Secure and Efficient Monitoring of Confidential VMs Using Programmable Read-ahead with eBPF

Kanta Uesugi
Kyushu Institute of Technology
uesugi@ksl.ci.kyutech.ac.jp

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

***Abstract*—In clouds, sensitive information in users' virtual machines (VMs) could be eavesdropped on by insiders. To prevent such information leaks, recent clouds provide confidential VMs (CVMs), whose memory is protected against insiders. Since CVMs cannot be monitored by intrusion detection systems (IDS) with VM introspection due to memory encryption, previous work enables IDS to obtain memory data via agents running inside CVMs. However, it is necessary to communicate with the agents whenever IDS needs the data of the guest operating system (OS). When IDS traverses OS data structures with pointers, it has to send many requests to the agents. This paper proposes *eBPFmonitor* for enabling efficient monitoring of CVMs by reading ahead OS data with eBPF. eBPFmonitor securely runs custom eBPF programs injected into CVMs for each IDS and obtains necessary memory data in batches. To address the issues of expressiveness, portability, and coverage in read-ahead with eBPF, it uses various methods for unbounded loops, BPF CO-RE, and LLVM Pass. It can perform the read-ahead of OS data synchronously or asynchronously in various data block sizes to maximize monitoring performance. We have implemented eBPFmonitor and showed that it was 3-7x faster than the previous work.**

***Index Terms*—virtual machine, intrusion detection system, eBPF, confidential VM, VM introspection**

I. Introduction

In clouds, insiders could eavesdrop on sensitive information in users' virtual machines (VMs) [1]–[3]. To counteract this risk, recent clouds provide confidential VMs (CVMs) [4]–[6], which are protected against insiders by using trusted execution environments (TEEs) such as AMD SEV [7] and Intel TDX [8]. Since processors manage encryption keys used by TEEs, even cloud insiders cannot decrypt the memory of CVMs. Unfortunately, CVMs cannot provide mechanisms to prevent attackers from intruding into VMs. Since TEEs perform decryption for memory access inside CVMs, intruders can also access decrypted memory. Therefore, intrusion detection systems (IDS) are required to detect intruders before sensitive information is eavesdropped on.

To prevent IDS itself from being compromised by intruders, IDS can be offloaded to the outside of VMs using VM introspection (VMI) [9]. However, offloaded IDS cannot monitor CVMs because their memory is encrypted. To enable this IDS offloading for CVMs, previous work enables IDS to monitor CVMs by communicating with agents running securely inside the CVMs [10], [11]. IDS sends a request to the agent whenever it requires the data of the guest operating system (OS) and receives memory data. The drawback of this approach is that monitoring performance is significantly degraded by communication overhead. In particular, when IDS traverses complex OS data structures with pointers, it needs communication to obtain memory data one by one.

This paper proposes *eBPFmonitor* for enabling IDS to efficiently monitor CVMs by reading ahead OS data with eBPF inside CVMs. eBPFmonitor isolates the target system using a container created in a CVM and securely runs an agent outside the container. When IDS needs to access complex OS data, it can execute custom eBPF programs injected into the CVM via the agent. For example, an eBPF program traverses the list of processes, obtains necessary memory data, and returns it to the IDS in batches. The injected eBPF programs can be safely executed because the verifier can detect illegal instructions and infinite loops. As such, eBPFmonitor can reduce communication overhead between IDS and the agent and improve monitoring performance.

To use eBPF programs for reading ahead OS data, eBPFmonitor addresses three issues: *expressiveness*, *portability*, and *coverage*. It uses various methods provided by eBPF, e.g., the BPF helper function [12], BPF iterators [13], and BPF Kernel Functions [14], to support unbounded loops, which are often used to access OS data. Using BPF CO-RE [15] enables developed eBPF programs to run in different kernel versions. In addition, eBPFmonitor converts the intermediate representation of eBPF programs using LLVM Pass [16] to automate the collection of OS data. We have implemented eBPFmonitor in Linux and KVM and conducted several experiments to show its effectiveness. As a result, eBPFmonitor could achieve 3-7x higher monitoring performance than the previous work.

II. Monitoring Confidential VMs

Even though the memory of CVMs is encrypted, it is not protected if attackers intrude into the CVMs. This is because the memory encryption is effective only against attacks from outside the CVMs. Since the memory is transparently decrypted for access inside the CVMs, intruders can easily eavesdrop on sensitive information in memory. From the perspective of processors, they cannot be distinguished from legitimate users in CVMs. Even if they cannot directly access the memory containing sensitive information inside CVMs, they could steal it by exploiting vulnerabilities in privileged software, such as the guest OS. Therefore, it is still necessary

to detect attacks against CVMs using IDS. In particular, host-based IDS is useful for monitoring the system states inside CVMs and detecting the symptoms of intrusion.

Since this type of IDS inherently runs inside VMs unlike network-based IDS, it can be easily disabled by intruders. To protect IDS from intruders, a technique called VM introspection (VMI) [9] is often used. This technique securely runs IDS outside VMs and enables it to monitor the systems inside VMs. Offloaded IDS detects attacks by analyzing OS data obtained from the memory of VMs. As such, intruders cannot disable offloaded IDS, while offloaded IDS can detect intruders. However, offloaded IDS cannot analyze OS data in the memory of CVMs because the memory is encrypted. From the perspective of processors, offloaded IDS cannot be distinguished from cloud insiders.

To support IDS offloading with VMI for CVMs, several systems have been proposed [10], [11]. They enable offloaded IDS to obtain memory data by communicating with agents running inside CVMs. 00SEVen [10] protects the agent using VM Privilege Levels (VMPLs) in SEV-SNP [17] and runs IDS at a remote host. SEVmonitor [11] confines the target system in a container created inside a CVM and securely executes the agent outside the container. It runs IDS in another CVM called an IDS VM. These systems encrypt communication between the IDS and the agent to protect the memory data obtained.

These systems reduce communication overhead by obtaining memory data for each page and preserving it in the cache. This cache can reduce communication if IDS accesses multiple OS data in a single memory page. However, IDS often needs communication whenever it traverses pointers if accessing complex data structures with pointers, e.g., the process list. In this case, frequent communication largely degrades monitoring performance. According to our experiment, it took up to 82x longer to obtain a set of OS data with pointers in SEVmonitor, compared with traditional VMI applied to a non-confidential VM.

The threat model of this paper is as follows. We trust processors providing TEEs and assume that TEEs are free of vulnerabilities. We trust the agent and the OS kernel running in a target CVM. However, we assume that the target system running in a container has vulnerabilities and could be compromised by external attackers. We also trust an IDS VM and assume that the system in it is not compromised. Since the IDS VM runs only IDS, it is easier to protect the IDS VM than the target VMs. We do not assume that denial-of-service (DoS) attacks against IDS because TEEs basically cannot protect VMs from DoS attacks.

III. eBPFmonitor

This paper proposes *eBPFmonitor* for enabling efficient monitoring of CVMs by programmable read-ahead with eBPF. Fig. 1 illustrates the system architecture of eBPFmonitor. Like the previous work [11], eBPFmonitor confines the target system in a container created in a CVM and runs an agent outside the container. Offloaded IDS runs in a dedicated CVM and obtains memory data from the agent running in the target

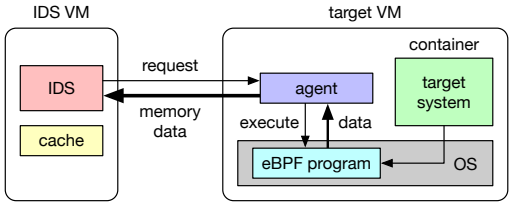


Fig. 1: The system architecture of eBPFmonitor.

CVM. It injects custom eBPF programs into the target CVM and securely runs them in the guest OS. Using the eBPF programs, it reads ahead complex OS data, obtains the memory data necessary for monitoring in batches, and preserves it in the cache. As such, eBPFmonitor reduces communication and achieves efficient monitoring by returning the necessary OS data in a single request.

eBPFmonitor achieves both safety and flexibility by using eBPF. eBPF is the kernel mechanism that can monitor the system in Linux and Windows. It is used to trace network and system behavior and enhance system security. eBPF programs are loaded into the OS kernel and executed when a system event occurs, e.g., network events, system calls, function entries, and kernel trace points. They can dynamically be loaded and executed without modifying the OS kernel or adding kernel modules. Therefore, eBPFmonitor can flexibly and efficiently obtain OS data using different eBPF programs for each IDS. For safety, eBPF programs fail to be loaded if the eBPF verifier detects illegal instructions or infinite loops. Therefore, eBPFmonitor can obtain OS data more safely than kernel modules, which can just restrict access to kernel functions and global variables.

However, it is not easy to collect OS data using eBPF programs due to three issues. First, unbounded loops are often necessary to traverse OS data structures, but eBPF does not allow unbounded loops to prevent excessive resource consumption. To address this *expressiveness* issue, eBPFmonitor uses various methods provided in eBPF [12]–[14]. Second, eBPF programs depend on OS data structures and can be injected only into the guest OS of a specific version. To address this *portability* issue, eBPFmonitor uses the BPF Compile Once – Run Everywhere (CO-RE) framework [15]. Third, it is not easy to write eBPF programs that obtain all the necessary OS data thoroughly because OS data structures are complex. To address this *coverage* issue, eBPFmonitor uses LLVM Pass [16] and automatically inserts code for collecting OS data into eBPF programs.

eBPFmonitor provides both synchronous and asynchronous read-ahead methods because there are tradeoffs between them. For synchronous read-ahead, IDS waits for the completion of receiving memory data after sending a read-ahead request to the agent. This method is simple but inefficient. In contrast, asynchronous read-ahead enables IDS to continue monitoring immediately after sending the request. If the read-ahead task has not yet stored memory data in the cache when IDS requires OS data, one method is to wait for the completion of the read-

ahead. The other is to send another request for the memory data to the agent. The former could avoid redundant requests, while the latter could reduce the latency for accessing OS data. The drawback of asynchronous read-ahead is the need for synchronization between IDS and the read-ahead task.

Since there also exist tradeoffs in terms of the block size of memory data that eBPF programs collect, eBPFmonitor supports various block sizes to obtain memory data. If eBPFmonitor obtains memory data in a fixed block size, e.g., 4 KB, it is easy to manage the cache. However, most of the OS data contained in each block could not be used for monitoring. The network transfer of such data becomes wasteful. In contrast, if eBPFmonitor obtains only the necessary OS data, it can reduce the network transfer of memory data. However, the total network transfer could increase because the number of metadata, such as the address and size, increases.

IV. Implementation

We have implemented eBPFmonitor in Linux and KVM using BPF CO-RE and LLVM Pass.

A. eBPF Programs Collecting OS Data

This subsection describes three issues in collecting OS data using eBPF programs and explains the solutions in further detail.

1) Expressiveness: To collect OS data, eBPF programs often traverse data structures using pointers until they satisfy conditions. Unfortunately, the verification of such eBPF programs fails to prevent excessive resource consumption because eBPF programs contain unbounded loops that may not stop. It is possible to use a finite loop with the maximum count specified and escape the loop if the condition is satisfied. In eBPF, however, a finite loop is always unrolled at compile time and increases the number of eBPF instructions. If complex processing is done in a loop or nested loops are used, that eBPF program easily exceeds more than one million instructions, which is the upper limit of one eBPF program.

Therefore, eBPFmonitor can use the BPF helper function [12] called `bpf_loop`, which allows a finite loop with up to eight million counts. This is sufficient for traversing kernel data structures. Since the loop is not unrolled, nested loops are possible. The processing in a loop is defined as a callback function that returns 1 when the loop ends. The variables used across loop iterations need to be passed as arguments to the `bpf_loop` function.

Instead of this BPF helper function, eBPFmonitor can use BPF iterators [13], which are one of the eBPF program types. This type traverses the list of a kernel data structure and invokes the specified callback function for each object. Compared with the `bpf_loop` function, it is more concise to write eBPF programs because it is not necessary to write programs for traversing lists. However, BPF iterators are available only for supported kernel data structures.

To express loops more intuitively without callback functions, eBPFmonitor can use BPF Kernel Functions (kfuncs) [14], which allow eBPF programs to use the Linux kernel functions. BPF kfuncs provides functions for finite loops and traversing kernel data structures. In general, BPF kfuncs can be used for a finite loop using a loop count. However, BPF kfuncs depends on the kernel version and can be changed or removed.

TABLE I: Three methods for unbounded loops

	expressiveness	generality	stability
BPF helper functions		✓	✓
BPF iterators	✓		✓
BPF kfuncs	✓	✓	

Table I summarizes the three methods for unbounded loops. Developers need to use the most appropriate method to write eBPF programs for the read-ahead of OS data.

2) Portability: eBPFmonitor dynamically injects eBPF programs into the target VM. This means that the kernel version of the target VM could differ from that of the development environment. Since kernel data structures change with kernel versions, eBPF programs compiled with the kernel header files in the development environment may not be compatible with the kernel in the target VM. This issue can be addressed by using BPF Compiler Collection (BCC) [18], which compiles eBPF programs at load time in the target VM. However, the target VM needs to install the development environment for compiling eBPF programs. In addition, loading eBPF programs takes longer.

Therefore, eBPFmonitor uses the BPF CO-RE framework [15] for the development and execution of eBPF programs. BPF CO-RE enables eBPF programs to be compiled in the development environment and be loaded in arbitrary target VMs. To accommodate the differences of the kernel data structures, it relocates the offsets of member variables of data structures at load time. To enable this relocation, eBPF programs use the `BPF_CORE_READ` macro instead of direct access, e.g., `task->pid`, to access the member variables of kernel data structures.

However, BPF CO-RE cannot directly handle kernel global variables. It is possible to embed the addresses of global variables, but such addresses depend on the kernel. Also, they can change at boot time even in the same kernel if Kernel Address Space Layout Randomization (KASLR) is enabled. Therefore, eBPFmonitor converts global variables from symbol names to kernel addresses at runtime using the BPF helper function. This enables eBPF programs to access global variables in the target VM.

3) Coverage: Developers can write eBPF programs that collect OS data using the kernel source code. Then, they need to add code for storing collected OS data in a BPF table using the BPF helper function. A BPF table is a ring buffer used to share collected OS data with the user-space agent. However, this task is cumbersome because access to OS data is spread across various locations. If eBPF programs cannot collect all the necessary OS data, IDS needs to send requests for the not-collected OS data on demand.

To reduce the burden on developers, eBPFmonitor automates storing OS data in a BPF table. It compiles eBPF

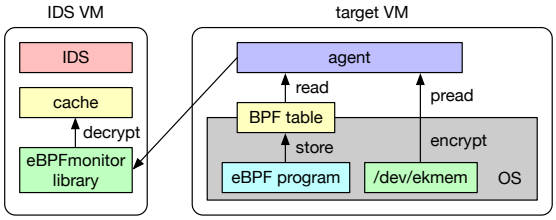


Fig. 2: Collecting memory data using a BPF table and the ekmem pseudo device.

programs using LLVM and converts the intermediate representation using LLVM Pass, which is a framework for optimizing LLVM bitcode. eBPFmonitor inserts code for storing OS data in a BPF table at all the call sites of the helper function used to read data from the kernel. It obtains the address and size of OS data from the arguments of this helper function.

When collecting only necessary OS data, eBPFmonitor stores the address, size, and value of OS data in a BPF table. For example, eBPF programs can collect global variables, member variables, arrays, and strings. When collecting OS data in a fixed block size, eBPFmonitor stores only the address of the block in a BPF table. This is because many eBPF instructions are required if eBPF programs store the entire memory data in a BPF table.

B. eBPFmonitor Agent

When the agent receives a load request for an eBPF program from IDS, it checks whether the eBPF program has already been loaded. It manages the load status of eBPF programs using their hash values. If the received hash value is not registered, the agent sends a request for an eBPF program to IDS and loads the received eBPF program. Then, it registers its hash value. When the agent receives the read-ahead request of OS data from IDS, it executes the eBPF program corresponding to the specified hash value. It directly executes an eBPF program using the `bpf` system call, although eBPF programs are usually executed when a system event occurs. Using encrypted communication, the agent accepts the load and read-ahead requests only from legitimate IDS that holds the correct encryption key.

When IDS collects memory data including OS data in a fixed block size, the agent first obtains the block address stored in the BPF table by the executed eBPF program, as illustrated in Fig. 2. Then, it obtains memory data via the ekmem pseudo device, which is provided as a kernel module by the guest OS. It stores the block address and size in a buffer and issues the `pread` system call with the buffer. The pseudo device encrypts memory data at the specified address using AES and stores it in the passed buffer.

When IDS collects only necessary OS data, eBPF programs store it directly in the BPF table. OS data stored in the BPF table is not encrypted because the encryption algorithm needs many eBPF instructions. Instead, the agent encrypts the obtained OS data before sending it back to IDS. Since we assume that the agent is not compromised, it is still secure for the agent to obtain unencrypted OS data.

C. eBPFmonitor Library

eBPFmonitor provides a library for transparently obtaining OS data via the agent to IDS. IDS invokes the library whenever it needs OS data to monitor the target VM. eBPFmonitor uses the LLView framework [19], [20] to automate this invocation of the library. The invoked library first checks whether the accessed OS data is cached. If the necessary OS data is already in the cache, the library returns it to IDS immediately. Otherwise, it sends a request to the agent, stores received memory data in the cache, and returns it to IDS. The cache is cleared after IDS completes one execution of the periodic monitoring.

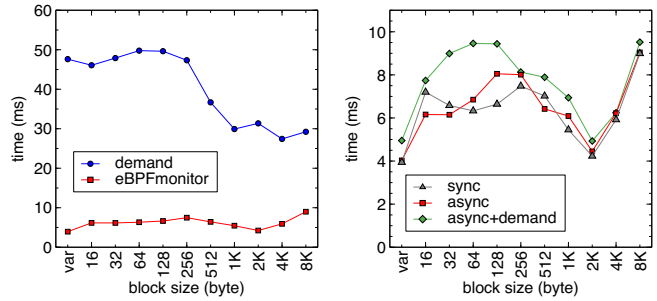
At the time of this demand read, the library can read ahead OS data by sending a read-ahead request to the agent. When it receives pairs of encrypted memory data and its size from the agent, it decrypts the memory data. It stores the address, size, and memory data of OS data in the cache. The library can read ahead OS data at various timings. For example, it can read ahead the entire list when IDS accesses the first node of the list or traverses several nodes in the list.

When the library uses asynchronous read-ahead, it creates a read-ahead thread, which receives memory data from the agent. Whenever the thread accesses the cache, it acquires a lock to ensure that the IDS thread does not access the cache at the same time. If the library uses asynchronous read-ahead with waiting, it stops when necessary OS data does not exist in the cache and then checks the cache periodically. If necessary OS data is stored by the read-ahead thread, it restarts the IDS thread. If the library uses asynchronous read-ahead with demand read, it sends an additional request to the agent and waits for memory data when necessary.

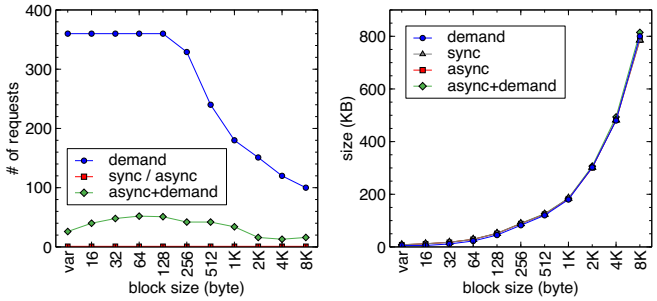
V. Experiments

To demonstrate the effectiveness of eBPFmonitor, we measured the time required to obtain system information by reading ahead OS data using eBPF programs. For comparison, we used (1) demand read, which obtained memory data via the agent on demand, and (2) direct VMI, which obtained OS data directly from the memory of a non-confidential VM. We used a server with an AMD EPYC 7443P processor and 256 GB of memory and ran Linux 5.15 and QEMU-KVM 6.2.0. We assigned two virtual CPUs and 2 GB of memory to both the target and IDS VMs and ran Linux 6.5 as the guest OS. We enabled SEV in both VMs, except for when using direct VMI.

We have developed various IDSes for eBPFmonitor, but we show the results of two IDSes due to space limitations. Proc-IDS traversed the entire list of processes using an eBPF program and obtained information on the processes running in the target VM. TCP-IDS traversed the large hash tables managing TCP sockets using another eBPF program and obtained information on TCP connections in the target VM.

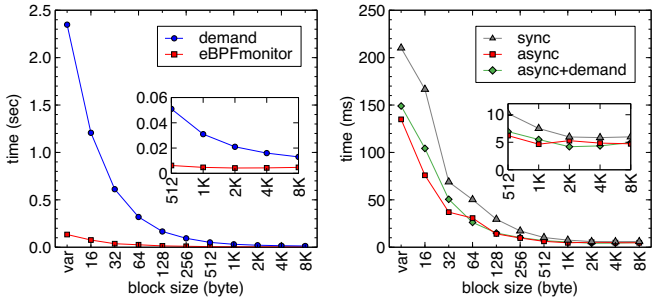


(a) Performance (b) Detailed Performance

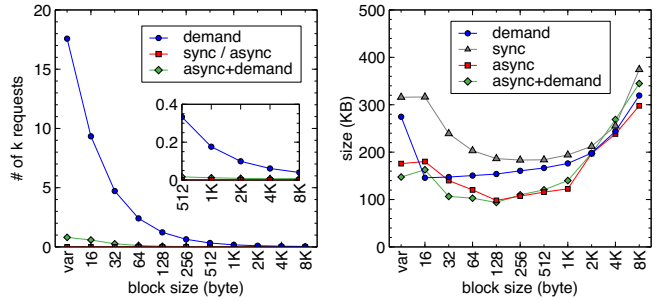


(c) Requests (d) Obtained data

Fig. 3: The monitoring of the process list.



(a) Performance (b) Detailed Performance



(c) Requests (d) Obtained data

Fig. 4: The monitoring of the TCP connections.

A. *Monitoring Performance*

We measured the monitoring time using the two IDSes. For eBPFmonitor, the monitoring time is the time elapsed from when IDS sends a read-ahead request to the agent until it completes monitoring. To examine the impact of the block size, eBPFmonitor obtained memory data in block sizes ranging from 16 bytes to 8 KB or in a variable block size from the agent. Also, we compared synchronous and asynchronous read-ahead. Demand read also obtained memory data in block sizes ranging from 16 bytes to 8 KB or in a variable block size. Direct VMI obtained memory data only in a variable block size due to its nature.

1) Process List: Fig. 3(a) shows the average monitoring time of Proc-IDS, which obtained the list of 120 processes. The results of eBPFmonitor are the time in the best read-ahead method. Compared with the results in demand read with the best block size, eBPFmonitor was 7.0x faster in the best case. It was the fastest when using a variable block size. In the same block size, it was 3.2-12x faster than demand read. Demand read was 82x slower than direct VMI, while eBPFmonitor was still 12x slower. When the block size was 4 KB, the execution time of the eBPF program was only 0.5 ms, most of which was spent storing OS data in the eBPF table. Most of the monitoring time, 5.9 ms, was spent obtaining memory data from the OS kernel and transferring it to the IDS.

For demand read, the monitoring time was long in a variable block size and did not decrease while the block size was less than or equal to 256 bytes. This is because the IDS accessed three small member variables spread across the large object of 9798 bytes. When the block size increased, the monitoring time suddenly decreased. Then, it became shorter and more stable after the block size was 1 KB or greater. The reason is that the number of requests decreased due to more cache hits, as shown in Fig. 3(c). In contrast, the overhead of obtaining larger memory data increased the latency.

For eBPFmonitor, the trend was much different from that of demand read. The monitoring time was the shortest in a variable block size because the eBPF program could efficiently collect only the necessary 28-byte data per process. The monitoring time increased until the block size reached 256 bytes and then decreased thereafter. Then, the results for a block size of 2 KB were almost the same as those for a variable block size. For larger block sizes, the monitoring time increased.

When we used synchronous read-ahead or asynchronous read-ahead with waiting, the number of requests was only one. This means that eBPFmonitor could fully automate storing OS data in the BPF table and collect all of the necessary OS data using the eBPF program at once. As shown in Fig. 3(b), the trends differ significantly among the three read-ahead methods. However, synchronous read-ahead was the best across most block sizes, while asynchronous read-ahead with demand read was consistently the worst because the number of requests increased.

2) TCP Connections: Fig. 4(a) shows the average monitoring time of TCP-IDS, which obtained information on nine TCP connections. Compared with the results in demand read with the best block size, eBPFmonitor was 3.1x faster in the best case. eBPFmonitor was the fastest with a block size of 2 KB. This improvement was smaller than Proc-IDS because it took longer to monitor the large hash tables. In the same block size, it was 2.8-17x faster than demand read. Compared

with direct VMI, demand read was at least 10x slower, while eBPFmonitor was 3.2x slower. The execution time of the eBPF program was 3.2 ms and much longer than that in Proc-IDS, but its execution could overlap with obtaining and transferring memory data.

For demand read, the monitoring time decreased as the block size increased. It was the shortest in a block size of 8 KB. This IDS traversed 16384 buckets in the TCP established hash table and 1024 buckets in the TCP listen hash table. Since these buckets are linear arrays, using a larger block size was more efficient. As the block size became larger, the number of requests became smaller, but the size of the obtained data did not increase much. Unlike Proc-IDS, the obtained data was much larger in a variable block size, as shown in Fig. 4(d). This is because the amount of metadata was much larger.

For eBPFmonitor, the monitoring time was also the worst when using a variable block size. This is because the number of obtained data was larger. Although TCP-IDS was much more complex than Proc-IDS, the number of requests was only one when we used synchronous read-ahead or asynchronous read-ahead with waiting, as shown in Fig. 4(c). This means that the eBPF program could collect all of the necessary OS data. Asynchronous read-ahead with waiting performed best across most block sizes, as shown in Fig. 4(b). However, synchronous read-ahead was always the worst. Although this IDS did not use all the OS data collected by the eBPF program, it had to wait for read-ahead to complete in this method. Fig. 4(d) shows the amount of data obtained until the IDS finished, but the size of the obtained data was largely different from that of Proc-IDS.

B. Security Analysis

In eBPFmonitor, intruders in the target system cannot disable the agent or injected eBPF programs because the agent and eBPF programs are isolated from the target system using a container. We assume that intruders cannot escape a container. Also, intruders cannot disable IDS running in the IDS VM because the target VM is isolated from the IDS VM. It is more difficult to intrude into the IDS VM, which runs only IDS. Similarly, intruders cannot access communication data between the agent and IDS.

Even cloud insiders cannot disable the agent or eBPF programs because the target VM is protected using a CVM. Due to the nature of a CVM, insiders could terminate the entire target VM using administrative privileges. However, this attack can be easily detected by users because it stops all the services provided by the target VM. Also, cloud insiders cannot disable IDS because the IDS VM is also running as a CVM. Cloud insiders can terminate the entire IDS VM, but users can detect this attack by periodically communicating with IDS. Although cloud insiders can access communication data between the agent and IDS, the data is securely encrypted. Since the data is encrypted using a pre-shared key between IDS and the agent, cloud insiders cannot inject eBPF programs into the target VM to eavesdrop on the internal information.

VI. Related Work

For far memory, several systems read ahead data from remote hosts. To achieve efficient read-ahead, AIFM [21] provides special data structures to applications that use the memory of remote hosts as swap space. However, it cannot be applied to the monitoring of the existing data structures in the OS because it needs to develop applications using the special data structures. DiLOS [22] enables the OS to efficiently obtain memory data stored in remote hosts. Upon a page fault, the application invoked by the OS specifies the necessary data using the application's knowledge. DiLOS reads ahead memory data for a sub-page, whose size is less than a page, and reduces network transfers for unnecessary memory data. When applications traverse data structures using pointers, they must specify the necessary data individually.

BPFd [23] enables users to transparently execute eBPF programs developed using BPF Compiler Collection (BCC) [18] on remote hosts. It forwards BCC functions, such as loading eBPF programs and configuring hook points, to remote hosts. However, it needs to compile eBPF programs using BCC on a local host and execute them on remote hosts. Therefore, the difference in the kernel version could affect the execution of eBPF programs. In contrast, TeleBPF [24] forwards BPF-related system calls to remote hosts. When used with BPF CO-RE, it can execute eBPF programs on a remote host whose kernel version differs from that of the local host.

SEVmonitor [11] runs an agent inside the guest OS or the inner hypervisor of the target VM to protect the agent. When the agent runs in the guest OS, it needs to load eBPF programs and access BPF tables inside the OS without using system calls. When the agent runs in the inner hypervisor, it can use the eBPF framework in the Linux kernel if the hypervisor is KVM. It can obtain OS data in the inner VM, which is created inside the target VM, using Hyperupcalls [25].

VII. Conclusion

This paper proposed eBPFmonitor for injecting eBPF programs into CVMs and enabling programmable read-ahead of OS data for IDS. eBPFmonitor reduces communication by obtaining necessary OS data in batches. We have implemented eBPFmonitor using BPF CO-RE and LLVM Pass and confirmed that IDS could obtain all the necessary OS data in a single request. As a result, IDS could significantly improve monitoring performance, compared with the previous work.

One of our future work is to apply eBPFmonitor to real clouds. It should be easy for clouds supporting CVMs because eBPFmonitor does not need to modify cloud infrastructure. Another direction is to integrate eBPFmonitor with 00SEVen [10], which can run the agent at VMPL0 [17] inside a CVM more securely. Using VMPL0, we do not need to trust the guest OS.

Acknowledgments

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

REFERENCES

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

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

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

[4] Amazon Web Services, Inc., "AMD SEV-SNP for Amazon EC2 Instances," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html, 2024.

[5] Google LLC, "Confidential VM Documentation," https://cloud.google.com/confidential-computing/confidential-vm/docs, 2025.

[6] Microsoft Corporation, "About Azure confidential VMs," https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview, 2024.

[7] Advanced Micro Device, Inc., "Secure Encrypted Virtualization API Version 0.24," 2020.

[8] Intel Corporation, "Intel Trust Domain Extension," Intel Corporation, Tech. Rep., 2023.

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

[10] F. Schwarz and C. Rossow, "00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents," in *Proc. USENIX Security Symp.*, 2024, pp. 1651–1668.

[11] T. Nono and K. Kourai, "Secure Monitoring of Confidential VMs with Isolated Agents," in *Proc. IEEE/ACM Int. Conf. Utility and Cloud Computing*, 2025.

[12] The Linux Kernel, "Helper functions," https://docs.kernel.org/bpf/helpers.html.

[13] ——, "BPF Iterators," https://docs.kernel.org/bpf/bpf_iterators.html.

[14] ——, "BPF Kernel Functions (kfuncs)," https://docs.kernel.org/bpf/kfuncs.html.

[15] A. Nakryiko, "BPF CO-RE (Compile Once - Run Everywhere), Andrii Nakryiko's Blog," https://nakryiko.com/posts/bpf-portability-and-co-re/.

[16] LLVM Project, "Writing an LLVM Pass," https://llvm.org/docs/WritingAnLLVMNewPMPass.html.

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

[18] IO Visor Project, "BPF Compiler Collection (BCC)," https://github.com/iovisor/bcc.

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

[20] ——, "Reliable and Accurate Fault Detection with GPGPUs and LLVM," in *Proc. IEEE Int. Conf. Cloud Computing*, 2023, pp. 540–546.

[21] Z. Ruan, M. Schwarzkopf, M. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated Far Memory," in *Proc. USENIX Symp. Operating Systems Design and Implementation*, 2020, pp. 315–332.

[22] W. Yoon, J. Oh, J. Ok, S. Moon, and Y. Kwon, "DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation," in *Proc. European Conf. Computer Systems*, 2023, pp. 266–282.

[23] J. Fernandes, "BPFd (Berkeley Packet Filter Daemon)," https://github.com/joelagnel/bpfd.

[24] K. Hori and K. Kourai, "Safe and Transparent Monitoring of VMs with Injected eBPF Programs," in *Proc. IEEE Cyber Science and Technology Congress*, 2025, pp. 55–63.

[25] N. Amit and M. Wei, "The Design and Implementation of Hyperupcalls," in *Proc. USENIX Annual Technical Conf.*, 2018, pp. 97–112.