SEmigrate: Optimizing Data Protection with VM Introspection

Shuhei Horio
Kyushu Institute of Technology
horio@ksl.ci.kyutech.ac.jp

Kouta Takahashi
Kyushu Institute of Technology
takahashi@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@csn.kyutech.ac.jp

Abstract—**Recently, virtual machines (VMs) with a large amount of memory are widely used. Since it is often difficult to migrate such a large-memory VM to one large destination host, split migration divides the memory of a VM into small fragments and transfers them to multiple destination hosts. The migrated VM exchanges its memory data between the hosts using remote paging. To prevent information leakage from and tampering with the memory data in an untrusted environment, memory encryption and integrity checking can be used. However, the overhead of such data protection affects the performance of the hosts and the VM more largely in faster networks. This paper proposes *SEmigrate* for optimizing data protection in split migration and remote paging. SEmigrate avoids decrypting memory data and integrity checking at most of the destination hosts to reduce the protection overhead and completely prevent information leakage. Also, it can *selectively* encrypt only sensitive memory data and check the integrity of only important memory data by analyzing the memory of the guest operating system and applications in a VM. SEmigrate could reduce the time for data-protected split migration by up to 43% and improve the performance of migrated VMs by up to 19% in 100 Gigabit Ethernet.**

Index Terms—**VM migration, VM introspection, selective data protection, split migration, remote paging**

1. Introduction

Recently, virtual machines (VMs) with a large amount of memory are widely used. For example, Amazon EC2 provides VMs with up to 24 TB of memory [1]. Upon host maintenance, a VM can be moved to another host using VM migration without disrupting the services provided by the VM. For large-memory VMs, however, it is not cost-efficient to always preserve large hosts with a sufficient amount of memory as the destination of occasional VM migration. In particular, private clouds may not afford to prepare a sufficient number of large hosts.

To make the migration of such large-memory VMs more flexible, split migration has been proposed [2]. Split migration divides the memory of a VM into small fragments and transfers them to multiple smaller destination hosts, which consist of one main host and one or more sub-hosts. It transfers likely accessed memory data to the main host as well as the state of the VM core such as virtual CPUs. The rest of the memory is transferred to sub-hosts. After split migration, the main host executes the VM core, while the sub-hosts provide the memory to the VM core. When the VM accesses the memory existing in a sub-host, it exchanges memory data between the main host and the sub-host using remote paging. The sub-host transfers the required memory data to the main host, while the main host transfers unnecessary memory data to the sub-host.

However, it is possible to eavesdrop on and tamper with the memory data of a VM during split migration and remote paging in an untrusted execution environment. For example, information leakage and manipulation easily occur if the memory data is transferred via untrusted networks. If the administrators of some of the hosts are untrusted, they can steal and alter the memory data held in the hosts. In general, the encryption and integrity checking of the memory data can prevent such attacks. However, the overhead of such data protection becomes relatively larger as higher-speed networks are being used, e.g., several hundred Gigabit Ethernet (GbE). Consequently, the overhead largely affects the performance of the hosts and the VM because encryption, decryption, and integrity checking are performed whenever memory data is transferred.

To address this performance issue, we propose *SEmigrate* for optimizing data protection in data-protected split migration and remote paging. SEmigrate avoids decrypting memory data and integrity checking at sub-hosts to reduce the overhead of data protection and completely prevent information leakage. Upon split migration, it encrypts memory data at the source host and then decrypts that data and checks its integrity only at the destination main host. Upon remote paging, it encrypts and decrypts memory data and checks its integrity only at the main host. In addition, SEmigrate can *selectively* encrypt only memory data containing sensitive information and check the integrity of only important memory data to further reduce the overhead of data protection.

To obtain information needed for such selective data protection, SEmigrate analyzes the memory of the guest operating system (OS) in a VM using a technique called *VM introspection (VMI)* [4]. For example, it considers that the free memory in a VM does not contain valid data. If SEmigrate determines that memory data to transfer is part of the free memory, it does not encrypt that memory

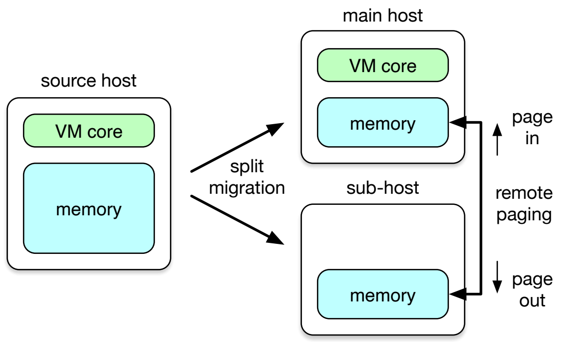


Figure 1. Split migration and remote paging.

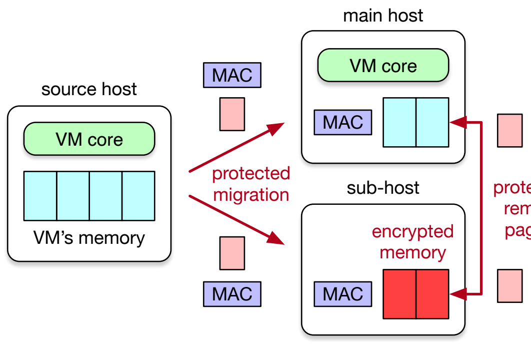


Figure 2. Split migration and remote paging with data protection.

data or check its integrity. When the user specifies that an application does not deal with sensitive information, SEmigrate does not encrypt the entire memory of the process executing that application. It also uses application-specific information to protect part of the memory in a fine-grained manner.

We have implemented SEmigrate in KVM [5] supporting split migration and remote paging. To confirm performance improvement by SEmigrate, we ran an application using a large amount of memory in a VM and examined the performance of data-protected split migration and the migrated VM with data-protected remote paging. As a result, it was shown that SEmigrate could reduce the migration time by up to 43% in 100 GbE. Also, we showed that SEmigrate could improve the performance of migrated VMs by up to 19%.

The organization of this paper is as follows. Section 2 describes split migration and remote paging and the issues of data protection of VMs. Section 3 proposes SEmigrate for optimizing data protection in split migration and remote paging. Section 4 explains the implementation of SEmigrate and Section 5 shows our experimental results. Section 6 describes related work and Section 7 concludes this paper.

2. Data Protection of VMs across Hosts

2.1. Split Migration

Split migration [2] divides the memory of a VM into small fragments and transfers them to multiple small destination hosts, as illustrated in Fig. 1. The destination hosts consist of one main host and one or more sub-hosts. Split migration transfers likely accessed memory data to the main host as much as possible. It also transfers the state of the VM core such as virtual CPUs and devices. The rest of the memory data is transferred to one of the sub-hosts.

After split migration, the migrated VM runs across the main host and the sub-hosts. The main host executes the VM core, while the sub-hosts provide the memory to the VM core. When the VM core requires the memory data existing in a sub-host, that data is exchanged between the main host and the sub-host using remote paging. The sub-host transfers the memory data required by the VM core, which is called a page-in. At the same time, the main host transfers unlikely accessed memory data to the sub-host, which is called a page-out.

2.2. Protection of Memory Data

In an untrusted execution environment, it is possible to eavesdrop on and tamper with the memory data of a VM transferred during split migration and remote paging. For example, information leakage and manipulation easily occur if the memory data is transferred via untrusted networks. In addition, the memory data can be stolen and altered by intruders and untrusted administrators at any host. Fortunately, several mechanisms have been proposed to protect the memory of a running VM using the hypervisor [6], [7], the security monitor below the hypervisor [8], and processors [9], [10]. Since we can use such memory protection mechanisms at the source host running the entire VM and the main host running the VM core, we assume that information leakage does not occur at these hosts in this paper.

In general, information leakage and manipulation can be prevented by encrypting the memory data of a VM and checking its integrity, respectively. Fig. 2 shows data-protected split migration and remote paging. Upon split migration, the memory data is encrypted at the source host by using an encrypted communication channel such as TLS [11]. Encrypted data is transferred to the destination main host and sub-hosts and is then decrypted by the channel. After that, the sub-hosts re-encrypt the decrypted memory data to securely hold it against intruders and untrusted administrators. The reasons why decryption and re-encryption are required are that the channel automatically decrypts memory data and that it is difficult to continue to use the decryption key created for the channel after the communication. Note that such re-encryption is not necessary at the main host because the memory data is managed in a protected manner [6]–[8].

Upon remote paging, the memory data held in a sub-host is first decrypted. Then, it is re-encrypted at the sub-host by using an encrypted communication channel established between the sub-host and the main host. It is transferred to the main host for a page-in and is then decrypted by

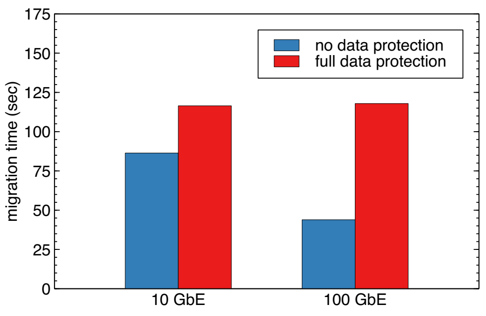


Figure 3. The migration time with data protection.

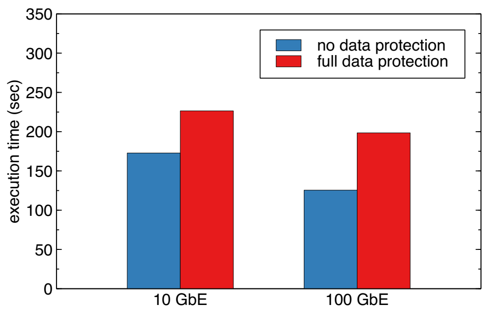


Figure 4. The benchmark time with data protection.

the channel. For a page-out, unnecessary memory data is encrypted at the main host by using this channel and is then transferred to the sub-host. It is decrypted at the sub-host and is then re-encrypted to be securely held.

These encrypted communication channels also check the integrity of transferred memory data. Upon split migration, the source host calculates the message authentication code (MAC) of memory data before it encrypts the data. MAC is a hash value of data and a secret key. After the destination host decrypts the received data, it re-calculates the MAC of memory data and compares it with the received MAC. If this comparison fails, the destination host can detect that memory data is tampered with during the transfer. Upon remote paging, a sender host calculates the MAC of memory data and transfers it, whereas a receiver host re-calculates MAC and compares it with the received one.

However, using encrypted communication channels imposes a large overhead because encryption, decryption, and integrity checking are performed whenever memory data is transferred. Fig. 3 shows the time needed for split migration when data protection is applied to memory data. We used the experimental setup in Section 5 and measured the performance in both 10 GbE and 100 GbE. Even in 10 GbE, the performance was degraded by 26% due to data protection. When we used 100 GbE, the performance degradation reached 63%. This is because the migration performance did not increase at all by the overhead of data protection. Fig. 4 shows the execution time of the benchmark that causes excessive paging when remote paging is performed with data protection. In this experiment, data protection affected a negative impact more largely in 100 GbE. This means that data protection in split migration and remote paging becomes more critical in faster networks.

In terms of security, the memory data of a VM is still exposed at sub-hosts due to re-encryption. Since it is decrypted temporarily by an encrypted communication channel, attackers can eavesdrop on the memory data before the data is re-encrypted. In addition, the memory data held in sub-hosts can be easily decrypted if its decryption keys are stolen by the administrators of sub-hosts.

3. SEmigrate

This paper proposes SEmigrate for optimizing data protection in data-protected split migration and remote paging. SEmigrate avoids decrypting the memory data of a VM and checking its integrity at sub-hosts to reduce protection overhead and completely prevent information leakage. To further reduce the overhead, it can selectively encrypt only sensitive memory data and check only the integrity of important memory data. This optimization is based on the fact that the entire memory data of a VM does not always need to be protected. To use memory attributes and process information for these optimizations, SEmigrate analyzes the memory of the guest OS in a VM using VMI [4]. In addition, it analyzes the memory of applications to use application-specific information.

3.1. Sub-host Optimization

SEmigrate always avoids the decryption of the memory data of a VM at sub-hosts, as illustrated in Fig. 5. Upon split migration, the source host encrypts memory data, while only the destination main host decrypts it. The destination sub-hosts hold it without decrypting it. To enable decrypting the encrypted memory data later, SEmigrate uses an encryption key that is available through the life cycle of a VM. Therefore, it does not use an encrypted communication channel that is established only for VM migration. This can reduce the overhead of the re-encryption of memory data as well as that of the decryption. Also, this can prevent information leakage by temporarily decrypting memory data in an encrypted communication channel. Since the sub-hosts do not manage any keys for decrypting encrypted memory data, even the administrators of the sub-hosts cannot decrypt or eavesdrop on the memory data.

Upon remote paging, SEmigrate encrypts and decrypts the memory data of a VM only at the main host. For a page-in, a sub-host does not decrypt the memory data requested by the main host. Then, it can securely transfer encrypted memory data to the main host without using an encrypted communication channel. The main host decrypts the received memory data and uses it. For a page-out, the main host encrypts unnecessary memory data without using

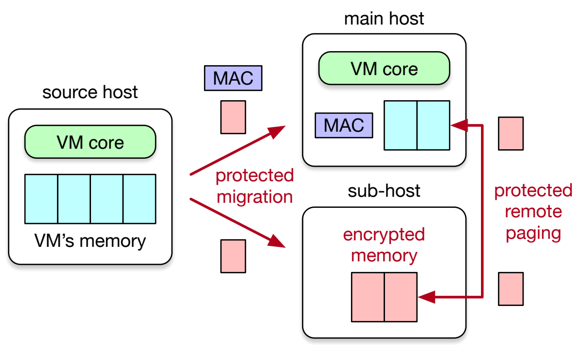


Figure 5. The optimization for reducing the overhead of data protection and enhancing security at sub-hosts.

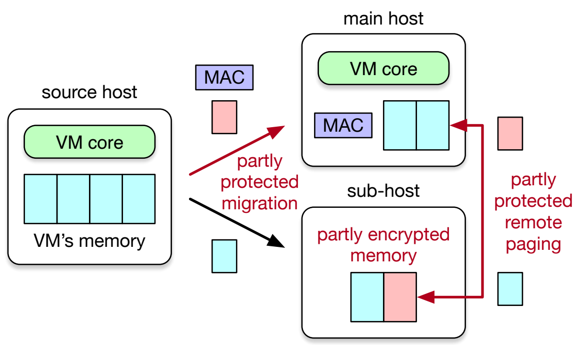


Figure 6. The optimization of selective encryption and integrity checking.

an encrypted communication channel and then securely transfers it to a sub-host. The sub-host holds it without decrypting it. SEmigrate uses the encryption and decryption keys shared with split migration to encrypt and decrypt memory data at the main host in remote paging.

Also, SEmigrate always avoids the integrity checking of memory data of a VM at sub-hosts. Upon split migration, the source host calculates the MAC of memory data transferred to a sub-host and transfers it to the main host, instead of the sub-host. Since integrity checking is not performed at the sub-host, it is not necessary to transfer the MAC to the sub-host. When the main host receives memory data from a sub-host upon a page-in, it decrypts the data and re-calculates the MAC of the memory data. Then, it compares the calculated MAC with the one held in the main host, which has been transferred on split migration. Upon a page-out, the main host calculates the MAC of transferred memory data and holds it without transferring it to a sub-host. Like this, SEmigrate does not transfer the MAC of memory data between the main host and a sub-host at all.

One disadvantage of avoiding integrity checking at sub-hosts is that the detection of tampering with memory data is delayed. Even if memory data is modified while split migration transfers it to a sub-host, the sub-host cannot detect that tampering immediately. When the memory data is transferred to the main host on a page-in, the main host can detect the previous tampering by integrity checking. If tampering is detected during split migration, we can cancel VM migration and continue to run the VM at the source host. However, we cannot continue the VM and have to stop it if tampering is detected after VM migration. Therefore, we need to consider a tradeoff between performance improvement and reliability. To address this issue, we can save memory data transferred to the sub-hosts in a disk at the source host. If tampering is detected at the first page-in, we can continue the VM by retransferring the saved memory data to the main host.

3.2. Selective Data Protection

SEmigrate selectively encrypts the memory data of a VM at the source host and the main host to reduce encryption overhead, as illustrated in Fig. 6. Upon split migration, the source host encrypts only the memory data that contains sensitive information, while it does not encrypt the other memory data. The destination main host decrypts the received data only if the memory data is encrypted. The destination sub-hosts hold it without encrypting it even if the memory data is not encrypted. Since that memory data does not contain sensitive information, it does not need encryption at sub-hosts as well.

Upon remote paging, a sub-host transfers the memory data requested for a page-in to the main host without encrypting it even if the memory data is not encrypted. The main host decrypts the received data only if the memory data is encrypted. For a page-out, the main host transfers unnecessary memory data to a sub-host without encrypting it if the memory data does not contain sensitive information. The sub-host holds it without encrypting it even if the memory data is not encrypted.

SEmigrate considers various memory regions in a VM to be not sensitive. For example, the memory regions that are not used by the guest OS or any applications, i.e., free memory, do not contain sensitive information. Therefore, SEmigrate does not encrypt free memory. When memory data is transferred, SEmigrate obtains its memory attribute from the guest OS using VMI and examines whether it is free memory or not. In addition, SEmigrate does not encrypt code segments of the guest OS or processes because programs do not contain sensitive information in general. When memory data is transferred, SEmigrate checks its memory attribute and determines that that memory is part of the code segments if it is executable.

When the user specifies an application that does not deal with sensitive information in a VM, SEmigrate does not encrypt the memory of the corresponding process used to execute that application. For example, if an in-memory database such as memcached [12] deals with only encrypted data, its memory data does not need to be further encrypted by SEmigrate when it is transferred. For an application created using Intel SGX [13], its memory does not need to be encrypted by SEmigrate if sensitive information is dealt with only inside memory regions called enclaves, which are protected by processors. When memory data is transferred, SEmigrate finds the memory region to which that data belongs and identifies the process that owns that memory region. If the name of that process is equal to the one specified by the user, SEmigrate does not encrypt the memory data to be transferred.

When the user specifies a specific memory region in a specific application, SEmigrate does not encrypt that memory region. For example, if an in-memory database holds both encrypted data and its decryption key in memory, only the encrypted data does not need to be encrypted by SEmigrate. When an SGX application saves the memory data of enclaves to its process memory in an encrypted form during VM migration [14], the encrypted data does not need to be encrypted by SEmigrate. If that application also deals with sensitive information outside enclaves, that data needs to be encrypted. When memory data is transferred, SEmigrate analyzes the memory of the target process in a VM and obtains application data. Then, it identifies the memory regions that do not need to be encrypted. If memory data to be transferred is contained in those regions, SEmigrate does not encrypt the memory data.

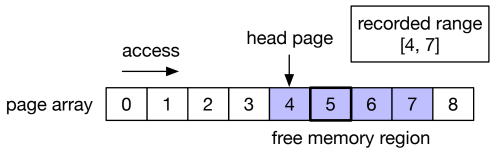
Similarly, SEmigrate reduces the overhead by selectively checking the integrity of memory data. Upon split migration, SEmigrate calculates and transfers MAC only for the memory that needs to preserve the integrity. Upon a page-in, only if the main host holds MAC, it calculates the MAC of received memory data and compares it with the holding MAC. Upon a page-out, it calculates MAC only for the memory that needs to preserve the integrity and holds it. For example, free memory always does not need integrity checking because it does not contain any information to be protected. If an application checks the integrity of its data by itself, SEmigrate does not need to perform integrity checking for the data.

4. Implementation

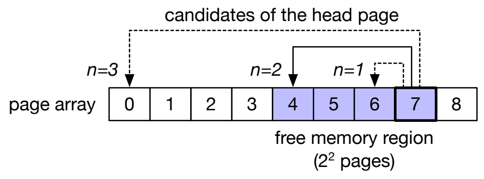
We have implemented SEmigrate in QEMU-KVM 7.1.0 [15] supporting split migration and remote paging. We assume Linux 4.18.17 as a guest OS to apply VMI for selective data protection, but we can easily support the other versions of Linux. We used OpenSSL 3.0.2 [16] for data protection. For encryption, we used AES-XTS with the AES-NI instruction set [17] and a 256-bit key. For integrity checking, we used SHA-256 with the SHA extensions [18] of Intel processors. Note that the SHA extensions were supported by Intel Core and Xeon processors in 2021, although they were supported by other processor architectures in 2013.

4.1. Detection of Free Memory

To optimize the protection of memory pages that are not used in a VM, SEmigrate analyzes the memory of the VM and checks whether a page to transfer is included in a free memory region or not. The Linux kernel allocates contiguous 2^n ($n = 0, 1, \cdots, 10$) physical pages at once using the buddy system [19]. As a result, it manages free memory pages as free memory regions that consist of 2^n pages. It sets the buddy page flag in the page structure



(a) sequential access



(b) random access

Figure 7. The detection of a free memory page.

for managing the head page of a free memory region. In addition, it stores the number of pages included in the region. However, this information is not stored in the page structures of the other pages. Therefore, it is not easy to check whether a page is included in a free memory region except for the head page.

In split migration, SEmigrate efficiently checks free memory by using the fact that most of the pages are transferred sequentially in ascending order of the page frame number. When SEmigrate transfers a page, it considers its page frame number as an index and obtains the page structure from the array of the structures managed by Linux, as shown in Fig. 7(a). If that page is the head page of a free memory region, SEmigrate determines that the page is free and records that page frame number and the number of pages contained in that region. Otherwise, it checks whether the page frame number is contained in the previously recorded range for a free memory region. If the number is within that range, SEmigrate determines that the page is free.

This method can be used only if SEmigrate first transfers the entire memory of a VM in split migration. When SEmigrate retransfers updated memory pages and transfers pages required by remote paging, these page frame numbers are usually not sequential. In this case, SEmigrate checks free memory by using the fact that the number of pages contained in a free memory region is 2^n, as shown in Fig. 7(b). First, it finds the candidates of the head page of the free memory region in which a transferring page can be contained. It can easily find the candidates by masking the lower n bits of the page frame number to zeros. If the buddy page flag is set in the found page and the transferring page is contained in the range of the free memory region, SEmigrate determines that the page is free. The pseudo code is shown in Algorithm 1.

4.2. Detection of Code

To optimize the protection of memory pages that store code, SEmigrate checks whether a page to transfer is contained in the code segments of the guest OS or the processes. For this purpose, it finds the virtual memory area that contains a target page in the guest OS, as shown in Fig. 8. First,

Algorithm 1 The free-memory detection algorithm on random access.

```
Input: page frame number (pfn)
1: n = 10
2: while (n > 0) do
3:     page = pfn_to_page(pfn & ~($2^n-1$))
4:     if (page_buddy(page) and freemem_size(page) $\geq 2^n$) then
5:         return true
6:     end if
7:     n = n$-$1
8: end while
9: return false
```

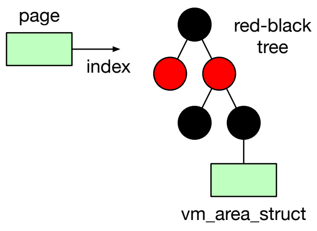


Figure 8. The search for a virtual memory area containing a target page.

it searches for the page structure corresponding to the page frame number, as in Section 4.1. From that page structure, it obtains the index in the red-black tree used for managing a virtual address space. Using this index, it traverses the red-black tree and finds the vm_area_struct structure used for managing the target virtual memory area. It examines the attribute of the virtual memory area and determines that the page contains code if the area is executable.

4.3. Detection of Process Memory

To optimize the memory protection of processes that do not deal with sensitive information, SEmigrate checks whether a page to transfer is contained in the memory of the specified processes. For this purpose, it finds a process that owns a target page. First, it finds a virtual memory area containing the page by searching for the corresponding red-black tree, as in Section 4.2. Next, it finds the process that owns the found virtual memory area and obtains the process name from the task_struct structure. If the process name matches one of the specified names, SEmigrate determines that the page is part of the process memory.

4.4. Detection of Application Data

To optimize the protection of insensitive memory regions in specific applications, SEmigrate checks whether a page to transfer is contained within the specified range of virtual memory addresses in the specified process, as shown in Fig. 9. First, it examines whether the process that owns the target page has one of the specified names, as in Section 4.3. If the process name matches the specified one, SEmigrate

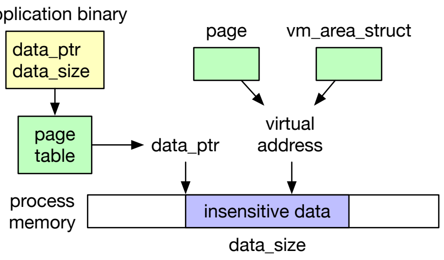


Figure 9. The detection of insensitive data in an application.

calculates the virtual address assigned to the target page from the address range of the target virtual memory area and the index in the red-black tree for the virtual address space of the process. If the obtained virtual address is within the specified range, SEmigrate determines that the page contains specific application data.

SEmigrate identifies the specified range of virtual addresses by analyzing the process memory. For example, suppose that the pointer to a target memory region and its size are stored in global variables in an application. SEmigrate obtains the addresses of the global variables from the binary file of the application. The Linux kernel assigns virtual addresses to the global variables at runtime by adding a fixed offset to these variable addresses. This offset can be changed by address space layout randomization (ASLR), but SEmigrate can obtain the value from the guest OS. Then, SEmigrate obtains the page tables for the process that owns the target page via the task_struct structure and translates the virtual addresses into physical ones. It accesses the memory of the target VM using the physical addresses and obtains the values stored in the global variables.

4.5. VMI for Obtaining Information in VMs

It is not desirable to explicitly communicate with the guest OS in a VM when SEmigrate obtains information inside the VM. This is because such communication requires modifications to the guest OS. Therefore, SEmigrate analyzes the memory of a VM and obtains necessary information. To make this analysis easier, it uses the LLView framework [20], which enables the user to obtain OS data by reusing the source code of the Linux kernel. LLView transforms analysis programs at compile time to seamlessly access the memory of a VM by translating the virtual addresses of OS data.

As shown in Fig. 10, SEmigrate first translates a virtual address into a physical one using the page tables of the init process in a VM. Then, it translates this physical address into the memory addresses in the QEMU-KVM process, which runs the VM. At this time, it also considers memory re-mapping by virtual hardware. In a VM, the physical addresses of 3 to 4 GB are often used for PCI memory-mapped areas. Therefore, the physical memory exceeding 3 GB is re-mapped to the physical addresses over 4 GB. QEMU-KVM contiguously assigns memory to a VM, but

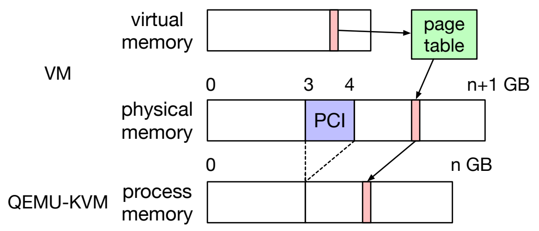


Figure 10. Address translation for VMI.

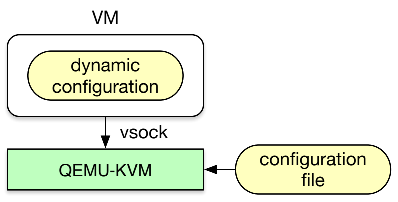


Figure 11. Two types of configurations for specifying optimization.

memory-remapping changes this memory layout. To bridge such a gap, SEmigrate subtracts 1 GB from a physical address when the address is over 4 GB and accesses the memory of a VM in QEMU-KVM.

4.6. Configuration of Selectively Protected Data

For free memory and code segments, the users need to specify nothing because SEmigrate automatically detects the memory regions from the memory attributes. When they specify a process, they need to write a process name in the configuration file. When the users specify application data, they write a process name and the addresses of two global variables that store the address and size of a memory region in the configuration file. It is our future work to support the other types of application data, e.g., a linked list. QEMU-KVM reads this configuration file at the boot time of a VM.

If the users specify a process using a process ID, which is dynamically assigned at runtime, they can configure selectively protected data inside the VM. This method is useful when the users want to specify some of the multiple processes with the same name. The users connect to QEMU-KVM using a VM socket (vsock) and send dynamic configuration, as illustrated in Fig. 11. Vsock is a mechanism for communication between a VM and the host and can be used in a manner similar to network sockets.

5. Experiments

We conducted several experiments to examine performance improvement by SEmigrate in data-protected split migration and remote paging. For a source host, a destination main host, and a destination sub-host, we used three PCs with an Intel Core i7-12700 processor and 128 GB of memory and ran Linux 5.15.60. The source host was equipped with a ConnectX-5 network adaptor, and the

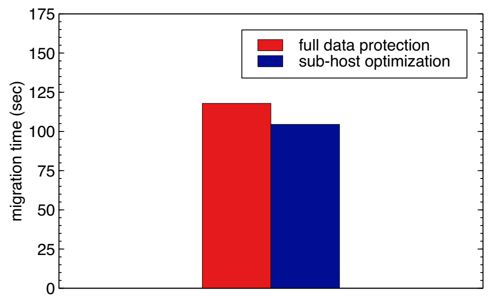


Figure 12. The migration time with sub-host optimization.

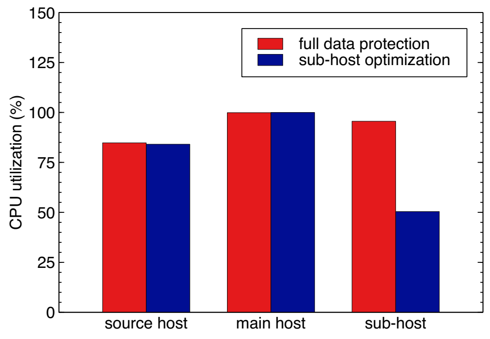


Figure 13. The CPU utilization during split migration with sub-host optimization.

destination hosts were equipped with a ConnectX-4 network adaptor.

Since we did not have a 100 GbE switch, we directly connected the source host and the main host using a 100 GbE cable. Also, we directly connected the source host and the sub-host using a 100 GbE cable. Then, we indirectly connected the main host and the sub-host by using the source host as a network bridge. When we used the iperf benchmark, the network bandwidth was 27 Gbps between the source host and the main host. In contrast, the bandwidth between the main host and the sub-host was 19 Gbps. The reason why these were much less than 100 Gbps is hardware limitations.

We created a VM with one virtual CPU and 96 GB of memory and ran Linux 4.18.17. Upon split migration, we equally divided the memory of the VM into two.

5.1. Sub-host Optimization

We first examined performance improvement by SEmigrate when we only optimized the data protection at sub-hosts. As shown in Fig. 12, the time needed for split migration was reduced by 11%. This is because SEmigrate neither decrypted the memory data received at the sub-host nor checked its integrity. Fig. 13 shows the average CPU utilization at the three hosts during split migration. The CPU utilization was decreased by 45% point at the sub-

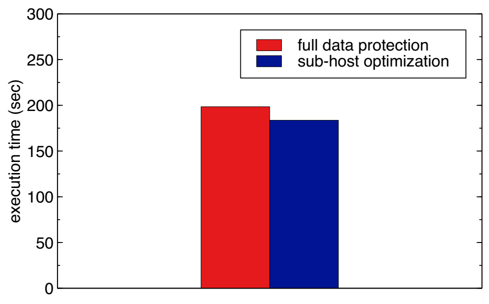


Figure 14. The benchmark time with sub-host optimization.

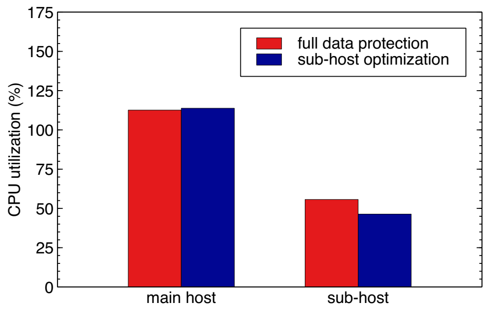


Figure 15. The CPU utilization during benchmark execution with sub-host optimization.

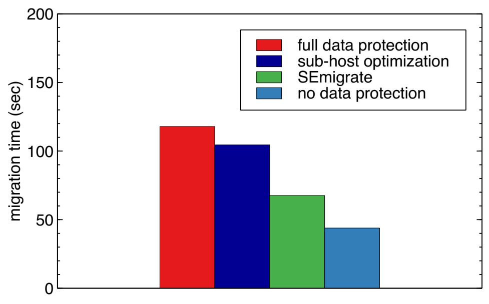


Figure 16. The migration time with SEmigrate.

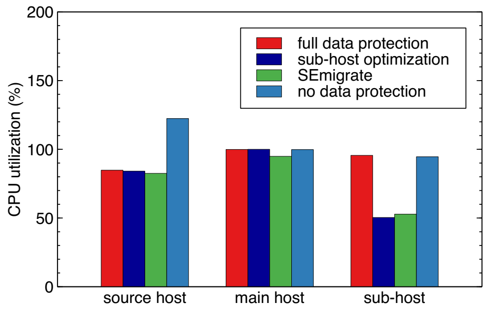


Figure 17. The CPU utilization during split migration with SEmigrate.

host, thanks to no decryption, no re-encryption, and no MAC calculation.

Next, we executed a benchmark that accessed 50 GB of memory in a VM after split migration. The execution time of this benchmark is shown in Fig. 14. Since the sub-host did not decrypt or re-encrypt memory data and did not calculate or transfer its MAC, the benchmark execution became 7% faster. Fig. 15 shows the average CPU utilization during this benchmark execution. Compared with when performing encryption and integrity checking at the sub-host, SEmigrate could decrease the CPU utilization at the sub-host by 9% point.

5.2. Selective Data Protection

We examined performance improvement by SEmigrate when we applied selective data protection as well as sub-host optimization.

5.2.1. Split Migration. We ran an application that used 50 GB of memory in a VM and configured that the memory of this process was not encrypted but its integrity was checked. This VM contained 42 GB of free memory, which was not encrypted or integrity-checked. Fig. 16 shows the time taken for split migration. SEmigrate was 43% faster than split migration with full data protection. Even compared with split migration with sub-host optimization, the migration time was reduced by 35%. SEmigrate took more time than split migration with no data protection, but the migration time was only 1.5x. This means that selective data protection is effective when a VM has much data that does not need to be protected.

Fig. 17 shows the CPU utilization during split migration. Compared with split migration with full data protection, SEmigrate could decrease the CPU utilization at the sub-host by 43% point. This is due to sub-host optimization, as shown in Section 5.1. In contrast, the CPU utilization was not reduced at the source host. We expected that the CPU utilization at the source host was also reduced because the source host neither encrypted 92 GB of memory data nor calculated MAC for 42 GB of memory data. The reason is that the source host could transfer more memory data per unit time by less data protection. In fact, the migration time was reduced. For a similar reason, the CPU utilization was high during split migration with no data protection.

To examine the breakdown of the performance improvement by selective data protection, we measured the migration time when we applied each optimization one by one. As shown in Fig. 18, the optimization of free memory reduced the migration time by 35%. This means that migration performance was largely improved by not protecting 42 GB of free memory. In contrast, the optimization of process memory reduced the migration time by 11%. This means that performance improvement by not encrypting 50 GB of process memory was not large. This suggests that the optimization of integrity checking is more effective even if relatively new SHA extensions are used. In addition, we examined the effectiveness of the optimization of application data. The migration time was rather 4% longer, compared

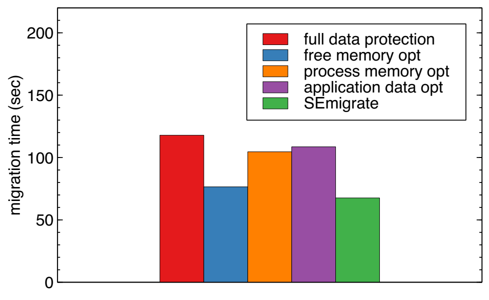


Figure 18. The migration time with each selective data protection.

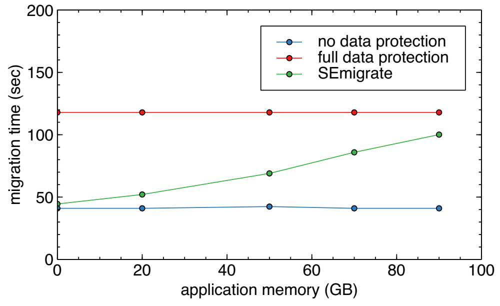


Figure 19. The migration time with various amounts of application memory.

with the optimization of the entire process memory due to the overhead of detecting application data.

Next, we measured the migration time when we changed the amount of memory used by the application. As shown in Fig. 19, migration performance was improved more largely as the application used less memory, i.e., there is more free memory. For example, SEmigrate was 63% faster than split migration with full data protection when the application almost did not use memory. In contrast, the migration time was reduced only by 15% when the application used 90 GB of memory.

5.2.2. Remote Paging. We measured the time needed to execute the memory benchmark used in Section 5.1 after split migration. As shown in Fig. 20, SEmigrate was 19% faster than the benchmark execution using remote paging with full data protection, thanks to selective data protection in remote paging. It was 13% faster than only sub-host optimization. Even compared with remote paging with no data protection, the execution time increased only by 28%. Fig. 21 shows the average CPU utilization during the benchmark execution. The CPU utilization was 9% lower than remote paging with full data protection at both hosts.

Fig. 22 shows the execution time when we applied optimization one by one. The optimization of the protection of free memory reduced the execution time by 15%. In contrast, the optimization of only the encryption of process memory improved the performance by 15%. Similarly, the optimization of only the encryption of application data reduced the execution time by 12%. Unlike the results in split

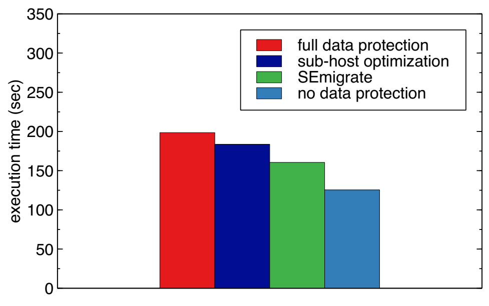


Figure 20. The benchmark time with SEmigrate.

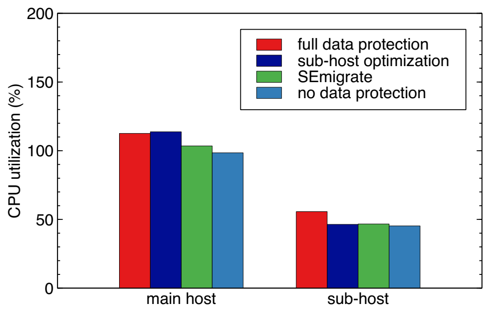


Figure 21. The CPU utilization during benchmark execution with SEmigrate.

migration, the optimization of encryption is more effective than that of integrity checking. This probably comes from the differences in memory access between split migration and remote paging. For example, split migration accesses memory at a high rate, whereas remote paging occurs at a lower rate.

5.3. Analysis Time

We examined the overhead of analyzing the memory of a VM for selective data protection. We ran the application used in Section 5.2 in the VM and configured that the process memory was not encrypted. Fig. 23 shows the average analysis time per memory page. The detection time of free memory was only 27 ns, whereas it took 6.2x longer to detect process memory. When we did not encrypt only specific application data, the detection time of application data took 48 ns longer than that of process memory. When we applied all the selective data protection, the detection time was 104 ns on average. It took 2.6 seconds for the VM with 96 GB. This overhead was 3.8% of the migration time in SEmigrate. As a result, SEmigrate is effective if performance improvement is more than this.

6. Related Work

For VM migration, various optimizations using VMI have been proposed, but there is no optimization of data protection. MiG [21] obtains memory attributes from the guest

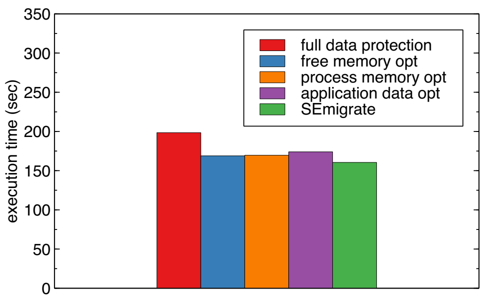


Figure 22. The benchmark time with each selective data protection.

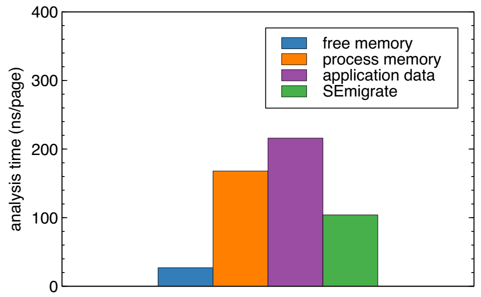


Figure 23. The analysis time per memory page.

OS in a VM and optimizes the compression algorithm for memory data using the obtained information. For example, it compresses free memory and transfers only its page frame number. It compresses the heap area using gzip because of its high redundancy. Such optimizations can reduce the amount of transferred memory by 51–61% and halve the migration time. However, MiG first saves all the states of a VM and then compresses the memory data. Therefore, it does not support live migration, which migrates a VM without stopping it, unlike SEmigrate.

IntroMigrate [22] identifies free memory in live migration of a VM and avoids transferring it to shorten the migration time. It obtains information on the entire free memory by simply scanning the array of the page structures in Linux at the beginning of VM migration. Therefore, it can apply the optimization based on old information for the memory that becomes in use during a long migration time. It relies on the retransfer mechanism to transfer the memory data that became in use. For the optimization of data protection, however, using old information is critical. If the memory including sensitive information is not encrypted, information leakage can occur.

Similar work [23] identifies memory types and avoids transferring not only free memory but also the page cache in a VM. It can identify free memory using information only in the target page structure in Linux on demand, but that information is not enough to exactly identify free memory. Since the page cache can occupy a large portion of the memory, the optimization of not transferring it can reduce the migration time largely. However, the performance of the migrated VM can degrade largely because the VM needs to access slow virtual disks, instead of the page cache.

FCtrans [24] does not transfer unused memory in a VM when the VM is migrated using split migration and runs using remote paging after the migration. It does not need to rely on VMI because it can identify unused memory only by examining the memory allocation to a VM. Unused memory in a VM becomes in use once it is accessed in a VM. Even if the memory becomes free in the VM, it does not become unused again. Therefore, FCtrans periodically identifies free memory and changes that memory to unused using VMI. This periodic reclamation imposes a large overhead.

The secure virtualization architecture [6] and VM-Crypt [7] prevent information leakage from the memory of a running VM. They provide the unencrypted version of the memory to the VM, while they provide the encrypted version to the management VM used by the administrators. Upon VM migration, they obtain and transfer the encrypted memory of a VM in the management VM. SEmigrate can apply such memory protection mechanisms to the source host and the destination main host to prevent information leakage unless the hypervisor is compromised. CloudVisor [8] can protect the memory of a VM without relying even on the hypervisor by running the security monitor under the hypervisor.

7. Conclusion

This paper proposed SEmigrate for optimizing data protection in split migration and remote paging[1]. SEmigrate avoids decrypting the memory data of a VM and checking its integrity at sub-hosts to reduce protection overhead and completely prevent information leakage. In addition, it selectively protects only necessary memory data to further reduce protection overhead. To enable this, SEmigrate analyzes the memory of the guest OS and applications in a VM using VMI. We have implemented SEmigrate in KVM and showed that SEmigrate could reduce the migration time by up to 43% in 100 GbE. Also, it could improve the performance of migrated VMs by up to 19%.

One of our future work is to examine the performance of SEmigrate using faster networks. Since the overhead of data protection becomes relatively larger if we can use the full capabilities of 100 GbE, SEmigrate could further improve the performance of split migration and remote paging. In addition, we are planning to apply SEmigrate to real applications and confirm that SEmigrate can perform selective encryption using application-specific information.

Acknowledgment

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan. These research results were partially obtained from the commissioned research (No.05501) by National Institute of Information and Communications Technology (NICT), Japan.

1. This paper is an extension of our workshop paper [3].

References

[1] Amazon Web Services, Inc., “Amazon EC2 High Memory Instances,” https://aws.amazon.com/ec2/instance-types/high-memory/, 2023.

[2] M. Suetake, T. Kashiwagi, H. Kizu, and K. Kourai, “S-memV: Split Migration of Large-memory Virtual Machines in IaaS Clouds,” in *Proc. IEEE Int. Conf. Cloud Computing*, 2018, pp. 285–293.

[3] S. Horio, K. Takahashi, K. Kourai, and L. A. Rahim, “Optimized Memory Encryption for VMs across Multiple Hosts,” in *Proc. Int. Workshop on Information Network Design*, 2021, pp. 307–315.

[4] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux Virtual Machine Monitor,” in *Proc. Ottawa Linux Symp.*, 2007, pp. 225–230.

[6] C. Li, A. Raghunathan, and N. Jha, “A Trusted Virtual Machine in an Untrusted Management Environment,” *IEEE Trans. Services Computing*, vol. 5, no. 4, pp. 472–483, 2012.

[7] H. Tadokoro, K. Kourai, and S. Chiba, “Preventing Information Leakage from Virtual Machines’ Memory in IaaS Clouds,” *IPSJ Online Transactions*, vol. 5, pp. 156–166, 2012.

[8] F. Zhang, J. Chen, H. Chen, and B. Zang, “CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization,” in *Proceedings of ACM Symposium on Operating Systems Principles*, 2011, pp. 203–216.

[9] Advanced Micro Devices, Inc., “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More,” White Paper, 2020.

[10] Intel Corporation, “Intel Trust Domain Extensions,” White Paper, 2022.

[11] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, 2018.

[12] B. Fitzpatrick, “memcached – A Distributed Memory Object Caching System,” http://memcached.org/.

[13] F. McKeena, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative Instructions and Software Model for Isolated Execution,” in *Proc. Int. Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

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

[15] F. Bellard, “QEMU,” https://www.qemu.org/.

[16] OpenSSL Project, “OpenSSL: Cryptography and SSL/TLS Toolkit,” https://www.openssl.org/.

[17] S. Gueron, “Intel Advanced Encryption Standard (AES) New Instructions Set,” White Paper, 2010.

[18] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, “Intel SHA Extensions: New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors,” White Paper, 2013.

[19] K. Knowlton, “A Fast Storage Allocator,” *Communications of the ACM*, vol. 8, no. 10, pp. 623–625, 1965.

[20] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, “Detecting System Failures with GPUs and LLVM,” in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 47–53.

[21] A. Rai, R. Ramjee, A. Anand, V. Padmanabhan, and G. Varghese, “MiG: Efficient Migration of Desktop VMs using Semantic Compression,” in *Proc. USENIX Annual Technical Conf.*, 2013, pp. 25–36.

[22] J. Chiang, H. Li, and T. Chiueh, “Introspection-based Memory De-duplication and Migration,” in *Proc. ACM Int. Conf. Virtual Execution Environments*, 2013, pp. 51–62.

[23] C. Wang, Z. Hao, L. Cui, X. Zhang, and X. Yun, “Introspection-based Memory Pruning for Live VM Migration,” *Int. J. Parallel Program*, vol. 45, no. 6, pp. 1298–1309, 2017.

[24] S. Tauchi, K. Kourai, and L. A. Rahim, “Optimizing VMs across Multiple Hosts with Transparent and Consistent Tracking of Unused Memory,” in *Proc. Int. Conf. Cloud Computing*, 2021, pp. 467–477.