MigSGX: A Migration Mechanism for Containers Including SGX Applications

Kenji Nakashima
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
rudh@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
kourai@csn.kyutech.ac.jp

ABSTRACT

Recently, containers are widely used to process big data in clouds. To prevent information leakage from containers, applications in containers can protect sensitive information using *enclaves* provided by Intel SGX. The memory of enclaves is encrypted by a CPU using its internal keys. However, the execution of SGX applications cannot be continued after the container running those applications is migrated. This is because enclave memory cannot be correctly decrypted at the destination host. This paper proposes *MigSGX* for enabling the continuous execution of SGX applications after container migration. Since the states of enclaves cannot be directly accessed from the outside, MigSGX securely invokes each enclave and makes it dump and load its state. At the dump time, each enclave re-encrypts its state using a CPU-independent key to protect sensitive information. For space- and time-efficiency, MigSGX saves and restores a large amount of enclave memory in a pipelined manner. We have implemented MigSGX in the Intel SGX SDK and CRIU and showed that pipelining could improve migration performance by up to 52%. The memory necessary for migration was reduced only to 0.15%.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Software and its engineering → Virtual memory**.

KEYWORDS

Intel SGX, enclaves, containers, migration, encryption

ACM Reference Format:
Kenji Nakashima and Kenichi Kourai. 2021. MigSGX: A Migration Mechanism for Containers Including SGX Applications. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3468737.3494088

1 INTRODUCTION

Recently, containers are widely used in clouds [4, 6]. A container is a virtual execution environment provided by the operating system. Compared with a virtual machine (VM), it is more lightweight because it does not virtualize hardware. In a container, some of the applications process big data, e.g., for artificial intelligence (AI) and Internet of Things (IoT). Big data often includes sensitive information such as users' privacy and healthcare data. It is confined into containers, but the protection provided by containers is weaker than that by VMs. If attackers intrude into containers, they could easily steal sensitive information. To prevent information leakage from applications, the operating system provides various security mechanisms [5, 18]. However, they can be disabled once the operating system is compromised.

To protect sensitive information in containers without relying on the operating system, a trusted execution environment called Intel SGX [11] is often used, e.g., in Microsoft Azure [12] and IBM Cloud [9]. SGX applications can create protection domains called *enclaves*. Since the memory of enclaves is encrypted using internal keys in a CPU, information leakage from enclaves is prevented. However, after SGX applications are moved to another host by container migration, their execution cannot be continued. The encrypted memory of enclaves is transferred to the destination host as it is, while the internal keys in the CPU are not. As a result, the enclave memory cannot be correctly decrypted at the destination host because the CPU executing applications is changed.

This paper proposes *MigSGX* for enabling the enclaves in SGX applications to continue their correct execution after container migration. Since the state of an enclave cannot be correctly saved or restored from the outside due to the memory protection of SGX, MigSGX makes an enclave itself dump and load its state. To save the state at the source host, each enclave re-encrypts its state using a CPU-independent key and dumps the encrypted state to the outside of it. To restore the state at the destination host, a re-created enclave decrypts the saved state using the same key and loads the decrypted state into itself. For space- and time-efficiency, MigSGX saves and restores a large amount of enclave memory in a pipelined manner using a small shared buffer. This pipelining can overlap dumping and loading enclave memory with saving and restoring it, respectively.

To enable the migration of an SGX application, MigSGX provides the MigSGX library to an enclave and runs the MigSGX runtime outside enclaves in an SGX application. The MigSGX manager runs outside a container and sends the save and restore requests to the runtime and then the runtime invokes the library to dump and load the state of the enclave. To achieve secure communication between the MigSGX manager and runtime, the MigSGX manager injects code into the runtime and shares a buffer for pipelining. We have implemented MigSGX in the Intel SGX SDK [10] and CRIU [15]. Using MigSGX, we examined the migration performance of an SGX application. As a result, the pipelined save and restore of enclave

UCC'21, December 6–9, 2021, Leicester, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8564-0/21/12...$15.00
https://doi.org/10.1145/3468737.3494088

memory could improve migration performance by up to 52%. The memory necessary for migration was reduced only to 0.15%.

The organization of this paper is as follows. Section 2 describes the issue in the migration of SGX applications. Section 3 proposes MigSGX for supporting that migration. Section 4 explains the detailed implementation of MigSGX. Section 5 shows the performance of MigSGX. Section 6 describes related work and Section 7 concludes this paper.

2 BACKGROUND

Clouds tend to become the targets of attackers because they consolidate too many containers into a small number of network locations. If attackers intrude into containers, they could steal sensitive information from applications processing big data such as AI and IoT. To prevent information leakage from applications, the operating system provides various security mechanisms. For example, SELinux [18] and AppArmor [5] can control access rights of applications in Linux. Such access restrictions can prevent attackers from stealing sensitive information and sending it to the outside. However, they can be disabled once the operating system is compromised.

To protect sensitive information in containers without relying on the operating system, a CPU feature called Intel SGX is used. SGX applications can create enclaves, which are protection domains whose memory is encrypted using internal keys in the CPU. The trusted computing base (TCB) is only the CPU and does not include the operating system. Since no software including the operating system can access enclave memory, enclaves can protect sensitive information from attackers and even a compromised operating system. In addition, the code and data inside a running enclave cannot be altered. Only a correctly signed code can be executed in an enclave.

In clouds, container migration is an indispensable functionality. It is a technique of moving a container to another host for various reasons. It is used to continue the execution of containers when the host running containers is maintained. When the load of a host becomes high, several containers are migrated to underloaded hosts for load balancing. Conversely, multiple containers are consolidated into a smaller number of hosts for power saving. Container migration first saves the states of all the processes in a container at the source host. Then, it transfers the saved states to the destination host and restores that container including the running processes from the received states.

However, it is difficult to migrate containers including SGX applications. If an SGX application is migrated as a normal process, its execution cannot be continued correctly. Upon migrating an SGX application, its enclave memory is also transferred as it is to the destination host. This means that the memory is encrypted using the internal key in the CPU at the source host. Since this key cannot be transferred to the destination host, the transferred enclave memory cannot be decrypted correctly. The SGX application itself might be able to re-create enclaves from scratch at the destination host, but all the states including a large amount of data are lost.

3 MIGSGX

This paper proposes MigSGX for enabling the continuous execution of the enclaves in SGX applications after container migration.

3.1 Container Migration with Enclaves

MigSGX makes an enclave itself dump and load its state because such a state cannot be accessed from the outside of the enclave due to the memory protection of SGX. When MigSGX invokes an enclave to save the state, the enclave dumps its state into the outside of it. For security, it encrypts data in its own memory using a CPU-independent key. This key is securely obtained from a key server using the secure communication channel established through remote attestation of the enclave. When MigSGX invokes a new enclave to restore the state, the enclave loads the saved state and overwrites its own state. At this time, it decrypts the data included in the saved state using the same key as used for dumping the state.

MigSGX provides a library to an application code running in an enclave. This *MigSGX library* transparently dumps and loads the state of an enclave during container migration. In addition, MigSGX runs a runtime outside enclaves in an SGX application. This *MigSGX runtime* invokes the MigSGX library in each enclave to save and restore the state. Therefore, the developers of SGX applications do not need to be aware of container migration. MigSGX also runs a manager outside a container. This *MigSGX manager* communicates with the MigSGX runtime and migrates a container including SGX applications.

Figure 1 illustrates container migration using MigSGX. At the source host, the MigSGX manager requests the runtime in each SGX application in a container to save the states of all the enclaves. When the runtime invokes the MigSGX library in each enclave, the library stops the enclave so that its state is not changed and dumps the state into the memory of the runtime. After that, the manager saves all the states of the application process. Finally, it saves the states of the container and transfers all the states. At the destination host, the MigSGX manager first restores the container and all the processes without enclaves. Then, it requests the MigSGX runtime in each process to restore the states of all the enclaves. The runtime re-creates enclaves and invokes the MigSGX library in each enclave. The library loads the saved state and restarts the enclave.

3.2 Pipelined Save/Restore

As described above, save-after-dump and load-after-restore are easy to implement, but such naive implementation is not suitable for SGX applications using a large amount of enclave memory. Since the recent third generation Intel Xeon Scalable processors enable efficient access to enclave memory of up to 512 GB, SGX applications would use bigger data in enclaves. If the MigSGX library dumps the entire enclave memory at once, the runtime temporarily needs a buffer of the same size as the enclave memory to store the dumped memory. If the memory size of the container is limited, this may cause swapping and slow down container migration. At worst, the runtime may fail to allocate the buffer and to save enclave memory. This situation can also happen when MigSGX restores SGX applications. In addition, it takes a long time to encrypt the entire enclave memory, dump the encrypted memory, and then save the dumped

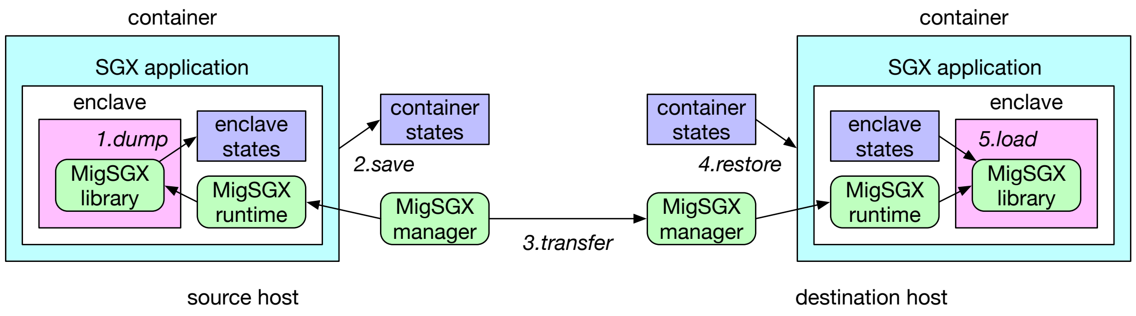


Figure 1: Container migration using MigSGX.

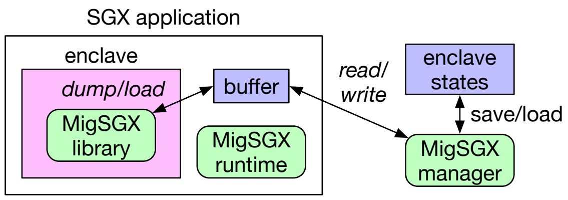


Figure 2: Pipelined save/restore of enclave memory.

memory sequentially. The sequential execution for restoring the state of an enclave can cause the same issue.

To address these issues, MigSGX saves and restores enclave memory in a pipelined manner, as illustrated in Fig. 2. Since it uses only a small buffer, it can prevent pressure on the memory of a container. Upon saving enclave memory, the MigSGX library dumps part of the enclave memory into this small buffer. Then, the MigSGX manager can save that data immediately. Similarly, upon restoring enclave memory, the MigSGX manager writes part of the dumped enclave memory to the small buffer. Then, the MigSGX library can load the data immediately. In addition, this pipelining enables parallel execution between dump and save and between restore and load. In particular, time-consuming encryption and decryption in an enclave can be overlapped with save and restore in the runtime. This overlap increases the migration performance.

To save and restore enclave memory in a pipelined manner, the MigSGX manager has to receive memory data from and send it to the MigSGX runtime. There are various communication methods between processes, but the capability of access control is mandatory. If there is no access restriction, attackers could actively make the MigSGX runtime stop all the enclaves by saving their states without the root privilege. For secure communication, the MigSGX manager injects code into the runtime and makes the code to establish shared memory for sending and receiving enclave memory. Since this code injection needs the root privilege, the availability of enclaves is kept as long as attackers cannot take the root privilege. A detailed comparison with the other communication methods is described in Section 4.4.

4 IMPLEMENTATION

We have implemented the MigSGX library and runtime in the Intel SGX SDK 1.9 [10] and the MigSGX manager in CRIU 3.9 [15].

4.1 Migrating an SGX Application

We focus only on the migration of one SGX application in a container. At the source host, the MigSGX manager first requests the process to save the states of all the enclaves to state files, as described in Section 4.2. After that, the manager saves the states of the process without enclaves as usual. It first suspends the process and saves memory mapping, mapped files, and open files by reading the proc filesystem. It also saves registers using the ptrace system call. Then, it saves the internal state that cannot be accessed from the outside of the process, e.g., memory pages and credentials, by using the *parasite* mechanism [13]. This mechanism injects parasite code into the process and makes the process itself save the states. Since it is also used for secure communication between the MigSGX manager and runtime, we describe the details of this mechanism in Section 4.4.

Next, the MigSGX manager transfers the state files to the destination host. This step can be skipped using several methods. If the state files are saved in network storage, the destination host can share the state files without explicit file transfers. As another method, CRIU provides disk-less migration [14]. Instead of saving state files in a local disk at the source host, the manager can directly save state files in a remote disk via the page server at the destination host. These methods can overlap saving the states with network transfers.

At the destination host, the MigSGX manager first restores the process without enclaves as usual. It creates a process using the fork system call and restores various states of the original process from state files. For example, it opens files, creates sockets, prepares namespaces, and maps private memory areas. Then, the manager requests the restored process to restore the states of enclaves, as described in Section 4.3.

4.2 Saving Enclave States

Figure 3 illustrates how to save the state of an enclave in an SGX application. The MigSGX manager first shares a small buffer with the runtime via the injection of parasite code, as described in Section 4.4. Then, the injected parasite code invokes the runtime. The runtime obtains the list of enclaves and saves the number of enclaves to its global variable, which is used on restoring the enclaves. For each enclave, the runtime saves the enclave ID that is assigned by the SDK and is used by the SGX application to its global variable. Also, it saves the base address where the enclave is located in the

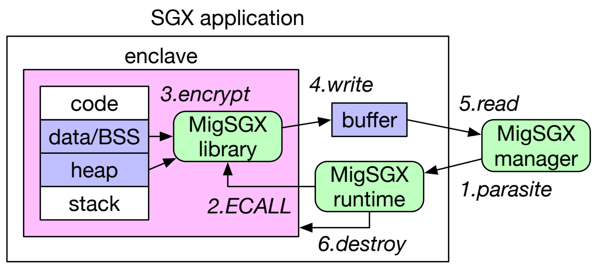


Figure 3: Saving the state of an enclave.

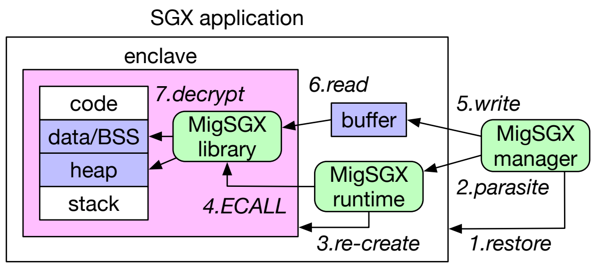


Figure 4: Restoring the state of an enclave.

process memory. In addition, it saves the path name of the enclave image file used for creating the enclave.

Then, it invokes the MigSGX library using the enclave call (ECALL) of SGX. ECALL is a mechanism for securely invoking only the functions exported by an enclave. To dump the state of an enclave, the library exports the dump function. The library implicitly defines and exports several functions that can be invoked by ECALL. In the Intel SGX SDK, the developers of SGX applications usually define such ECALL functions using the enclave definition language (EDL). Using this framework requires all the SGX applications to explicitly define the dump function in EDL. Also, the load function for restoring the state of an enclave needs to be defined. Since this is troublesome for developers, the library internally defines these ECALL functions without EDL. Then, it assigns special ECALL IDs to these internal functions and the MigSGX runtime uses these IDs to execute ECALL.

The dump function in the MigSGX library first obtains information on the heap area and the data and BSS segments of the enclave. It can obtain the address and size of the heap area using the internal API of the SDK. However, there is no information on the data and BSS segments inside the enclave at runtime. Therefore, MigSGX extracts the offsets from the beginning of the enclave image file and the sizes for these segments at compile time. Then, it embeds them into the enclave image file by modifying the global data structure used by the SDK. The library provides an API for obtaining this embedded information.

It should be noted that the MigSGX library does not dump the code or stack segment. For the code segment, enclave code is not modified and is loaded again from the enclave image file when the enclave is re-created at the destination host. For the stack segment, the library waits for the other ECALLs to finish before starting to dump the state. In addition, the MigSGX runtime suspends threads that attempt to execute new ECALLs. This means that the stacks in the enclave are empty and there is no useful information to be dumped. Similarly, the library does not dump the other states such as threads. We assume that enclaves provide only functions executed in a short time.

The MigSGX library dumps enclave memory in a pipelined manner. It first encrypts a chunk of enclave memory using the AES-NI functions ported from wolfSSL [20]. Note that the library implements these functions without using the standard malloc function so that the heap area and the data and BSS segments are not changed during the execution of the dump function. Instead, the library allocates variable-length arrays in the stack. Next, the library writes the encrypted chunk to the shared buffer allocated in the runtime. In SGX, the code inside an enclave can freely access the memory outside the enclave although the code outside an enclave cannot access enclave memory. Then, it sets the flag in the shared memory and waits for the flag to be reset by the MigSGX manager. When the flag is set, the manager reads the data from the shared buffer and resets the flag. It saves the read data to a state file created for saving the state of the enclave.

After saving the state of the enclave, the MigSGX runtime destroys that enclave. This is because releasing enclave memory can reduce the amount of process memory to be saved. Even if enclave memory is transferred, it could not be reused at the destination host. In addition, the MigSGX manager based on CRIU cannot save the memory used by the enclave because that is a special region. After all the enclaves are destroyed, the runtime closes the SGX device used by the SDK because the manager cannot save the state of this special device.

4.3 Restoring Enclave States

Figure 4 illustrates how to restore the state of an enclave in an SGX application. The MigSGX manager first shares a small buffer in the runtime using the parasite mechanism. Then, the injected parasite code invokes the runtime. The runtime re-creates the necessary number of enclaves, which is saved to its global variable. When it creates a new enclave, it specifies the path name of the enclave image file, which is also saved to its global variable. This restores the code segment of the enclave, but the data and BSS segments are initialized. Then, it re-assigns the enclave ID saved to its global variable to the enclave.

Upon re-creating an enclave, the operating system locates the memory of the enclave in a random address. Even after the enclave memory is restored correctly, the enclave cannot continue to run if the enclave memory contains addresses. To address this issue, the MigSGX runtime locates the enclave memory in the same address as at the source host, which is saved to its global variable. It specifies the saved address to the parameter of the mmap system call used for allocating enclave memory. The enclave memory can be always located in the original memory region because that region is used by the enclave just before the enclave is destroyed at the source host.

Then, the MigSGX runtime invokes the library in the enclave using ECALL to restore the enclave memory in a pipelined manner. The invoked load function waits for the flag in the shared buffer to be set. The MigSGX manager reads a chunk of encrypted memory data from a state file and writes it to the shared buffer. When it sets the flag in the shared memory, the library reads the chunk from the shared buffer and resets the flag so that the MigSGX manager can process the next chunk. The library decrypts the chunk using the same CPU-independent key as used at the source host and overwrites the heap area and the data and BSS segments in the enclave memory. This does not affect the execution of the load function itself in the enclave. This function uses only the code and stack segments, which are not overwritten by restoring the enclave memory.

4.4 Secure Communication

MigSGX achieves secure communication for pipelining between the manager and runtime using the parasite mechanism of CRIU. This mechanism can dynamically inject the parasite code for communication into a process. This code injection can be done only in the root privilege. The operating system traditionally provides many methods for communication between independent processes, which mean processes without parent and child relationship. However, no methods are enough secure or provide sufficient communication capabilities.

Network communication is the most common method, but it is difficult to perform access control in a finer-grained manner than an IP address and a port number. Authentication with public-key infrastructure is possible, but the management of the digital certificate of the MigSGX manager is troublesome. A Unix domain socket and a named pipe can perform access control by setting appropriate permission to a created special file. However, it is necessary to permit access for the user who executes an SGX application. Even if only the privilege of the user is taken, attackers could freely communicate with the application process. For System V shared memory and message queues, any processes can access the data if they know the identifier of a segment or a message queue. A signal can perform access control based on the user ID of a sender process, but it cannot send information except for a signal number.

Figure 5 illustrates how to inject parasite code into the target process of an SGX application. First, the MigSGX manager suspends the process and saves part of the code segment, which is pointed by the current instruction pointer, using the ptrace system call. Then, it writes a code fragment for executing the memfd_create system call to that code segment. When it resumes the process, the process automatically executes that system call and creates an anonymous file in the process memory. Since the manager receives the returned descriptor of that file, it maps the corresponding pseudo file exported by the proc filesystem onto its memory. Finally, the manager makes the process map the file onto its memory by executing the mmap system call using the parasite mechanism. As a result, the manager can share part of the process memory.

It is difficult for attackers to map this anonymous file and share the memory with the target process. First, its descriptor is returned internally via the ptrace system call. Attackers cannot eavesdrop on the exact number of the file descriptor. Second, the target process closes the file soon after it maps that file. Therefore, attackers have only a very short time to illegally map the pseudo file exported by the proc filesystem.

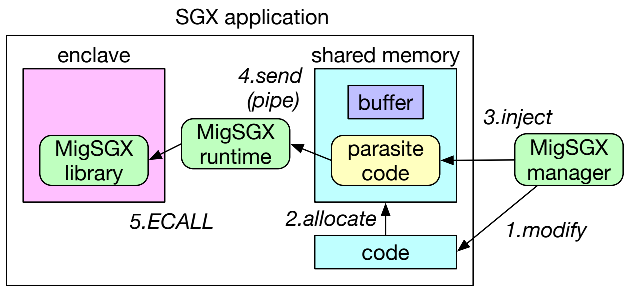


Figure 5: Injecting parasite code.

Once the shared memory is established, the MigSGX manager allocates a small buffer used for pipelining in it. Then, it writes parasite code to the shared memory and makes the process execute it. The parasite code invokes the MigSGX library in each enclave, but it cannot do that by directly executing ECALL. This is because the parasite code cannot use any functions of the SDK. Therefore, the runtime runs two threads for invoking the library and creates two anonymous pipes between the main thread and either thread. Since anonymous pipes cannot be accessed outside the process, they can be used securely. To enable the parasite code to access these anonymous pipes, the runtime assigns the fixed numbers to their file descriptors. When the parasite code sends a request to the thread via the pipe, the thread invokes the library using ECALL.

4.5 Key Management

MigSGX provides a trusted key server and manages CPU-independent keys used for encrypting and decrypting enclave memory. When it saves the state of an enclave at the source host, the key server generates a new key and shares it with the enclave. First, it remotely attests to the enclave and establishes a secure communication channel to the enclave. Then, it securely sends the generated key to the enclave. Note that it holds that key during migration. When the enclave is re-created at the destination host, it shares the same key with the key server. The key server first performs remote attestation to the new enclave. If this enclave is identical to the enclave saved at the source host, the key server securely sends the corresponding key to that enclave. This prevents attackers from illegally obtaining CPU-independent keys and decrypting enclave memory.

5 EXPERIMENTS

We conducted several experiments to show that MigSGX could migrate an SGX application efficiently. For comparison, we used a naive migration method, which saved and restored enclave memory without pipelining, as described in Section 3.1. We used a PC with an Intel Xeon E3-1225 v5 processor, 32 GB of memory, 1 TB of HDD, and Gigabit Ethernet as the source host. As the destination host, we used another PC with an Intel Core i7 8700 processor, 32 GB of memory, 1 TB of HDD, and Gigabit Ethernet. At both hosts, we ran Linux 4.4 with the Intel SGX driver, the Intel SGX SDK 2.2, and CRIU 3.9.

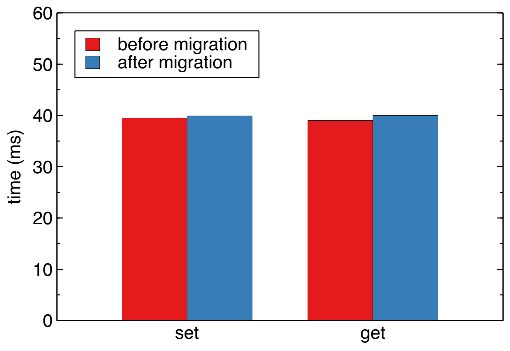


Figure 6: The performance of the in-enclave key-value store.

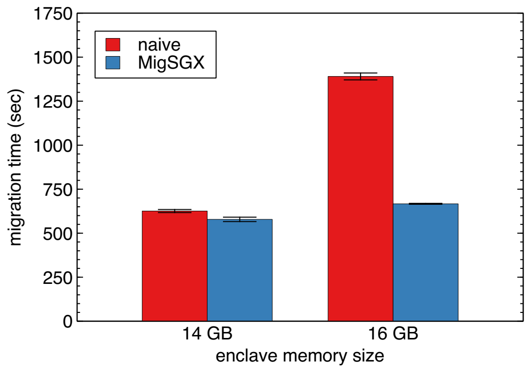


Figure 7: The migration time of the SGX application.

5.1 Migration of a Key-value Store

As an SGX application, we have developed a key-value store running in an enclave. This application stores data in enclave memory and manages it using a hash table. It handles requests for the set and get operations and invokes ECALL for storing and retrieving data, respectively. We stored a large amount of data in this key-value store by sending the set requests to the source host. Then, we migrated this application using MigSGX and retrieved the stored data by sending the get requests to the destination host. Since we could obtain the same data, it was shown that MigSGX could migrate this SGX application correctly.

Next, we examined the impact of the migration on the application performance. We measured the time needed for storing and retrieving data to and from the database before and after the migration. Figure 6 shows the performance of the set and get operations. The performance degraded by 1–3% after the migration, but the root cause of this small overhead is under investigation.

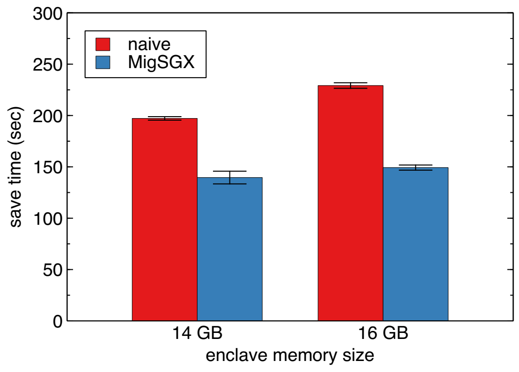


Figure 8: The save time of the SGX application.

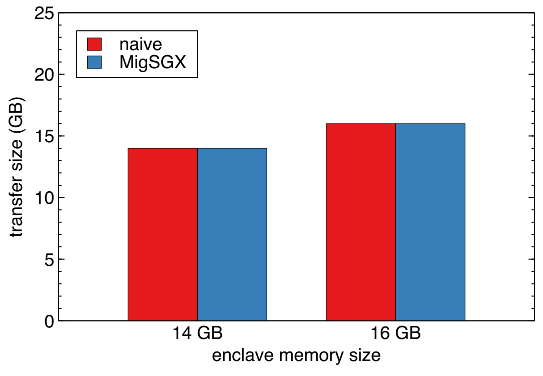
5.2 Migration Performance

To examine the migration performance of an SGX application with a large amount of enclave memory, we measured the time needed for migrating the developed key-value store. In this experiment, we allocated 14 GB or 16 GB of memory to its enclave. When the application used 14 GB of enclave memory, more than 16 GB of free memory was left in the host. This free memory was enough for dumping the entire enclave memory at once. However, when the application used 16 GB of enclave memory, less than 16 GB of free memory was left. Swap space was necessary to dump the entire enclave memory without pipelining. We used a buffer of 20 MB for pipelining in MigSGX.

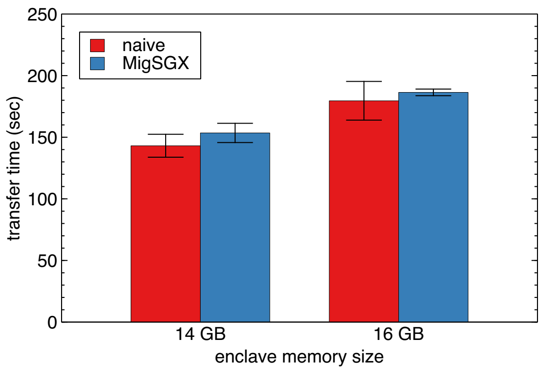
Figure 7 shows the migration time in MigSGX and the naive migration method. For 14 GB of enclave memory, MigSGX could reduce the migration time only by 7.6%. In contrast, it became 52% faster for 16 GB of enclave memory. We examined the breakdown of the migration time to clarify the reason for this performance improvement.

The migration of the SGX application consists of saving the states of its process, transferring them, and restoring them. Figure 8 shows the save time of the states of the process. Compared with the naive method, MigSGX could reduce the save time by 29% for 14 GB of enclave memory and 35% for 16 GB of that. This is because MigSGX could overlap encrypting and dumping enclave memory with saving the dumped memory by pipelining. In contrast, the naive method needed to first encrypt and dump the entire enclave memory and then save the process memory including the dumped memory sequentially. Due to this, it caused swapping due to out-of-memory for 16 GB of enclave memory. MigSGX could suppress swapping by using only a small buffer.

As shown in Fig. 9(a), the size of transferred data was the same between MigSGX and the naive method. It was dominated by the size of enclave memory. MigSGX saved enclave memory independently of the states of the process, while the naive method saved it with the states of the process. In any case, the size of the saved states was the same. However, MigSGX took a slightly longer time to transfer the saved states, as shown in Fig. 9(b). This is because the current implementation of MigSGX creates one state file for each chunk of enclave memory and sequentially sends them. It needs a longer time to transfer multiple files, compared with transferring one large file like the naive method. If we overlap dumping enclave memory with transferring the created state files, the transfer time should become much shorter. Furthermore, we can transfer the



(a) Transfer size



(b) Transfer time

Figure 9: The transfer size and time of the SGX application.

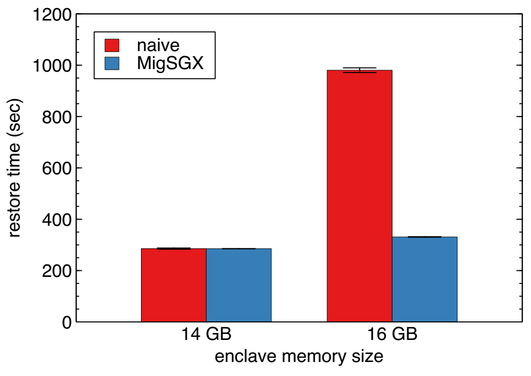


Figure 10: The restore time of the SGX application.

dumped memory without creating state files. These are our future work.

Figure 10 shows the restore time of the saved states of the process. Unlike saving enclave memory, the restore time was almost the same between MigSGX and the naive method when the SGX application used 14 GB of enclave memory. MigSGX could reduce the restore time only by 5 seconds although the naive method had to restore a 14-GB larger amount of process memory including the dumped enclave memory. This is because MigSGX just mapped the state file to which the process memory was saved onto the re-created process without reading the file. For both methods, it took 110 seconds to re-create the enclave after the states of the process wad restored. In MigSGX, it took 6 seconds longer time to load the enclave memory due to the overhead of pipelining. Therefore, MigSGX could not improve the restore performance in this case.

For 16 GB of enclave memory, in contrast, the restore time was much shorter in MigSGX. This is because MigSGX could avoid swapping due to out-of-memory. When re-creating an enclave, the naive method took 225 seconds longer time by the memory pressure of the enclave memory saved in the process memory. Similarly, the time for loading the enclave memory was 396 seconds longer.

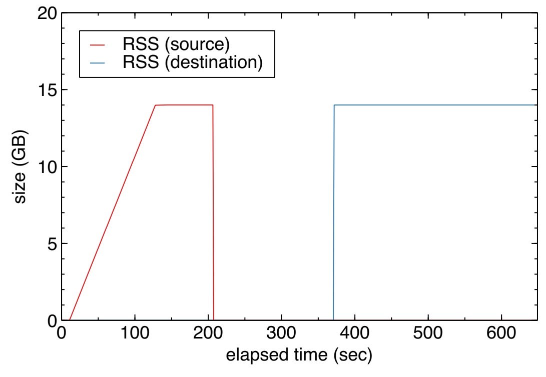
5.3 Memory Usage

To examine the memory usage of the SGX application during migration, we measured the changes in consumed physical memory and swap space at each host. The physical memory consumed by the process is called the resident set size (RSS). Note that the RSS does not include the size of enclave memory because the operating system deals with that memory region as special. As in the previous section, we migrated the application using 14 GB or 16 GB of enclave memory.

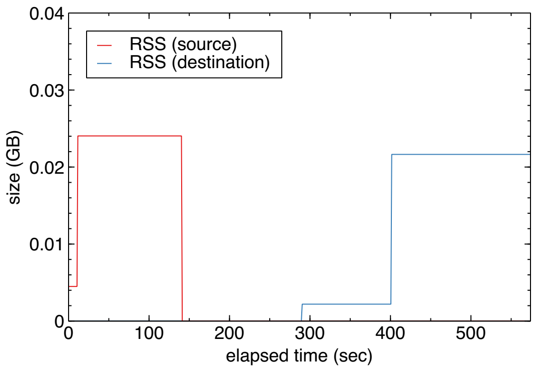
Figure 11(a) shows the changes in memory usage when we used the naive method to migrate the application with 14 GB of enclave memory. At the source host, the RSS gradually increased until it became 14 GB. This is because the enclave encrypted and dumped its memory into the process memory. The naive method started to save the states of the process at 130 seconds and terminated the process at 207 seconds. At the destination host, the RSS suddenly increased to 14 GB at 372 seconds by restoring the process memory including enclave memory. The RSS did not change after that. Swap space was not used at both hosts.

The changes in memory usage for MigSGX are shown in Fig. 11(b). At the source host, the RSS increased by 20 MB at 10 seconds because the process allocated a small buffer for pipelining. The RSS did not change until the process was terminated. At the destination host, the RSS increased only slightly after the states of the process were restored. When MigSGX started to load the enclave memory at 400 seconds, the RSS increased by 20 MB, which was used as the buffer for pipelining. Swap space was not used at both hosts.

On the other hand, Figure 12(a) shows the changes in memory usage when we migrated the application with 16 GB of enclave memory using the naive method. At the source host, the RSS gradually increased to 16 GB by dumping the enclave memory into the process memory. Unlike 14 GB of enclave memory, swap space was used up to 3.6 GB because physical memory ran out. At the destination host, the RSS suddenly increased to 16 GB, but it decreased by 1.3 GB at 482 seconds while an enclave was re-created. At the same time, swap space increased by 3.5 GB. After the naive method started to load the enclave memory at 771 seconds, the RSS and the swap size increased gradually.

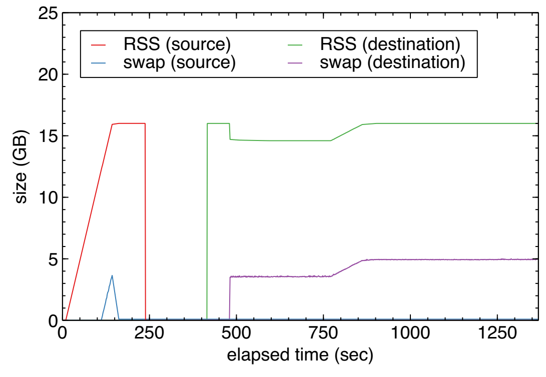


(a) Naive

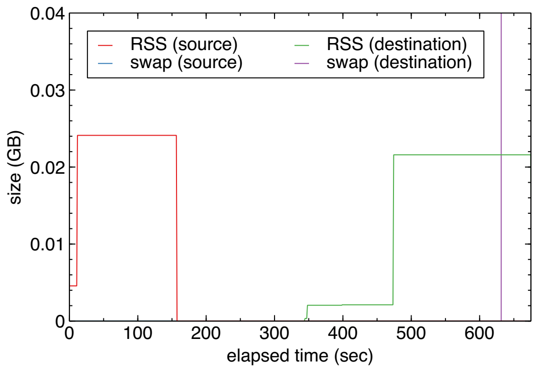


(b) MigSGX

Figure 11: The memory usage during the migration of the SGX application with 14 GB of enclave memory.



(a) Naive



(b) MigSGX

Figure 12: The memory usage during the migration of the SGX application with 16 GB of enclave memory.

The changes in memory usage for MigSGX are shown in Fig. 12(b). The memory usage was almost the same as the migration of the application with 14 GB of enclave memory. Only one difference was that the destination host started to use swap space in the middle of loading the enclave memory. The swap size increased by 2.1 GB finally. Since MigSGX did not cause swapping at the source host, physical memory should not run out at the destination host as well. The Intel SGX SDK might use more memory in a hidden manner, but the reason for this swapping was under investigation.

5.4 CPU Usage

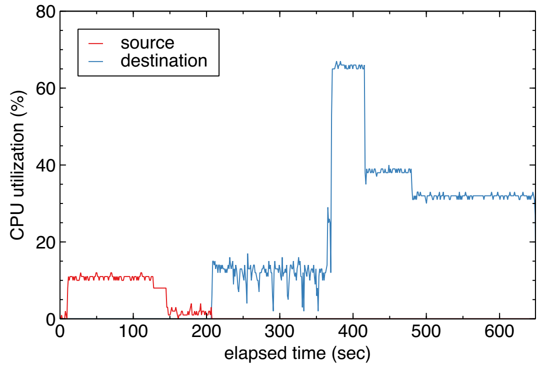
To examine how pipelining in MigSGX affected the CPU usage during migration, we measured the entire CPU utilization at both hosts. Figure 13 shows the changes in CPU utilization when we migrated the application with 14 GB of enclave memory. CPUs were used very similarly in both methods. The CPU utilization for dumping enclave memory was about 10%. That for re-creating the enclave was 65% at first and then reduced to 40%. That for loading the enclave memory was about 30%. It should be noted that the CPU utilization for saving the states of the process was different. The naive method slightly used more CPUs for 77 seconds to save the dumped enclave memory.

On the other hand, Fig. 14 shows the changes in CPU utilization when we migrated the application with 16 GB of enclave memory. The CPU utilization in MigSGX was similar to the above, but that for the naive method was largely different at the destination host. While the enclave was re-created, the CPU utilization was high at first but became much lower after that. While the enclave memory was loaded, the CPU utilization slightly increased but much lower. This is because the naive method caused frequent swapping and was IO-intensive.

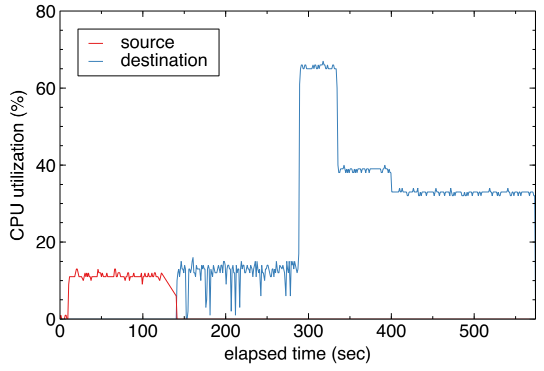
6 RELATED WORK

This paper focuses on the migration of containers including SGX applications, but that of VMs running SGX applications has been already studied.

The SGX extension called eMotion [16, 17] introduces a new instruction set. This method prepares migration enclaves at the source and destination hosts, performs remote attestation each other, and exchanges a master key. SGX generates a migration key from the master key. At the source host, the hypervisor saves

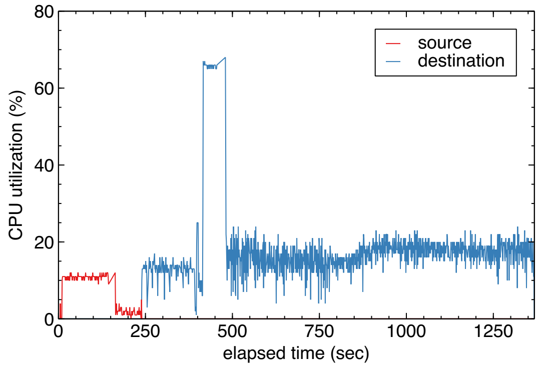


(a) Naive

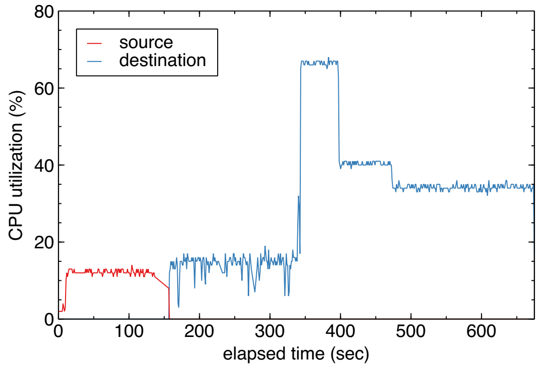


(b) MigSGX

Figure 13: The CPU usage during the migration of the SGX application with 14 GB of enclave memory.



(a) Naive



(b) MigSGX

Figure 14: The CPU usage during the migration of the SGX application with 16 GB of enclave memory.

enclave memory re-encrypted using the migration key by SGX. At the destination host, it restores enclave memory, which is re-encrypted using another key by SGX. This method does not require any modifications to SGX applications, but hardware modification by Intel is necessary.

TEEnder [8] uses hardware security modules (HSMs) to securely re-encrypt enclave memory. At the source host, an enclave sends its memory data to the HSM and receives encrypted memory data. At the destination host, a re-created enclave sends the encrypted memory data to the HSM and receives decrypted memory data. This method can avoid vulnerabilities in remote attestation and migration enclaves, but it is costly to use hardware appliances.

A software-only method has been also proposed [7]. This method makes an enclave itself dump and load its state like MigSGX, but it has four drawbacks. First, an enclave dumps the entire enclave memory into the outside at once. When the size of enclave memory is large, swapping in the VM can largely degrade migration performance, as shown in our experiments. MigSGX addresses this issue using pipelining. Second, the guest operating system in a VM sends signals to SGX applications to save the states of enclaves when it receives a virtual interrupt from the hypervisor on VM migration. In this method, it is difficult to prevent attackers from illegally sending the save requests to SGX applications. MigSGX uses the parasite mechanism to securely communicate with SGX applications. In addition, it is necessary to modify the operating system unlike MigSGX. Finally, this method is not applicable to applications developed using the standard Intel SGX SDK because it is implemented using its own SDK. MigSGX is implemented in the Intel SDK.

In addition to the internal states of enclaves, its external persistent states can be maintained after VM migration [3]. For example, such states include data written to the outside after being sealed with a secret key per enclave and monotonic counters. This method provides the library for VM migration to an enclave and the enclave seals data using the secret key generated in the library. Since the key is migrated with the state of the enclave, the sealed data can be unsealed at the destination host. For monotonic counters, the library transfers the values and resets them at the destination host.

AMD SEV [1] provides another trusted execution environment. It encrypts the memory of VMs and natively supports VM migration. At the source host, the hypervisor saves the target VM's memory, which is re-encrypted for transmission by SEV. At the destination host, the hypervisor restores the VM's memory, which is re-encrypted using another key by SEV. For SEV-SNP [2], the hypervisor swaps out the entire memory of a VM at the source host. The swapped memory is re-encrypted by the offline encryption key. At the destination host, the hypervisor swaps in the VM's memory, which is re-encrypted using another key.

The migration of GPGPU applications [21] has a similarity to that of SGX applications. GPUs have states that cannot be accessed from the outside. In addition, GPU kernels cannot be suspended from the outside. This method allows kernels running in GPUs to cooperate with the migration framework. It achieves the migration of GPGPU applications using GLoop [19], which is a framework for enabling cooperative multitasking in GPGPU applications. Using GLoop, it makes GPU kernels themselves suspend their execution and save all the execution contexts of GPUs. Then, it migrates the process and restores the states of GPUs using GLoop at the destination host.

7 CONCLUSION

This paper proposed MigSGX for enabling the migration of containers including SGX applications. Since the state of an enclave cannot be correctly saved or restored from the outside, MigSGX makes the enclave itself dump and load its state. To securely transfer the state, the enclave encrypts and decrypts the state using a CPU-independent key. For space- and time-efficiency, MigSGX saves and restores enclave memory in a pipelined manner. It protects the communication with the target SGX applications using the parasite mechanism. We have implemented MigSGX in the Intel SGX SDK and CRIU. Our experiments show that pipelining could improve migration performance by up to 52% and that the memory necessary for migration was reduced only to 0.15%.

One of our future work is to support live migration of SGX applications. Currently, MigSGX stops the services provided by enclaves and then saves the states of the enclaves. We need to overlap the execution of the services with saving the states of enclaves to reduce the service downtime. To enable this, it is necessary to efficiently detect the memory regions modified during saving the states.

ACKNOWLEDGMENTS

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan.

REFERENCES

[1] Advanced Micro Devices, Inc. 2020. *Secure Encrypted Virtualization API Version 0.24.*

[2] Advanced Micro Devices, Inc. 2021. *SEV Secure Nested Paging Firmware ABI Specification.*

[3] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan. 2018. Migrating SGX Enclaves with Persistent State. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 195–206.

[4] Amazon Web Services, Inc. 2014. *Amazon Elastic Container Service.* Retrieved October 17, 2021 from https://aws.amazon.com/ecs/

[5] AppArmor Security Project. 1998. *AppArmor.* Retrieved October 17, 2021 from https://gitlab.com/apparmor

[6] Google LLC. 2015. *Google Kubernetes Engine.* Retrieved October 17, 2021 from https://cloud.google.com/kubernetes-engine

[7] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li. 2017. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 225–236.

[8] J. Guerreiro, R. Moura, and J. Silva. 2020. TEEnder: SGX enclave migration using HSMs. *Computers & Security* 96 (2020), 101874.

[9] IBM Corp. 2020. *IBM Cloud Data Shield.* Retrieved October 17, 2021 from https://www.ibm.com/cloud/data-shield

[10] Intel Corp. 2016. *Intel Software Guard Extensions SDK for Linux.* Retrieved October 17, 2021 from https://01.org/intel-softwareguard-extensions

[11] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy.*

[12] Microsoft Corp. 2017. *Azure Confidential Computing.* Retrieved October 17, 2021 from https://azure.microsoft.com/en-us/solutions/confidential-compute/

[13] OpenVZ Team. [n.d.]. *Compel.* Retrieved October 17, 2021 from https://criu.org/Compel

[14] OpenVZ Team. [n.d.]. *Diskless Migration.* Retrieved October 17, 2021 from https://criu.org/Disk-less_migration

[15] OpenVZ Team. 2012. *CRIU.* Retrieved October 17, 2021 from https://criu.org/Main_Page

[16] J. Park, S. Park, B. Kang, and K. Kim. 2019. eMotion: An SGX Extension for Migrating Enclaves. *Computers & Security* 80 (2019), 173–185.

[17] J. Park, S. Park, J. Oh, and J. Won. 2016. Toward Live Migration of SGX-Enabled Virtual Machines. In *Proceedings of World Congress on Services.* 111–112.

[18] SELinux Project. 2000. *SELinux Project.* Retrieved October 17, 2021 from https://github.com/SELinuxProject

[19] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. 2017. GLoop: An Event-driven Runtime for Consolidating GPGPU Applications. In *Proceedings of the 8th ACM Symposium on Cloud Computing.* 80–93.

[20] wolfSSL Inc. 2006. *wolfSSL Embedded SSL/TLS Library.* Retrieved October 17, 2021 from https://www.wolfssl.com/

[21] S. Yuhara, Y. Suzuki, and K. Kono. 2018. An Application Framework for Migrating GPGPU Cloud Applications. In *Proceedings of the 2008 IEEE International Conference on Cloud Computing Technology and Science.*