machines. In *2016 IEEE World Congress on Services (SERVICES)*. IEEE, 111–112.

[29] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143* (2019).

[30] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[31] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *NDSS*.

[32] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 645–658.

[33] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 469–480.

[34] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.