Secure Offloading of User-level IDS with VM-compatible OS Emulation Layers for Intel SGX

Takumi Kawamura
Kyushu Institute of Technology
takumi24@ksl.ci.kyutech.ac.jp

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

***Abstract*—Since virtual machines (VMs) provided by Infrastructure-as-a-Service clouds often suffer from attacks, they need to be monitored using intrusion detection systems (IDS). For secure execution of host-based IDS (HIDS), IDS offloading is used to run IDS outside target VMs, but offloaded IDS can still be attacked. To address this issue, secure IDS offloading using Intel SGX has been proposed. However, IDS development requires *kernel-level* programming, which is difficult for most IDS developers. This paper proposes *SCwatcher* for enabling *user-level* HIDS running on top of the operating system (OS) to be securely offloaded using VM-compatible OS emulation layers for SGX. SCwatcher provides the standard OS interface used in a target VM to in-enclave IDS. Especially, the virtual proc filesystem called *vProcFS* analyzes OS data using VM introspection and returns the system information inside the target VM. We have implemented SCwatcher using Xen supporting SGX virtualization and two types of OS emulation layers for SGX called SCONE and Occlum. Then, we confirmed that SCwatcher could offload legacy HIDS and showed that the performance could be comparable to insecure IDS offloading.**

***Index Terms*—virtual machines, Intel SGX, VM introspection, intrusion detection systems, proc filesystem**

I. Introduction

Infrastructure-as-a-Service clouds provide virtual machines (VMs) to users. Since clouds are connected to the Internet, such VMs tend to suffer from various attacks. Therefore, it becomes important to monitor VMs using intrusion detection systems (IDS). Since host-based IDS (HIDS) has to be executed inside target VMs to obtain system information, it can be disabled if attackers intrude into VMs. To protect HIDS, IDS offloading using VM introspection (VMI) [1] is used to execute IDS outside target VMs. This technique can prevent IDS from being disabled even if attackers intrude into VMs because IDS does not exist in the VMs. However, offloaded IDS could be compromised by external attackers to clouds. If insiders exist in clouds, they could easily attack offloaded IDS [2]–[4].

To address this issue, SGmonitor [5] has been proposed to securely offload HIDS using Intel SGX. SGX is a security feature of Intel processors and enables IDS to securely run in a protection domain called an *enclave*. However, it is not easy to develop IDS in SGmonitor. Since offloaded IDS needs to analyze the data structures of the operating system (OS) in the memory of target VMs, SGmonitor requires kernel-level programming. This is much more difficult than application-level programming for IDS running on top of the OS. IDS developers need to develop IDS for each OS version because kernel-level programs are largely affected by changes in the OS. Moreover, it is necessary to use the library dedicated to SGX. This makes legacy IDS difficult to run.

This paper proposes *SCwatcher* for enabling *user-level* HIDS to be offloaded into enclaves using VM-compatible OS emulation layers for SGX. SCwatcher provides the standard OS interface such as the standard C library used in a target VM to in-enclave IDS. Especially, the virtual proc filesystem called *vProcFS* returns the system information on a target VM. It obtains memory data of the target VM via the underlying hypervisor, analyzes OS data in it, and creates pseudo files. IDS can obtain system information inside the VM by accessing these pseudo files. As such, even legacy HIDS can be offloaded because it can use exactly the same OS interface as in the VM. Moreover, IDS developers can develop new HIDS like traditional one.

We have implemented SCwatcher using Xen-SGX 4.7 [6], which supports SGX virtualization. Many OS emulation layers for SGX have been proposed [7]–[11], but there are various trade-offs between them, particularly in terms of performance and security. Among them, SCwatcher supports two OS emulation layers: SCONE [8] and Occlum [11]. For SCwatcher with SCONE, we have implemented vProcFS independently of SCONE because SCONE does not provide the library OS or any filesystems. To invoke the hypervisor in closed-source SCONE, SCwatcher leverages the system-call interface provided by SCONE. It also transforms IDS programs at compile time to enable IDS to invoke the additionally linked vProcFS. For SCwatcher with Occlum, we have extended the proc filesystem embedded into the Occlum library OS so that it returns system information inside a target VM. This vProcFS invokes the outside runtime using the interface provided by SGX and then the hypervisor to obtain memory data of a target VM.

We executed legacy HIDS called chkrootkit [12], which consisted of shell scripts and external commands, using SCwatcher. Then, we confirmed that this IDS could monitor networks and processes in a target VM. We examined the performance of this IDS and showed that the overhead of invoking many external commands was large in SCwatcher. Therefore, we have re-implemented this IDS using Python and C so that IDS did not invoke external commands. As a result, the overhead of IDS written in C was 6-39% and SCwatcher could achieve performance comparable to traditional insecure IDS offloading.

The organization of this paper is as follows. Section II describes issues of existing IDS offloading in clouds. Section III proposes SCwatcher for securely offloading user-level HIDS using SGX. Section IV explains the implementation of SCwatcher using SCONE and Occlum and Section V shows experimental results. Section VI describes related work and Section VII concludes this paper.

II. IDS Offloading in Clouds

IDS offloading using VMI [1] is a technique for securely executing HIDS outside target VMs. Using this technique, offloaded IDS cannot be disabled even if attackers intrude into VMs because IDS does not exist inside the VMs. Unlike HIDS running inside target VMs, offloaded HIDS obtains memory data of target VMs, analyzes OS data in it, and monitors the system information. For example, HIDS can detect illegal network communication by examining network sockets used inside target VMs. It can also detect the execution of malware by checking the list of running processes.

Even if it is offloaded to the outside of target VMs, IDS can still be attacked. This is because external attackers could attack offloaded IDS running in clouds. In addition, insiders could exist in clouds running offloaded IDS. In fact, system administrators stole personal information and violated users' privacy in Google [2]. It is reported that 28% of cybercrimes were done by insiders [3] and that 35% of system administrators have eavesdropped on sensitive information [4]. If offloaded IDS is compromised, it could no longer monitor the system correctly. It could leak sensitive information obtained from target VMs.

To address this issue, SGmonitor [5] has been proposed to securely offload HIDS using Intel SGX, as illustrated in Fig. 1. SGX is a security feature of Intel processors. It enables secure program execution in a protection domain called an *enclave*. It checks the digital signature of a program at the load time to an enclave. Since it preserves the integrity of the memory of an enclave, it is impossible to tamper with the program running inside an enclave. In addition, SGX can prevent information leakage from the memory of an enclave by encrypting it. SGmonitor offloads IDS into an enclave and protects it from attacks. Offloaded IDS obtains OS data from the memory of a target VM via the hypervisor underneath the VM.

However, SGmonitor requires *kernel-level* programming to develop IDS. Offloaded IDS needs to analyze the kernel memory of a target VM and monitor the system in the VM using obtained OS data. Such IDS development is not easy for average developers because kernel-level IDS is not popular. Also, it is necessary to develop IDS for each version of the OS running in a target VM. Kernel-level programs are subject to changes in the internal data structures of the OS. In addition, IDS developers need to use the SDK dedicated to SGX. They cannot develop IDS using standard libraries.

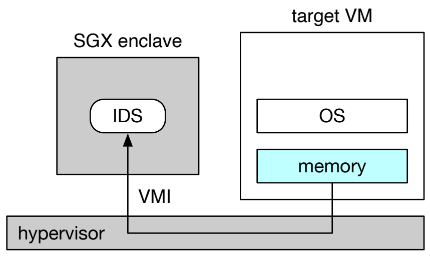


Fig. 1: The system architecture of SGmonitor.

Several systems have been proposed to run *user-level* IDS using SGX. User-level IDS is developed using standard user-level libraries and runs on top of the OS. S-NFV [13] runs only a sensitive part of user-level IDS in an enclave. As an example, it stores the states of virtual network functions of Snort [14] in an enclave. SEC-IDS [15] enables running the entire Snort in an enclave by obtaining network packets using DPDK [16]. However, these systems are for network-based IDS (NIDS) and are not applicable to HIDS. NIDS monitors only packets, while HIDS needs to monitor various system states such as networks and processes.

III. SCwatcher

A. Threat Model

This paper assumes the following threat model. First, we trust cloud providers because it is critical for the providers to lose the trust of users. This assumption is often used [17]–[19]. Consequently, we trust hardware managed by cloud providers, including processors with the SGX feature. We assume that SGX and software executing in its enclaves have no vulnerabilities. In addition, we trust the hypervisor managed by cloud providers. This assumption of trusting both SGX and the hypervisor is also used [20]. We can validate that the hypervisor works correctly in various methods. For example, cloud providers and their users can confirm that an unmodified hypervisor is booted by remote attestation with TPM [21]. The runtime modification to the hypervisor can be detected using hardware mechanisms such as the system management mode (SMM) [22]–[24].

On the other hand, we do not trust software except for IDS inside enclaves and the hypervisor, e.g., the host OS running IDS. We assume that external attackers or insiders in clouds attack offloaded IDS.

B. IDS Offloading with VM-compatible OS Emulation Layers

To enable *user-level* HIDS to be offloaded into SGX enclaves, SCwatcher provides the standard OS interface to in-enclave IDS. For this purpose, SCwatcher uses a *VM-compatible OS emulation layer* for SGX, which has been developed for running legacy applications inside enclaves. Thanks to this layer, IDS can use the traditional OS interface such as the standard C library. Figure 2 illustrates the system architecture of SCwatcher. An OS emulation layer consists of the library provided to IDS inside an enclave and the runtime running outside an enclave.

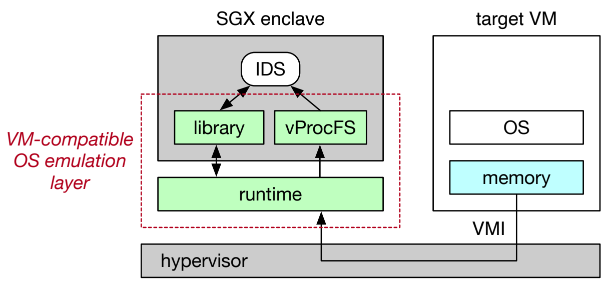


Fig. 2: The system architecture of SCwatcher.

In addition, SCwatcher provides the virtual proc filesystem for VM monitoring, which is called *vProcFS*, to in-enclave IDS. The proc filesystem is the interface for obtaining system information and is often used by user-level IDS. In SCwatcher, IDS can transparently obtain system information in a target VM by accessing pseudo files provided by vProcFS. Unlike regular files stored in persistent storage, pseudo files are special files whose data is dynamically generated in memory at access time. For example, /proc/[pid]/stat, where [pid] is a process ID, returns information on the state, memory usage, and so on of the specified process.

vProcFS obtains necessary OS data in the memory of a target VM using VMI when IDS starts to access a pseudo file. First, it invokes the hypervisor via the SCwatcher library and runtime and obtains memory data in which target OS data is contained from the VM. To prevent information leakage from the obtained memory data, the hypervisor encrypts it and the library in the enclave decrypts it. The encryption key is securely shared only between the hypervisor and the library using the public key infrastructure. Then, vProcFS analyzes OS data structures in the VM and generates the data of the pseudo file. It returns a requested part of the generated file data whenever IDS reads the pseudo file.

C. *Trade-offs between OS Emulation Layers*

Many OS emulation layers for SGX have been proposed [7]–[11], but there are various trade-offs, particularly in terms of performance and security. One of the design choices is the library OS. Some of the OS emulation layers provide the library OS inside an enclave [7], [9], [11]. The library OS enables its processes to use various OS features including process management and filesystems. It requires exposing only a small interface to the outside of an enclave, while it increases the size of the trusted computing base (TCB) inside an enclave. A larger TCB leads to more vulnerabilities.

In contrast, the other OS emulation layers provide only thin libraries, e.g., the standard C library, inside an enclave without a large library OS [8], [10]. This architecture can decrease the TCB size and avoid the extra overhead of the library OS. However, the external interface to the outside of an enclave tends to be larger, e.g., many system calls or many functions of the standard C library. This can expand the attack surface against an enclave.

Multi-process support is another design choice. It is necessary for some type of IDS to execute external commands as a helper. Some of the OS emulation layers can execute only a single process in an enclave [8]–[10]. The isolation between processes is strong because processes share nothing. However, it is necessary to create a new enclave whenever a child process is created by the fork system call and whenever a new program is executed by the execve system call. Therefore, the cost of process management is high. Multiple processes can securely run in one enclave by isolating them with software fault isolation [11]. This is an advantage in terms of performance, but the isolation between processes is weaker because each process can interfere with the others via the shared library OS.

To enable users to take trade-offs between these, SCwatcher supports two types of OS emulation layers. One is a *thin* OS emulation layer. It provides no library OS and runs only a single process in an enclave. vProcFS is provided independently inside an enclave because this OS emulation layer does not provide filesystems. This vProcFS leverages the existing external interface and obtains the memory data of a target VM to keep the interface as narrow as possible. When it accesses a pseudo device called the *VM memory device* installed in the host OS, the device invokes the hypervisor. The vProcFS is additionally linked to IDS as a library, but legacy IDS cannot use any functions of vProcFS because it is not aware of vProcFS. To glue vProcFS to IDS, SCwatcher transforms IDS programs at compile time so that IDS invokes the in-enclave *virtual filesystem (VFS)*, which is also linked to IDS. The VFS dispatches file requests to vProcFS when necessary.

The other is a *rich* OS emulation layer. It provides the library OS and runs multiple processes in an enclave. vProcFS is embedded into the library OS because the library OS includes filesystems. This vProcFS uses a newly added external interface and obtains the memory data of a target VM. This slightly expands the interface, but it can avoid the overhead of the library OS, e.g., extra data copy.

IV. Implementation

We have implemented SCwatcher using Xen-SGX 4.7 [6], which supports SGX virtualization. In Xen, IDS is usually offloaded into Dom0, which is a privileged VM, but Xen-SGX cannot create enclaves in Dom0 because Dom0 is para-virtualized. Therefore, SCwatcher creates enclaves in a VM dedicated for IDS, which is fully virtualized and called the IDS VM, and offloads IDS into them.

SCwatcher supports two largely different OS emulation layers for SGX: SCONE [8] and Occlum 0.24.0 [11]. The reasons why we chose these two are that they support fork/exec and that they can run on top of Xen-SGX 4.7. Both OS emulation layers provide the standard C library, while Occlum provides the library OS as well. Most of the Occlum library OS is written in Rust, which is the memory-safe language, and is therefore less vulnerable than one written in C. Occlum

TABLE I: Examples of pseudo files provided by vProcFS.

pseudo file	description
/proc/meminfo	statistics of memory usage
/proc/stat	kernel/system statistics
/proc/uptime	time spent since the system boot
/proc/net/{tcp,udp}	TCP/UDP socket table
/proc/[pid]/auxv	loader information passed to the process
/proc/[pid]/cmdline	command line for the process
/proc/[pid]/stat	status information about the process
/proc/[pid]/status	human-readable format of /proc/[pid]/stat
/proc/sys/kernel/osrelease	kernel version
/proc/sys/kernel/pid_max	maximum process ID
/proc/tty/drivers	list of tty drivers

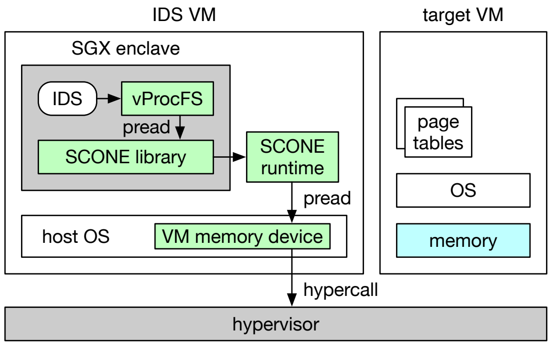


Fig. 3: The system architecture of SCwatcher/SCONE.

can run multiple processes in one enclave on top of the library OS.

A. In-enclave vProcFS

vProcFS generates the data of pseudo files by obtaining OS data in a target VM. We have implemented the function of VM monitoring in vProcFS using LLView [25]. LLView is a framework for analyzing OS data in the memory of VMs using the source code of the OS. It enables developing IDS using data structures, global variables, macros, and inline functions by including the header files of the Linux kernel. LLView compiles the source code of vProcFS and generates the LLVM intermediate representation called bitcode. Then, it transforms the bitcode so that vProcFS obtains the memory data from a target VM whenever it needs to access OS data.

Specifically, LLView inserts the function call for obtaining OS data from a target VM just before each load instruction in bitcode. The invoked function obtains memory data from the VM and stores it in the enclave memory. Then, LLView transforms bitcode so that the load instruction reads the data stored in the enclave memory. The obtained memory data is cached in the enclave. If vProcFS needs the same OS data, it can read it from the cache. It provides a special pseudo file of /proc/drop_caches as an interface for flushing the stale cache. If IDS reads this pseudo file, LLView removes the entire cache. This is necessary for long-running IDS to obtain up-to-date information.

Table I shows the pseudo files used by the netstat and ps commands, which were executed in our experiments. For example, /proc/net/tcp is the pseudo file that stores the IP addresses, port numbers, and states of TCP connections. vProcFS generates the data of this file by traversing all the entries of two hash tables for the sockets of the LISTEN and ESTABLISHED states in the Linux kernel. It obtains necessary information from the sock structures. /proc/[pid]/cmdline is the pseudo file created for the process whose ID is [pid]. vProcFS generates the data of this file by reading information on the command line, which is stored in the process memory. It obtains the memory address of the command line from the mm_struct structure.

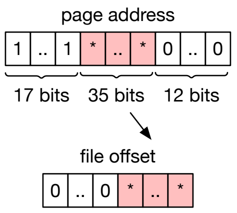
B. SCwatcher/SCONE

To build legacy IDS for SCONE, the user needs to slightly change the build procedure to use the SCONE compiler. The SCONE compiler compiles the source code as position independent code. It uses the musl library as the standard C library. Then, it links the object files as a shared library. Finally, it signs that shared library and creates a binary file that contains the signed shared library in the .enclave section. When this binary file is executed on top of the host OS, the shared library for IDS is loaded to an enclave. When this in-enclave IDS invokes other programs, it executes the corresponding binaries in the host filesystem.

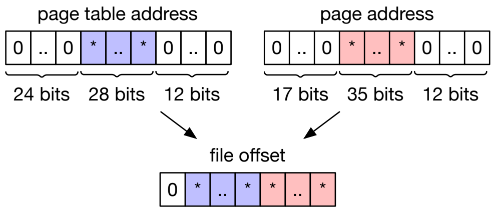
1) Accessing VM Memory Device: The VM memory device is a pseudo device installed in the host OS. It is used to obtain memory data of target VMs using closed-source SCONE. It provides the interface for the memory of VMs. The minor number of this device specifies the domain ID of a target VM. vProcFS accesses this device using the pread system call, as illustrated in Fig. 3. This system call is usually used to obtain file data from the specified offset. vProcFS uses this offset to specify a memory address in the target VM.

vProcFS first masks the lower 12 bits of the 64-bit virtual address of OS data and calculates the first address of the 4-KB page containing that OS data, which is called a page address. Then, it specifies that page address as a 64-bit file offset and executes the pread system call. However, the page address of the kernel data is too large for SCONE to accept. Therefore, vProcFS compresses the page address, as in Fig. 4(a). From the above definition of a page address, the lower 12 bits of the page address are always zero. In addition, the upper 17 bits of the kernel address are always one. Since it is not necessary to pass these fixed bits to the VM memory device, vProcFS extracts the remaining 35 bits and stores them in the lower 35 bits of a file offset. The generated file offset is within the acceptable range by setting the upper 29 bits to zero.

To obtain process data in a target VM, on the other hand, vProcFS needs to pass the address of the page tables as well as the page address of process data. It stores these two addresses in one file offset, as illustrated in Fig. 4(b). First, it compresses the page address of process data into the lower 35 bits of a file offset, as in kernel data. Note that the upper 17 bits of the process address are zero. Then, it translates the virtual address of the page tables into the physical address and uses only its lower 40 bits. This limits the memory that can be assigned



(a) Kernel data



(b) Process data

Fig. 4: The compression of address(es) into a file offset.

to a target VM to 1 TB, but this is usually sufficient. Since the lower 12 bits of the address of the page tables is always zero, vProcFS uses the remaining 28 bits as the next 28 bits of the file offset. It makes the generated file offset acceptable by setting the most significant bit to zero.

When vProcFS issues the pread system call, it invokes the VM memory device via the SCONE library and runtime. The read function of this device obtains the minor number of the accessed device and identifies the domain ID of a target VM. Then, the VM memory device extracts and decompresses the page address of OS data and the address of the page tables from the passed file offset. If the address of the page tables is zero, the VM memory device issues the hypervisor call for obtaining the kernel memory of the target VM. It uses the page tables currently used in the VM. Otherwise, it issues the hypervisor call for obtaining the process memory of the target VM using the specified page tables.

2) In-enclave VFS: When IDS invokes the standard file functions, SCwatcher/SCONE makes the IDS transparently invoke the in-enclave VFS, as illustrated in Fig. 5, instead of the SCONE library. To enable this, we have developed a tool called REPLLVM, which transforms IDS programs at compile time. It replaces the standard file functions invoked by the call instructions in bitcode with the corresponding VFS functions with the prefix of vfs_. Note that we cannot use any OS mechanisms for replacing invoked functions, e.g., LD_PRELOAD, in enclaves. The VFS provides such replaced functions and invokes vProcFS if the accessed file or directory is a pseudo one of vProcFS. It manages pseudo files using the PFILE structure, which has the buffer storing the data generated for the pseudo file, the size of the pseudo file, and the current file offset. It also manages pseudo directories using the PDIR structure, which has the list of the dirent structures.

When IDS issues the open function, the vfs_open function is invoked instead. This function first examines the file path and determines whether the opening file is a pseudo one in vProcFS or not. If it is a pseudo file, the VFS assigns the pseudo-file number consecutively from zero. It uses the assigned pseudo-file number as an index of the array of the PFILE structures. Then, it generates a file descriptor by adding the fixed value to the assigned pseudo-file number. Using a large fixed value, it avoids the conflict with file descriptors assigned to regular files.

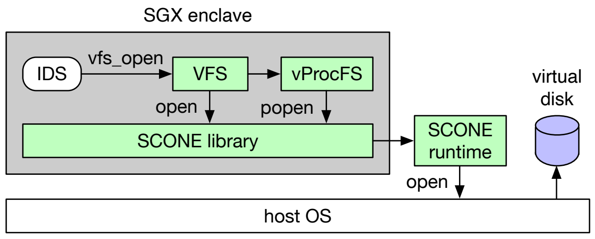


Fig. 5: The virtual filesystem in an enclave.

Next, the vfs_open function invokes the vProcFS function corresponding to the path. vProcFS generates the data of the specified pseudo file and stores it in the corresponding PFILE structure as a cache. Finally, the vfs_open function returns the value of the generated pseudo-file descriptor. For vfs_fopen function, the VFS handles a pseudo file similarly, but it returns a pseudo-file pointer instead of a file descriptor. It generates a pseudo-file pointer by adding a pseudo-file number to the fixed base address. This enables distinguishing a regular file pointer to the FILE structure.

When a VFS function with a file descriptor such as vfs_read is invoked, the VFS determines whether the specified file descriptor is of a pseudo file or not. If the value is equal to or more than the fixed value, the VFS calculates the pseudo-file number by subtracting that fixed value from the file descriptor. For a VFS function with a file pointer, it calculates the pseudo-file number similarly by using the fixed base address. Then, it searches the array of the PFILE structures using the pseudo-file number as an index. Finally, it executes the corresponding file operation to the found PFILE structure. Note that the VFS discards the cache in the PFILE structure when vfs_close is invoked.

When IDS issues the opendir function to open a pseudo directory in vProcFS, the invoked vfs_opendir function creates directory entries, which include pseudo files and sub-directories. For /proc, vProcFS traverses the list of the task_struct structures in the memory of the target VM. Then, the VFS creates sub-directories whose names are obtained process IDs. For a pseudo directory, it assigns a pseudo-directory number consecutively from zero. Then, it makes a pointer to the DIR structure by adding the assigned pseudo-directory number to the fixed base address. When a VFS function with a pointer to the DIR structure is invoked, the VFS first calculates the pseudo-directory number. Then, it executes the corresponding directory operation to the found DIR structure.

If IDS accesses a non-pseudo file or directory, the VFS accesses the virtual disk of a target VM. SCwatcher makes the

IDS VM share the virtual disk with the target VM using NFS. Then, it mounts that disk on /tmp/vm[domid], where [domid] is the domain ID of the target VM, in advance. At access time, the VFS modifies the path specified to the standard file functions. For example, the vfs_open function appends /vm[domid] to the specified absolute path and executes the standard open function with the modified path through the SCONE library. Finally, the host OS accesses the virtual disk. IDS needs to access the host filesystem as well if it invokes other binaries, but such binaries are located in the special directory in SCwatcher/SCONE. Therefore, IDS can monitor all the files in the virtual disk.

For more secure access, SCwatcher should provide filesystems such as ext4 in an enclave and handle encrypted virtual disks like SGmonitor. This is our future work, but it is not difficult that the VFS invokes the in-enclave ext4 filesystem.

C. SCwatcher/Occlum

To build legacy IDS for Occlum, the user needs to slightly change the build procedure to use the Occlum compiler. Like the SCONE compiler, the Occlum compiler also compiles the source code as position independent code using the musl library. Then, it links the object files as a shared library. In addition, the user prepares an Occlum filesystem image including that shared library. When the occlum run command is executed, it loads the Occlum library to an enclave and then that library loads the IDS from the in-enclave filesystem. When this IDS invokes other binaries, it loads them from the in-enclave filesystem as well.

Since Occlum is open-source software unlike SCONE, we have extended the proc filesystem in the Occlum library OS to implement vProcFS. Unfortunately, the original proc filesystem in Occlum provides only a small subset of pseudo files and directories. Therefore, we added necessary pseudo files and directories. Unlike SCwatcher/SCONE, SCwatcher/Occlum generates the data of a pseudo file at the first read. This is because the file open operation is executed outside vProcFS. Our design policy is to minimize the modification to the Occlum library except for its proc filesystem.

Since most of the Occlum library is written in Rust, vProcFS invokes C code to generate the data of pseudo files using LLView. That C code is transformed to issue a newly added outside call (OCALL) when memory data of a target VM is required. An OCALL is the SGX interface for securely invoking untrusted code outside an enclave from trusted code inside the enclave. SCwatcher/Occlum provides two types of OCALLs. For kernel data, vProcFS calculates the page address of its virtual address and passes it to one OCALL. For process data, vProcFS passes its page address and the address of the page tables to another OCALL. When these OCALLs invoke the extended Occlum runtime, the runtime directly issues the corresponding hypervisor calls for obtaining the specified memory data of a target VM, as illustrated in Fig. 6. It does not invoke the VM memory device in the host OS unlike SCwatcher/SCONE.

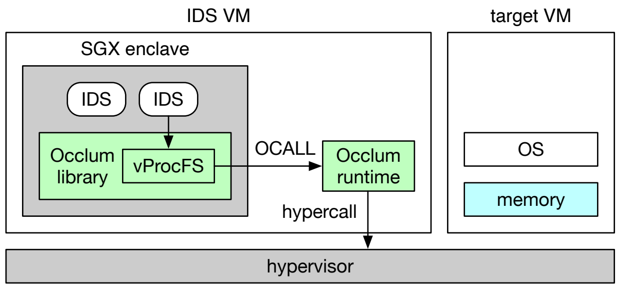


Fig. 6: The system architecture of SCwatcher/Occlum.

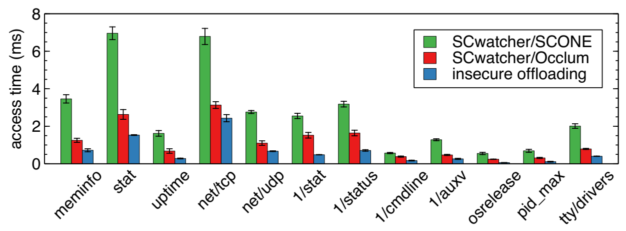
vProcFS itself preserves the generated data of pseudo files as a cache. Since it cannot recognize the file close operation, it flushes the cache after the specified period to prevent a stale cache. In addition, SCwatcher/Occlum needs to periodically flush the cache of the memory data obtained by LLView even for short-running IDS. Unlike SCONE, Occlum can run multiple processes in one enclave. If IDS is invoked many times, the LLView cache can become stale. Currently, we sometimes flush the LLView cache manually to reduce the overhead of checking cache expiration whenever memory data is obtained.

To monitor the virtual disk of a target VM, SCwatcher/Occlum merges the in-enclave root filesystem and the root filesystem in the target VM. It makes the IDS VM share the virtual disk with the target VM using NFS and mounts that disk on /host/vm in the in-enclave filesystem. If it cannot find a file in the lookup operation, it appends /host/vm to the specified path and recursively searches the root filesystem in the target VM. For this purpose, we have slightly modified the VFS in Occlum. Since the in-enclave filesystem provides only a minimum set of files, IDS can substantially monitor all the files in the virtual disk.

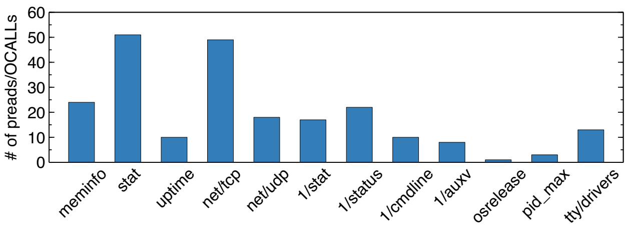
V. Experiments

We conducted several experiments to show the effectiveness of SCwatcher. We offloaded chkrootkit [12], which was one of the legacy HIDS, to enclaves. Since chkrootkit is written in shell scripts, we first executed the bash shell in an enclave and then ran chkrootkit on bash. We used SCwatcher/SCONE and SCwatcher/Occlum. For SCwatcher/Occlum, we created an enclave for IDS in advance. Since chkrootkit is usually executed for periodic intervals, we ran the shell script that executed chkrootkit periodically on bash. For comparison, we used traditional insecure IDS offloading, which ran chkrootkit without enclaves by reusing the implementation of vProcFS in SCwatcher/SCONE.

We used a PC with Intel Core i7-8700, 16 GB of memory, and a 2-TB HDD. For the IDS VM, we assigned two virtual CPUs, 2 GB of memory, and an 80-GB virtual disk. For a target VM, we assigned two virtual CPUs, 2 GB of memory, and a 50-GB virtual disk.



(a) Access time



(b) Issued preads/OCALLs

Fig. 7: The performance of accessing pseudo files of vProcFS.

A. *Secure Offloading of chkrootkit*

chkrootkit invokes many external commands, but only two commands, netstat and ps, access the proc filesystem. The netstat command obtains information on network sockets and the ps command obtains information on running processes. First, we offloaded and executed these two commands using SCwatcher. As a result, we confirmed that the outputs of these offloaded commands were almost exactly the same as those when we ran them inside a target VM.

Next, we offloaded and executed chkrootkit. We used one function for network-level malware detection and three functions for process-level malware detection in chkrootkit. Only these functions use the proc filesystem through the execution of external commands. For network-level malware detection, chkrootkit checks whether the slapper warm infects the system or not. For process-level malware detection, chkrootkit checks whether inetd, sshd, and tcpd are compromised or not. In this experiment, we made the target VM be infected by malware. According to the execution results, we confirmed that the offloaded chkrootkit could detect malware correctly.

B. *Performance of vProcFS*

We examined the performance of accessing each pseudo file of vProcFS. We measured the time needed for opening, reading, and closing pseudo files. We performed this measurement ten times and calculated the average and the standard deviation. Fig. 7(a) shows the results for pseudo files used by netstat and ps. Compared with the insecure IDS offloading, SCwatcher/SCONE took 2.8-9.8x longer. In contrast, SCwatcher/Occlum was only 1.3-4.2x slower. This access time was basically proportional to the number of issued pread system calls or OCALLs, as shown in Fig. 7(b). SCwatcher/SCONE issued the pread system calls to obtain memory data, while SCwatcher/Occlum issued OCALLs. The

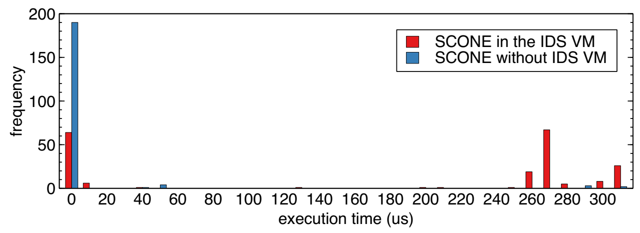


Fig. 8: The execution time of each pread system call.

execution time of the pread system calls or OCALLs occupied a large portion of the access time.

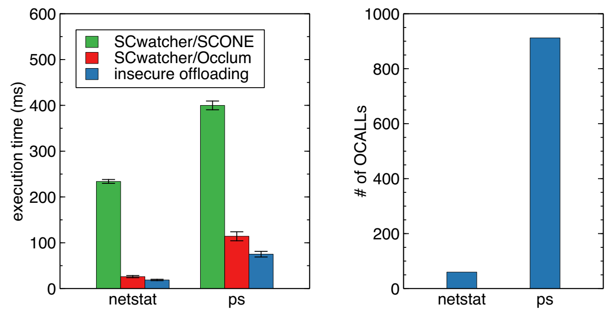
Although SCwatcher/SCONE was much slower than SCwatcher/Occlum, it should be faster to invoke system calls in SCONE thanks to asynchronous system calls [8]. To analyze the large overhead in SCwatcher/SCONE, we measured the execution time of the pread system call in an enclave inside the IDS VM 200 times. For comparison, we measured the execution time in an enclave without using the IDS VM. In both cases, the VM memory device returned dummy data of 4 KB.

In this experiment, we needed to obtain the execution time in the order of microseconds. In SCONE, a program running in an enclave can get time via the clock_gettime system call, but it is not accurate because the invocation overhead of the system call is not negligible. To get time inside an enclave as accurately as possible, we obtained time information directly from the vsyscall area in the process memory, on which part of the kernel memory was mapped. However, the granularity of the obtainable time was only 4 ms. It was necessary to execute the rdtsc instruction to adjust the obtained time, but that instruction could not be executed in an enclave. Therefore, we developed a kernel module to periodically execute rdtsc and store the result in the vsyscall area.

Fig. 8 shows the histogram of the execution time of pread. The execution time was 3-320 μs regardless of the IDS VM. However, it took about 300 μs more than 100 times when SCONE ran in the IDS VM. Without the IDS VM, it was rare to take more than 4 μs. As a result, it was clarified that virtualization in Xen increased the overhead of the pread system call in SCONE.

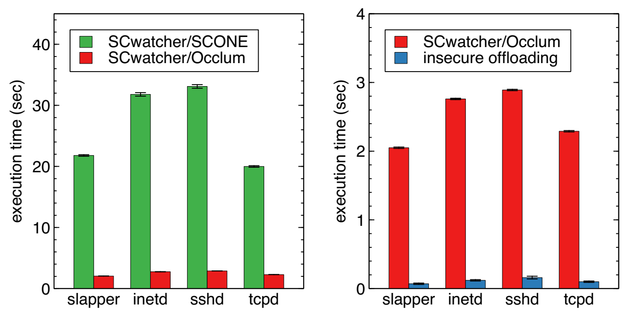
C. *Performance of Commands Using vProcFS*

We measured the execution time of the netstat and ps commands offloaded to enclaves. Fig. 9(a) shows that SCwatcher/Occlum was 3.5-9.0x faster than SCwatcher/SCONE. This is because SCwatcher/Occlum did not need to create a new enclave for the command execution. In terms of security, it is more secure to create an enclave for each command execution like SCwatcher/SCONE. From this result, SCwatcher could take a trade-off between performance and security. Note that the reason why the overhead in netstat was much larger is that the time for enclave creation occupied the large portion of the execution time.



(a) Execution time (b) Issued OCALLs

Fig. 9: The performance of executing external commands.



(a) SCwatcher (b) vs. insecure offloading

Fig. 10: The execution time of four detection functions in chkrootkit.

Compared with insecure IDS offloading, SCwatcher/Occlum was 39% slower in netstat and 52% slower in ps. One reason for this overhead is the posix_spawn system call used for creating a process and executing the command. This system call does not create a new enclave, but it needs to load a new program from the Occlum filesystem. The other reason is the overhead of vProcFS. As shown in Fig. 9(b), vProcFS issued many OCALLs to obtain memory data of the target VM. The difference in the overhead between netstat and ps came from the number of issued OCALLs. SCwatcher/Occlum needed much more OCALLs during the execution of ps.

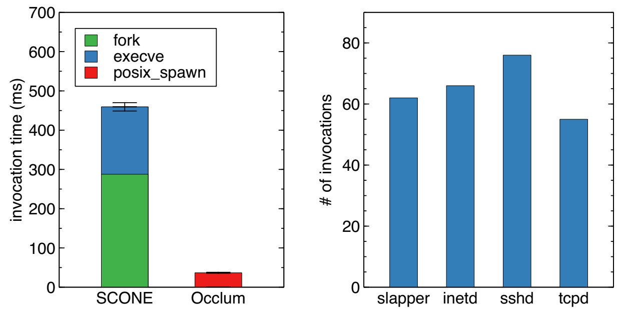
D. Performance of chkrootkit

We measured the execution time of the above four detection functions for networks and processes in chkrootkit. Fig. 10(a) shows the time needed for detecting malware using SCwatcher. SCwatcher/Occlum was 8.9-12x faster than SCwatcher/SCONE. To examine the reason for such a large difference, we first measured the average time of invoking each external command. As shown in Fig. 11(a), SCONE needed 20x longer time than Occlum to invoke one external command. In SCwatcher/SCONE, two enclaves were created whenever bash executed one external command by issuing the fork and execve system calls. In SCwatcher/Occlum, in contrast, bash modified by the Occlum developer issued only one posix_spawn system call, instead of fork and execve. In addition, Occlum did not create an enclave by this system call.

On the other hand, even SCwatcher/Occlum took 18-29x longer than insecure IDS offloading, as in Fig. 10(b). Since chkrootkit executed external commands many times, as shown in Fig. 11(b), a large portion of the execution time was spent by the posix_spawn system calls even if new enclaves were not created. This experiment shows that SCwatcher is not suitable for IDS that creates many processes by invoking external commands.

E. Performance of chkrootkit/Python

To examine the performance of IDS that does not invoke external commands in SCwatcher, we have re-implemented the four detection functions in chkrootkit by using Python. This chkrootkit is executed using only one process. It accesses only /proc/[pid]/cmdline and /proc/net/{tcp,udp} in vProcFS. Fig. 12 shows the execution time in SCwatcher/Occlum and insecure IDS offloading. We omit the result of SCwatcher/SCONE because its execution time was much longer due to creating an enclave at first. It was shown that the performance of chkrootkit written in Python was improved by 18-25x, compared with that of the original chkrootkit written in shell scripts. However, SCwatcher/Occlum was still 2.2-2.7x slower than insecure IDS offloading. One of the reasons was that the Python interpreter accessed many files at startup time. It is reported that the average overhead of reading files was 39% in Occlum [11].



(a) Invocation time (b) # of invocations

Fig. 11: The invocation overhead of external commands.

F. Performance of chkrootkit/C

To reduce the number of file accesses, we have re-implemented chkrootkit by using C. This chkrootkit is also executed using only one process and accesses only monitored files. Fig. 13 shows the execution time of this chkrootkit in SCwatcher/Occlum and insecure IDS offloading. The performance of chkrootkit written in C was further improved by 4.5-5.0x, compared with that of chkrootkit written in Python. The performance degradation in SCwatcher/Occlum

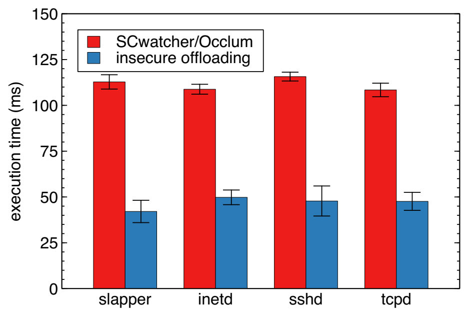


Fig. 12: The execution time of chkrookit/Python.

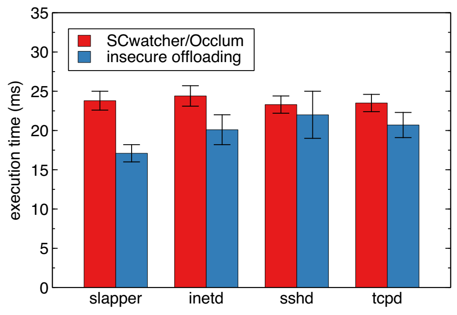


Fig. 13: The execution time of chkrookit/C.

was only 5.9-39%, compared with insecure IDS offloading. Consequently, SCwatcher/Occlum could execute IDS written in C in performance comparable to insecure IDS offloading. This result is consistent with the performance of external commands, as shown in Section V-C.

VI. Related Work

S-NFV [13] enables NIDS to be offloaded into SGX enclaves. It stores the states of virtual network functions in enclaves and runs code that handles them in enclaves. As an example, it runs Snort in an enclave so that Snort can securely manage the state for each network flow. As such, S-NFV can minimize the TCB size inside an enclave, but it is not easy to divide existing IDS into two without introducing a new attack surface. In contrast, SCwatcher can run the entire IDS in enclaves.

SEC-IDS [15] runs the entire Snort in an enclave with almost no modification. To enable this, it uses the OS emulation layer for SGX called Graphene-SGX [9], which provides the library OS but runs a single process in an enclave. This is different from SCONE and Occlum used by SCwatcher. SEC-IDS uses DPDK [16] to efficiently obtain network packets even in an enclave. As long as the state of Snort is smaller than the enclave page cache in SGX, SEC-IDS can achieve the performance comparable to running Snort without SGX. However, its TCB is relatively large because Graphene-SGX uses glibc as the standard C library. SCONE and Occlum use much smaller musl. In addition, SCwatcher can offload the entire HIDS.

Transcall [26] enables legacy user-level IDS to be offloaded from VMs. It provides an execution environment so that IDS can monitor a target VM. This execution environment is similar to a container and provides the standard OS interface, e.g., the standard C library, the proc filesystem, and system calls. For the proc filesystem and several system calls, Transcall obtains system information from the memory of a target VM and provides it to IDS. Also, it provides the same filesystem as used in a target VM. However, the provided execution environment can be easily attacked by external attackers and insiders because it is not suitably protected unlike SGX enclaves.

VMST [27] can also offload legacy IDS from VMs. Unlike Transcall, it provides a monitoring VM for each target VM. A monitoring VM runs exactly the same system as a target VM and provides the standard OS interface to IDS. The OS in a monitoring VM transparently obtains system information from the memory of a target VM and provides it to IDS. A monitoring VM is isolated more strictly than an execution environment in Transcall, but its protection is much weaker than SGX enclaves. Recently, AMD SEV [28] is used to encrypt the memory of VMs transparently. If AMD SEV is applied, a monitoring VM could be hardened.

RemoteTrans [29] can offload legacy IDS into trusted remote hosts outside clouds. Unfortunately, remote hosts can be attacked because they need to be connected to the Internet. It is also troublesome for users to maintain remote hosts by themselves. V-Met [30] runs the entire virtualized system in an outer VM using nested virtualization [31] and offloads legacy IDS to the outside of the outer VM. However, the overhead of nested virtualization largely degrades the performance of the entire virtualized system.

Besides, various systems have been proposed for the secure execution of offloaded IDS. Co-pilot [32] checks OS data using a dedicated PCI add-in card. Flicker [33] securely runs IDS using AMD SVM and Intel TXT. HyperCheck [23] and HyperSentry [24] monitor the hypervisor using the SMM of Intel processors. Self-service cloud [34] runs IDS in VMs isolated by the hypervisor. BVMD [35] embeds IDS into the hypervisor and checks the disk I/O of VMs. However, these systems cannot run legacy user-level IDS.

VII. Conclusion

This paper proposed SCwatcher for offloading user-level HIDS into enclaves using VM-compatible OS emulation layers for SGX. SCwatcher provides the standard OS interface using SCONE and Occlum. In addition, in-enclave IDS can obtain system information inside a target VM via vProcFS. vProcFS obtains memory data of the target VM, analyzes OS data in it, and generates the data of its pseudo files. We confirmed that SCwatcher could run legacy HIDS and monitor target VMs. Also, it was shown that IDS written in C could achieve monitoring performance comparable to insecure IDS offloading without SGX.

Our future work is to reduce the overhead of obtaining memory data from target VMs. For this purpose, the hypervisor could perform read-ahead of kernel data structures such as lists and obtain a larger amount of memory data at once. Another direction is to reduce the overhead of invoking external commands from IDS. One possible solution is to use the process pool. For SCwatcher/SCONE, we could preserve enclaves after command execution and reuse them by restarting the execution. For SCwatcher/Occlum, similarly, we could preserve processes in the enclave after command execution and reuse them. Since the recent third-generation Intel Xeon Scalable processors have up to 512 GB of the enclave page cache, the process pool in enclaves becomes realistic.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 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.

[2] TechSpot News, "Google Fired Employees for Breaching User Privacy," http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, 2010.

[3] PwC, "US Cybercrime: Rising Risks, Reduced Readiness," 2014.

[4] CyberArk Software, "Global IT Security Service," 2009.

[5] T. Nakano and K. Kourai, "Secure Offloading of Intrusion Detection Systems from VMs with Intel SGX," in *Proc. IEEE Int. Conf. Cloud Computing*, 2021, pp. 297–303.

[6] K. Huang, "Introduction to Intel SGX and SGX Virtualization," Xen Project Developer and Design Summit, 2017.

[7] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proc. OSDI*, 2014.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. USENIX Symp. Operating Systems Design and Implementation*, 2016, pp. 689–703.

[9] C. Tsai, D. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proc. USENIX Annual Technical Conf.*, 2017, pp. 645–658.

[10] S. Shinde, D. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," in *Proc. Network and Distributed System Security Symp.*, 2017.

[11] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[12] N. Murilo and K. Steding-Jessen, "chkrootkit – Locally Checks for Signs of a Rootkit," http://chkrootkit.org/.

[13] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV States by Using SGX," in *Proc. ACM Int. Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2016, pp. 45–48.

[14] Snort Team, "Snort – Network Intrusion Detection & Prevention System," https://www.snort.org/.

[15] D. Kuvaiskii, S. Chakrabarti, and M. Vij, "Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX)," in *arXiv:1802.00508*, 2018.

[16] Linux Foundation, "Data Plane Development Kit (DPDK)," https://www.dpdk.org/.

[17] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Proc. Workshop on Hot Topics in Cloud Computing*, 2009.

[18] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proc. ACM Symp. Operating Systems Principles*, 2011, pp. 203–216.

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

[20] H. Shuang, W. Huang, P. Bettadpur, L. Zhao, I. Pustogarov, and D. Lie, "Using Inputs and Context to Verify User Intentions in Internet Services," in *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 76–83.

[21] Trusted Computing Group, "Trusted Platform Module Library Family "2.0" Specification," 2019.

[22] J. Rutkowska and R. Wojtczuk, "Preventing and Detecting Xen Hypervisor Subversions," Black Hat USA, 2008.

[23] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A Hardware-assisted Integrity Monitor," in *Proc. Int. Symp. Recent Advances in Intrusion Detection*, 2010, pp. 158–177.

[24] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky, "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity," in *Proc. ACM Conf. Computer and Communications Security*, 2010, pp. 38–49.

[25] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Detecting System Failures with GPUs and LLVM," in *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 47–53.

[26] T. Iida and K. Kourai, "Transcall," http://www.ksl.ci.kyutech.ac.jp/oss/transcall/, 2012.

[27] 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. IEEE Symp. Security and Privacy*, 2012, pp. 586–600.

[28] Advanced Micro Devices, Inc., "Secure Encrypted Virtualization API Version 0.24," 2020.

[29] K. Kourai and K. Juda, "Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds," in *Proc. IEEE Int. Conf. Cloud Computing*, 2016, pp. 43–50.

[30] S. Miyama and K. Kourai, "Secure IDS Offloading with Nested Virtualization and Deep VM Introspection," in *Proc. European Symp. Research in Computer Security, Part II*, 2017, pp. 305–323.

[31] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles Project: Design and Implementation of Nested Virtualization," in *Proc. 9th USENIX Symp. Operating Systems Design and Implementation*, 2010, pp. 423–436.

[32] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proc. 13th USENIX Security Symp.*, 2004.

[33] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. 3rd ACM SIGOPS/EuroSys European Conf. Computer Systems*, 2008, pp. 315–328.

[34] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service Cloud Computing," in *Proc. ACM Conf. Computer and Communications Security*, 2012, pp. 253–264.

[35] Y. Oyama, T. Giang, Y. Chubachi, T. Shinagawa, and K. Kato, "Detecting Malware Signatures in a Thin Hypervisor," in *Proc. Annual ACM Symp. Applied Computing*, 2012, pp. 1807–1814.