Transparent IDS Offloading for Split-Memory Virtual Machines

doi: https://doi.org/10.1109/COMPSAC48688.2020.0-160

Transparent IDS Offloading for Split-memory Virtual Machines

Kouki Yamato
Kyushu Institute of Technology
yamato@ksl.ci.kyutech.ac.jp

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

Tarek Saadawi
City University of New York
saadawi@ccny.cuny.edu

***Abstract*—To enable virtual machines (VMs) with a large amount of memory to be flexibly migrated, split migration has been proposed. It divides a large-memory VM into small pieces and transfers them to multiple hosts. After the migration, the VM runs across those hosts and exchanges memory data between hosts using remote paging. For such a *split-memory VM*, however, it becomes difficult to securely run intrusion detection systems (IDS) outside the VM using a technique called IDS offloading. This paper proposes *VMemTrans* to support transparent IDS offloading for split-memory VMs. In VMemTrans, offloaded IDS can monitor a split-memory VM as if that memory were not distributed. To achieve this, VMemTrans enables IDS running in one host to transparently access VM's remote memory. To consider a trade-off, it provides two methods for obtaining memory data from remote hosts: *self paging* and *proxy paging*. We have implemented VMemTrans in KVM and compared the execution performance between the two methods.**

***Index Terms*—Intrusion detection systems, IDS offloading, virtual machines, VM introspection, VM migration**

1. Introduction

Since efficient processing of big data requires a large amount of memory, large-memory virtual machines (VMs) are being used recently. As an extreme example, Amazon EC2 provides High Memory instances with 24 TB of memory. Such large-memory VMs are used for fast in-memory databases [1], [2] and efficient big data analysis [3], [4]. One advantage of using VMs is service continuity on host maintenance by using VM migration. VM migration transfers the state of a VM including its memory to another host. For large-memory VMs, however, it becomes more difficult to find destination hosts with sufficient memory for VM migration. Since hosts with a large amount of memory are expensive, on-premise and private clouds may not be able to afford to prepare large hosts as the destination of occasional VM migration. Even in public clouds, it would not be cost-effective to always preserve many large hosts in preparation to large-scale maintenance of data centers.

To address this issue, a migration method called *split migration* has been proposed [5]. It divides a large-memory VM into small pieces and transfers them to multiple smaller hosts, i.e., one main host and several sub-hosts. After the migration, the migrated VM runs across those hosts by performing *remote paging*. VM's *remote memory* in a sub-host is paged in to the main host when necessary, while unlikely accessed *local memory* in the main host is paged out to the sub-host. However, it becomes difficult to monitor such *split-memory VMs* using *IDS offloading* with VM introspection [6]. IDS offloading runs IDS outside target VMs and securely monitors them. It has been well studied for normal VMs [6]–[9], but it cannot be applied to split-memory VMs as it is. Offloaded IDS needs to be modified so as to handle remote memory specially, but this is troublesome for the developers.

This paper proposes VMemTrans to support IDS offloading for split-memory VMs. VMemTrans offloads IDS to the main host and enables IDS to transparently access VM's remote memory existing in sub-hosts. Using VMemTrans, offloaded IDS can monitor a split-memory VM as if that memory were not distributed across multiple hosts. To obtain data in VM's remote memory, VMemTrans provides two methods: *self paging* and *proxy paging*. When using self paging, the VMemTrans runtime itself obtains necessary memory data from sub-hosts. For proxy paging, it lets a VM itself perform remote paging to do that. To offload legacy IDS from split-memory VMs without modification, we integrated VMemTrans with Transcall [10]. Transcall provides the shadow proc filesystem so that offloaded IDS can obtain the internal state of the target VM through the standard interface of the proc filesystem.

We have implemented VMemTrans in KVM [11] supporting split migration and remote paging. VMemTrans enables offloaded IDS to share the memory of a split-memory VM by mapping a *memory file*. To handle VM's distributed memory, VMemTrans uses a special file called a *sparse file* as a memory file. When offloaded IDS accesses memory pages that do not exist in the main host and a page fault occurs, the VMemTrans runtime traps that fault using the userfaultfd mechanism in Linux. When using self paging, it directly adds memory data obtained from a sub-host to the memory mapping. For proxy paging, it instead sends a control command to QEMU-KVM [12] running a VM. Our experimental results showed that VMemTrans enabled existing chkrootkit [13] to be offloaded from a split-memory VM. The performance of offloaded chkrootkit was better for self paging, while the impact of pages-ins by offloaded IDS

was smaller for proxy paging.

The organization of this paper is as follows. Section 6 mentions related work and Section 2 describes split-memory VMs and IDS offloading. Section 3 proposes VMemTrans to support IDS offloading for split-memory VMs and Section 4 explains its implementation. Section 5 reports the results of our experiments. Section 7 concludes this paper.

2. Background

To enable large-memory VMs to be migrated flexibly, a migration method called *split migration* has been proposed [5]. It divides the memory of a large-memory VM into small pieces and transfers them to multiple smaller hosts, i.e., one main host and several sub-hosts. Then, it transfers the memory likely to be accessed after the migration to the main host. In contrast, it transfers unlikely accessed memory to one of the sub-hosts. Such access prediction of VM's memory is performed using the least recently used (LRU) algorithm on the basis of the memory access history of the VM at the source host. Since split migration transfers the VM core such as virtual CPUs and devices to the main host, memory splitting is done in this way to improve the performance of a migrated VM.

After split migration, the migrated VM runs across multiple hosts. This VM is called a *split-memory VM*. Since the memory of the VM is distributed, a split-memory VM runs by performing *remote paging* between the main host and one of the sub-hosts. When the VM requires *remote memory* existing in a sub-host, that memory is paged in from the sub-host to the main host via the network. In exchange, the least likely accessed *local memory* is paged out from the main host to that sub-host. Such memory is also selected on the basis of LRU. Since likely accessed memory has been transferred to the main host in advance at the migration time, the frequency of remote paging is suppressed just after split migration.

However, a split-memory VM raises one new issue on using a technique called *IDS offloading* [6]–[9]. This technique enables IDS to securely run outside its target VMs. Even if attackers intrude into a target VM, they cannot disable offloaded IDS. The enabling technology for monitoring the system inside a VM from the outside is *VM introspection* [6]. Specifically, memory introspection analyzes the memory of a VM to obtain the internal state of the target system. For example, IDS can obtain the process list and detect malicious processes. Disk introspection analyzes the filesystem used in a virtual disk to access files and directories. IDS can find hidden files and examine the file contents. Network introspection analyzes captured packets to detect attacks from the outside of a VM.

Since the memory of a split-memory VM is distributed across multiple hosts unlike a normal VM running in one host, memory introspection cannot be applied in a traditional form. If IDS is offloaded to the main host, it cannot directly access remote memory existing in sub-hosts as local memory. Even if it is offloaded to one of the sub-hosts, it cannot seamlessly access the memory in the main host or the other sub-hosts. Remote paging is automatically performed only when a split-memory VM itself accesses remote memory, but it does not work when offloaded IDS accesses VM's remote memory. Consequently, it is necessary that IDS itself obtains remote memory. This is a troublesome task for the developers of offloaded IDS.

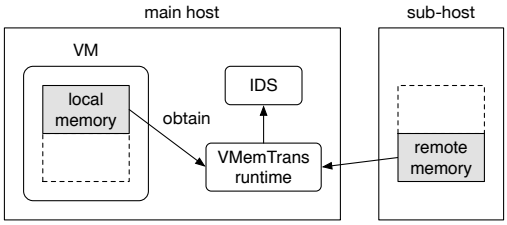


Figure 1: IDS offloading for split-memory VMs.

3. VMemTrans

This paper proposes *VMemTrans* to support IDS offloading for split-memory VMs. VMemTrans offloads IDS to the main host and provides a runtime system to offloaded IDS, as illustrated in Fig. 1. The VMemTrans runtime enables IDS to transparently access remote memory existing in sub-hosts. When IDS accesses VM's remote memory, the runtime automatically detects that access on behalf of the IDS. Then, it obtains data of accessed remote memory from the corresponding sub-host and prepares it to that IDS. Using the VMemTrans runtime, IDS can monitor a split-memory VM as if the target were a normal VM, that is, its memory is not distributed.

To consider a trade-off, VMemTrans provides offloaded IDS with two methods for accessing VM's remote memory resident in sub-hosts. One is *self paging*, in which the VMemTrans runtime itself directly obtains memory data from sub-hosts. When the VMemTrans runtime detects access to VM's remote memory by IDS, it enables the IDS to access that memory data on its own responsibility. This method is non-intrusive to the monitored VM in that it obtains memory data without causing remote paging to the split-memory VM. Therefore, it does not affect the memory access performance of the VM. In contrast, the VMemTrans runtime cannot keep all the obtained memory data due to a limited amount of free memory in the main host. In addition, obtained memory data can become stale after a while. As a result, the VMemTrans runtime has to keep only a minimum amount of memory data. This can affect the performance of offloaded IDS.

The other method is *proxy paging*, which makes a split-memory VM itself indirectly obtain VM's remote memory resident in sub-hosts by performing remote paging. If the VMemTrans runtime detects access to VM's remote memory by IDS, it requests remote paging to the VM. The requested memory data is paged in from a sub-host to the main host and can be finally accessed by the IDS. This method enables IDS to always monitor up-to-date memory data. In addition, it is likely that memory data that IDS frequently accesses is

kept in the main host. In contrast, this method can degrade the performance of offloaded IDS due to the communication overhead between the VMemTrans runtime and the VM and the overhead of page-outs performed in exchange for page-ins. Furthermore, it is intrusive to the monitored VM and can affect the memory access performance of the VM.

Using VMemTrans, legacy IDS can be offloaded from split-memory VMs in cooperation with Transcall [10]. Transcall provides an execution environment for legacy IDS to introspect a VM without any modification. It consists of the system call emulator and the shadow filesystem. The system call emulator traps the system calls issued by IDS and obtains necessary information on the operating system from the memory of a VM. The shadow filesystem provides the same filesystem view as that in a VM. To achieve this, Transcall generates the shadow proc filesystem, which provides system information in a VM as pseudo files, as done by the proc filesystem inside the VM. VMemTrans allows transparent analysis of the memory of split-memory VMs.

4. Implementation

We have implemented VMemTrans in KVM [11]. In KVM, the QEMU-KVM process provides memory to a VM. We used QEMU-KVM extended for split migration and remote paging.

4.1. Introspection of Local and Remote Memory

To achieve memory introspection of a split-memory VM from the outside, VMemTrans stores VM's memory in a file called a *memory file*. VMemTrans maps the memory file to both QEMU-KVM and offloaded IDS. To make memory updates by a VM accessible to IDS, the shared mapping is used. Through the mapped memory, IDS can introspect VM's memory. Also, QEMU-KVM can use the mapped memory like internally allocated memory, which is traditionally used. To support remote memory of a split-memory VM, VMemTrans uses a special file called a *sparse file*, instead of a normal "dense" file, as a memory file. A sparse file can contain data only in a part of file blocks. The other empty blocks are called *holes*. This memory file contains only data of local memory existing in the main host, while it does not contain that for remote memory in sub-hosts. VMemTrans creates a memory file on the tmpfs filesystem. Since all the file data is maintained only in the page cache, the overhead of a memory-mapped file is negligible.

To detect access to remote memory by offloaded IDS, VMemTrans uses the userfaultfd mechanism in Linux. This mechanism enables user-level processes to handle page faults for registered memory regions. It is also used for the implementation of remote paging. First, the VMemTrans runtime registers the memory region to which the memory file is mapped to the userfaultfd mechanism. When IDS accesses a memory region corresponding to a hole in the memory file, a page fault occurs because that region means

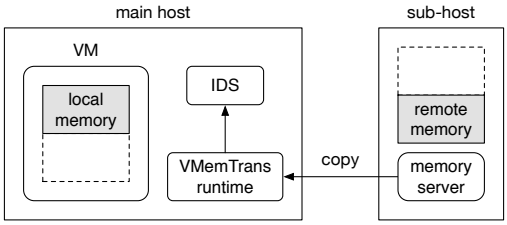


Figure 2: The procedure of self paging.

remote memory. At this time, an event is notified to the VMemTrans runtime and the thread of the IDS is suspended. The VMemTrans runtime translates the faulting address into VM's physical address. Then, it prepares memory data for the IDS using self paging or proxy paging. Finally, it resumes the suspended thread of the IDS.

4.2. Self Paging

For self paging, the VMemTrans runtime directly obtains necessary data of remote memory from sub-hosts. When it traps access to remote memory, it sends a page-ref request to the memory server in an appropriate sub-host and receives the data of the specified page, as illustrated in Fig. 2. The memory server is used for split migration and remote paging. Unlike the page-in request used for remote paging, the page-ref request does not remove the data from the sub-host. To avoid frequent page faults, the VMemTrans runtime sends multiple requests for all the pages contained in the same memory chunk at a time, as performed in remote paging.

The VMemTrans runtime temporarily adds the received memory data to the mapped memory. When the data is added to the faulting page, the suspended thread of the IDS is automatically resumed. However, if the memory file were actually modified and the change were propagated to VM's memory, the integrity could not be guaranteed because that data exists in both the main host and the sub-host. To prevent this situation, the VMemTrans runtime maps the memory file in a private copy-on-write mapping. Updates to the mapped memory in the runtime are visible only to the IDS, not to the VM. In contrast, updates in the VM are still visible to the IDS.

The VMemTrans runtime manages the temporarily added memory pages on its own responsibility. Since it does not page out unlikely accessed pages when it receives memory data from sub-hosts, it has to minimize the amount of maintained memory data to avoid memory pressure. Therefore, the runtime uses a queue to manage memory data received from sub-hosts. If it receives new memory data from a sub-host when the queue is full, the runtime removes the oldest entry from the queue and removes the mapping of all the pages in the corresponding memory chunk. Then, it adds a new entry to the queue and adds all the pages in the corresponding memory chunk to the mapping.

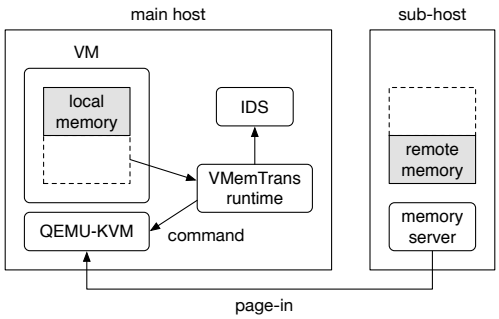


Figure 3: The procedure of proxy paging.

4.3. Proxy Paging

For proxy paging, VMemTrans makes QEMU-KVM indirectly obtain necessary data of remote memory from sub-hosts by using remote paging, as illustrated in Fig. 3. As a result, offloaded IDS can also access the pages that are paged in to the main host, which are shared with the VM. When IDS accesses remote memory and a page fault occurs, that fault is trapped by the runtime using userfaultfd. Then, the runtime sends a remote-paging command to QEMU-KVM using the QEMU monitor protocol (QMP). After QEMU-KVM pages in a memory chunk including the requested page, the runtime receives a notification for completing remote paging. Unlike self paging, the suspended thread of the IDS is not resumed automatically because the runtime does not add the page to the mapped memory using userfaultfd. To explicitly make the IDS continue, the runtime wakes up the suspended IDS using userfaultfd.

When QEMU-KVM receives the command, it can perform remote paging in two methods: *direct* and *fault*. There is a trade-off between these two methods. Using the direct method, the command handler in QEMU-KVM directly executes the function for remote paging. The function pages in multiple pages including the requested page at once. Unlike normal remote paging, the command handler has to synchronously perform page-ins of all the pages and the following page-outs. This is because it is not easy to return the result before remote paging of one memory chunk is not completed.

On the other hand, the fault method causes a page fault by accessing the requested page again in QEMU-KVM. This is almost the same as a page fault caused by a VM itself. An advantage of this method is that the command handler can continue immediately after the faulting page is paged in. The command handler does not need to wait for the other pages in the same chunk to be paged in and the following page-outs. However, this method suffers from the overhead of an extra page fault in QEMU-KVM.

For page-outs, QEMU-KVM selects the least likely accessed memory chunk on the basis of LRU and sends memory data in the chunk to the sub-host to which the page-ins are performed. At this time, it removes the mapping of the corresponding pages. In addition, it also removes the data from the memory file using the fallocate system call and makes the file blocks holes. As a result, page faults occur again when IDS accesses the removed pages later.

5. Experiments

We conducted several experiments to show the effectiveness of VMemTrans. For the main host and a sub-host, we used two PCs with an Intel Xeon E3-1225 v5 processor, 8 GB of memory, 1 TB of HDD, and a Gigabit Ethernet. We ran modified Linux 4.11 and QEMU-KVM 2.4.1 modified for split migration, remote paging, and VMemTrans. These two PCs were connected with a Gigabit switch. For a VM, we assigned one virtual CPU and 2 GB of memory and divided the memory into 1 GB each. We ran Linux 3.13 in the VM because Transcall supported this version of Linux kernel.

5.1. Execution Test of Offloaded IDS

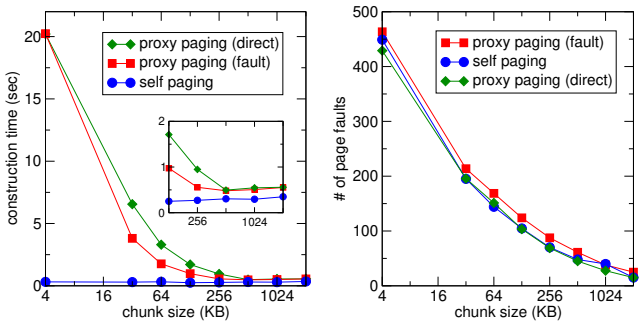
To test the behavior of IDS offloaded from a split-memory VM, we offloaded chkrootkit [13] to the main host using VMemTrans and Transcall. Chkrootkit examines the system state and several files to detect rootkits that are installed in the system. Offloaded chkrootkit accesses the shadow proc filesystem for obtaining the system state and the shadow filesystem for inspecting filesystems. Compared with traditional IDS offloading for a normal VM, we could obtain the same result for a split-memory VM.

5.2. Performance of VMemTrans

We measured the time needed for constructing the shadow proc filesystem for a split-memory VM. To obtain memory data from a sub-host, we compared the performance between self paging and proxy paging. For proxy paging, we also compared the performance between the direct and fault methods. In this experiment, we changed the size of a memory chunk from 4 KB to 2 MB, which is one to 512 pages.

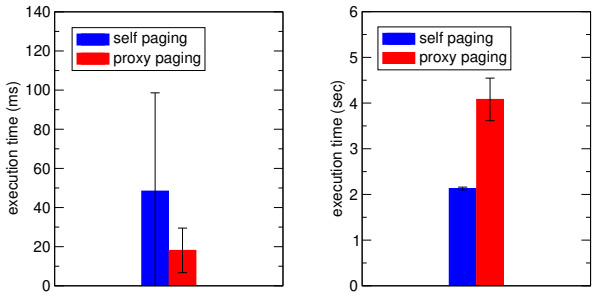
Fig. 4(a) shows the average construction time and includes the magnification of only the data for the chunk size that is equal to or larger than 128 KB. For self paging, the construction time almost did not depend on the chunk size. For proxy paging, in contrast, the time was largely decreasing as the chunk size was increasing. When we compared the construction time between self paging and proxy paging, it was shown that self paging was always faster than proxy paging. Even for the chunk size of 2 MB, self paging was 58% faster. As the chunk size became smaller, proxy paging took a longer time. It was 64 times slower when the chunk size was 4 KB. This is because more page faults occurred, as shown in Fig. 4(b).

For proxy paging, the construction time in the fault method was shorter than that in the direct one between chunk sizes of 32 KB and 256 KB. At maximum, the fault method was 87% faster. This is because the fault method did not need to wait for page-ins of all the pages contained in



(a) Construction time (b) Page faults

Figure 4: The performance of constructing the shadow proc filesystem.



(a) ps (page-ins) (b) chkrootkit (page-outs)

Figure 5: The execution time of commands inside a VM after paging by offloaded IDS.

a memory chunk and page-outs. The execution of Transcall could continue more quickly after a page fault. For the chunk size of more than 256 KB, the construction time was also similar between the two methods. This reason is probably that the overhead of paging in and out large memory chunks was a dominant factor.

For the fault method, the number of page faults was larger, as shown in Fig. 4(b). Note that this data does not include extra page faults intentionally caused by the VMemTrans runtime. This is because page faults could occur for multiple pages in one memory chunk. Since the execution of Transcall continued immediately after only the faulting page was paged in, Transcall could access the rest of the pages in the same memory chunk while QEMU-KVM was still paging in those pages. In contrast, the direct method caused only one page fault for the same memory chunk.

Next, we compared the construction time for a split-memory VM using VMemTrans with that for a normal VM using traditional IDS offloading. When the chunk size was 1 MB, the traditional IDS offloading was faster than any methods in VMemTrans. The construction time in the traditional IDS offloading was 35% shorter. Note that the execution time of chkrootkit itself was 8.1 seconds and therefore this difference of the construction time was almost negligible.

5.3. Impact of Page-ins by Offloaded IDS

We examined the impact of pages-ins caused by offloaded IDS on the memory access performance of a split-memory VM. First, we constructed the shadow proc filesystem for offloaded IDS. When we used self paging, this construction did not cause any remote paging in QEMU-KVM because the VMemTrans runtime itself obtained memory data. For proxy paging, in contrast, it caused many page-ins for the VM. Then, we measured the execution time of the ps command, which accessed the proc filesystem, inside the VM. In this experiment, we used the fault method for proxy paging because this method and the direct method resulted in almost the same memory layout. For paging, we configured the size of a memory chunk to 1 MB.

Fig. 5(a) shows the average execution time of the ps command and its standard deviation. The execution time in proxy paging was 63% shorter than that in self paging. This is because the memory data used for constructing the shadow proc filesystem tended to be stored in the main host. This led to less frequent remote paging during the command execution. When we used self paging, the variance was very large. The reason is that the number of page faults depended only on VM's memory layout just after split migration in self paging.

5.4. Impact of Page-outs by Offloaded IDS

We examined the impact of page-outs caused by offloaded IDS on the memory access performance of a split-memory VM. First, we executed chkrootkit inside the VM and caused necessary page-ins. Next, we executed a memory-intensive IDS that accessed a large amount of VM's memory outside the VM. This execution did not change the memory layout of the split-memory VM for self paging, while it caused many page-ins and page-outs for proxy paging. Then, we measured the execution time of chkrootkit inside the VM.

Fig. 5(b) shows the execution time of chkrootkit. The time in self paging was 48% shorter than that in proxy paging. For proxy paging, the memory data used by chkrootkit tended to be paged out by the memory-intensive IDS. This led to many page-ins when chkrootkit was executed after that IDS. For self paging, the memory data used by chkrootkit was kept in the main host even after the memory-intensive IDS was executed.

6. Related Work

MemX [14] runs a VM using the memory of multiple hosts. In the MemX-VM mode, the guest operating system in a VM provides a block device to access remote memory. Since that device is created inside a VM, offloaded IDS cannot access it. In the MemX-DD mode, Dom0 in Xen provides such a block device. IDS offloaded to Dom0 can access that device, but it is difficult to introspect remote memory via that device because the device is used as swap

space of the guest operating system. In the MemX-VMM mode, MemX provides a VM with the memory extension to access remote memory transparently. It may be possible that offloaded IDS accesses VM's remote memory, but that would need a considerable effort.

vNUMA [15] enables running one large VM with not only the memory but also CPUs of multiple hosts. The VM can transparently access the memory of all the hosts using distributed shared memory (DSM). This is similar to proxy paging in VMemTrans. According to our experiments, proxy paging was slower than self paging although there are several trade-offs. Using DSM cannot allow us to consider such trade-offs. Unlike a split-memory VM, active memory is distributed across multiple hosts because all the hosts run virtual CPUs in vNUMA. Therefore, IDS offloading can affect the system performance more largely.

RemoteTrans [16] enables IDS to be offloaded to remote hosts, instead of the host running a target VM. Offloaded IDS communicates with the hypervisor running the target VM to obtain memory data. Memory introspection in RemoteTrans is similar to self paging in VMemTrans. The RemoteTrans runtime running in remote hosts always obtains necessary memory data from the target VM via the Internet. Therefore, the overhead of accessing VM's memory is quite large.

LibVMI [17] is an open source library for VM introspection and supports Xen and KVM. If LibVMI is used to a split-memory VM, QEMU-KVM would automatically perform remote paging when the target memory does not exist in the main host. This is similar to proxy paging in VMemTrans. However, the performance of memory introspection is low in LibVMI because memory data have to be transferred from QEMU-KVM to IDS using inter-process communication.

KVMonitor [18] is another system for VM introspection in KVM. It uses a memory file to share VM's memory with IDS and maps it to both a target VM and IDS. Therefore, its memory introspection is much more efficient than LibVMI. This mechanism of memory sharing is similar to that of our VMemTrans. However, KVMonitor is not applicable to split-memory VMs because it cannot handle remote memory unlike LibVMI. If IDS accesses a memory region corresponding to remote memory, any page faults do not occur and consequently IDS would read empty memory data from the memory file.

7. Conclusion

This paper proposed VMemTrans to support IDS offloading for split-memory VMs. VMemTrans offloads IDS to the main host and its runtime enables IDS to transparently access remote memory existing in sub-hosts. Using VMemTrans, IDS can monitor a split-memory VM as if the memory were not distributed. We have implemented VMemTrans in KVM supporting split migration and remote paging. Our experimental results showed that VMemTrans enabled chkrootkit to be offloaded from a split-memory VM successfully. The performance of chkrootkit was better when using self paging, while the impact of pages-ins by offloaded IDS was smaller for proxy paging.

One of our future work is to offload IDS to not only the main host but also one of the sub-hosts or the other hosts. To run offloaded IDS in the other hosts, it is necessary to access VM's memory efficiently from remote hosts. Another direction is to continue the monitoring by offloaded IDS after split-memory VMs are migrated. To seamlessly monitor migrated VMs, it is necessary to virtualize access to the memory file and switch the file between the source and destination hosts of VM migration.

Acknowledgment

The research results have been achieved by the "Resilient Edge Cloud Designed Network (19304)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

References

[1] SAP SE. SAP HANA. https://www.sap.com/products/hana.html.
[2] Microsoft Corporation. SQL Server 2017 on Windows and Linux.
[3] Apache Software Foundation. Apache Spark – Unified Analytics Engine for Big Data. http://spark.apache.org/.
[4] Facebook, Inc. Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io/.
[5] M. Suetake, T. Kashiwagi, H. Kizu, and K. Kourai. S-memV: Split Migration of Large-memory Virtual Machines in IaaS Clouds. In *Proc. Int. Conf. Cloud Computing*, pages 285–293, 2018.
[6] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symp.*, pages 191–206, 2003.
[7] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction. In *Proc. Conf. Computer and Communications Security*, pages 128–138, 2007.
[8] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. Symp. Security and Privacy*, pages 297–312, 2011.
[9] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proc. Symp. Security and Privacy*, pages 586–600, 2012.
[10] T. Iida and K. Kourai. Transcall. http://www.ksl.ci.kyutech.ac.jp/oss/transcall/.
[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proc. Ottawa Linux Symp.*, pages 225–230, 2007.
[12] F. Bellard. QEMU: the FAST! Processor Emulator. https://www.qemu.org/.
[13] N. Murilo and K. Steding-Jessen. chkrootkit – Locally Checks for Signs of a Rootkit. http://chkrootkit.org/.
[14] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan. MemX: Virtualization of Cluster-Wide Memory. In *Proc. Int. Conf. Parallel Processing*, pages 663–672, 2010.
[15] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-Memory-Multi Processor. In *Proc. USENIX Annual Technical Conf.*, 2009.
[16] K. Kourai and K. Juda. Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds. In *Proc. Int. Conf. Cloud Computing*, pages 43–50, 2016.
[17] B. Payne. LibVMI: Simplified Virtual Machine Introspection. http://libvmi.com/.
[18] K. Kourai and K. Nakamura. Efficient VM Introspection in KVM and Performance Comparison with Xen. In *Proc. Pacific Rim Int. Symp. Dependable Computing*, pages 192–202, 2014.