JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Unified Enclave Abstraction and Secure Enclave Migration on Heterogeneous TEE Architectures

Jin-Yu Gu[1,2], Hao Li[1,2], Yu-Bin Xia[1,2,*] Senior Member, CCF, Member, ACM, IEEE, Hai-Bo Chen[1,2] Distinguished Member, CCF, ACM, Cheng-Gang Qin[3], Zheng-Yu He[3]

[1]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

[2]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China*

[3]*Ant Group, Hangzhou 310099, China*

E-mail: {gujinyu, lihao, xiayubin, haibochen}@sjtu.edu.cn, {chenggang.qcg, zhengyu.he}@antgroup.com

Abstract Nowadays, application migration becomes more and more attractive. For example, it can make computation closer to data sources or make service closer to end-users, which may significantly decrease latency in edge computing. Yet, migrating applications among servers that are controlled by different platform owners raises security issues. We leverage hardware-secured Trusted Execution Environment (TEE, aka., enclave) technologies, such as Intel SGX, AMD SEV, and ARM TrustZone, for protecting critical computations on untrusted servers. However, these hardware TEEs propose non-uniform programming abstractions and are based on heterogeneous architectures, which not only forces programmers to develop secure applications targeting some specific abstraction but also hinders the migration of protected applications. Therefore, we propose UniTEE which gives a unified enclave programming abstraction across the above three hardware TEEs by using a microkernel-based design and enables the secure enclave migration by integrating heterogeneous migration techniques. We have implemented the prototype on real machines. The evaluation results show the migration support incurs nearly-zero runtime overhead and the migration procedure is also efficient.

Keywords Enclave, Heterogeneous, Migration

1 Introduction

As an emerging computing paradigm, edge computing [1, 2, 3, 4] has gained more attention in recent years because it allows services to become closer to clients or data production sources. Owing to the promising feature of "close-to-data/client", edge computing can significantly reduce network communication cost and thus bring better quality of service, e.g., extremely low latency for requests. Nowadays, it has been used in plentiful application domains, such as computation offloading from cloud to smart home or city [5, 6], real-time analytics [7, 8], and so on, to address the concerns of response time requirement and bandwidth cost limitation.

The mobility of clients (e.g., mobile users) and data sources (e.g., intelligent vehicles) makes runtime service migration become an indispensable requirement in edge computing [9, 10, 11, 12]. With migration, a service can keep running on the edge server nearest to the client, which may change from time to time, in order to keep latency low. Otherwise, dramatic performance degradation may occur, and qualified service continuity is difficult to ensure. In addition, migration is also important for meeting other demands of edge computing, like relieving congested edge servers and leaving servers that may fail (e.g., before running out of battery).

However, research on security issues of service migration in edge paradigms is still nascent and lim-

Regular Paper

*Corresponding Author: Yu-bin Xia. This work is supported in part by the National Key Research and Development Program of China (under Grant No. 2020AAA0108502), the National Natural Science Foundation of China (under Grant No. 61972244, U19A2060, 61925206), and the HighTech Support Program from Shanghai Committee of Science and Technology (under Grant No. 19511121100).

ited [9, 13, 14]. Compared with traditional cloud computing, there are two main security-related differences when edge servers involve. First, the owners of the edge servers may be different from the cloud providers and could be curious or even malicious. Thus, the providers of the service applications (e.g., the application developers) have concerns about their intellectual property which may be easily stolen by the edge server owner who controls the physical machine as well as the whole system software stack. The end users also cannot ensure the service application is correctly running on the edge servers. Second, the edge servers are easier to be attacked compared with cloud servers because cloud servers usually face remote attacks only while attackers are easier to physically access edge servers and thus have more attack means (e.g., conducting physical attacks). Such differences are obstacles for migrating services among edge servers as well as from cloud to edge.

In this paper, we propose to utilize the hardware-assisted Trusted Execution Environment (TEE) to mitigate the above security threats and enable secure service migration. TEE is suitable for protecting private code and data on untrusted platforms. For example, Intel SGX has been adopted by some major cloud providers and ARM TrustZone has been well-used on smartphones. Generally speaking, when accommodated in a hardware TEE, a benign service application can protect itself and users' input from malicious software, including OS and compromised peripherals. Nevertheless, edge servers can deploy CPUs from different vendors, such as Intel, AMD and Huawei, which, inherently, means that their equipped TEEs are heterogeneous, like Intel SGX [15], AMD SEV [16], and ARM TrustZone [17].

Therefore, we propose UniTEE which gives a unified TEE abstraction for hiding the hardware heterogeneity from applications. UniTEE adopts the programming model of SGX applications for its flexibility and popularity. Specifically, an application can partition itself into secure-(in)sensitive parts and build one or more hardware-secured TEEs (named enclaves) to run the secure-sensitive ones. An enclave can offer strong guarantees of both confidentiality and integrity for the secure code/data inside despite being executed in an untrusted environment, which can be extremely suitable for outsourced computation [18, 19, 20, 21]. No matter what the underlying hardware TEEs are, UniTEE provides unified programming APIs including creating, attesting, invoking, and destroying enclaves. As AMD SEV and ARM TrustZone do not provide enclaves like Intel SGX, we leverage hardware-software co-designs for building SGX-like enclaves on those platforms. AMD SEV uses virtual machine (VM) as the granularity of its TEE and supports concurrently running at most 15 secure VMs, which does not fit the programming model of UniTEE. Therefore, we deploy a trusted microkernel in the supervisor mode of a secure VM and then let the microkernel to build user-level isolated enclaves. An application can construct its enclaves in the secure VM by sending requests to the trusted microkernel. ARM TrustZone enables the CPU to have two modes named *normal world* and *secure world*, respectively. UniTEE achieves the same enclave abstraction by deploying the trusted microkernel in the secure world to be the enclave manager. Thus, by combing the tiny software layer (the trusted microkernel) and the hardware TEE (either a secure VM of SEV or the secure world of TrustZone), UniTEE provides SGX-like enclave abstractions and thus unifies the TEE programming model on Intel SGX, AMD SEV, and ARM Trustzone. Besides, for easing programming, it provides an enclave-management library for an application to control its enclaves' life cycle, including creation, attestation, interaction and deletion. It also provides a C library (based on musl-libc) to ease the development and deployment of in-enclave code, as well as to be compatible with legacy code.

A unified enclave abstraction enables programmers to develop secure applications without considering the differences of the underlying TEEs. Nevertheless, it is not enough for migrating applications between edge nodes because heterogeneous TEEs use different instruction set architectures (ISAs). Therefore, UniTEE further integrates heterogeneous-ISA migration techniques [22, 23, 24] to hide the heterogeneity of enclave ISAs and support enclave migration [25] at runtime. The enclave code will be compiled into different binaries for different ISAs, but every symbol (a variable and a function) has the same offset in different binaries. No matter on which architecture, those symbols will always be loaded at the same virtual addresses at runtime, which significantly simplifies the (cross-architecture) migration procedure because the pointers to them will still be valid after migration. For migrating an enclave running on the source machine, the target machine will first launch a virgin enclave with the binary for its architecture and then receive and restore the enclave checkpoint (memory data and execution context) from the source machine. For ensuring security, the checkpoint generation should not rely on the untrusted software including the OS. Thus, an enclave on the source machine will generate a consistent checkpoint by itself. Specifically, UniTEE adds a *control thread* in each enclave as a part of the framework. After receiving a migration request, this thread will wait for all the enclave threads to enter a quiescent state and then make a checkpoint by encrypting and dumping the enclave states. The encryption key is negotiated by the source enclave and the target enclave and it will protect both the confidentiality and the integrity of the checkpoint during the transfer process.

UniTEE provides a software development kit (SDK) for programmers, and they can develop secure edge applications without awareness of the underlying TEE hardware or the migration mechanisms. We present a prototype implementation and evaluation on an Intel (Skylake i7-7700) machine, an AMD (EPYC 7281) machine, and an ARM (HiKey970) machine, respectively. The evaluation results show: 1) Our SDK can support many real-world applications and the migration mechanism incurs negligible overhead; and 2) The latency of heterogeneous enclave migration is acceptable and mainly decided by the network latency.

In summary, this paper makes the following contributions:

- A unified enclave abstraction on Intel SGX, AMD SEV, and ARM TrustZone exposed by UniTEE.
- A design of secure enclave migration between heterogeneous TEEs enabled in UniTEE.
- A real implementation and evaluation of UniTEE.

2 Motivation and Background

2.1 Motivation

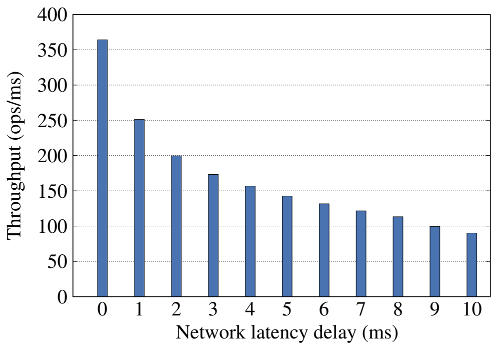


Fig.1. Throughput of the Vedis service that runs on an Intel Skylake machine. The client device is an ARM HiKey board. The network latency varies from 1 to 10 ms (typical latencies in edge computing [11, 26])

.

Application (service) Migration is One of the Most Critical Features in Edge Computing [8, 9, 10, 11, 27, 28, 29]. First, migration can relieve congested edge servers and thus achieve better load balance. Second, migration can bring better fault tolerance (e.g., migrating applications that run on a low-battery edge machine) and ease the edge server upgrading (e.g., migrating running applications to other servers first and then update an idle machine). Third, migration is important to ensure Quality-of-Service (QoS) in edge computing because of the high mobility of client devices such as smartphones or intelligent cars. Specifically, the latency between a client and an edge service may vary because of the client's mobility, which can impact the overall performance, i.e., the service quality. Fig. 1 presents such an example which shows the latency impact on Vedis, a popular key-value store on edge. If latency-sensitive services can be migrated to follow clients, they can show much better performance.

Security Concerns on Migration for Both Service Providers and Clients. Although service providers have the above motivations to migrate their services between different servers, security concerns may force them to abandon migration. Different from cloud servers, which are aggregated together in data centers and managed by the same cloud provider, edge servers are more disaggregated and can be managed by different owners. If one service application is allowed to migrate from cloud to edge and between different edge nodes, the service provider faces the risk of leaking digital property because the owner of some edge server may be curious or even malicious. The owner has the full control of the edge server and can deploy malicious system software (e.g., OS and hypervisor) or compromise them. If a service runs on such a server, the server owner can easily retrieve all the code, data, and runtime states of the service, which means the loss of the digital property to the service provider. Moreover, when the service is controlled by an untrusted server owner, the clients of the service also worry about the security: the client data sent to the service may be stolen, or the service may not faithfully handle the requests.

Hardware TEEs Bring a Potential to Solve the Security Problems. Nowadays, hardware support for secure computing, i.e., Intel SGX, ARM TrustZone, and AMD SEV, gains more and more attention in both the academic and the industry area. These hardware security extensions can protect security-sensitive applications from attackers through providing a hardware-secured trusted execution environment (TEE). A TEE can shield an application's code and data from external accesses by other software, including higher-privileged software like OS and hypervisor. Besides memory protection, it can also provide tamper-resistant execution for the protected application. So, hardware-supported TEE technology is a promising candidate for protecting applications in untrusted cloud/edge servers where the entire software stack and the infrastructure owner are not trustworthy.

However, the Heterogeneity of Servers in Edge Computing Leads to Two-Fold Challenges. In terms of application programming, different servers are equipped with different kinds of TEEs which give heterogeneous programming abstractions (*Challenge-1*). Writing code for every abstraction not only makes the application development inefficient but also brings difficulties to runtime migration. In addition, heterogeneous hardware TEEs make the migration procedure of protected applications challenging (*Challenge-2*) for two reasons. First, they use architecture-specific instructions, registers, etc., which are different from each other. Second, they cannot be accessed by privileged system software such as OS and hypervisor which play important roles in traditional migration (e.g., OS will stop an application and then send its memory data).

In this paper, we make an attempt to solve the above two challenges on how to program security-sensitive applications with different heterogeneous TEEs and how to migrate them between the heterogeneous TEEs. Besides in edge computing as said above, our work may also be used in cloud computing where heterogeneous hardware TEEs and migration are also required [22, 25, 30, 31, 32, 33, 34], for example, secure cross-cloud migration is needed in joint cloud computing [35].

2.2 Background of Hardware-secured TEEs

Intel SGX. Intel SGX [15] can protect user-level computations through providing a hardware-secured execution environment called an *enclave*. An enclave's memory pages reside in the EPC (Enclave Page Cache) which is a part of the memory region that will be automatically protected by the CPU. Although the hypervisor and OS retain their ability to manage EPC memory (e.g., swap EPC page), they cannot break the memory data's confidentiality and integrity. Moreover, Intel SGX also provides tamper-resistant execution to enclaves and enables remote attestation, which means that an enclave can prove its identity to a remote party. Thus, researchers have proposed to leverage SGX to protect outsourced applications [19, 21, 36, 37, 38, 39] and cloud vendors have started to explore the commercial usage of SGX [40]. Specifically, an application can be separated into trusted and untrusted parts, and the trust parts can be executed in one or more enclaves. An SGX enclave resides in the address space of its host application while its memory can only be accessed by itself. A thread has to enter an enclave through executing an *EENTER* instruction and exit from the enclave with an *EEXIT* instruction. Moreover, the CPU can help an enclave to produce a verifiable proof that identifies its memory contents. A remote party, e.g., the enclave owner, can leverage official attestation services like Intel Attestation Service (IAS) to assess the trustworthiness of the proof. Such a procedure is called SGX remote attestation.

AMD SEV. AMD proposed SEV [16] to protect outsourced computing on untrusted servers, whose support has been integrated into existing system software stacks. Different from Intel SGX, which can build TEEs inside applications, the granularity of a TEE in SEV is a secure virtual machine (VM). Tenants can run their applications inside a secure VM which is protected as a whole by the SEV hardware. SEV supports at most 15 secure VMs, and each of them has a unique identifier (ASID). Inside the CPU, all the secure memory of the VM is tagged with the VM's ASID, which prevents the memory content from being accessed by anyone other than the owner VM. When the secure memory data leaves/enters the CPU, it is automatically encrypted/decrypted by the memory controller with a key bound to its owner VM. These keys are managed by a secure co-processor and will never be exposed. A secure VM can decide whether a memory page is secure by setting one bit (C-bit) in the corresponding guest page table entry. Once the bit is set, the CPU will treat the memory page as secure and then protect it transparently. Otherwise, access to the memory page is not restricted, i.e., the page can be accessed by the hypervisor. Besides memory protection, SEV also protects a secure VM's execution states during runtime. Recently, AMD proposes further extensions named SEV-SNP [1] which helps to mitigate the memory integrity problems of SEV [41, 42]. Therefore, even in the face of compromised privileged software, SEV can also protect both the confidentiality and the integrity of its TEEs.

ARM TrustZone. ARM proposes TrustZone [43] technology as its hardware security extension since

[1]SEV Secure Nested Paging. `https://www.amd.com/system/files/TechDocs/56860.pdf`. Referenced February 2021.

ARMv6 architecture. With TrustZone, the CPU has two execution environments, named normal world and secure world, separately. Both worlds have their own user space and kernel space, while the latter is used as the TEE on ARM. Usually, a commodity OS and non-security-sensitive applications run in the normal world while a secure OS (e.g., OPTEE [2]) and security-sensitive applications run in the secure world. The two worlds can switch to each other through the highest privilege mode, monitor mode. One world can execute an SMC (Secure Monitor Call) instruction to trap into monitor mode, and then a secure monitor in the monitor mode helps to finish the world switch. TrustZone can also partition all the physical memory resources into the normal part and secure part, and ensure that the normal world cannot access the secure memory part while the secure world can access the entire memory. Thus, two worlds can exchange data through the normal memory part. Moreover, the secure world can adjust the memory resource partition according to runtime requirements. As TrustZone is widely deployed in ARM platforms such as smartphones and tablets, it has already been used to protect security-critical computation and data [1, 44, 45, 46, 47, 48].

3 Unified TEE Programming Abstraction

In this section, we first give a brief analysis of the three commercial TEE abstractions, which will explain why UniTEE chooses the SGX-like abstraction as the unified one. Then, we describe how to achieve the unified TEE abstraction on different security hardware. Last, the main programming interfaces of such an abstraction will be introduced.

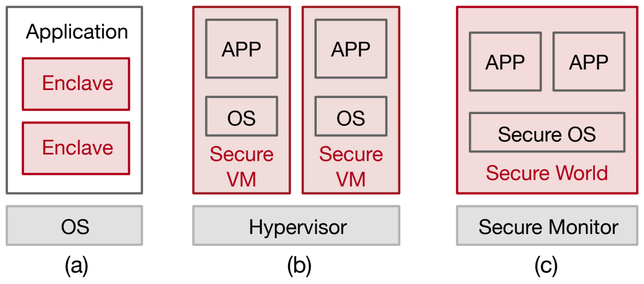


Fig.2. Three commercial TEE abstractions: (a) Intel SGX, (b) AMD SEV, and (c) ARM TrustZone.

3.1 Abstraction Analysis

As shown in Fig. 2, Intel SGX supports constructing multiple enclaves (fine-grained TEEs) inside an application, i.e., in the application's address space, which allows programmers to divide an application into one untrusted part and one or more trusted parts. The latter ones are used to protect security-sensitive code, data, as well as execution. There are several typical usages of SGX for application protection (different granularity). First, programmers can put an unmodified application together with a library OS into a single enclave (e.g., Graphene-SGX [21]) which may also be deployed as a guest VM on untrusted servers (e.g., Haven [36]). Second, programmers can run a container in an enclave to enhance security (e.g., SCONE [19]). Third, programmers can partition an application into mutual-distrusted parts manually [39] or automatically [49] and then utilize enclaves for isolation. In brief, SGX enclave abstraction not only allows a relatively unlimited number of TEEs and but also promises flexibility in the isolation granularity.

In contrast, AMD SEV supports at most 15 secure VMs as TEEs, which leads to two drawbacks: 1) The TEE number is too limited to accommodate different applications; 2) the TEE granularity is too coarse-grained to meet different requirements. Similarly, ARM TrustZone provides only-one secure world as TEE. Al-

[2]`https://github.com/OP-TEE/optee_os`. Referenced February 2021.

though prior studies proposed to multiplex the secure world by deploying a secure OS or using virtualization [32], they only considered protecting a whole application instead of fine-grained protection enabled by SGX.

Therefore, UniTEE embraces the flexible abstraction of SGX and allows to build SGX-like enclaves with any of the three hardware-security technology. Programmers can develop secure applications against a unified abstraction without concerning the underlying hardware TEEs.

3.2 System Architecture

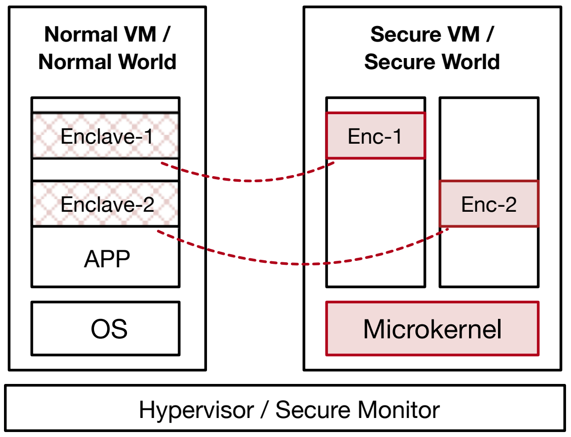


Fig.3. UniTEE gives SGX-like enclave abstraction based on AMD SEV or ARM TrustZone. Enclave is abbreviated as Enc.

To provide SGX-like abstraction with AMD SEV or ARM TrustZone, the first problem to solve is that they cannot provide an unlimited number of hardware TEEs as enclaves. To this end, UniTEE deploys a security-oriented microkernel in one hardware TEE, i.e., the secure world of ARM TrustZone or a secure VM of AMD SEV, and then leverages the microkernel to construct an unlimited number of software TEEs as enclaves. The microkernel runs in kernel mode while the enclaves managed by it are running in user mode.

As shown in Fig. 3, the microkernel creates a new address space for building a new enclave, which is similar to a traditional process on the microkernel. Nevertheless, an enclave logically belongs to some application that runs in the normal VM on an SEV-capable machine or in the normal world on a TrustZone-capable machine. The trustworthy microkernel guarantees both the isolation between different enclaves and the isolation between an enclave and all the untrusted software in the normal VM or the normal world. Specifically, it assigns different enclaves with different page tables and thus achieves the memory isolation between them; it leverages the hardware-security mechanism to ensure the memory isolation between an enclave and the untrusted software, i.e., the enclave uses the secure memory (in the secure VM or the secure world) that cannot be accessed by the untrusted ones, including the privileged OS. Besides, the microkernel also manages the enclaves' runtime states (execution context) and ensures the states' confidentiality and integrity.

An application still runs on the untrusted OS while its enclaves are created by and run on the microkernel. The microkernel is only responsible for the enclave life-cycle management, which mainly involves enclave construction/destruction, enclave memory management, and enclave thread scheduling. The application cannot access its enclaves' memory while an enclave can access its host application's memory only if the microkernel maps the normal memory belonging to the application into the enclave's address space. By default, an enclave and its host application have shared memory for communication.

In brief, UniTEE leverages the microkernel to multiplex a single hardware TEE and thus allows an application to create an arbitrary number of fine-grained enclaves on an AMD SEV machine or an ARM TrustZone machine, just like on an Intel SGX machine. The tradeoff is our microkernel enlarges the trusted computing base (TCB) of an enclave. Nevertheless, our microkernel has a small code base (around 5,000 lines of code) and thus is relatively easier to be implemented correctly. With more efforts in the future, formal verification can be used to make our microkernel more secure. Although prior work on ARM TrustZone also proposes to deploy a secure OS in the secure world (e.g., OPTEE), the secure OS is for running multiple trusted applications instead of enclaves (belong to the host applications), which makes the secure OS and the microkernel of UniTEE different.

3.3 Programming Interfaces

As the SGX programming model is easy to use and adopted by the public, UniTEE preserves similar (or even can be the same) interfaces as listed in Table 1. By providing such interfaces, existing secure applications targeting SGX can be more easily ported to UniTEE, which can make our work more practical.

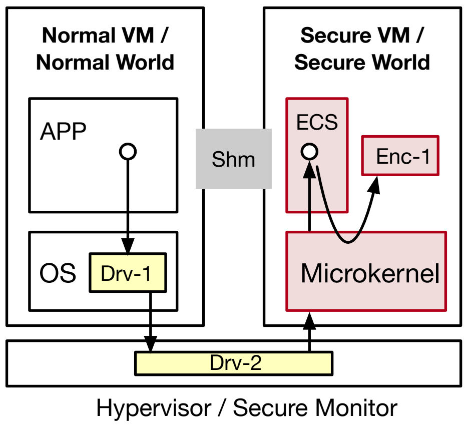


Fig.4. The procedure of enclave construction on SEV and TrustZone platforms. ECS represents Enclave Construction Service. Enclave is abbreviated as Enc. Drv-1 and Drv-2 are two software components deployed by UniTEE.

3.3.1 Enclave Creation

Fig. 4 shows how a host application creates an enclave on SEV and TrustZone platforms (the enclave creation procedure on SGX platforms is just like before, i.e., using official Intel SGX Driver). First, the host application prepares the enclave image and the corresponding configuration. Second, it invokes *create_enclave*, which traps into a kernel module (Drv-1) deployed by UniTEE. Third, the kernel module transfers the control flow to the microkernel. For transferring the control flow, *SMC* instruction is used on TrustZone-enabled platforms while *VMMCALL* and *VMRUN* instructions are used on SEV-enabled platforms. In the former case, *SMC* instruction makes the CPU trap into monitor mode, and another tiny module (Drv-2) deployed by UniTEE helps to finish the switch between the normal world and secure world. In the latter case, *VMMCALL* instruction triggers a *VMExit* and makes the CPU trap into hypervisor mode and, thus, a similar tiny module (Drv-2) in the hypervisor executes *VMRUN* instruction to notify the microkernel. Fourth, a system service of the microkernel, named Enclave Construction Service (ECS), receives the enclave creation request and then constructs the enclave according to the image and configuration passed through the shared memory. On SGX platforms, UniTEE provides the same interface but constructs enclaves just like how official Intel-SGX SDK does. Specifically, a kernel module like Drv-1 in the OS builds enclaves using SGX instructions (*ENCLS*).

Components of UniTEE: For SEV, the components include a kernel module in the normal VM's guest OS, a tiny module in the hypervisor, and a microkernel OS in the secure VM. For TrustZone, the components include a kernel module in the normal world OS, a tiny module in the monitor mode, and a microkernel OS in the secure world. For SGX, the components include a kernel module, i.e., the SGX driver. Besides, the components also include a library in each enclave for all the three platforms.

Table 1. Main Interfaces in the Enclave-Management Library (for Host Applications) and the Modified C Library (for Enclaves)

Declaration	Description
int create_enclave(Buf enclave_img, Buf enclave_config)	Used by the host application to create an enclave. The two arguments give the locations of the enclave image and the configuration, respectively.
int attest_enclave(int enclave_id, Buf input, Buf output)	Used by the host application to generate an attestation for an enclave. The last two arguments give the locations of the input message and the final attestation data, respectively.
int call_enclave(int enclave_id, Buf buffer) [ecall]	Used by the host application to invoke an enclave function. The first argument specifies which enclave to call. The second one is the shared buffer between the host application and the enclave, which is used for storing both input arguments and output results.
int call_host(Buf buffer) [ocall]	Used by an enclave to invoke its host application's function.
int get_seal_key(Buf output_key)	Used by an enclave to get a sealing key which can be used to encrypt some persistent data outside the enclave.
Most interfaces in musl-libc	An enclave can also invoke common POSIX interfaces in musl-libc just like a normal application.

3.3.2 *Enclave Attestation*

Remote attestation enables a remote user to attest whether an enclave is correctly launched and further allows the remote user and the enclave to build a secure communication channel (i.e., exchanging a session key). UniTEE provides the corresponding interface named *attest_enclave*. On SGX-enabled platforms, UniTEE simply uses the hardware-provided remote attestation mechanism. On the other two platforms, UniTEE leverages the reliable microkernel to implement a two-phase attestation. Briefly speaking, in the first phase, it boots the microkernel by using the secure boot mechanism provided by the hardware and allows remote users to negotiate secure keys with the microkernel; in the second phase, the ECS of the microkernel is responsible for launching enclaves, computing the enclave measurement, and signing the measurement by secure keys. As the results of *attest_enclave*, the signed measurement will be returned to the host application and it can be further sent back to remote users for attestation.

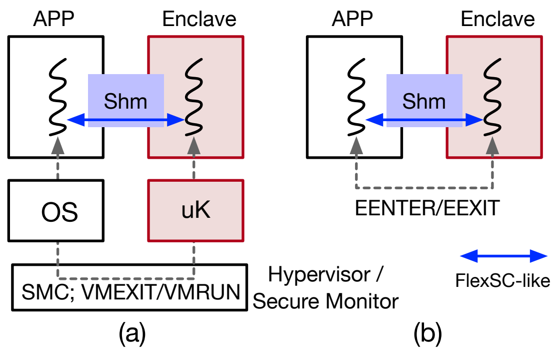


Fig.5. UniTEE supports two modes of communication between an enclave and its host application. The shared communication buffer is abbreviated as Shm.

3.3.3 *Enclave Interaction*

An enclave and its host application can expose function routines for each other, called *ecall* and *ocall* in the official Intel SGX SDK. UniTEE also supports the interaction between enclaves and the host application, and the programming interfaces are *call_enclave* and *call_host*. When the host application thread invokes an enclave function, it needs to transfer the control flow to an enclave thread. As shown in Fig. 5(a), UniTEE can implement the cross-boundary invocation (passive interaction mode) by using a similar method for creating an enclave. However, in such mode, the invocation cost is high because both the *SMC*-based world switching of TrustZone and the *VMMCALL*/ *VMRUN*-

based VM switching of SEV bring both expensive direct cost (thousands of CPU cycles) and indirect cost (pollution to CPU internal structures like cache and TLB). For the sake of performance, UniTEE also provides an alternative interaction mode (proactive interaction mode) which integrates the FlexSC-like mechanism [50]. There is a shared buffer between a host application thread and an enclave thread. The buffer is not only for transferring data but also for transferring control flow. The enclave thread will poll on a request ready flag in the buffer. When the flag is set, which means the application thread makes an invocation request, the enclave thread starts to handle the request and sets a replay ready flag after finishing the request. So, when invoking *call_enclave*, the host application thread writes the arguments of the request to the shared buffer, sets the corresponding request ready flag, and waits for the reply ready flag. After the enclave thread sets the reply ready flag, it retrieves the results of the request and continues the execution. During the request handling procedure, the enclave thread may also invoke functions provided by the host application through *call_host*. If so, it sets the reply ready flag to a specific value, which means it makes an invocation request to the host application thread. Since the latter thread polls on the reply ready flag, it can detect and finish such a request. With this proactive interaction mode, the invocation can be much faster while it requires more CPU cores. The application programmers can select either mode according to the requirements.

Fig. 5(b) shows that UniTEE also enables the two modes of interaction on SGX platforms. In the passive interaction mode, expensive *EENTER* and *EEXIT* instructions are used. In the proactive interaction mode, the FlexSC-like mechanism described above is used instead.

3.3.4 Enclave Library

UniTEE provides an in-enclave C library based on musl-libc for easing programming and supporting running legacy code inside an enclave. The vanilla musl-libc will finally invoke system calls by executing *syscall* (x86-64) or *svc* (AArch64) instruction. Unlike that, the modified library changes the system calls into invocations to the host application and then the host application will invoke the requested system calls on the OS for the enclave. In other words, the system calls issued by an enclave are redirected to the OS on which the host application runs. Note that an enclave belongs to its host application, and the OS will serve it for most system calls. Although the microkernel of UniTEE does implement various system calls like an OS, it provides the ones related to enclave memory management. The enclave library will transparently dispatch the system calls without the involvement of programmers. An existing application can be linked against the modified C library and then directly run in an enclave as a whole. In this case, UniTEE will start a simple host application which just creates the enclave and handles the system calls at runtime for the enclave.

Besides, the library supports another interface named *get_seal_key* which can be invoked by an enclave to get a sealing key. If an enclave needs to store some data for use in the next boot, it can seal the data with this key and store the encrypted data on some untrusted storage. Enclave Key Service, another system service of the microkernel, manages the relationship between the sealing key and the enclave measurement. Therefore, the same enclave (i.e., the same measurement) can retrieve the same sealing key after every boot. For SGX platforms that directly support this functionality, the enclave library uses *EGETKEY* instruction to get the sealing key.

4 Heterogeneous TEE Migration

The unified programming abstraction of UniTEE benefits secure applications development by hiding the heterogeneity from programmers. Furthermore, based on the unified abstraction, UniTEE transparently enables enclave migration between different platforms. We focus on the enclave migration in this paper because: the migration of an application's non-enclave part has no significant difference from traditional migrations, which has been detailedly presented in prior studies. In this section, we first give an overview of the whole enclave migration process and then explain the detailed techniques used during the migration.

4.1 Overview of the Migration

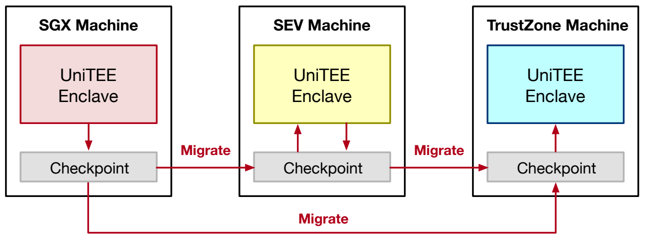


Fig.6. Secure enclave migration between different platforms.

Briefly speaking, as shown in Fig. 6, a migration process includes the following three steps: first, an enclave checkpoint is generated on the source machine; second, the checkpoint is transferred to the target machine through the network; third, the checkpoint is used to restore the running states and resume the execution of the enclave on the target machine.

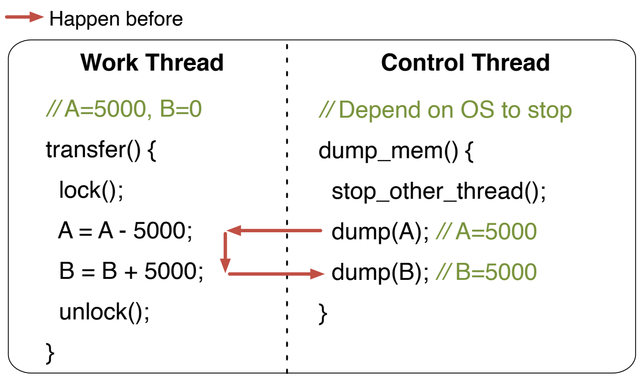
Compared with the traditional checkpoint generation of application migration, there are three differences that make the enclave checkpoint generation challenging. 1) The enclave states cannot be accessed by any system software (e.g., OS), which means they cannot help to generate the enclave checkpoint; 2) the system software may be compromised and launch consistency attacks during the generation procedure; 3) the instructions and calling convention used by enclaves are different on heterogeneous TEEs.

To overcome the first challenge, UniTEE enables each enclave to generate its own checkpoint, i.e., an enclave encrypts and then dumps out its states as a checkpoint without the involvement of others like the OS. Considering state consistency, an enclave needs to stop the enclave threads before generating the checkpoint. Otherwise, the checkpoint may be inconsistent, i.e., it consists of both old and new data. Also, a malicious OS may schedule enclave threads during checkpoint generation to break the state consistency. To overcome this second challenge, UniTEE enables an enclave to make all its threads enter into a quiescent point (make no further updates) before checkpoint generation. Besides, the underlying hardware TEEs on the source and the target machines can be heterogeneous. UniTEE integrates the heterogeneous migration techniques proposed in Popcorn [51] to solve the heterogeneity challenge (the third one). During the generation process, UniTEE will transform the architecture-dependent states according to the target machine's architecture.

Before receiving the enclave checkpoint, the target machine will first launch a virgin enclave with the enclave binary for its architecture. The virgin enclave will receive the checkpoint from the source enclave and use the checkpoint to resume the execution. For securely transferring the checkpoint from the source enclave to the target enclave, the two enclaves will negotiate a migration key with each other by using the widely-used Diffie-Hellman key exchange protocol whose crux is the mutual authentication between the two participants. As UniTEE enables remote attestation (introduced in Subsection 3.3), the source enclave and the target enclave can attest each other to finish the key exchange protocol and then generate the migration key (stored inside the enclaves). Before writing the checkpoint out, the source enclave will first calculate the checksum of the checkpoint and then encrypt it together with the checksum by using the migration key. Since the checkpoint is encrypted when it is outside the enclave or in the network, the untrusted software like OS cannot break confidentiality and integrity.

4.2 Preparation for Checkpointing

UniTEE introduces *control thread*, an extra enclave thread, to assist migration. Since the control thread runs within an enclave, it can traverse and dump the entire memory data within the enclave boundary as the checkpoint. To ensure state consistency of the generated checkpoint, it has to make all the other enclave threads (worker threads) suspend running before starting the generation. Otherwise, it may get a checkpoint with inconsistent data because a worker thread may update some memory during the generation process. As a user-level thread, the control thread cannot directly suspend all worker threads. However, if it asks the OS for help, a malicious OS can deceive the control thread that all enclave threads are suspended but actually not, which will violate the consistency of checkpoint.



```
//A=5000, B=0              //Depend on OS to stop
transfer() {               dump_mem() {
  lock();                    stop_other_thread();
  A = A - 5000;              dump(A); //A=5000
  B = B + 5000;              dump(B); //B=5000
  unlock();                }
}
```

Fig.7. An example of data consistency attack.

Fig. 7 presents a simple example of such a data consistency attack. When a migration begins, a worker thread in an enclave is transferring money from account A to account B. The control thread calls *stop_other_thread()* to ask the OS to stop all other enclave threads. However, the malicious OS returns OK but actually does not stop the worker threads. Thus, the control thread may get an old version of account A (5000) and a new version of account B (5000), which violates the invariant that the sum of accounts should be 5000.

```
void migration_stub(void)
{
  if (global_flag == set) {
        transform_stack();
        local_flag = spin;
        while (global_flag == set) ;
  }
}
```

Fig.8. Pseudo code of the migration stub.

Instead of relying on the untrusted OS, the control thread makes the worker threads to reach a quiescent point as follows. When receiving a migration notification (e.g., through a user-defined signal like SIGUSR1), the control thread is wakened up and then sets a global flag in the enclave to indicate the start of the suspending process. There is a global flag for each enclave and a local flag for each worker thread. Initially, the global flag is unset, and the local flags are *free*. Each worker thread sets its local flag to *busy* and *free* at the enclave entry point and exit point, respectively. Each worker thread normally runs until meeting a migration stub (see Fig. 8), in which it first checks whether the global flag is set. If not, it continues to execute normally. It yes, it performs stack transformation, sets its corresponding local flag to *spin*, and then enters the spin region. The stack transformation is for transforming the execution stack according to the target architecture, which will be explained in Subsection 4.3. When running in the spin region, a worker thread will not change any memory and will keep in the region, until it finds that the global flag is unset. The control thread will wait for the point when all the local flags of worker threads are either *free* or *spin* (i.e., not running or in the spin region) before generating the enclave checkpoint. Therefore, it can ensure the consistency of the checkpoint without the help of the untrusted OS.

UniTEE inserts migration stubs before the ocalls. So, a running enclave thread will respond to the migration when it invokes an ocall. Nevertheless, some worker threads may execute in the enclave for a long time without performing an ocall. It is very likely that such a thread has already set its local flag to *busy* when the control thread sets the global flag. If so, the control thread needs to wait for a long time, which will block the process of migration. To this end, UniTEE also allows programmers to insert migration stubs in their code as they want.

4.3 Hiding Heterogeneity for Migration

Since the three popular hardware TEEs, namely Intel SGX, AMD SEV, and ARM TrustZone, are provided on different architectures, UniTEE also has to transparently hide the heterogeneity during the enclave migration. We explain the detailed techniques from the following four main aspects.

1) How to Migrate Code. Different hardware TEEs must use the corresponding CPU instructions. Therefore, UniTEE compiles different enclave binary codes for different hardware TEEs, and an enclave will use the corresponding binary code according to the underlying hardware TEEs. The key point of the compilation is that each function in the different binary is located at the same start address. Therefore, function pointers are always valid after migration, which eases the migration, i.e., no need to update the pointers.

2) How to Migrate Data. UniTEE targets 64-bit and little-endian because the first two types of TEEs support 64-bit only and all three types use little-endian. Thus, the data format needs no transformation across the three architectures, e.g., the data format of a struct written in C is always the same for the three TEEs. Like the function start address, each global variable address is also located at the same address. Therefore, the global data section and the heap area (using the same heap start address) can be directly copied from the source enclave to the target enclave. The validity of data pointers is inherently preserved after migration, which significantly eases the migration process.

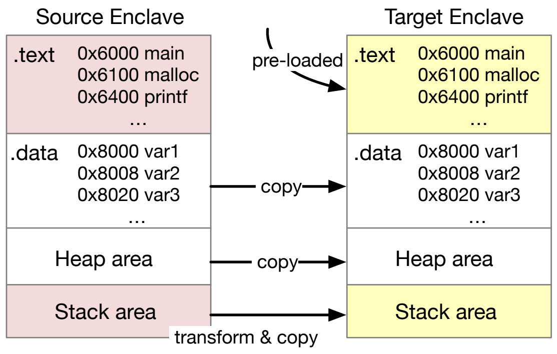


Fig.9. The memory layout of the source enclave and the target enclave. Colored parts are architecture-dependent.

As shown in Fig. 9, UniTEE generates multiple enclave binaries for each architecture. An enclave binary can be distributed to the target machine before migration or when migration is triggered. When a migration begins, the target machine boots the virgin enclave that will receive the checkpoint from its source enclave. Thus, the code section does not need to be transferred during migration. As described above, the enclave binaries for different TEEs share a uniform address space layout, i.e., every symbol address is kept the same, and every data structure uses the same memory format. Therefore, simply copying the data section and heap area will not make any pointer invalid. In other words, these two areas are transferred without any transformation.

3) How to Migrate Execution Context as well as Execution Stack. Different architectures provide different numbers of general-purpose registers (GPR) and use different calling conventions. For example, there are 16 GPRs for Intel SGX (x86-64) while 32 GPRs in ARM TrustZone (AArch64). And the calling conventions for them vary widely, which makes the execution stack different. A simple approach to solving this challenge is to use the same number of registers and the same calling convention. In such a way, it is easy to give an explicit one-to-one mapping to connect registers in different ISAs and simply copying the stack area can also work for migration. However, this approach leaves many registers unused and abandons originally applicable optimizations, which may hurt the performance enormously.

Instead, UniTEE adopts and implements the stack-/register transformation proposed by prior work [51]. The basic idea is recording each stack frame's information at compile time and then reconstructing the execution stack frame by frame for the target architecture at migration time. The inserted migration stubs ensure a migration always happens at the function boundaries, which means the stack frame to transform is always intact. Specifically, the compilation toolchain is based on LLVM and the information (including live variables and the calling site) of each stack frame is recorded according to the Intermediate Representation (IR) of LLVM. For each specific architecture, a live variable is either mapped to a register or on the stack. Therefore, according to such information, the transformation procedure will reconstruct a new stack as well as set the registers for the target architecture.

Implementation details for the transformation: During the compilation process, all the stack frame information is recorded in a particular section of the binary, which mainly includes locations of live variables (either on the stack or in a register). The transformation procedure first calculates the size for the new stack according to the recorded information. Then it rebuilds the new stack from the outermost frame to the innermost frame (frame by frame). For rewriting one stack frame, it gets all the live variables of that frame in the source enclave binary, queries their locations in the target enclave binary, and then copies them to the new locations. A special case is that a variable is a pointer that points to some stack address. In this case, the variable cannot be directly copied because its value (some stack address) should be changed after the stack is transformed. Instead, it will be recorded in a fixup list for resolving later. Every time the stack transformation procedure copies a variable, it checks whether the variable (*V1*) is pointed by some variable (*V2*) in the fixup list. If so, it removes *V2* from the list and sets the new location of *V1* (on the target enclave stack) to *V2* for the target enclave. Also, the return address of each stack frame is also rewritten according to the calling site information.

At the migrate point, some live variables may be stored in registers. Therefore, besides transforming each stack frame, it is also necessary to restore these in-register variables for the target enclave. According to the information recorded during compilation, the transformation procedure knows the location of each live variable on different architectures and thus can simply set the corresponding registers for the target enclave.

4) How to Migrate OS-Related States. UniTEE also allows an enclave to invoke system calls, as described in Subsection 3.3. Therefore, it is also necessary to migrate OS-related states from the source enclave to the target enclave. Currently, UniTEE supports restoring file descriptors and TCP connections. For file descriptors, it records the states (e.g., file path, file descriptor, and cursor) of each opened file in the modified enclave library that redirects the system calls. These states and related files are also transferred during migration. During the restoring process, the target enclave reopens each file and sets the cursor to the right position. For migrating TCP connections, UniTEE refers to CRIU [3]. The Linux kernel (since version 3.5) has supported the TCP connection repair mechanism to help with sockets transmission. UniTEE first uses TCP_REPAIR option to switch the socket into a special mode for the source enclave. It then collects and transfers necessary TCP states. Last, on the target side, the TCP states will be restored, and the socket mode will be reset to normal for the target enclave.

5 Evaluation

We conduct performance evaluations on the prototype of UniTEE and present the results in this section. The experiments include the enclave migration between Intel SGX and ARM TrustZone, between Intel SGX and Intel SGX, and between Intel SGX and AMD SEV.

5.1 Between SGX and TrustZone

We conduct experiments on two machines that support Intel SGX and ARM TrustZone, respectively. One is equipped with Intel Core i7-9700 CPU and 16 GB memory, and the other is an HiKey970 board with 6 GB memory. We run Ubuntu 16.04 on both machines while the Linux kernel versions are 4.15.0 and 4.9.78. We run each experiment over 30 times and report the average of the results. The standard deviation is within 5% across all the experiments.

We select several SPEC CPU 2006 benchmarks and vedis (a popular key-value store) as applications. An application is protected as a whole and runs in an enclave of UniTEE. We do not modify the source code of the applications except for inserting some migration stubs. For the SPEC CPU benchmarks, the workloads are the built-in *ref* test suites. For vedis, we generate 10 million random keys and perform PUT and GET operations (50% PUT and 50% GET) randomly.

Overhead of Migration Support. We first present an experiment on the overhead introduced by supporting migration. We compile the chosen benchmarks with and without migration support, run them in enclaves, and measure the execution time. To disable migration support, we do not link the applications with migrated-related libraries or insert any migration stubs. Fig. 10 shows the results of some benchmarks (others are similar). We normalize the results to the no-migration-support version for better readability.

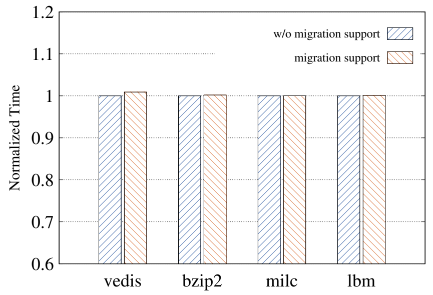


Fig. 10. Execution time with and without migration support. The results are normalized to the no-migration-support version.

Migration Cost. Table 2 gives the breakdown time of migrating applications from the Intel machine to the ARM machine. To facilitate analysis, we divide the whole migration procedure in both machines into different phases: preparation phase, checkpoint phase, transmission phase, and restore phase. The time consumed by booting the virgin target enclave is not reported because this procedure is out of the critical path, i.e., has finished before the final restore phase.

This evaluation shows the migration support brings nearly zero overhead. It makes sense because UniTEE only requires each thread to do two extra operations

[3]CRIU: Checkpoint/Restore In Userspace. `https://www.criu.org/Main_Page`. Referenced February 2021.

Table 2. Breakdown of the Migration Cost

Benchmark	**Preparation phase (μs)**	**Checkpoint Phase (μs)**	**Transmission Phase (μs)**	**Restore Phase (μs)**
vedis	932	155,818	1,140,314	71,888
bzip2	543	618,690	2,526,885	162,563
milc	916	399,044	1,880,252	113,955
sjeng	713	529,231	2,173,368	150,601
libquantum	755	294,514	1,378,012	90,352
h264ref	598	206,130	1,261,305	76,975
lbm	701	1,251,807	4,629,325	410,445

Note: Preparation and checkpoint phases happen on the source machine. Transmission phase is for transferring the enclave checkpoint through the network. Restore phase happens on the target machine.

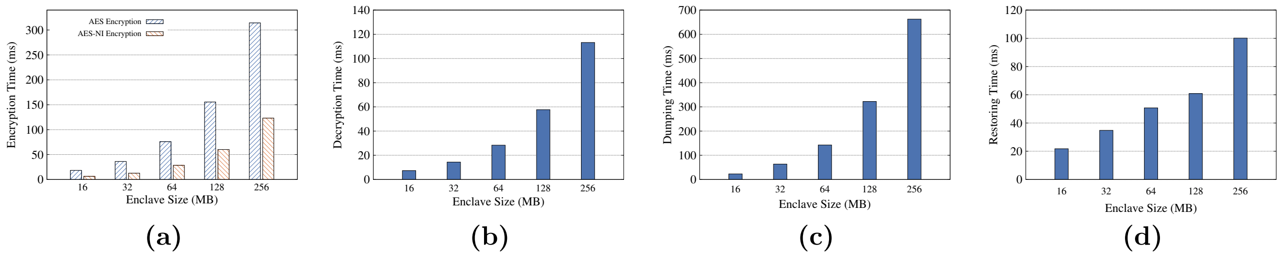


Fig.11. Time consumed in the checkpoint/restore phase. Figure (a), (b), (c) and (d) show the encryption, decryption, dumping and restoring time, respectively.

during normal execution for supporting migration. The first one is to initialize the local migration flag, which can be ignored since it only happens once. The second one is to check the global migration flag at each migration point. However, checking the flag only requires several instructions, which is negligible compared with real workloads. Therefore, migration support does not influence performance during normal execution.

Preparation Phase. This phase mainly consists of the time of waiting for the quiescent point and performing stack/register transformation. As shown in Table 2. it takes less than 1% of total migration time in all benchmarks. Here each enclave only has one worker thread, so the quiescent point is reached once it meets the first migration point. Stack transformation can be done in a short time because all necessary information has been stored in binaries during compilation and the depth of the stack is usually not deep.

Checkpoint/Restore Phase. After the preparation phase, the enclave (control thread) encrypts its memory data and dumps the encrypted data outside the enclave, which is called checkpoint phase here. Similarly, for restoring the checkpoint in the target enclave, the checkpoint needs to be copied into the enclave and then decrypted. For better performance, the checkpoint only contains the enclave memory in use. Firstly, the code section of the enclave does not need to be dumped, since the binary code for different architectures is different and can be placed on the target machine in advance. Secondly, the size of the data section is known at compile time and does not change at runtime. The data section is included in the checkpoint. Thirdly, only the in-use enclave heap region is dumped to the checkpoint. To be specific, the in-use heap region consists of two parts: one is from the heap base to the heap top; the other is a list of memory-mapped areas (i.e., through mmap). Fourthly, only the valid stack regions are dumped according to the stack pointers.

The cost of the checkpoint phase (on the Intel SGX machine) is obviously higher than that of the restore phase (on the ARM TrustZone machine). This is because accessing the SGX-protected memory is much more expensive, especially when SGX page swapping happens.

The cost of such two phases is related to the in-use enclave memory size. Therefore, we give a further analysis of the cost and Fig. 11 shows the results. Fig. 11(a) and 11(b) show the time spent on encryption and decryption. The encryption mechanism is Advanced Encryption Standard (AES) and some hardware-assisted acceleration can be used. For example, Fig. 11(a) shows Intel AES-NI [52] instructions can reduce over 60% of encryption time. The time consumed in dumping (writing the checkpoint to outside) and restoring (copying the checkpoint into an enclave) is reported in Fig. 11(c) and 11(d), respectively. The result shows the dumping and restoring time grows linearly as the enclave size increases, because dumping and restoring are actually memory copy operations.

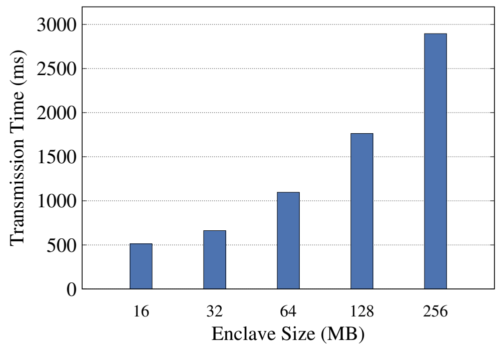


Fig.12. The network transmission time.

Transmission Phase. Fig. 12 presents the time of transferring enclave checkpoints with different sizes. The two machines connect to the same LAN, and the bandwidth is about 115 MB/s. The evaluation results show the time consumed in the transmission phase increases along with the enclave size grows. According to Table 2, this phase takes most of the migration time (over 78%). With faster network, the total migration latency can be significantly reduced.

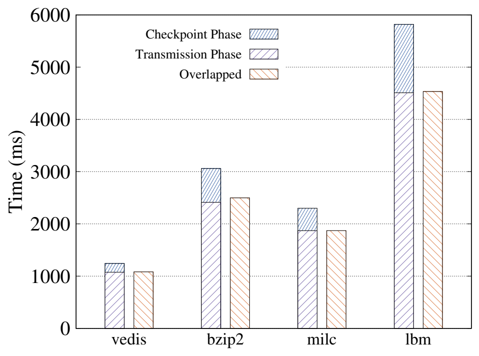


Fig. 13. Time saved by overlapping phases in the source machine.

Overlapping Phases. To further decrease the migration latency, we pipeline the execution of the checkpoint/transmission/restore phases by dividing the whole enclave checkpoint into pieces. Fig. 13 shows the time saved by using this strategy on the source machine. The total latency can be reduced by up to 30%. Therefore, the total downtime for the enclave applications is bottlenecked/decided by the network speed. We conclude the solution proposed by UniTEE is feasible.

5.2 Between Two SGX Machines

We also measure the performance of enclave migration between two laptops with Intel Core i7-6700HQ 2.6GHz CPU and 8 GB memory. The experiment is migrating a virtual machine (VM) with and without enclaves running inside. KVM is chosen as the underlying hypervisor, and the version of QEMU is 2.5.0. The guest VM has 4 VCPUs (Virtual CPU) and 2 GB memory.

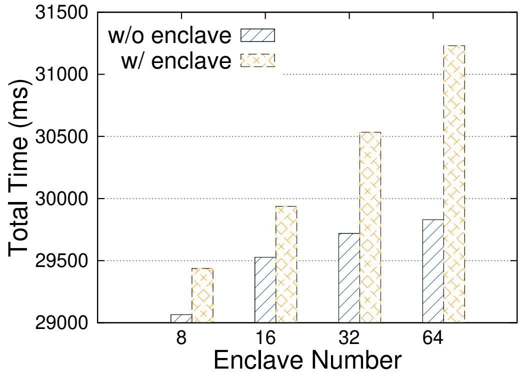


Fig.14. Total migration time w/ and w/o enclaves. Note that the x-axis does not starts from 0.

We run two VMs respectively, one with some running enclave applications, the other with the same number of original applications. The enclaves run either *libjpeg* or *mcrypt*, which are real-world applications. The enclave size is 1MB, and the workload is an endless loop of picture decoding or encryption. Fig. 14 shows the total migration time. The migration of VM with no more than 32 enclaves has about 2% overhead. The overhead increases to 5% when the number of enclaves reaches 64.

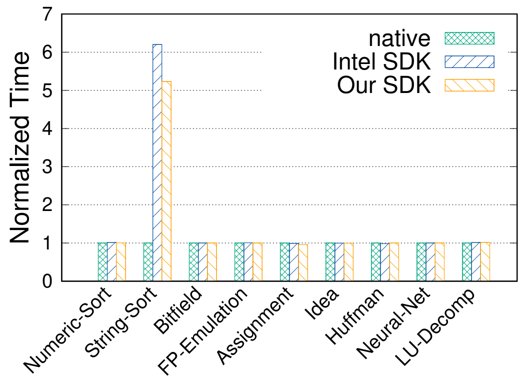


Fig.15. SDK performance comparison on nbench.

Besides, we conduct an experiment to compare the SDK performance, i.e., UniTEE and Intel official SGX SDK. The benchmark is nbench 2.2.3 in which most applications are computation intensive. *String Sort* is the one that accesses much more secure memory, which leads to high SGX paging overhead. As shown in Fig. 15, our SDK shows similar or better performance compared with Intel SDK.

5.3 Between SGX and SEV

Intel SGX and AMD SEV are two security extensions to x86-64 and they share the same general purpose registers as well as calling convention. Compared with the migration between SGX and TrustZone, the migration between SGX and SEV needs no stack transformation while the other procedures are the same. Fig. 16 shows the time for generating the enclave checkpoint. The AMD machine is equipped with EPYC 7281 CPU that supports SEV.

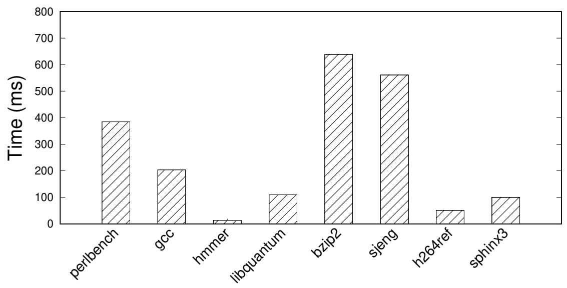


Fig.16. The cost for generating checkpoints.

Besides, we measure the execution time of 12 applications in SPEC CPU 2006 benchmarks on the SEV machine and present the results in Fig. 17. This experiment is to show the protection overhead of UniTEE on the SEV machine. For CPU-intensive benchmarks such as bzip2 and gobmk, the performance of enclaves is nearly the same as or even better than the native execution performance. Two reasons can explain: first, UniTEE will not bring overhead to enclave applications when they do not invoke ocalls; second, when executing ocalls (system calls), the FlexSC-like design (described in Section 3.3) avoids the context switches for the enclave threads although it requires some extra cycles. Context switches between user-mode (ring 3) and kernel-mode (ring 0) may incur indirect costs like TLB/cache pollution. Other applications show less than 5% overhead which mainly comes from the memory copies (transferring arguments and results) during ocalls.

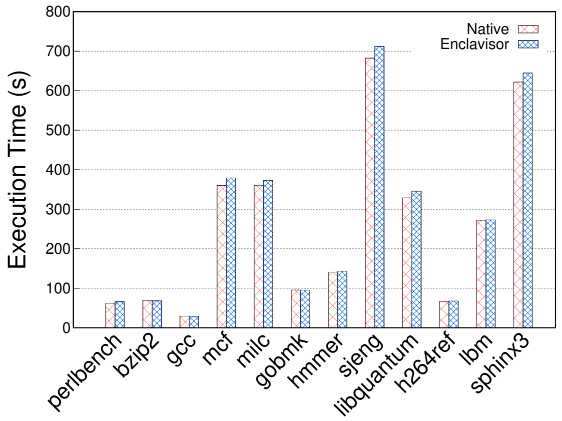


Fig.17. Runtime overhead of UniTEE on the SEV machine.

6 Discussion

As mentioned in Section 3.2, the microkernel used in UniTEE is included in the TCB. It makes sense to assume the microkernel is trusted because it only provides simple and clear functionalities and has a small code base. This is also a common assumption (e.g., TrustVisor [53], CloudVisor [54], Nested Kernel [55]). It may also be feasible to build UniTEE's microkernel over a formally verified OS kernel (e.g., Hyperkernel [56] and seL4 [57]) and further verify the entire one with more effort. Besides, the implementations of existing hardware enclaves also heavily rely on software. For example, as mentioned in "Hardware is the new software" [58], much of SGX's logic is implemented by microcode, which can be patched on-the-fly just as software. Meanwhile, many researchers try to build enclaves from software (with the help of hardware), like Komodo [59]. We certainly agree that from the perspective of security, it is more preferable to construct the TCB in a more simple and predictable way. But we argue that the point here is more about the level of semantics instead of being hardware or software. In our design, we try to move some of the hardware logic from firmware (e.g., on AMD PSP) to software running in the secure TEE, which has low-level semantics, instead of developing a new complex software like a guest OS.

7 Related Work

Enclave Programming Model. The strong security insurance of hardware TEEs motivates a variety of prior studies [18, 19, 21, 36, 45, 60, 61] to protect applications by leveraging one of Intel SGX, ARM TrustZone, and AMD SEV. However, they do not focus on providing a unified enclave programming model for hiding the underlying hardware security technologies. Open Enclave SDK [4] aims to provide consistent API surface across enclave technologies as well as all platforms from cloud to edge, which shares the same goal of the unified programming abstraction of UniTEE. Nevertheless, Open Enclave SDK considers SGX and TrustZone while UniTEE further considers SEV. Moreover, UniTEE enables enclave migration across those platforms while Open Enclave SDK does not.

Microkernel Usages. There is a long line of research on microkernel OS [57, 62, 63, 64, 65, 66]. Owing to the desired advantages including good security and fault isolation, microkernel has been used in some safety-critical scenarios like vehicles. Designing microkernels for general-purpose scenarios is also on the way. Nevertheless, UniTEE leverages the microkernel for constructing isolated enclave instances in a single hardware TEE. A recent work [67] also proposes to design a TEE OS based on the microkernel architecture. Different from that, the microkernel of UniTEE only manages the enclave life cycle without providing various OS services through system calls because most system calls are redirected to the full-fledged OS which runs the

[4]Open Enclave SDK. `https://openenclave.io/sdk/`. Referenced February 2021.

host application. The microkernel used here is derived from [65].

Heterogeneous Migration. Live migration between heterogeneous architectures has been studied by prior work [22, 24, 68, 69]. UniTEE adopts and extends the existing migration techniques of Popcorn [22, 24] to migrate the secure enclaves among different TEE hardware. The major difference is that the OS is not trustworthy. During migration, UniTEE relies on the OS functionalities without trusting it. For example, UniTEE requires the OS to transfer the enclave checkpoint through the network but protects the consistency, confidentiality, and integrity of the checkpoint. [25] makes efforts to securely migrate SGX enclaves on untrusted cloud, which, however, does not provide unified enclave abstraction or enclave migration support on heterogeneous platforms.

8 Conclusion

This paper proposed UniTEE whose target is twofold. It provides a unified enclave programming abstraction that can help programmers to write enclave applications without considering the underlying hardware TEEs. Further, with the unified abstraction, it enables secure enclave migration between heterogeneous platforms.

References

[1] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. Streambox-tz: secure stream analytics at the edge with trustzone. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 537–554, 2019.

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[3] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing-a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[4] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[5] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[6] Biljana L Risteska Stojkoska and Kire V Trivodaliev. A review of internet of things for smart home: Challenges and solutions. *Journal of Cleaner Production*, 140:1454–1464, 2017.

[7] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[8] Andrew Machen, Shiqiang Wang, Kin K Leung, Bong Jun Ko, and Theodoros Salonidis. Live service migration in mobile edge clouds. *IEEE Wireless Communications*, 25(1):140–147, 2017.

[9] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. A survey on service migration in mobile edge computing. *IEEE Access*, 6:23511–23528, 2018.

[10] Mofijul Islam, Abdur Razzaque, and Jahidul Islam. A genetic algorithm for virtual machine migration in heterogeneous mobile cloud computing. In *2016 International Conference on Networking Systems and Security (NSysS)*, pages 1–6. IEEE, 2016.

[11] Antonio Barbalace, Mohamed L Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: the case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 73–87, 2020.

[12] T. G. Rodrigues, K. Suto, H. Nishiyama, N. Kato, and K. Temma. Cloudlets activation scheme for scalable mobile edge computing with transmission power control and virtual machine migration. *IEEE Transactions on Computers*, 67(9):1287–1300, 2018.

[13] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.

[14] Z. Ning, J. Liao, F. Zhang, and W. Shi. Preliminary study of trusted execution environments on heterogeneous edge platforms. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 421–426, 2018.

[15] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[16] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper, Apr*, 2016.

[17] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

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

[19] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[20] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[21] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

[22] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. *ACM SIGARCH Computer Architecture News*, 45(1):645–659, 2017.

[23] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[24] Antonio Barbalace, Mohamed L Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: the case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 73–87, 2020.

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

[26] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2012.

[27] Matthew Furlong, Andrew Quinn, and Jason Flinn. The case for determinism on the edge. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[28] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017.

[29] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142. IEEE, 2017.

[30] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud migration research: a systematic review. *IEEE transactions on cloud computing*, 1(2):142–157, 2013.

[31] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1450–1465. IEEE, 2020.

[32] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556, 2017.

[33] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, 2009.

[34] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. A framework and algorithm for energy efficient container consolidation in cloud data centers. In *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*, pages 368–375. IEEE, 2015.

[35] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1846–1855, 2017.

[36] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 33(3):8, 2015.

[37] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, 2016.

[38] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Sebastian Nowozin, Aastha Mehta, Felix Schuster, and Kapil Vaswani. Sgx-enabled oblivious machine learning. 2016.

[39] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proc. of the Annual Network and Distributed System Security Symp.(NDSS)*, 2017.

[40] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *S&P*, pages 38–54, 2015.

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

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

[43] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[44] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure onetime password tokens. In *CCS*. ACM, 2015.

[45] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ASPLOS*, 2014.

[46] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution on arm processors. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 72–90. IEEE, 2016.

[47] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501, 2017.

[48] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1723–1740, 2019.

[49] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.

[50] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 33–46. USENIX Association, 2010.

[51] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[52] Jeffrey Rott. Intel advanced encryption standard instructions (aes-ni). Technical report, Technical report, Intel, 2010.

[53] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010.

[54] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, pages 203–216, 2011.

[55] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. *ACM SIGPLAN Notices*, 50(4):191–206, 2015.

[56] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[58] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS'17, pages 132–137, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 287–305,

New York, NY, USA, 2017. Association for Computing Machinery.

[60] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[61] Jinyu Gu, Xinyu Wu, Bojun Zhu, Yubin Xia, Binyu Zang, Haibing Guan, and Haibo Chen. Enclavisor: A hardware-software co-design for enclaves on untrusted cloud. *IEEE Transactions on Computers*, 2020.

[62] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[63] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[64] Francis M David, Ellick Chan, Jeffrey C Carlyle, and Roy H Campbell. Curios: Improving reliability through operating system structure. In *OSDI*, volume 8, pages 59–72, 2008.

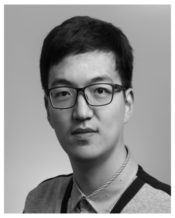
[65] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference*, pages 401–417, 2020.

[66] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[67] Dongxu Ji, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. Microtee: Designing tee os based on the microkernel architecture. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 26–33. IEEE, 2019.

[68] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 261–272, 2012.

[69] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, 2012.



Jin-yu Gu received his BS degree in software engineering from Shanghai Jiao Tong University, Shanghai, in 2016. He is now a Ph.D. candidate at the School of Software, Shanghai Jiao Tong University. His research interests include operating systems, computer architecture, and security.

Hao Li received his BS degree in software engineering from Shanghai Jiao Tong University, Shanghai, in 2020. He is a Master student at the School of Software, Shanghai Jiao Tong University. His research interests include computer architecture and security.

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

Hai-Bo Chen received his B.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004 and 2009, respectively. He is currently a Professor and the Director of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a Distinguished member of both CCF and ACM. His research interests include operating systems, and parallel and distributed systems.

Cheng-gang Qin received the PhD degree in computer science from Graduate School of The Chinese Academy of Sciences, in 2012. He is currently a senior technicial expert in the Ant Group. His research interests include operating system, computer architecture and system security.

Zheng-yu He received the PhD degree in computer engineering from Georgia Institute of Technology, in 2012. He currently leads the technical infrastructure in Ant Group, and his research interests include operating systems, programming models and software engineering.