Secure Live Migration of SGX Enclaves on Untrusted Cloud

Jinyu Gu[†‡], Zhichao Hua[†‡], Yubin Xia[†‡], Haibo Chen[†‡], Binyu Zang[†], Haibing Guan[‡] and Jinming Li[*]
[†]Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University
[‡]Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
[*]Huawei Technologies, Inc.
Email:{gujinyu,huazhichao,xiayubin,haibochen,byzang,hbguan}@sjtu.edu.cn, lijinming@huawei.com

***Abstract*—The recent commercial availability of Intel SGX (Software Guard eXtensions) provides a hardware-enabled building block for secure execution of software modules in an untrusted cloud. As an untrusted hypervisor/OS has no access to an enclave's running states, a VM (virtual machine) with enclaves running inside loses the capability of live migration, a key feature of VMs in the cloud. This paper presents the first study on the support for live migration of SGX-capable VMs. We identify the security properties that a secure enclave migration process should meet and propose a software-based solution. We leverage several techniques such as two-phase checkpointing and self-destroy to implement our design on a real SGX machine. Security analysis confirms the security of our proposed design and performance evaluation shows that it incurs negligible performance overhead. Besides, we give suggestions on the future hardware design for supporting transparent enclave migration.**

I. Introduction

Along with a long line of active research [11], [12], [5], [4], [29], [25], the recent release of Intel Skylake processors marks the transition of hardware-secured execution (i.e., Intel SGX [1], [9], [13]) from research proposals to mainstream reality. By providing a hardware-secured execution environment called *enclave*, Intel SGX is a promising technique to directly provide data confidentiality and tamper-resistant execution to applications, without trusting the whole software stack including the hypervisor, operating system and runtime environment. The features such as minimized TCB (Trusted Computing Base) and strong security protection are extremely useful for outsourced computation in the multi-tenant cloud, where the infrastructure owner may be curious or even malicious.

There have been several prior studies of using Intel SGX for security protection. Baumann et al. [3] illustrated how to leverage SGX to protect an entire LibOS including both OS and applications; Schuster et al. [16] presented a case of using SGX for trustworthy data analytics in an untrusted cloud; Hunt et al. [10] used SGX to implement a distributed sandbox for hosting untrusted computation on secret data; Arnautov et al. [2] designed a secure Linux container with SGX.

Unfortunately, with SGX being increasingly deployed to outsourced computation in an untrusted cloud, a VM with enclaves running inside loses the capability of live migration, which is widely used in cloud computing, e.g., for load balancing, fault tolerance. This is because the SGX hardware prevents an untrusted hypervisor/OS accessing an enclave's running states, which is necessary for traditional live migration.

In this paper, we present the first design and implementation of securely live migrating [1] a VM with enclaves running inside. First, we identify all the challenges of enclave migration introduced by SGX and list all the security properties that such migration should preserve. Then we design a security-oriented software-based migration mechanism that allows an enclave to transfer its running states from the source machine to the target machine on which it resumes the execution. The migration process requires the cooperation between enclaves and untrusted privileged software.

Specifically, we introduce a *control thread* running in each enclave to assist migration, which securely dumps all of the enclave's states from inside. For the states like data in memory and CPU context, the control thread will encrypt them and use checksum for integrity protection before dumping; for the states that cannot be accessed directly by software, e.g., the CSSA (Current State Save Area) maintained by processor, we carefully design a bookkeeping mechanism to infer the value within the enclave. Besides, being aware of possible *data consistency attacks* from an untrusted OS and inspired by the *two-phase commit* [24] in distributed computing, we reinforce our design through a *two-phase checkpointing* scheme to create a *quiescent point* when all the enclave threads reach a quiescent state. To further defend against *fork attack* and *rollback attack*, we leverage *remote attestation* (without user involvement) and *self-destroy* to guarantee that each enclave instance will not be rolled back or generate multiple instances after migration.

We have implemented the above design based on KVM by providing an SGX library for applications and SGX support in KVM as well as guest OS. We also provide an SDK (Software Development Kit) for developers so that they can write code running in an enclave without awareness of our mechanism for migration, e.g., the control thread. We present a real implementation and evaluation on the Intel Skylake machine and share our experience of successful implementation without relying on Intel SGX SDK for Linux [2]. The evaluation results show that our system's performance overhead is negligible: to migrate a VM with 64 enclaves running inside, the total time of migration grows by 4.7%, and the downtime increases by only 3 milliseconds. At last, we propose several suggestions on hardware support for migrating an enclave transparently,

[1]We uniformly term VM suspension, resuming and live migration as live migration since the key steps of live migration involve suspending and resuming a VM.

[2]We have implemented our own SDK before the release of Intel Linux SDK.

which may be considered in future extension of SGX.

In summary, this paper makes the following contributions:

- A study on the challenges and possible attacks of secure live migration of enclaves in a VM.
- A design and implementation with a set of techniques to address subtle security issues in migration.
- A detailed security analysis and performance evaluation as well as a set of design suggestions for future SGX extensions.

II. Motivation

A. Background of Intel SGX Technology

Intel SGX (SGX for short) allows an application to instantiate a protected execution environment, referred to as an *enclave*, in the application's address space. Figure 1 gives the memory layout of an enclave. Accesses to the enclave memory area from any software not resident in the enclave are forbidden. It reduces the TCB to only the processor and the code running in enclaves. In this subsection, we summarize the SGX functionality relevant to our work.

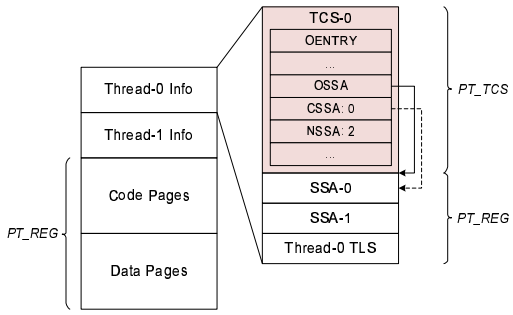


Fig. 1. Memory layout of an enclave with two threads. A thread's states include one TCS and multiple SSAs. A TCS is saved in a PT_TCS page. Other data is saved in PT_REG pages, which can be accessed in the enclave.

Memory Protection: **EPC** (Enclave Page Cache) is a secure storage used by the processor to store enclave pages when they are a part of an executing enclave. The EPC is divided into chunks of 4 KB pages. The processor tracks the metadata of the EPC in a secure structure called **EPCM** (Enclave Page Cache Map), which is only accessible by hardware. SGX provides a set of instructions for adding and removing content to and from the EPC pages. Typically, BIOS can reserve a range of main memory for EPC at boot time. The contents of the EPC are protected by Intel MEE (Memory Encryption Engine). The enclave pages can only be accessed by the corresponding enclave owner.

Memory Swapping: The EPC is a finite platform asset and should be managed by privileged software. Intel SGX includes the *EWB* instruction for securely evicting pages. *EWB* encrypts a page in the EPC and writes it to unprotected memory. In addition, *EWB* also creates a cryptographic MAC (Message Authentication Code) of the page and stores it in unprotected memory and writes a version number in a VA (Version Array) slot. VA is a special type of EPC page. A page can be reloaded back to the EPC with *ELDB* or *ELDU* instruction only if the data, version and MAC match. Note that although a guest OS can make a checkpoint of an enclave by swapping all its memory into an image file on disk, the image cannot be swapped in by another machine, since the evicted pages are encrypted by *Page Encryption Key*, which is unique for each CPU and will never be retrieved outside the CPU.

Important Data Structures: Each enclave has a **SECS** (SGX Enclave Control Structures), which contains the metadata of an enclave. Each enclave needs to have one or more **TCS**s (Thread Control Structure). These two structures are located in EPC and can only be accessed by hardware. When entering an enclave, a thread has to specify one *TCS* which designates one fixed entry. When an enclave's execution is interrupted, the context is saved in the thread's current **SSA** (State Save Area) frame, which is decided by a field of *TCS* called **CSSA** (Current SSA). An enclave can access its own *SSA*s.

Control Flow Transfer: A processor can enter an enclave through *EENTER* instruction with a *TCS*, and exit from an enclave by issuing *EEXIT*. If an enclave's execution is interrupted due to interrupts or traps, the processor will invoke a special internal routine called **AEX** (Asynchronous Enclave Exit) which saves the context, scrubs the processor state and sets the faulting instruction address to a value specified by *EENTER*. *AEX* increases the *CSSA* by 1. Here, the host application of the enclave has two choices: one is issuing *ERESUME* to restore the interrupted context, which will decrease the *CSSA* by 1; the other is issuing *EENTER* to enter the exception handler defined in the enclave. In the latter case, if another exception happens, the processor will perform an *AEX* again and save the current context in a new *SSA*, and increase the *CSSA* again. Then the processor needs to issue two *ERESUME* instructions to resume the initial execution.

Attestation: In SGX, attestation is the mechanism by which another party can gain confidence that the correct software is securely running within an enclave. During enclave construction, the processor computes a digest of the enclave which represents the whole enclave layout and memory contents. **Local attestation** allows an enclave to prove to another enclave that it has a particular digest and is running on the same processor. This mechanism can be used to establish authenticated shared keys between local enclaves. Moreover, SGX enables a particular enclave, called the *Quoting Enclave*, which is devoted to **remote attestation**. Enclave binary should not be shipped with plaintext sensitive data. After launched successfully, the enclave can contact its owner to get the sensitive data. With the help of Quoting Enclave, an enclave can produce a secure assertion that identifies the hardware environment and itself. The enclave owner can use attestation services, e.g., IAS (Intel Attestation Service), to assess the trustworthiness of the assertion. Based on this mechanism, the owner can establish a secure communication channel and provide sensitive data to the enclave.

B. Differences of VM Migration with & without Enclaves

Traditional Live VM Migration: Live VM migration has been a widely-used technology in the cloud. A VM *checkpoint* is usually an image that contains the VM's entire memory data as well as its CPU context. The image is a snapshot of the VM's current state and can be used to resume the execution

of the VM in the future. Live VM migration can be done by simply making a checkpoint of a VM on the current physical machine, transferring the checkpoint to a target machine, and resuming the VM on that machine. A typical optimization of live VM migration is to overlap the checkpointing phase and the transferring phase so that a VM can keep running while its checkpoint data is being transferred through the network.

Difference-1: Enclave states are sealed in hardware. Traditionally, the hypervisor can access all the running states of a guest VM. On SGX platform, the hardware has many new states for each running enclave. All these states are necessary for resuming the enclave's execution on the target machine, including the enclave memory, TCS structure, etc., as presented in Section II-A. However, neither the hypervisor nor the guest OS can access an enclave's memory (EPC) directly. Even if the guest OS can swap all of an enclave's data from EPC to normal memory, these data will be encrypted by hardware and the encryption key will never leave the processor, which makes it impossible for the target machine to restore the execution of the enclave. Moreover, some of the enclave states, e.g., the CSSA, cannot be accessed by any software (even the enclave itself), which brings a significant challenge to enclave migration.

Difference-2: Neither the hypervisor nor the guest OS is trusted. Traditionally, the hypervisor is trusted and can access any state of a guest VM. On SGX platform, the hardware provides an execution environment for each enclave, which is isolated from the potentially malicious hypervisor or guest OS. However, the migration process requires that the enclave states are dumped out of the enclave where there is no hardware protection. A malicious hypervisor may steal or tamper with the states during the process. In Section IV-A, we also show that a malicious guest OS may violate the consistency of states of a multi-threaded enclave by controlling the scheduling. It is necessary to protect the enclave states during migration without trusting the hypervisor or guest OS.

Difference-3: Cloning or rolling back an enclave could be malicious. Traditionally, a cloud operator is allowed to clone a guest VM to get multiple instances or rollback a guest VM to some previous checkpoint. For example, a VM running a web server can be automatically replicated once its workload increases significantly, which is known as "resilient computing". Nevertheless, cloning or rolling back an enclave might be considered as a malicious behavior. On SGX platform, once an enclave starts to run, all of its states are protected by hardware. Thus the OS has no way to clone or rollback the enclave. However, if enclave migration is enabled, a malicious cloud operator might issue *fork attack*s or *rollback attack*s, as presented in Section V-A. How to defend against these attacks while enabling enclave migration is also a challenge.

C. Security Properties

In order to enable secure migration of a guest VM with enclaves, the system needs to achieve the following security properties:

- **P-1: State confidentiality**. The states of a migrating enclave will not be leaked during migration.
- **P-2: State integrity**. The states of a migrating enclave will not be tampered with during migration.
- **P-3: State consistency**. An enclave will generate a consistent checkpoint (consisting of the enclave's memory data and execution context), which will be used for enclave restoration.
- **P-4: State continuity**. The states of an enclave will not be rolled back to some previous states due to migration.
- **P-5: Single instance**. It should be guaranteed that *only one* enclave instance will be restored after migration.
- **P-6: Minimal TCB**. The migration process should not trust any software other than the code running in the enclave.

In addition to these security properties, the system should also avoid any user (owners of guest VMs or owners of enclaves) involvement during migration.

D. Threat Model

Analogous to prior work [3], [26], [2], we assume a powerful and active adversary who has superuser access to the system and the physical hardware. All other softwares and off-the-chip hardwares, including the hypervisor, the guest OS, DRAM, peripherals, are not trusted. We only trust the enclaves and the Intel SGX mechanism.

We take no steps to prevent the enclaves from intentionally leaking their own secrets (or via bugs) or defend against side channel attacks (e.g., using cache timing attack to steal secret keys). Some orthogonal solutions [18], [17] for software reliability can be adopted to mitigate this. DoS attacks, e.g., the hypervisor refusing to migrate a VM or the guest OS only resuming 3 out of 4 enclaves on the target VM, are also not considered. Such attacks are not introduced by migration and we focus on the new threats introduced by migration.

We define a secure migration process as follows: *For a single VM, the process of its migration should not lead to any running state that will not exist if without the migration*. In another word, the migration should not degrade the security level of the original system.

III. System Overview

Briefly, the enclave migration process, as shown in Figure 2, includes the following three operations: first, the source machine dumps one enclave's running states out. Second, the dumped states are transferred to the target machine through the network. Third, the target machine creates a new enclave and restores the running states to resume execution.

Since SGX hardware ensures that the memory content and other states of an enclave can only be accessed by the enclave itself, the states dumping and restoring in the first and third operation mentioned above can only be done within enclaves. Thus we introduce *control thread*, a new thread that runs within each enclave, to assist migration. The control threads on the source and the target machine will establish a *secure channel* for communication during migration. Control threads are totally transparent to enclave developers as long as the developers use our SDK, which will insert a control thread to each enclave automatically.

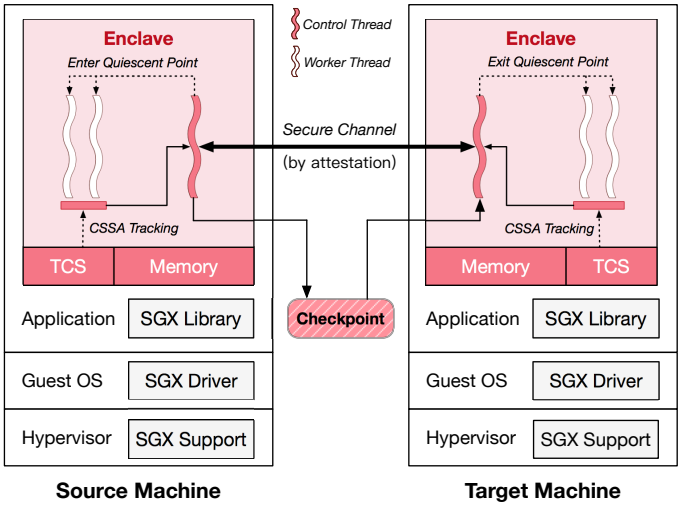


Fig. 2. Overview of migration process of a VM with an enclave.

On the source machine, the control thread is responsible for generating a checkpoint that contains all the enclave's memory and execution context, including the software-unreadable states maintained by hardware: the CSSA field in TCS. Specifically, we have designed a software mechanism running in the enclave that can track CSSA by monitoring all the entry and exit events of the enclave, without any dependency on the untrusted guest OS or hypervisor. The control thread also needs to protect the confidentiality (**P-1**) and integrity (**P-2**) of the generated checkpoint during the transfer. The migration key is shared by the control threads on the source and target enclave, which is exchanged through a secure channel. We also illustrate that a malicious guest OS may violate the consistency of a checkpoint by suspending and resuming enclave threads. Thus we propose a *two-phase checkpointing* mechanism to ensure the consistency of the checkpoint (**P-3**).

On the target machine, the restore process contains the following four steps.

- *Step-1*: the target machine creates and initializes a virgin enclave using the same image of the migrated enclave.
- *Step-2*: the source control thread will remotely attestate the newly created enclave. If the attestation succeeds, it will establish a secure channel with the target control thread to deliver the key encrypting the checkpoint.
- *Step-3*: the target control thread will restore all the memory using the checkpoint, and utilize the SGX library (out-of-enclave) to restore CSSA. In the meantime, the target control thread will also track the CSSA using the same method mentioned above.
- *Step-4*: before resuming execution, the target control thread will check whether the tracked CSSA is the same as the one in the checkpoint.

Step-1 is the only way to restore part of the hardware maintained enclave states such as SECS (SGX Enclave Control Structure). Note these states do not change during enclave execution. At this point, the code and data of the virgin enclave are encrypted except for the part of migration.

In Step-2, the remote attestation is done by source control thread without involving the enclave owner. This design significantly eases the process of migration. Otherwise, the owner will have to do the attestation for each migration, which will make the migration not practical. The attestation in step-2 will be done for only once. After step-2, the source control thread will destroy its own enclave (*self-destroy*) to ensure that only one enclave instance is running (**P-4 & P-5**). The entire migration process is done without trusting any software components other than the code running inside the enclave (**P-6**).

IV. Generating Enclave Checkpoints

At the beginning of a migration, the control thread will traverse the entire used memory within the boundary of the enclave and dump the data. Once dumped from EPC to normal memory, the state data is no longer protected by the hardware. To guarantee the privacy and integrity of the data in checkpoint, before dumping it to the normal memory, the source control thread first calculates a hash value of the checkpoint and then uses a randomly generated migration key ($K_{migrate}$) to encrypt the data together with the hash value. However, a malicious guest OS may still violate the consistency of the checkpoint when it is being generated.

A. Data Consistency Attack by Guest OS

Since the control thread runs concurrently with other worker threads of the same enclave, it has to ask all the other threads to suspend running to reach a *quiescent point* before dumping. Otherwise, it may get a checkpoint with inconsistent data. As a user-level thread, the control thread cannot directly suspend all worker threads, so it has to ask the OS for help. However, a malicious OS may deceive the control thread that the enclave has reached a quiescent point but is actually not, in order to violate the consistency of checkpoint. We call such attacks *data consistency attack*, as demonstrated in Figure 3.

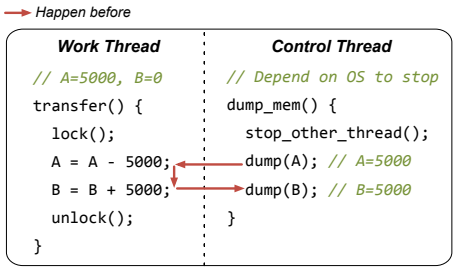
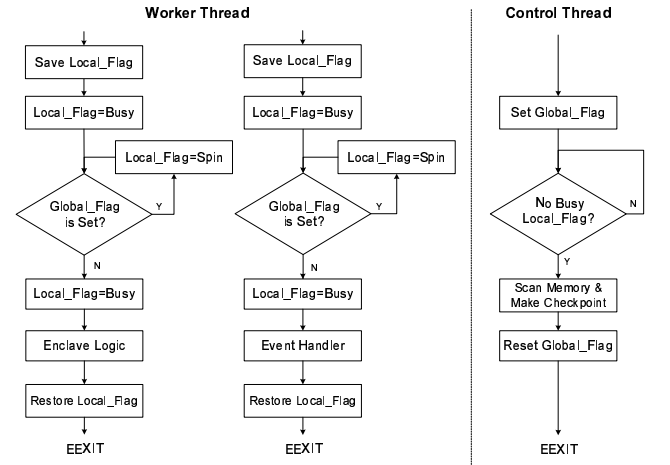


Fig. 3. An example of data consistency attack.

As Figure 3 shows, 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 threads. However, the malicious OS returns OK but actually does not stop the worker thread. 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.

B. Two-phase Checkpointing

To ensure that one enclave has reached a quiescent point before memory dumping, we propose a mechanism named *two-phase checkpointing*. In the first phase, the control thread will set a global flag in the enclave to indicate the start of the suspending process. Each worker thread will enter a predefined *spin region* as long as it finds that the global flag is set. When running in the spin region, a 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 all the worker threads to enter the spin region (or not running) before starting the second phase. In the second phase, the control thread will scan the enclave memory and generate a checkpoint. Thus, the control thread can ensure the consistency of the checkpoint without the help of the untrusted guest OS.



(a) Normal logic execution (b) Event handling (c) Control thread process

Fig. 4. The control flow of a worker thread and a control thread.

Implementation: More specifically, we introduce 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**. As Figure 4 shows, when entering the enclave, a worker thread will first save its local flag and set this flag to **busy**. Then it will check the global flag. If the global flag is set, it will set its local flag to **spin** which means it is ready for migration and spin till the end of phase two. Otherwise, it will set its local flag to **busy** and move forward. Before exiting, it will restore its local flag to the previous value.

Some worker threads may execute in the enclave for a long time. 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. Such a thread will be interrupted periodically, so we can leverage AEX to make it enter the exception handler in the enclave and then check the global flag. If the global flag is set, the thread will also set its local flag to **spin** and spin in the exception handler until the end of migration.

The control thread waits until a quiescent point when all the worker threads are in either **free** or **spin** state. Next, it dumps all the enclave states. The memory layout of an enclave is decided during development. Our SDK puts the global flag at the beginning of enclave, so the address of the global flag can help the control thread to determine the address range of the enclave. If a page is evicted (swapped out), it will be swapped back into EPC when the control thread accesses it. If having executable, writable and non-readable permission, one EPC page cannot be migrated because the control thread cannot read its content. This is a limitation of our solution in SGX v1. However, such pages are rare in common applications and this problem can be fixed in SGX v2 which supports dynamically changing page permissions.

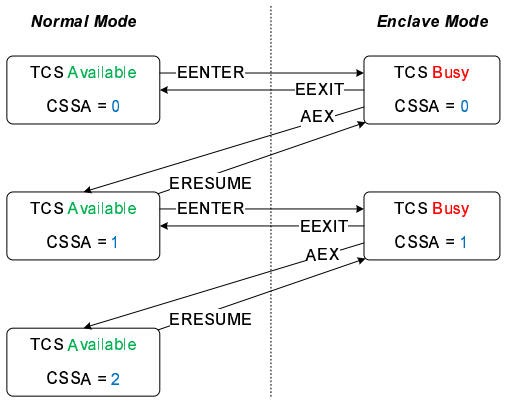


Fig. 5. TCS states change. EENTER and EEXIT are one pair of operations, while AEX and ERESUME is another pair. The former does not change the value of CSSA, while the latter does.

Note that the control thread only needs to ensure a fixed number of enclave threads (the number of TCS) are in the predefined *safe state* (**free** or **spin**). This is because the number of TCS is the maximum number of enclave threads. The code to implement *two-phase checkpointing* is added by our SDK, which will not burden enclave developers.

C. In-enclave CSSA Tracking

The CSSA (Current State Save Area) field of TCS is a counter that indicates the nesting level of enclave exception handling, as mentioned in Section II-A. Figure 5 shows an example of the state change of *TCS*. If an exception happens (e.g., page fault) when an enclave is running, the hardware will save the context of execution to the SSA-0 (SSA is a memory region within an enclave and indexed by CSSA, as shown in Figure 1) and increase CSSA to 1 before exiting the enclave. Later, if the SGX library decides to resume the execution from the interrupted point (by using *ERESUME*), the hardware will restore the context from the previous SSA, and decrease CSSA to 0. Or the SGX library may decide to enter the enclave's exception handler (by using *EENTER*), then no restoring will occur. In this case, if an exception happens again when the enclave is executing, its context will be saved in the SSA-1, and CSSA will be 2.

As the above example shows, the value of CSSA is critical to the control flow of an enclave and should be a part of the checkpoint. The challenge is that there is no software way to read or write CSSA **directly**. As a result, although all the SSA regions (they are in the enclave) will be transferred to the target enclave, there is no way to know or set the correct value of CSSA. We observe that there are two possible solutions to **infer** CSSA by software.

Solution-1: Infer CSSA from AEX and ERESUME by the (untrusted) outside-enclave SGX library. When an AEX happens, the control flow will be transferred to the event handler of OS, and finally return to an address named "AEP" (Asynchronous Exit Pointer). AEP is *EENTER*'s parameter which is an address located in SGX library. The location contains the *ERESUME* instruction which can transfer control back to the enclave. It is observed that the value of CSSA is only changed when an *AEX* happens or an *ERESUME* instruction is executed. An intuitive choice is leveraging the SGX library to track the value of each CSSA. However, the SGX library could be compromised because it is outside the enclave. An attacker may offer a faked CSSA value to hijack the control flow when resuming the enclave's execution.

Solution-2: Infer AEX and ERESUME from EENTER and EEXIT by the enclave itself. In order not to rely on untrusted components (**P-6**), we provide a novel method to indirectly trace the CSSA in the enclave. Although the CSSA field cannot be read, its current value will be stored in register *rax* as the return value of *EENTER* instruction. Nevertheless, the value of CSSA may be changed due to *AEX* during enclave execution. We observe that there are two possible cases between *EENTER* and *EEXIT*. In one case, when a thread is executing in the enclave, the execution times of *ERESUME* and *AEX* must be equal. The value returned by *EENTER* is equal to the accurate value of CSSA. In the other case, when a thread is outside the enclave, the difference between the execution times of *AEX* and *ERESUME* must be one. So the return value of *EENTER* is smaller than the real value of CSSA by one.

Implementation: Specifically, when a worker thread is prepared for migration, its local flag can only be **free** or **spin**. The local flag is **free** if and only if the thread has executed *EENTER* and *EEXIT* for the same times. Therefore, the number of times of *AEX* and *ERESUME* must be equal too, which means the thread's CSSA must be 0 in this case.

At the entry of enclave, the stub code will record $\mathcal{CSSA}_{EENTER}$ (the return value of *EENTER*). If the local flag is **spin** after migration, the CSSA's value must be greater than $\mathcal{CSSA}_{EENTER}$ than one. Because the corresponding thread must be outside the enclave when the VM is migrated from the source to the target machine.

On the target machine, CSSA tracking is also leveraged to verify the process of restoring. Note that although the control thread knows the accurate value of CSSA, it cannot directly restore that value on the target machine (only accessed by hardware). Only the untrusted SGX library together with guest OS can restore the value of CSSA through executing the *EENTER* and triggering the *AEX* repeatedly. While the control thread has to rely on their functionalities, it will check the restoring process to ensure the CSSA is correctly restored.

V. Secure Enclave Migration

Thanks to the hardware-enforced isolation as well as remote attestation provided by SGX, the owner of an enclave can ensure that there is only one running instance of the enclave, because the OS can neither clone (requires EPC access) nor re-create (requires remote attestation) the enclave without its owner's awareness. Some applications may depend on the *uniqueness* feature of SGX as one important security assumption. However, once migration is enabled, a malicious cloud operator may try to migrate one enclave to multiple target machines (aka., fork attack) or let the target machine resume from a stale version of the checkpoint (aka., rollback attack). In this section, we present how our migration mechanism leverages remote attestation to build a secure channel, and further uses *self-destroy* to ensure the uniqueness of the enclave instance at all time.

A. Fork Attack & Rollback Attack

Figure 6 shows an example of *fork attack*. There is a mail server running in an enclave on some untrusted cloud. A client connects to the server and does three operations. ①: create an email with a list of friends as recipients. ②: delete one of the friends, Eve, from the recipients. ③: send the mail. The client will not do the next operation unless receiving the acknowledgment of the previous operation from the server. The untrusted cloud controls the forwarding of requests.

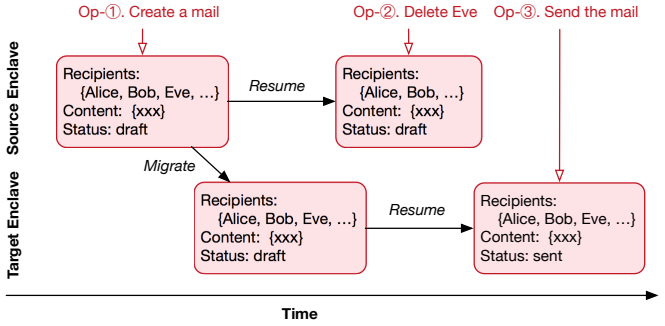


Fig. 6. An example of fork attack.

If there is only one instance of the mail server enclave running and no migration enabled, the client can ensure that the mail will not be sent to Eve. However, if a malicious cloud operator migrates the enclave to a *target machine* after operation ①, then resumes the execution of the enclave on the *source machine* to handle operation ②, the client will receive the acknowledgment of operation ② and start to do operation ③. At this moment, the operator resumes the enclave on the *target machine* to handle operation ③. In such case, operation ② is actually bypassed on the *target machine*, and the email will be sent to Eve. The key step of the attack is that the operator resumes the enclave on the *source machine* after migration, which causes two enclave instances with the same intermediate states running at the same time.

Rollback attack can also be an undesired side effect of enclave migration. For example, a mail server running in an enclave requires a client to enter a password for authentication. To mitigate brute-force attacks, the server sets a policy that a client can make at most three failed attempts, otherwise some predefined alarm will be triggered. However, a malicious hypervisor can utilize enclave migration to get thousands of enclave instances to guess the password. Also, a malicious operator can utilize enclave migration to generate a checkpoint at first, and commit a brute-force attack by restoring the enclave to the checkpoint after three failed attempts. So, it should be banned that a migration restores more than one

enclaves. And the operator cannot resume an enclave from a checkpoint at any time without the enclave owner's awareness.

B. Enforcing Single Instance of Enclave

In order to ensure the uniqueness and state-continuity of an enclave during migration, it is necessary to ensure that:

- The enclave instance on the source machine will not resume its execution after migration.
- The enclave instance on the source machine will only generate one checkpoint, which can be restored by one target enclave.

We develop a mechanism named *self-destroy* to ensure the above properties without trusting any privileged software components.

Self-destroy: The source control thread will refuse to resume the source enclave after it transfers the $K_{migrate}$ (migration key) to the target control thread through the secure channel. In order to prevent the case that the migration process is canceled (e.g., due to network problem) after the source enclave's self-destroy, the $K_{migrate}$ will only be sent after all other data transferring has been done. If a migration is canceled, the source enclave will delete the $K_{migrate}$ immediately so the checkpoint will be useless. By the self-destroy mechanism, once the source enclave generates one checkpoint for one target enclave, it will not resume any more. This is done simply by keeping the global flag unchanged so that all the work threads will spin forever.

Secure Channel: So as to protect the $K_{migrate}$ and ensure *only one* target control thread can get it, a secure channel between the source and the target control thread is essential. The source and the target control threads leverage Diffie-Hellman key exchange protocol to build a secure channel. It is depicted in Figure 7 that the crux is the mutual authentication of both the source and the target enclaves.

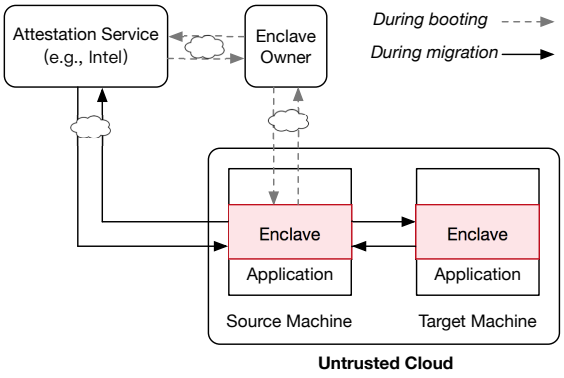


Fig. 7. Attestation process. During booting, the enclave owner attests the enclave; during migration, the source enclave attests the target enclave.

The source authenticates the target: The source control thread uses *remote attestation* to ensure that the received message is from the target enclave. Here the source enclave acts like the enclave owner in the remote attestation at launch time.

The target authenticates the source: We put a pair of keys into the enclave image. The public key is in plaintext while the private key is in ciphertext. After the migration, the target enclave can get the plaintext private key from the source enclave. When an enclave is first created, it can rely on *remote attestation* to get the plaintext private key from the enclave owner, as mentioned in Section II-A. Since the source enclave has been attested before, it has a plaintext private key. All the messages from the source enclave to the target enclave are encrypted by this private key. Thus, the target control thread can verify the received message with the public key.

Note that the communication channel between the source enclave and the attestation service is also protected by another two pair of public/private keys. One belongs to the source enclave. The other belongs to the attestation service. The public key of the source enclave was registered in the attestation service before. The source enclave has its own plaintext private key because it has been attested before while the target enclave does not. So the target enclave cannot use *remote attestation* to authenticate the source enclave.

Besides, the source control thread ensures that it will use Diffie-Hellman key exchange protocol to build only one secure channel even if receiving many exchange requests from different targets. Therefore, our design guarantees that only one authenticated target enclave will get the $K_{migrate}$.

C. Supporting Checkpoint/Resume

Technically, there is no difference between rollback attacks and legal checkpoint/resume operations. Some prior work [21] simply disables the checkpoint/resume mechanism for security, which, however, renders some normal operations usually offered by a cloud platform (like snapshot) unavailable.

We want to support not only secure live migration but also legal checkpoint/resume operations. Similar to the first step of migration, the control thread will generate a checkpoint. The only difference is that for encrypting the checkpoint, the control thread will retrieve an encryption key ($K_{encrypt}$) *from the enclave owner* instead of generating a random one. When resuming, the control thread must use remote attestation to retrieve the corresponding $K_{encrypt}$ from the enclave owner. Thus, all the checkpoint/resume operations are logged. By auditing the log, an owner can check suspicious rollbacks, constrain the operations on enclaves and ask the cloud operator to prove the necessity of such operations. In short, the checkpoint/resume operations need the owners' involvement while the migration does not.

VI. Implementation

We have implemented a working system on a real Intel SGX machine for Linux and KVM. We added about 50 LoC in Qemu, 1584 LoC in KVM and 5798 LoC (except the source code of libc inside enclave) in the guest VM which consists of the SGX driver and SDK. When we built our system, Intel has not provided any SGX support for Linux environment. To our knowledge, we are the first to report the design and implementation of migrating a VM with enclaves on a real machine. Our SDK is also compatible with Intel SGX driver for Linux. Integrating our solution of Enclave Migration into Intel SGX SDK is our future work.

A. SGX Support in Hypervisor

EPC Management: EPC (Enclave Page Cache) is a finite platform asset. The hypervisor is in charge of managing EPC and letting VMs share EPC without affecting each other. At boot time, hypervisor maps all the EPC into its virtual address space. For each page located in this region, hypervisor needs to maintain the metadata, including state (used or free), owner (which VM it belongs to), etc. When creating a guest VM, the hypervisor will first reserve a range of guest physical address which will be used as the guest's EPC region later. New hypercalls are provided for the guest VM to learn about the location and size of its EPC. However, the hypervisor only maps part of this region to real EPC and leaves the remaining part unmapped. On the one hand, the guest VM can use the EPC without triggering EPT (Extended Page Table) violations before it uses up the mapped part, which benefits the performance. On the other hand, the hypervisor can use the on-demand paging strategy to save EPC resources. If the hypervisor has already used up all the physical EPC and receives a new request for EPC allocation, it will revoke some EPC resource from a chosen VM by evicting EPC pages and clearing the mappings in EPT. The hypervisor overcommits the EPC resources through swapping which is transparent to the VMs.

VMExit Inside an Enclave: Once a VMExit event occurs when the CPU is running an enclave, the hardware will set a bit, named "Enclave Interruption" bit, in the *Guest Interruptibility State* field of the VMCS (Virtual Machine Control Structure) as well as in the *EXIT_REASON* field before delivering the VMExit to the hypervisor. Thus the hypervisor can invoke the correct handlers. For EPT violation triggered during enclave execution, if the fault address is located in the virtual EPC of guest VM, the hypervisor will allocate a physical EPC page and fill the corresponding EPT entry with the address of this page; if not, the original EPT violation handler will be invoked. For other events such as illegal instruction and timer interrupt, currently we clear the bit in *EXIT_REASON* field and then reuse the original handlers in the hypervisor and guest OS.

B. SGX Support in Guest OS

Virtual EPC Management: Our SGX driver in the guest OS first asks the hypervisor for the address of EPC and then maps the whole EPC into the kernel virtual address space. The management of EPC is similar to that of normal memory. The difference is in the process of evicting a page. If the SGX driver needs to allocate a new EPC page when it has already used up all its EPC, it will first choose some EPC pages based on a simplified LRU (Least Recently Used) algorithm and then use SGX instructions to swap them into normal memory. Evicting an EPC page generates a cryptographic MAC, an encrypted copy of the page data and an 8-byte version number. The driver needs to record such information for loading back this evicted page in the future.

Enclave Creation and Destruction: Our SGX driver provides some new interfaces (through *ioctl*) for applications. The SGX driver first reserves a range of virtual address for the enclave in the application's address space. Next, the driver uses the *ECREATE*, *EADD*, *EEXTEND* and *EINIT* instructions in turn to create a runnable enclave. Last, the driver returns an enclave *ID* to the application if succeeding in creating the enclave, and maintains the relationship among the application, the ID and the enclave. An application can invoke one system call with an enclave ID to destroy its own enclave. The driver uses the *EREMOVE* instruction to free the enclave's resources.

C. SDK and SGX Library for Applications

We provide an SDK for developing applications with enclaves. Although our enclave migration design needs applications' awareness, it will not bother developers because the migration task is in the charge of our SDK.

Interactions between Enclaves and Applications: Our SDK hides the details of interaction between the enclaves and their host applications from the developers. For data communication, we pass arguments through shared memory outside the enclave. For control transfer, we insert trampolines into an enclave, which enables the enclave to call the outside functions without leaking any security information; there are other trampolines in SGX library (outside the enclave) for transferring the control flow into the enclave. Besides, if the developer defines an exception handler in the enclave, the SGX library will use *EENTER* to invoke that handler after the enclave is interrupted, and then use *ERESUME* to resume the execution.

Stubs and Functions in Enclaves: A developer needs to specify the legal entries of an enclave. Our SDK is responsible for adding a stub for the legal entry of the enclave. The entry stub saves all the registers (including the stack pointer) and changes the stack (using the stack inside the enclave). Accordingly, there is also a stub, which restores all the registers and the original stack, at the exit. Moreover, these stubs ease the support of two-phase checkpointing. They are in charge of checking the global flag, setting the local flags, as well as recording the return value of *EENTER*. The SDK also adds the code of control thread, and another TCS (Thread Control Structure) for invoking the thread, without the developers' involvement. Besides, our SDK supports most of libc functions in enclave through statically linking a simplified libc within enclave. For some functions, such as malloc and free, the SDK implements them in enclave directly. For other functions requiring invoking system calls, such as read and write, they will eventually be forwarded to the outside SGX library.

D. Putting All Together: Migration of a VM with Enclaves

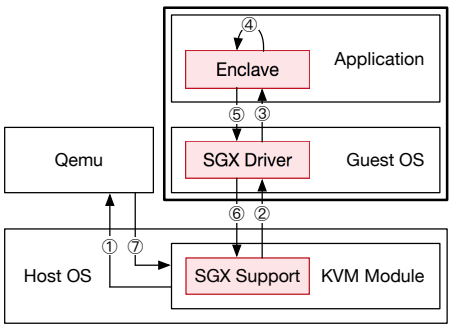


Fig. 8. The overview of suspending process on the source machine.

This subsection describes the whole process of migrating a VM with enclaves. An overview of suspending process is depicted in Figure 8, which consists of the following steps:

① When the QEMU monitor receives a migration command, it will tell the hypervisor to migrate the VM through a new system call.

② The hypervisor locates the VM of the QEMU process and informs the guest OS that it should be prepared for migration. Here we implement an upcall mechanism by injecting a special interrupt.

③ After the guest OS receives the migration notification, it will refuse to create any new enclaves till the end of migration and ask applications to make enclaves prepared for migration. Specifically, it sends a migration signal (SIGUSR1) to each enclave process. Our SGX library has already registered the handler for this signal before creating enclaves.

④ After receiving the signal of migration, the SGX library will invoke the control threads to start the two-phase checkpointing.

⑤ After a control thread returns, the SGX library will tell the guest OS that one enclave is ready. At this point, the encrypted checkpoint of this enclave is stored outside the enclave.

⑥ When all enclaves are ready for migration, the guest OS will tell the hypervisor that it is ready for migration through a new hypercall.

⑦ The hypervisor returns the control flow back to the QEMU process. Then the QEMU can continue the migration process without caring about enclaves.

On the target machine, the resuming process is similar with the previous one (without enclaves). The differences are as follow: First, the guest OS rebuilds all the enclaves according to the records of enclave creation and destruction. In the meantime, the hypervisor restores the EPC mapping for the guest OS. Next, the guest OS tells the SGX library to invoke control threads for each application. Last, each control thread restores each enclave.

An Optimization of Remote Attestation: As shown in Figure 7, one remote attestation needs at least two network round trips: one between the enclave application and the owner, the other between the owner and the attestation service. The source control thread only sends the $K_{migrate}$ to the target after building the secure channel with remote attestation. The latency of remote attestation could harm the performance of migration if not hidden. Therefore, we propose an optimized solution that hides the latency while guaranteeing the same security.

The application developer needs to provide another enclave called the *agent enclave*. A developer can use one *agent enclave* to serve all his/her enclaves. During a migration (or even before a migration), the source control thread first remotely attests the *agent enclave* on the target machine and then transfers the $K_{migrate}$ to it in advance. Hence, when the VM is resumed on the target machine, all its enclaves can get their migration keys from *agent enclave*s through **local attestation**. *Agent enclave*s can be destroyed after the VM resuming. The implementation of *agent enclave* is straightforward and our SDK provides a framework to easily develop it.

VII. Security Analysis & Discussion

A. Security Analysis

Our software-based enclave migration solution meets all the security properties proposed in Section II-C. The privacy (**P-1**) and integrity (**P-2**) of the checkpoint are protected by encryption and checksum. The consistency (**P-3**) is ensured by our two-phase checkpointing mechanism. Each checkpoint can only be used for migration once (**P-4**), after which the source enclave will destroy itself to keep a single running instance (**P-5**). The whole design does not depend on the trustworthiness of any software other than the enclave itself (**P-6**).

Consistency attack among multiple enclaves: There are cases that a VM may contain multiple interrelated enclaves. For instance, an application may distribute states to multiple enclaves. In such cases, a malicious guest OS may try to violate the consistency of the VM's checkpoint that contains all of the enclaves' checkpoint:

$C_{All-Enc} = \{C_{Enc-1}, C_{Enc-2}, ..., C_{Enc-n}\}$, where $C_{All-Enc}$ means checkpoint of the VM and C_{Enc-n} means checkpoint of enclave-n. Our checkpoint generating mechanism can inherently enforce the consistency of $C_{All-Enc}$. Specifically, as long as the state continuity (**P-4**) and single instance (**P-5**) are ensured, the system will not generate a $C_{All-Enc}$ whose states will not exist in the original system (the one without migration).

DoS attack by the malicious hypervisor or guest OS: In our design, a successful migration still relies on several critical functionalities of the hypervisor and guest OS, including the I/O operations, scheduling, enclave construction, etc. It is possible that the system software refuses to finish such functionalities. Such a DoS attack can be detected by the tenants and is not introduced by migration.

Side-channel attack: The migration process may leak two new kinds of information of an enclave to a potential attacker: The first is the size of the checkpoint image. For example, the attacker may get the size of stack and heap of an enclave to infer its running states or even certain data. However, since the checkpoint is encrypted, the attacker has no way to get its internal structure. A simple solution is dumping out the whole enclave memory instead of only the used part. Or we can simply add some random size of data to the checkpoint (e.g., padding data) so that the size of checkpoint does not accurately reflect the size of memory used by the enclave. The second is the length of time of checkpoint generation. However, such length only relates to the time of AEX occurrence, which does not reflect any running information of the enclave.

Other attacks: Our migration mechanism does not leak any information in plaintext during the process, thus it can defend against physical attacks as well. Resending all the network packets to a target enclave cannot launch a replay attack successfully, because the control threads will establish a new secure channel (with random session key) for each migration so that the stale checkpoint will be considered invalid.

B. Suggestions on Hardware Design for Migration

Currently, due to hardware limitations (both SGX v1 and v2), an enclave cannot be migrated transparently and securely by system software. In this section, we give some suggestions on hardware design to assist transparent enclave migration.

Since different SGX equipped CPUs use different keys to encrypt the EPC (Enclave Page Cache) and these keys cannot leave the CPU, there must be some way of letting two CPUs share the same *migration keys* (an encryption key and a signing key). We suggest that Intel can provide a special enclave, e.g., *control enclave*, for two machines to share the migration keys. The *control enclave*s on the source and target machines can use remote attestation to authenticate each other and agree on randomly generated migration keys. Both *control enclave*s install the migration keys into the CPUs with a new instruction *EPUTKEY*, which can only be executed by the *control enclave*.

For migrating an enclave, another new instruction called *EMIGRATE* can be used to make the enclave non-executable, which will deny all accesses to the enclave memory. So the enclave states will not change during migration, which can avoid consistency problems.

Another new instruction called *ESWPOUT* can be used to swap an enclave page from EPC to normal memory. This instruction works as follow: first, decrypt the EPC page; then, encrypt it with the *encryption key*; last, generate a MAC with the *signing key*. It can ensure the privacy and integrity of enclave pages.

Some enclave pages may have been evicted to normal memory before migration. For such pages, a new instruction called *ECHANGEOUT* can change its original encryption key to the migration encryption key and generate a new MAC. Note that *ESWPOUT* and *ECHANGEOUT* can only be executed after *EMIGRATE*.

On the target machine, new *ESWPIN* and *ECHANGEIN* instructions perform the opposite function of *ESWPOUT* and *ECHANGEOUT*; a new *EMIGRATEDONE* instruction can verify the whole state of a migrated enclave and make the enclave runnable. Besides, the other EPC pages such as VA (Version Array) pages should be migrated in a similar way.

VIII. Performance Evaluation

To measure the performance of our design, we have conducted experiments on two laptops with Intel Core i7-6700HQ 2.6GHz CPU and 8 GB memory, which run a 64-bit Ubuntu Linux. To be specific, their model is DELL Inspiron 7559. We use the same software stack as those on servers. 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. We run each experiment 50 times and report the average of the results. The standard deviation is within 5% across all the experiments and is not reported.

A. Overhead of SGX and Migration Support

We first present an experiment on the overhead introduced by SGX protection. Here we utilize our SDK to make nbench 2.2.3 execute in an enclave. Also, we leverage Intel SDK to port nbench into an enclave. As shown in Figure 9(a), the overhead caused by SGX is not obvious if the workload is computation intensive and has small memory footprint. Conversely, if a workload in enclave requires more safe memory, the overhead introduced by SGX significantly increases. String Sort is such an example. We use the performance results reported by the benchmark itself. In particular, nbench runs its benchmarks multiple times, taking the mean and standard deviation until it reaches a confidence threshold such that the results are 95% statistically certain.

To measure the overhead caused by supporting migration, we also choose some real world applications which have security requirements, change them to applications with enclave, and evaluate their performance with and without migration support. The results in Figure 9(b) show that migration support brings almost no overhead. It makes sense because in our implementation, the extra work for each enclave thread only involves checking the global flag, setting the local flag and maintaining the counter for CSSA. Compared to real workloads, these operations are negligible. The control thread of each enclave is only active during migration. Hence, it does not influence the performance during normal execution.

Putting less code in enclave can enhance security due to small TCB (Trusted Computing Base) and get better performance due to using less EPC (Enclave Page Cache). These are the reasons why we choose the above benchmarks and applications (not very large). For other larger applications, a proper way to use SGX is splitting them and putting the security part into enclaves.

B. Performance of Migration

In the following experiments, the enclaves run either *libjpeg* or *mcrypt* and have two worker threads.

Two-phase Checkpointing: Figure 9(c) shows how long it takes an enclave to generate a checkpoint with different numbers of concurrently executing enclaves. The average time is about 255 microseconds when the total number of enclaves is no more than 4. The time increases to 263 microseconds when there are 8 enclaves. It is because the VM only has 4 VCPUs while each enclave has two worker threads as well as a control thread.

The total time of generating the checkpoint also depends on the size of output states and the encryption algorithm. In the above evaluation, we use RC4 as the encryption method and the output size is 20KB. The encryption process takes about 200us. If DES is chosen as the encryption method, the encryption process will take about 300us. An optimized method is to utilize hardware support for encryption and hashing [15], [7]. Another optimization opportunity is only encrypting the sensitive data. We also make Memcached-1.4.22 run in an enclave to test the performance of two-phase checkpointing when the output size increases. During this experiment, there are four threads running inside the enclave and the output states are encrypted with AES-CBC which is implemented with AES-NI. The results are shown in Figure 11.

Time of Waiting for All Enclaves: To make the enclave migration independent of the underlying hypervisor, our design first asks the VM to prepare all the enclaves. The guest OS

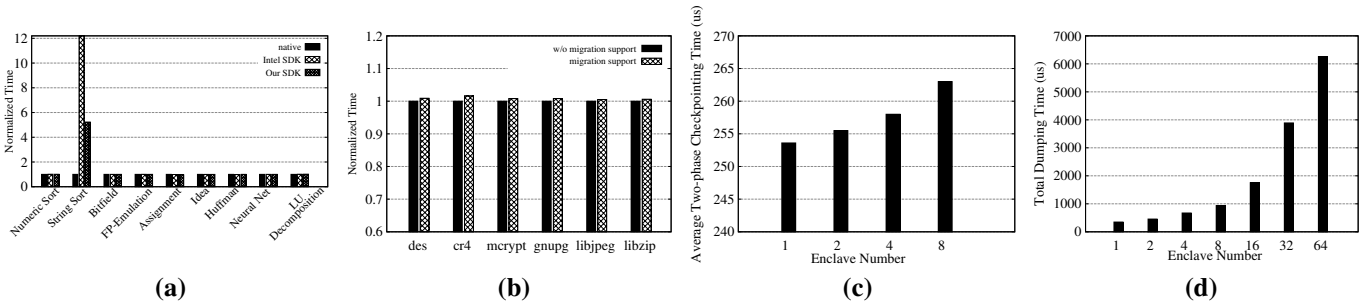


Fig. 9. Performance evaluation: Figure (a) shows the overhead on nbench. Figure (b) shows the overhead caused by migration support. Figure (c) shows the time of two-phase checkpointing and Figure (d) shows the time of suspending all enclaves.

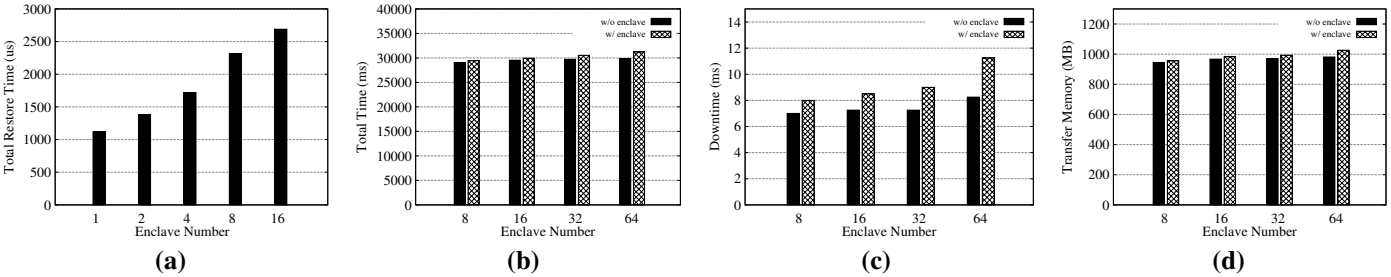


Fig. 10. Performance evaluation: Figure (a) shows the time of restoring all enclaves. Figure (b), (c), (d) show the overhead of total migration time, downtime and the amount of transferred data.

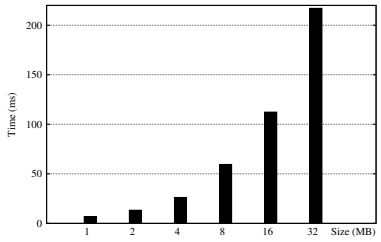


Fig. 11. The time of two-phase checkpointing on Memcached.

informs all the enclave processes of migration and waits until they are ready. The total dumping time of such procedures is shown in Figure 9(d). The time is calculated from the guest OS receiving a migration notification to all the enclaves getting ready. When the number of enclaves is no more than 8, the total time is within 940us. When the number reaches 16, the total time is about 1700us. As the total number of enclaves increases, the number of threads also increases and it takes more time to schedule the control and worker threads.

Restoring Enclaves on Target VM: Figure 10(a) shows the time of restoring all the enclaves on the target machine. The total time grows linearly as the number of enclaves increases, because the enclaves are rebuilt one by one. An optimized method is using multiple threads to rebuild enclaves to make maximum use of parallelism. Since there are some concurrency restrictions of SGX instructions, building an enclave is hard to be accelerated. For instance, the hardware disallows EADD/EEXTEND to concurrently run on one SECS (SGX Enclave Control Structure).

Live Migration: We run two VMs respectively, one with some running enclave applications, the other with the same number of original applications. We compare the important performance indicators of their live migration based on shared storage. Figure 10(b) 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. This is consistent with the overhead of transferring memory, as shown in Figure 10(d). The reasons for the overhead are that each enclave needs to dump its states and the guest OS needs to record each enclave. Figure 10(c) represents that the downtime grows as enclave number increases. This is because it takes more time for more enclaves to generate the checkpoints. Although we count the time of two-phase checkpointing in the downtime, the other applications without enclaves in the VM can run when the enclaves are generating the checkpoints.

IX. Related Work

SGX-assisted TCB Reduction: There are also several prior work that leverage SGX to provide trustworthy execution. SCONE [2] builds secure Linux containers with Intel SGX. Ryoan [10] puts a sandbox into an enclave, which can prevent untrusted applications from leaking secret data. VC3 [16] uses SGX to run distributed MapReduce computations while keeping the code and data secret and ensuring the correctness and completeness of the results. However, they assume the cloud providers are benign and do not consider VM migration. Haven [3] leverages the hardware-enforced protection of SGX to defend against privileged code and physical attacks, but also addresses the dual challenges of executing unmodified legacy binaries and protecting them from a malicious host. Yet, it does not consider how to securely migrate the enclave among physical machines.

Secure Live VM Migration: Live VM migration [6], [8] is a key enabling technique in nowadays data center and cloud platforms. However, the security of a live VM migration would notably affect the security of end-user applications. Hence, the default Xen migration tool uses SSL to secure the transfer.

PALM [30], [28] pioneers in providing security-preserved live migration of a VM under the protection of a secure hypervisor (i.e., CHAOS [5], [4]). Compared to PALM, our work faces a stronger threat model and a unique challenge where no system software is trusted and can access the protected enclave states. Given that there is currently no support for VM as well as VM migration in SGX, the goal of our work is to empower a VM with SGX protection with the ability of live migration, and thus to embrace both strong security and rich functionality.

Consistency and Rollback Attacks: Watson exploits concurrency vulnerabilities in system call wrappers [22]. Yang et al. [27] show attacks caused by concurrency bugs. These attacks are similar to our consistency attack. The difference is that our attack works on SGX enclave which has more security constraints. Weichbrodt et al. [23] exploit synchronization bugs in Intel SGX enclaves on a single machine. Some previous work [14], [19], [20] ensures rollback resistance without making the system vulnerable to crashes. Raoul and Frank propose Ariadne [19], [20] to solve the state-continuity problem of stateful protected modules such as SGX enclaves. Such solutions can be adopted in our system to prevent rollback attacks on the single machine, to an extent. However, they do not consider about state-continuity among multiple machines (e.g. under the VM migration situation).

X. Conclusion

This paper focuses on the migration of VMs with SGX enclaves running inside. Traditional methods of VM migration do not work because, with SGX protection, the running states of a VM cannot be entirely transferred, since the hypervisor cannot access the memory and CPU context of enclaves. We present a software-based mechanism that supports enclave migration, which is able to enforce all the security properties required. We further implement it on real SGX enabled hardware and also propose a set of design suggestions for future SGX extensions.

XI. Acknowledgment

We thank the anonymous reviewers for their insightful comments. This work is supported in part by National Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61303011 and 61572314), a research grant from Huawei Technologies, Inc., National Top-notch Youth Talents Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (TS0220103006), Singapore NRF (CREATE E2S2), NSF via grant number CNS 1513687, and ONR via grant PHD.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, volume 13, 2013.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, 2016.

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

[4] H. Chen, J. Chen, W. Mao, and F. Yan. Daonity–grid security from two levels of virtualization. *Information Security Technical Report*, 12(3):123–138, 2007.

[5] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P.-c. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. *Parallel Processing Institute Technical Report*, (FDUPPITR-2007-08001), 2007.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[7] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[8] J. G. Hansen. *Virtual Machine Mobility with Self-Migration*. PhD thesis, Citeseer, 2009.

[9] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, page 11, 2013.

[10] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, 2016.

[11] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177, 2000.

[12] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, page 10, 2013.

[14] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *S&P*, 2011.

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

[16] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *S&P*, pages 38–54, 2015.

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

[18] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS*, 2015.

[19] R. Strackx, B. Jacobs, and F. Piessens. Ice: A passive, high-speed, state-continuity scheme. In *ACSAC*, pages 106–115. ACM, 2014.

[20] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[21] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *ASPLOS*, pages 437–450, 2012.

[22] R. N. Watson. Exploiting concurrency vulnerabilities in system call wrappers. *WOOT*, 7:1–8, 2007.

[23] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[24] Wikipedia. https://en.wikipedia.org/wiki/Two-phase_commit_protocol. *Two-phase Commit Protocol*.

[25] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, pages 246–257, 2013.

[26] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, 2015.

[27] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *HotPar*, 2012.

[28] F. Zhang and H. Chen. Security-preserving live migration of virtual machines in the cloud. *Journal of network and systems management*, 21:562–587, 2013.

[29] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, pages 203–216, 2011.

[30] F. Zhang, Y. Huang, H. Wang, H. Chen, and B. Zang. Palm: security preserving vm live migration for systems with vmm-enforced protection. In *Asia-Pacific Trusted Infrastructure Technologies Conference*, pages 9–18. IEEE, 2008.