Safe and Transparent Monitoring of VMs with Injected eBPF Programs

Kyosuke Hori
Kyushu Institute of Technology
hori@ksl.ci.kyutech.ac.jp

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

***Abstract*—Clouds provide virtual machines (VMs) to users and often collect internal information using agents running in VMs for system monitoring. One issue with using agents is that agents themselves can become vulnerabilities in VMs. Clouds can also obtain internal information by directly accessing the memory of VMs with VM introspection (VMI). However, VMI is too powerful for clouds to monitor users' VMs. Unlike the agent method, users cannot control system monitoring with VMI. This paper proposes TeleBPF for safer system monitoring of VMs. TeleBPF enables clouds to dynamically inject verifiable eBPF programs into VMs as agents and obtain their internal information. To transparently run monitoring tools outside VMs, TeleBPF intercepts eBPF-related system calls and forwards them to VMs. In addition, it enables monitoring tools to obtain information from eBPF programs by sharing ring buffers with VMs. We confirmed that existing monitoring tools could run using TeleBPF and showed that they could be executed faster by reducing virtualization overhead.**

***Index Terms*—eBPF, agent, VM introspection, virtual machine**

1. Introduction

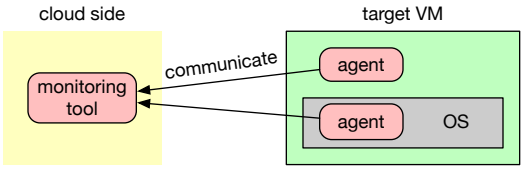
Infrastructure-as-a-Service (IaaS) clouds provide virtual machines (VMs) to users. Users can freely manage the entire systems in VMs, whereas clouds often monitor the states of the systems in VMs to enhance observability and security. They usually use software called an agent to monitor VMs. This method installs an agent in a VM and obtains the internal information of the VM from the agent. The agent can be executed as a process [1] or a kernel module [2]. One issue with this agent method is that users need to keep the agent up-to-date by themselves. If they do not update the agent, the agent itself could become vulnerabilities in the system. If the agent is executed as a process to suppress the impact on the system, it cannot monitor all the system information. If it is executed as a kernel module, its vulnerabilities could affect the entire system.

To address this issue, clouds can use VM introspection (VMI) [3], which obtains the internal information of the system by directly analyzing the memory of a VM. However, VMI is too powerful for clouds to monitor users' VMs. Unlike the agent method, users cannot control system monitoring with VMI at all. In addition, it cannot be used for emerging confidential VMs, which are provided by recent clouds such as Amazon Web Services [4], Google Cloud [5], and Microsoft Azure [6]. The memory of confidential VMs is encrypted by processor-based trusted execution environments (TEEs) such as AMD SEV [7] and Intel TDX [8]. Therefore, clouds cannot obtain the necessary information by accessing the memory of such VMs.

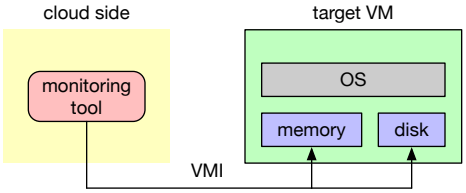
This paper proposes TeleBPF to monitor the systems in VMs by injecting eBPF programs from clouds. eBPF [9] is a mechanism that extends the Berkeley packet filter in Linux. eBPF programs can monitor the state of the system at a specified point, e.g., when events occur in the operating system (OS). TeleBPF uses eBPF programs as safe agents. eBPF programs can safely run in the OS kernel because they are checked by the eBPF verifier at load time. TeleBPF enables clouds to transparently execute existing monitoring tools developed as eBPF applications on the cloud side and obtain internal information from VMs. Since clouds can dynamically load eBPF programs as agents, users do not need to maintain agents by themselves. eBPF programs running inside VMs are not affected by the memory encryption of confidential VMs.

TeleBPF provides eBPF applications with a shared library that intercepts eBPF-related system calls. It efficiently achieves the interception of the system calls by binary rewriting with zpoline [10]. Then, the shared library forwards the intercepted system calls to VMs using the VM-specific communication mechanism called the VM socket [11]. The TeleBPF proxy running in the VMs issues the forwarded system calls and returns the results of the system calls to the eBPF applications. In addition, TeleBPF shares ring buffers used for returning information from eBPF programs between VMs and eBPF applications and enables eBPF applications to directly access ring buffers without network communication. We confirmed that various eBPF applications ran with TeleBPF and that the monitoring performance outperformed traditional in-VM execution thanks to the reduction of virtualization overhead.

The organization of the paper is as follows. Section 2 describes the advantages and disadvantages of the agent method and VMI. Section 3 proposes a new monitoring system with eBPF, called TeleBPF. Section 4 explains the implementation of TeleBPF, specifically its shared library and proxy. Section 5 shows experimental results using TeleBPF. Section 6 describes related work, and Section 7



(a) Agent method



(b) VM introspection

Figure 1. Two existing monitoring methods.

concludes the paper.

2. System Monitoring in VMs

In IaaS clouds, users manage their systems inside provided VMs, whereas clouds often monitor information inside VMs and use it for various purposes. For example, clouds can perform autoscaling accurately by using performance metrics inside VMs. They can also detect intrusion into VMs by analyzing system logs. To obtain system information inside VMs, they often use an agent method, which installs agent software in VMs and receives system information via network communication, as illustrated in Fig. 1(a). The agent can be implemented as a process or a kernel module. For example, the Amazon CloudWatch agent [1] is a process-based agent that collects system logs and metrics inside VMs to analyze logs and traces. An example of a kernel-based agent is the monitoring agent in IBM Cloud [2], which also collects information on executed system calls.

One issue with the agent method is that the users of VMs have to maintain the agent by themselves. They need to install the agent in VMs and periodically update it. If they neglect the agent updates, the agent could become new vulnerabilities that are attacked from the outside. Another issue is that there are trade-offs between process-based and kernel-based agents. The process-based agent does not largely affect the monitored system because it runs on top of the OS, whereas it can be easily disabled by intruders. In addition, it cannot collect information hidden in the kernel. In contrast, the kernel-based agent can be protected from intruders more strongly and collect information on the entire system. However, its vulnerabilities and instability could largely affect the entire system because it is embedded in the kernel.

On the other hand, a method called VMI has been proposed [3] to collect information from the outside of VMs, as illustrated in Fig. 1(b). For example, it can obtain the states of the system by analyzing the kernel data structures

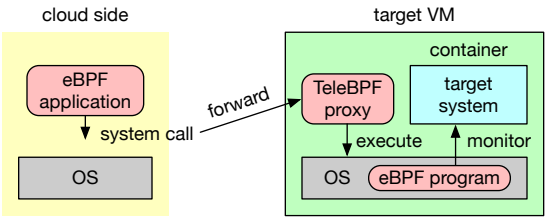


Figure 2. The system architecture of TeleBPF.

in the memory of VMs. It can also check the integrity of files and inspect configuration files by analyzing the file systems used in the virtual disks of VMs. Unlike the agent method, VMI does not need users to install or update the agent in VMs. Since there is no agent running inside VMs, VMI does not introduce new vulnerabilities or instability to the systems in VMs. In addition, it can access any information hidden in the kernel and the file systems.

However, VMI has various issues as well. First, it needs a low-level analysis of the memory and virtual disks of VMs. It is more difficult to develop monitoring tools with VMI. Second, VMI is too powerful for clouds to monitor users' VMs. Since it cannot be controlled by the systems in VMs, sensitive information could be easily stolen. Third, VMI is not applicable to confidential VMs, which are introduced to protect users' VMs from clouds [4]–[6]. Since the memory of confidential VMs is encrypted by TEEs such as AMD SEV and Intel TDX, clouds cannot analyze the memory to collect internal information. Similarly, VMI cannot analyze the virtual disks if they are encrypted inside VMs.

3. TeleBPF

This paper proposes TeleBPF to monitor the systems in VMs by injecting eBPF programs. eBPF is a mechanism provided in the Linux kernel and enables users to collect various kinds of information by loading programs into the OS. TeleBPF uses eBPF programs as safe agents. As illustrated in Fig. 2, TeleBPF transparently executes existing monitoring tools developed as eBPF applications on the cloud side and loads eBPF programs into the OS via the TeleBPF proxy in a VM. The eBPF programs collect information in the OS and return it to the eBPF applications via the proxy. Although the proxy needs to be installed in a VM, it provides only simple functions and rarely needs to be updated. TeleBPF confines the target system in a container provided by the OS and runs the proxy outside it to protect the proxy from intruders in the target system.

The threat model of TeleBPF is as follows. From the viewpoint of the users of monitored VMs, we do not trust clouds or eBPF programs injected by them. Also, we do not trust the target systems running in VMs, which attackers can intrude into. In contrast, we assume that the OS in a VM is trusted. This means that there are no vulnerabilities exploitable by intruders who reside in the target system. In addition, we assume that the TeleBPF proxy in a VM is

trusted. The attack surface against the OS and the proxy is narrowed down by running the target system in a container.

Unlike the traditional agent method, TeleBPF enables clouds to dynamically inject eBPF programs into VMs when necessary. The users of VMs do not need to install or update the agent by themselves. Since eBPF programs run in the OS, they can collect more information than the agent running as a process. Thanks to the protection of the OS, eBPF programs are not easily affected by external attackers. Conversely, dynamically loaded eBPF programs do not affect the OS kernel by the load-time check with the eBPF verifier. Unlike VMI, TeleBPF does not need the low-level analysis of VMs because eBPF programs run inside VMs. It can collect information inside VMs even if their memory and virtual disks are encrypted. In addition, the capabilities of the injected eBPF programs can be controlled by the OS inside VMs.

To transparently execute eBPF applications on the cloud side, TeleBPF forwards eBPF-related system calls to the TeleBPF proxy running in a VM. There are three types of eBPF-related system calls. First, the system calls for controlling eBPF are used to execute commands related to eBPF programs. Using these commands, users can load eBPF programs in the OS and manage BPF maps, which are used to pass data between an eBPF application and eBPF programs. Second, the system calls for controlling events are used to associate events that occur in the system with specific eBPF programs. Users can configure the system so that an eBPF program is invoked by an event, e.g., when a specific kernel function is executed. Third, the file-related system calls are used to access special files for eBPF. For example, such files provide information on trace points.

TeleBPF forwards such eBPF-related system calls to a target VM as follows. First, it intercepts a system call issued by an eBPF application. If the intercepted system call is for controlling eBPF, TeleBPF always forwards it to the target VM because this type of system call needs to be executed inside the VM. If the intercepted system call is for controlling events, TeleBPF checks its type and arguments and forwards it if necessary. This type of system call is also used to control events that are not related to eBPF. If the intercepted system call is for file access, TeleBPF determines whether it should be forwarded by checking its arguments because it is a generic system call. If TeleBPF forwards the system call to the VM, it serializes the arguments and converts them into a byte sequence. In the VM, the TeleBPF proxy deserializes the received data and issues the forwarded system call. Then, the results are forwarded back to the eBPF application.

In addition, TeleBPF supports data collection using ring buffers between an eBPF application and eBPF programs. eBPF applications often use ring buffers to efficiently obtain information from eBPF programs. A ring buffer is created in the kernel memory and shared with the process of an eBPF application. Since the eBPF application accesses the ring buffer without using any system calls, it is difficult for TeleBPF to forward that access to the target VM by intercepting system calls. TeleBPF could trap memory access and forward it, but a large performance degradation is inevitable. Therefore, TeleBPF shares the memory used for the ring buffer in the VM with the cloud side, so that the eBPF application can directly access the ring buffer. For this purpose, TeleBPF forwards the system call for sharing the ring buffer created in the kernel with a process to the VM. Then, it enables that memory to be accessed by the eBPF application outside the VM.

4. Implementation

We have implemented TeleBPF for Linux 5.15 and QEMU-KVM 8.0.0.

4.1. Intercepting System Calls

To intercept system calls, various techniques are used. The `ptrace` system call enables a monitoring process to intercept all the system calls issued by a target process. It intercepts system calls in the OS kernel and switches the contexts between the two processes. Therefore, the cost of the invocation of system calls becomes very large. The `LD_PRELOAD` environment variable enables the specified shared library to indirectly intercept the function calls that issue system calls in the target application. Since only the specified functions are replaced with the functions provided by the shared library, the overhead of intercepting system calls is much smaller. However, if a function to be replaced is an inline function, all the functions that invoke that inline function need to be replaced. This strongly depends on the implementation of the target application and the used libraries. In general, it is not easy to replace all the necessary functions. An eBPF-specific method is to directly modify the libraries for eBPF, e.g., libbpf [12] and libbcc [13]. This method can intercept inline functions that issue system calls as well. It is much more flexible, but it can support only specific eBPF frameworks. Also, it is a troublesome task to follow the updates of the frameworks.

Therefore, TeleBPF uses a new method called zpoline [10]. zpoline rewrites the binary of the target application at runtime. It is not possible to replace the invocation of system calls with arbitrary function calls because the instruction for invoking a system call is only two bytes. Therefore, zpoline replaces all the invocations of system calls with the invocations of the trampoline code. Then, the trampoline code invokes the function defined for each system call in the TeleBPF shared library, as illustrated in Fig. 3. This method can directly intercept system calls like the `ptrace` method. Nevertheless, it is much faster than the `ptrace` method, although it is slower than the `LD_PRELOAD` method. Note that we used the `LD_PRELOAD` method for the `faccessat` system call because zpoline crashes the interception of that system call.

4.2. Forwarding System Calls

After the TeleBPF shared library intercepts the invocation of a system call, it first transfers the number assigned to

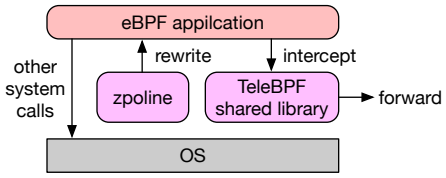


Figure 3. Intercepting the invocation of eBPF-related system calls.

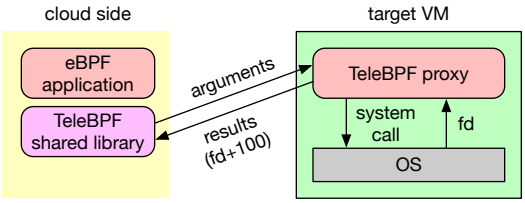


Figure 4. The details of the forwarding system calls.

the system call to the TeleBPF proxy running in the target VM. Then, it serializes the arguments of the system call using Protocol Buffers [14] and transfers the resultant byte sequence to the proxy, as illustrated in Fig. 4. For Protocol Buffers, TeleBPF defines messages for target system calls and target commands used by several system calls. In the VM, the proxy deserializes the passed arguments using Protocol Buffer. According to the received number of the system call, it issues the corresponding system call with the received arguments. It transfers the return value of the system call back to the shared library. If the execution of the system call fails, the proxy transfers the error number stored in the `errno` global variable as well. If any, it transfers data returned from the system call via its arguments. These transfers are performed using VM sockets (Vsock) [11], which are a communication mechanism specific to VMs. Vsock is used to communicate between the host and a VM and is faster than TCP/IP.

The TeleBPF proxy converts the value of the file descriptor returned from the forwarded system call. This enables the TeleBPF shared library to determine whether a system call using a file descriptor should be forwarded or not. The system call issued by the proxy can return a file descriptor via its return value or argument. The value of the file descriptor is uniquely distinguishable only in the proxy process. If the proxy simply returns the value to the eBPF application, the eBPF application cannot distinguish the returned file descriptor from that returned from a locally executed system call. Therefore, the proxy duplicates the file descriptor using the dup2 system call and creates a new file descriptor whose value is larger than the original by a pre-defined constant, e.g., 100. This file descriptor can be handled by the shared library as that returned from the forwarded system call.

4.3. Forwarding the bpf System Call

When an eBPF application issues the `bpf` system call, which is used for controlling eBPF, the TeleBPF shared library intercepts and forwards it. This system call supports various eBPF commands, which are specified by the first argument. Table 1 lists some of the eBPF commands. To distinguish each eBPF command, the shared library transfers a unique number assigned to each command, instead of the number of the system call. Each command uses the specific members of the `bpf_attr` union, which is also specified by the second argument. For efficiency, the shared library transfers only the necessary members of the union. In the target VM, the TeleBPF proxy creates a new `bpf_attr` union from the received data and issues the `bpf` system call with it. The proxy transfers the return value from the system call back to the shared library. Finally, the return value is passed to the eBPF application.

For example, an eBPF application issues the `bpf` system call with the `BPF_PROG_LOAD` command to load an eBPF program. The TeleBPF shared library intercepts this system call and transfers the type, name, byte code, etc. of the eBPF program, which are stored in the `bpf_attr` union of the argument, to the TeleBPF proxy. Then, it receives a file descriptor for manipulating the loaded eBPF program as the return value of the system call from the proxy. After that, the eBPF application can issue the `bpf` system call with the `BPF_RAW_TRACEPOINT_OPEN` command to attach an eBPF program to a trace point in the kernel. The shared library intercepts this system call and transfers the file descriptor of the loaded eBPF program and the name of a trace point, e.g., `sys_enter`, to the proxy. As a result, the eBPF program is executed whenever system calls are issued in the target VM.

An eBPF application can issue the `bpf` system calls to manipulate BPF maps, which consist of keys and values and are used to pass data between the eBPF application and eBPF programs. When it creates a new BPF map using the `BPF_MAP_CREATE` command, the TeleBPF shared library transfers a map type and the sizes of the key and the value, which are stored in the `bpf_attr` union, to the TeleBPF proxy. This information is necessary when the shared library forwards map-related eBPF commands such as `BPF_MAP_LOOKUP_ELEM` and `BPF_MAP_UPDATE_ELEM`, but it is not specified in the arguments of these commands. Therefore, both the shared library and the proxy save the sizes of the key and the value on the creation of the BPF map. Using the saved information, the shared library can transfer the key and the value, and the proxy can receive them.

4.4. Forwarding Event-controlling System Calls

There are various types of system calls for controlling events. For example, the `perf_event_open` system call is used to configure events for performance monitoring. As the first argument, it takes the `perf_event_attr` structure, which consists of the type of an event, the name and address of a monitored function, the identifier of a trace point, etc. When an eBPF application issues this system call, the TeleBPF shared library transfers the members of this structure to the TeleBPF proxy. Then, it receives a file descriptor for manipulating the event as a return value.

TABLE 1. THE EXAMPLES OF THE EBPF COMMANDS.

eBPF command	description
BPF_PROG_LOAD	Load an eBPF program and return a file descriptor associated with it
BPF_BTF_LOAD	Load BPF type format (BTF) metadata and return a file descriptor associated with it
BPF_RAW_TRACEPOINT_OPEN	Attach an eBPF program to a trace point and return a file descriptor managing the event
BPF_LINK_CREATE	Attach an eBPF program to a file descriptor and return a file descriptor managing the link
BPF_MAP_CREATE	Create a BPF map and return a file descriptor referring to it
BPF_MAP_LOOKUP_ELEM	Look up an element by key in a BPF map and return its value
BPF_MAP_UPDATE_ELEM	Update an element (a key and a value) in a BPF map
BPF_OBJ_GET_INFO_BY_FD	Return information about the eBPF object specified by a file descriptor

TABLE 2. THE EXAMPLES OF THE IOCTL OPERATIONS FOR PERFORMANCE MONITORING.

ioctl operation	description
PERF_EVENT_IOC_SET_BPF	Attach an eBPF program to the event of a kernel trace point
PERF_EVENT_IOC_ENABLE	Enable an event specified by a file descriptor
PERF_EVENT_IOC_DISABLE	Disable an event specified by a file descriptor

TABLE 3. THE EXAMPLES OF SPECIAL FILES USED BY EBPF.

special file	description
/sys/kernel/debug/tracing/*	Various information on events, e.g., trace point identifiers
/sys/devices/system/cpu/*	Various information on CPUs, e.g., online CPUs
/sys/kernel/btf/vmlinux	BTF information on the running kernel

The eBPF application issues the `ioctl` system call with the returned file descriptor to attach an eBPF program to that event and enable that event. Table 2 lists some of the `ioctl` operations for performance monitoring. The TeleBPF shared library intercepts this system call, but this system call is used for various purposes. Therefore, the shared library checks the value of the specified file descriptor and forwards this system call only if the value is larger than the pre-defined constant. This is possible because the TeleBPF proxy adds the pre-defined constant to the value of a file descriptor returned from the forwarded `perf_event_open` system call. Since the `ioctl` system call can take an arbitrary type of data as the third argument, the shared library transfers the data of an appropriate size according to the operation specified by the second argument. For example, a file descriptor of the integer type is specified for the `PERF_EVENT_IOC_SET_BPF` operation.

The epoll-related system calls are used to wait for events. An eBPF application uses these system calls to wait for eBPF programs to store information in ring buffers. When it issues the `epoll_create1` system call to create an epoll instance, the TeleBPF shared library transfers the flag to the proxy and receives a file descriptor for manipulating the created instance. Then, the eBPF application issues the `epoll_ctl` system call to add a file descriptor used for manipulating a ring buffer to the epoll instance. The shared library transfers the two file descriptors for the epoll instance and a ring buffer, and an event type to the proxy. After that, the eBPF application repeats the `epoll_wait` system call to wait for epoll events. When the waiting events occur in the target VM, the shared library receives the array of the `epoll_event` structures from the proxy. These system calls can be used for non-eBPF events, but it is not possible to identify the event type until the `epoll_ctl` system call is executed. In the current implementation, TeleBPF always forwards the epoll-related system calls.

4.5. Forwarding File-related System Calls

Table 3 lists some of the special files used by eBPF applications. There are two types of system calls for accessing eBPF-related files: one with a path name and one with a file descriptor. For a system call with a path name, the TeleBPF shared library first checks the path name. If the path name is for special files used for obtaining information on eBPF, e.g., events and CPUs, the shared library forwards that system call. For a system call with a file descriptor, the shared library checks the value of the file descriptor. If the value is larger than the pre-defined constant, it forwards that system call. For example, it forwards the `read` system call with the file descriptor returned by opening an eBPF-related file and receives the data in that file as well as the return value. For a system call with both a path name and a file descriptor for a directory, e.g., `openat` and `newfsstat`, the shared library checks both.

4.6. Forwarding the mmap System Call

The TeleBPF shared library intercepts the `mmap` system call to share ring buffers between an eBPF application and eBPF programs. An eBPF application first executes the `bpf` system call with the `BPF_MAP_CREATE` command to create a new ring buffer. Unlike a normal BPF map, it specifies `BPF_MAP_TYPE_RINGBUF` or `BPF_MAP_TYPE_PERF_EVENT_ARRAY` in the map type of the `bpf_attr` union. It receives a file descriptor for manipulating the created ring buffer via the TeleBPF shared library. Then, the eBPF application issues the `mmap` system call with that file descriptor to map the ring buffer created in the kernel onto its address space, as illustrated in Fig. 5.

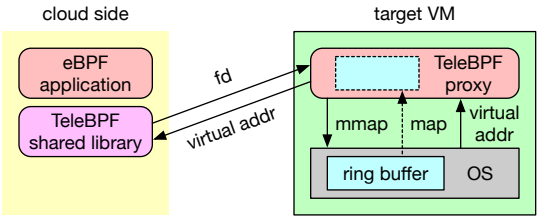


Figure 5. Mapping the memory of the ring buffer by mmap.

The shared library forwards this system call only if the value of the file descriptor is larger than the pre-defined constant. In the target VM, the TeleBPF proxy maps the ring buffer and returns its virtual address. However, the shared library cannot access the ring buffer in the VM using that virtual address because that address is of the TeleBPF proxy process.

To address this issue, the TeleBPF shared library first translates the received virtual address of the proxy into the physical address of the VM. Then, it accesses the memory of the VM using the physical address. To perform this address translation, it needs the page tables of the TeleBPF proxy process. Therefore, it obtains the physical address of the page tables from the proxy when it connects to the proxy. The proxy obtains that address using the kernel module because the process cannot obtain information on the page tables by itself. That kernel module accesses the `task_struct` structure used for the current process and finds the `mm_struct` structure used for memory management. Then, it translates the virtual address of the page tables into a physical one and returns it to the proxy.

To enable the shared library to access the page tables and the ring buffer in the memory of the VM, TeleBPF boots the VM using the memory backed by a file called a memory file. The shared library maps this memory file onto its address space. Since a ring buffer consists of multiple memory pages, the shared library translates the virtual address of each page into a physical address. Next, it maps the region of the memory file corresponding to each physical address onto its address space so that all the regions become contiguous, as illustrated in Fig. 6. Finally, it returns the virtual address of the entire region to the eBPF application. For confidential VMs, the memory of the page tables and the ring buffer needs to be unencrypted by the OS in the VMs.

5. Experiments

We conducted several experiments to confirm that existing monitoring tools could collect information on the system in a VM from the cloud side using TeleBPF. Also, we examined the overhead of TeleBPF. For comparison, we measured the performance of monitoring tools executed inside a VM without TeleBPF (in-VM execution). In addition, we measured the performance when we used TeleBPF with TCP/IP instead of Vsock. In this experiment, we used a PC with an Intel Core i7-10700 processor and 64 GB of memory. We ran Linux 5.15 and QEMU-KVM 8.0.0. We

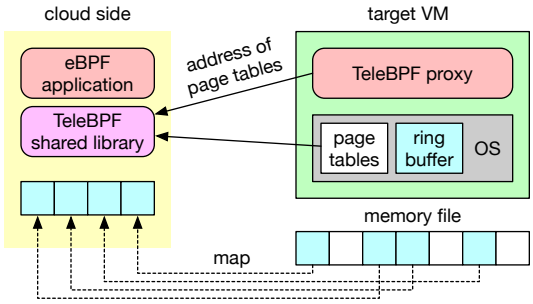


Figure 6. Mapping the ring buffer in a VM.

```
Event SYSMONEVENT_CREATE_PROCESS
        RuleName: -
        UtcTime: 2024-02-11 06:37:45.
        ProcessGuid: {c937e8a0-6b39-
        ProcessId: 18000
        Image: /usr/bin/ls
        FileVersion: -
```

(a) Sysmon

TIME(s)	T	BYTES	LAT(ms)
449231.197230000	R	0	0.61
449231.261284000	W	0	0.56
449231.261699000	W	0	0.37
449231.262067000	W	0	0.34
449231.262456000	W	0	0.36
449231.262745000	W	0	0.26

(b) Disksnoop

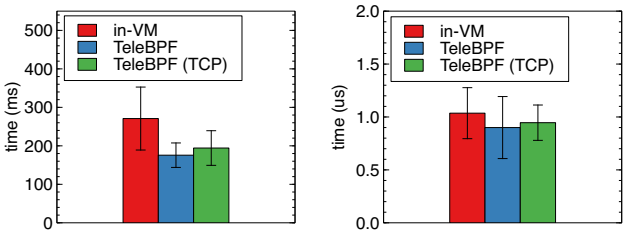
Figure 7. The results of monitoring tools with TeleBPF.

assigned one virtual CPU and 1 GB of memory to a VM and ran Linux 5.15. We used protobuf-c 1.3.3 [15] as Protocol Buffer for C.

5.1. System Monitoring

First, we ran Microsoft Sysmon for Linux [16] using TeleBPF. Sysmon is an eBPF application that monitors the creation and termination of processes, file writes, network connections, etc., and records those system events in a log file. It is written in C using libbpf [12]. In this experiment, we disabled network monitoring because TeleBPF has not yet supported network-related system calls. When we ran Sysmon outside the VM, Sysmon loaded its 19 eBPF programs into the VM. Fig. 7(a) is the log recorded on the cloud side by Sysmon when we executed commands in the VM. We confirmed that Sysmon with TeleBPF could monitor system events inside the VM correctly.

Next, we ran the monitoring tool of disk access named `disksnoop` using TeleBPF. This tool is a sample eBPF application included in BPF Compiler Collection (BCC) [13] and monitors the type, size, and latency of each disk access. It is written in Python and runs using libbcc. When we ran this monitoring tool outside the VM, the tool loaded one eBPF program written in C into the VM. We confirmed that TeleBPF enabled this tool to monitor disk access in the VM, as shown in Fig. 7(b).



(a) Initialization (b) Data collection

Figure 8. The performance of Sysmon.

As such, TeleBPF is applicable to various eBPF frameworks, including libbpf and BCC.

5.2. Performance of Sysmon

To examine the overhead of TeleBPF, we measured the time needed for Sysmon to perform initialization for eBPF 10 times. Fig. 8(a) shows the average initialization time and its standard deviation for TeleBPF with Vsock and TCP/IP and in-VM execution. From these results, it was shown that TeleBPF could initialize the eBPF-related task 54% faster than when we executed Sysmon in the VM. This is because the virtualization overhead of running Sysmon is larger than the overhead of forwarding system calls in TeleBPF. For communication, TeleBPF with Vsock was 11% faster than that with TCP/IP. The data transfer rate in forwarding system calls was 978 KB/s for Vsock and 885 KB/s for TCP/IP.

Next, we examined the performance of collecting information from its eBPF programs after initialization. We executed the `ls` command in the VM and measured the time needed for Sysmon to obtain that information from the ring buffer. Fig. 8(b) shows the data collection time. Even in this case, TeleBPF was 13% faster than in-VM execution. This is because TeleBPF enables Sysmon to directly share the ring buffer in the VM and can reduce the virtualization overhead.

5.3. Performance of System Calls

To examine the performance of the system calls used at initialization and data collection in Sysmon, we measured the execution time of eBPF-related system calls. For the `bpf` system call, we measured the execution time for each command. Sysmon executed only the `bpf` system call with `BPF_MAP_LOOKUP_ELEM` for data collection, while it executed the others for initialization. It executed the `bpf` system call with the `BPF_PROG_LOAD` command 19 times and loaded eBPF programs of various sizes. It also executed that system call with the `BPF_BTF_LOAD` command 9 times and loaded BPF type format (BTF) metadata of various sizes. Therefore, we calculated the mean execution time for each system call. We excluded the `epoll_wait` system call because its execution time was almost always 0.1 seconds, which was the specified timeout.

Fig. 9 shows its average and standard deviation. The execution time of most of the system calls was longer in TeleBPF. In particular, the execution time of the `mmap` system call was 15.8 ms longer because it took longer to share the ring buffer in the VM. In contrast, the `bpf` system call with the `MAP_CREATE` command was 11.3 ms shorter. This is due to extreme outliers in in-VM execution. If we excluded the outliers exceeding the 1.5x quartile range, the difference shrank to 1.4 ms.

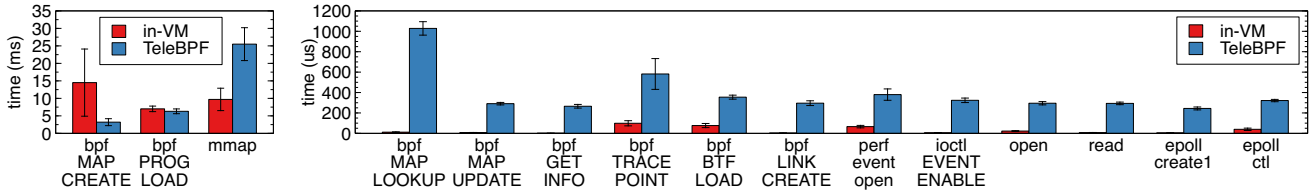
To examine the breakdown of the execution time of system calls, we measured the communication time, the execution time of the forwarded system calls, and the miscellaneous time. The communication time is the time needed to transfer the arguments of system calls and the results between the TeleBPF shared library and the proxy using Vsock. As shown in Fig. 10, the communication time was 75%, and the execution time of the forwarded system calls was 23% on average. From this result, the communication time was usually much longer than the others.

However, the execution time of the forwarded system calls was the longest in the `bpf` system call with the `BPF_MAP_CREATE` and `BPF_PROG_LOAD` commands and the `mmap` system call. In particular, the `mmap` system call took 4.2x longer than in-VM execution. In this system call, the miscellaneous time was more than 10%. This is because it took much time to translate the virtual address of each page in the ring buffer into a physical address and map the regions of the memory file corresponding to all the physical addresses. The size of the used ring buffer was 64 MB, i.e., 16,384 pages.

Next, we examined the performance improvement of the execution of system calls by using Vsock in TeleBPF. Fig. 11 shows the comparison of the execution time of system calls between Vsock and TCP/IP. Vsock made the execution of system calls 10% faster on average. In particular, the performance was largely improved in the `bpf` system call with the `BPF_MAP_LOOKUP_ELEM` commands.

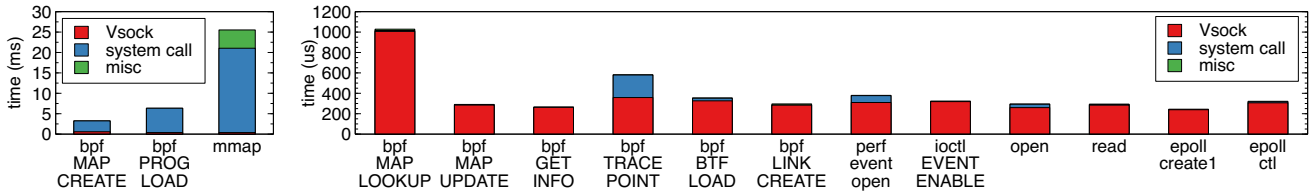
5.4. Performance of Data Collection

eBPF applications usually collect information from eBPF programs using BPF maps or ring buffers. For data collection from BPF maps, they issue the `bpf` system call with the `BPF_MAP_LOOKUP_ELEM` command. For data collection from ring buffers, they do not issue any system calls. To compare the performance of data collection, we measured the time needed to collect information from a BPF map and a ring buffer. Fig. 12(a) shows the collection time for a BPF map. The collection time in TeleBPF was 199 μs longer than in in-VM execution. Using Vsock improved the performance by 5x, compared with using TCP/IP. The collection time for a ring buffer is shown in Fig. 12(b). TeleBPF was 1.8x faster than in-VM execution. Both TeleBPF and in-VM execution could enable obtaining information directly from the ring buffer, but virtualization overhead degraded the performance when the eBPF application ran in the VM.



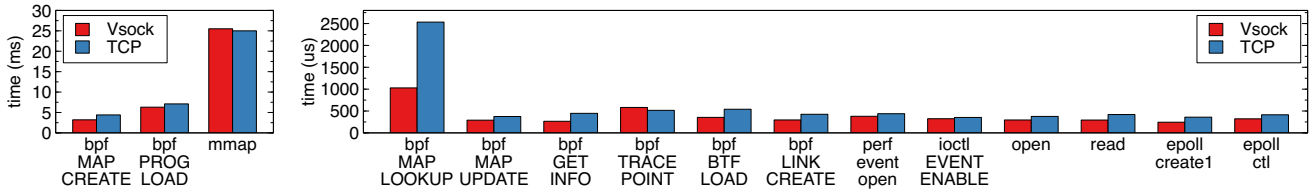
(a) long system calls (b) short system calls

Figure 9. The execution time of system calls.



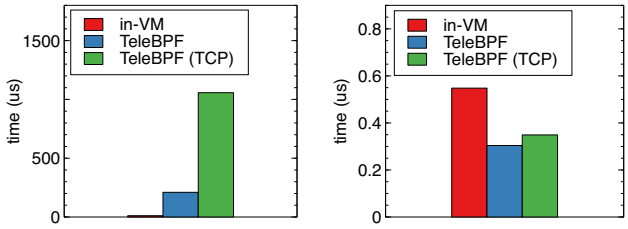
(a) long system calls (b) short system calls

Figure 10. The breakdown of the execution time of system calls.



(a) long system calls (b) short system calls

Figure 11. The performance improvement of system calls by Vsock.



(a) BPF map (b) Ring buffer

Figure 12. The performance of data collection.

6. Related Work

BPFd [17] enables the applications developed with BPF Compiler Collection (BCC) [13] to collect information from a remote host. It was developed to use the functionalities of BCC in Android devices where BCC cannot be installed. It forwards BCC commands for loading eBPF programs and creating BPF maps from a BCC application to the BPFd daemon running in a remote host. Then, the BPFd daemon executes the forwarded commands. BPFd is similar to TeleBPF, but it forwards the function calls of the BCC library. Therefore, it is subject to changes in the implementation of BCC. In contrast, TeleBPF forwards the invocation of the more primitive system calls and is not affected by the development of BCC. Since the specification of the system calls is rarely changed and usually keeps backward compatibility, TeleBPF is more generic than BPFd. As a result, TeleBPF is applicable to any eBPF framework including BCC. In addition, BPFd cannot support ring buffers because it assumes running eBPF programs at a remote host.

Sysdig [18] is used as a monitoring agent in IBM Cloud [2] and monitors the execution of system calls by installing the kernel module in VMs. Recently, it achieves the same function as eBPF. Using eBPF programs enables more secure monitoring, while the monitoring performance degrades. Therefore, Sysdig still provides the kernel module as well. Even if it uses eBPF programs, users need to install and update Sysdig in their VMs. In TeleBPF, the cloud side can maintain the entire monitoring tools.

Several frameworks have been proposed to make the development of monitoring tools with VMI easier. VMST [19] enables users to run existing monitoring tools in a monitoring VM and monitor the system in the target VM. It automatically identifies the data needed for VMI and redirects that data access to the memory of the target VM. LLView [20] enables users to develop monitoring tools using the source code of the OS. It converts the developed programs at compile time so that they access the kernel data in the memory of the target VM. Using the proc file system

developed with LLView, existing monitoring tools can be run outside the target VM. However, these tools for VMI cannot be used for confidential VMs.

SEVmonitor [21] enables intrusion detection systems (IDS) to monitor the systems inside confidential VMs using VMI. Since traditional VMI cannot obtain information from the encrypted memory of a VM, SEVmonitor runs a small agent inside the VM and obtains memory data from it. To protect the agent inside the VM, SEVmonitor confines the target system in a container and runs the agent in the OS kernel. This system is a bit similar to TeleBPF, but eBPF programs are much safer than the agent implemented in the kernel.

Similarly, 00SEVen [22] securely runs an agent for VMI using VM Privilege Levels (VMPLs) provided by AMD SEV-SNP in a confidential VM. VMPLs enable a VM to divide its address space into four levels. The agent runs in the highest privilege called VMPL0, while the target system runs in a lower privilege. The agent can obtain not only the memory data but also the state of virtual CPU registers of the target system. In addition, it supports event-based analysis by trapping memory access and calls to kernel functions. However, the agent cannot be dynamically injected or updated, unlike TeleBPF. In addition, 00SEVen cannot be applied to confidential VMs with the other types of TEEs such as Arm CCA [23] because it strongly relies on VMPLs in SEV-SNP. TeleBPF does not depend on such a specific hardware mechanism.

7. Conclusion

This paper proposed TeleBPF for safely and transparently monitoring VMs by dynamically injecting eBPF programs. When an eBPF application on the cloud side issues eBPF-related system calls, the TeleBPF shared library intercepts them. Then, it forwards them to the TeleBPF proxy in the target VM using the VM-specific communication mechanism. The proxy executes the forwarded system calls and returns the results to the eBPF application. TeleBPF enables the eBPF application to directly and efficiently obtain information from the eBPF programs via the ring buffers created in the target VM. The experimental results show that TeleBPF can run monitoring tools faster than traditional in-VM execution.

One of our future work is to support more eBPF-related system calls and special files in TeleBPF. This is necessary to run various existing monitoring tools developed as eBPF applications. In addition, we are planning to protect the TeleBPF proxy in a VM, e.g., by running it in the OS kernel.

Acknowledgment

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

References

[1] Amazon Web Services, Inc., "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/.

[2] IBM Corporation, "IBM Cloud Monitoring," https://www.ibm.com/products/cloud-monitoring.

[3] 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.

[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 Devices, Inc., "Secure Encrypted Virtualization API Version 0.24," 2020.

[8] Intel Corp., "Intel Trust Domain Extensions," 2023.

[9] A. Starovoitov and D. Borkmann, "eBPF – Introduction, Tutorials & Community Resources," https://ebpf.io/, 2014.

[10] K. Yasukata, H. Tazaki, P. Aublin, and K. Ishiguro, "Zpoline: A System Call Hook Mechanism Based on Binary Rewriting," in *USENIX Annual Technical Conf.*, 2023, pp. 293–300.

[11] S. Hajnoczi, "virtio-vsock: Zero-configuration Host/guest Communication," KVM Forum 2015, 2015.

[12] "libbpf," https://github.com/libbpf/libbpf.

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

[14] Google Developers, "Protocol Buffers," https://developers.google.com/protocol-buffers.

[15] D. Benson, "protobuf-c: Protocol Buffers Implementation in C," https://github.com/protobuf-c/protobuf-c.

[16] Microsoft Corp., "Sysmon for Linux," https://github.com/Sysinternals/SysmonForLinux.

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

[18] Sysdig, Inc., "Sysdig: Security Tools for Containers, Kubernetes, and Cloud," https://sysdig.com/.

[19] 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*, 2012, pp. 586–600.

[20] 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.

[21] T. Nono and K. Kourai, "Secure Monitoring of Virtual Machines Protected by AMD SEV in Public Clouds," Int. Symp. Applied Engineering and Sciences, 2022.

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

[23] Arm Limited, "Arm Confidential Computing Architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2023.