P4 Shield: Secure Execution of Users' P4 Programs with In-VM Information inside Clouds

Masaki Iwai
Kyushu Institute of Technology
iwai@ksl.ci.kyutech.ac.jp

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

***Abstract*—Recently emerging programmable network switches can control packet forwarding by software, e.g., using the P4 language. For virtual machines (VMs), virtual P4 switches have been developed to execute P4 programs *inside* virtual switches. If users could load their own P4 programs into virtual P4 switches, it would enable custom packet forwarding using information inside their VMs. In clouds, however, users cannot fully trust virtual P4 switches, nor can virtual P4 switches trust users' P4 programs. To address these issues, this paper proposes *P4 Shield* for enabling the secure execution of each user's P4 programs *outside* clouds' virtual switches by using *P4 VMs* prepared per user. P4 Shield protects P4 programs from clouds by running P4 VMs as *confidential VMs*, whose states are protected by processors. It also protects clouds' virtual switches from users' P4 programs by confining them in *uBPF sandboxes* created in P4 VMs. To allow P4 programs to use information inside users' VMs, P4 Shield provides *P4 external functions* for accessing information in the shared memory between a P4 VM and a user's VM. We have implemented P4 Shield using Open vSwitch and confirmed its effectiveness.**

***Index Terms*—P4, uBPF, Open vSwitch, confidential VM, cloud computing.**

I. Introduction

Recently, it has become possible to program packet forwarding in network switches using the P4 language [1]. Such P4 switches can perform advanced packet filtering and rewriting in the data plane. They can enable emerging technologies without waiting for the release of new network switches. They are also used to validate new network protocols that aim for future standardization. In addition to physical P4 switches, virtual switches supporting P4 have also been developed [2], [3]. Virtual P4 switches are used to connect virtual machines (VMs) to virtual networks and execute P4 programs for all the packets from and to VMs *inside* the switches.

Currently, P4 programs are loaded into virtual P4 switches by network or host administrators. If users could load their own P4 programs, more flexible packet forwarding would be possible for each VM. In addition, if P4 programs could use information inside users' VMs, advanced packet processing would be achieved. In clouds, however, virtual switches are provided by clouds and are not necessarily trustworthy for users. For example, clouds could tamper with the P4 programs loaded by users and eavesdrop on in-VM information obtained by the P4 programs. Conversely, users' P4 programs could negatively affect virtual switches if they have some issues [4].

This paper proposes *P4 Shield* for enabling the secure execution of each user's P4 programs *outside* clouds' virtual switches by using *P4 VMs* prepared per user. P4 Shield prevents attacks against users' P4 programs from clouds by running P4 VMs as *confidential VMs*. The states of confidential VMs are encrypted and integrity-checked by processors using trusted execution environments (TEEs) such as AMD SEV [5] and Intel TDX [6]. In addition, P4 Shield protects clouds' virtual switches from the abnormal behavior of users' P4 programs by confining P4 programs in *uBPF sandboxes* [7] created in P4 VMs. uBPF is a framework for safe program execution with load-time verification.

In P4 Shield, a virtual switch first passes a received packet to an appropriate P4 VM to execute P4 programs and then forwards the packet according to the execution results. At this time, P4 Shield allows P4 programs to use information inside users' VMs (user VMs). Since P4 programs cannot access in-VM information directly, P4 Shield provides dedicated *external functions* to obtain in-VM information. The external function in P4 is used to execute programs defined outside P4 programs. In-VM information is stored in the shared memory established between a P4 VM and a user VM. To access the shared memory, the P4 external functions invoke a helper function provided by the uBPF runtime.

We have implemented P4 Shield using Open vSwitch [8], the uBPF runtime, and confidential VMs with AMD SEV. We conducted several experiments to show the effectiveness of P4 Shield. First, we examined whether P4 Shield could perform advanced packet filtering using in-VM information on TCP memory. The results show that P4 Shield could deny new TCP connections when the used TCP memory exceeded a custom threshold in a user VM. We also measured the latency and throughput of TCP and UDP communications in a user VM and confirmed that performance degradation due to P4 Shield was not so large for the secure execution of P4 programs.

The remainder of this paper is organized as follows. Section II presents challenges for allowing users to load P4 programs into virtual switches in clouds. Section III proposes P4 Shield to securely execute users' P4 programs using in-VM information in P4 VMs. Section IV explains the implementation of P4 Shield, and Section V describes experiments to show its effectiveness. Section VI presents related work, and

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

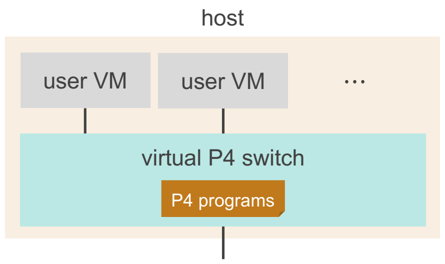

Fig. 1. A virtual P4 switch.

Section VII concludes this paper.

II. Virtual P4 Switches in Clouds

The P4 language [1] is used for the programming of the network data plane. It is a domain-specific language and enables programmers to define the behavior of the data plane in a high-level and declarative manner. A P4 program takes packet data as input and analyzes the packet headers in the parser block. Then, it executes a pipeline of packet processing, such as packet filtering and the modification of packet headers, in the control blocks. Finally, it reconstructs packet data from the modified packet headers in the deparser block. Such P4 programs are executed in P4 network switches.

In addition to physical P4 switches, P4-compatible virtual switches such as P4rt-OVS [2] and IPDK Networking Recipe [3] have been developed for VMs. In a virtualized environment, a virtual switch is created inside each host, which runs VMs on top of the hypervisor. It connects the virtual NICs of all the VMs running in the same host. Then, a virtual network is constructed by connecting virtual switches across multiple hosts via physical NICs. Each virtual switch receives a packet from a VM or a physical NIC in the same host and forwards it to another VM or a physical NIC. Virtual P4 switches execute P4 programs on packet forwarding, as illustrated in Fig. 1.

Since P4 programs are handled by virtual switches, which are a part of virtualized infrastructure, they are usually loaded into virtual switches only by network or host administrators. For example, P4rt-OVS provides the ovs-ofctl command, which needs administrator privileges to be executed. If users, i.e., the owners of VMs, could load their own P4 programs into virtual switches, more flexible packet forwarding could be performed for each VM. Furthermore, if users' P4 programs could use information inside their VMs, advanced packet processing would become possible. One example of such packet processing is packet filtering based on information about processes and users transmitting and receiving packets [9].

However, there are three challenges for enabling such user-created P4 programs using in-VM information to be executed in virtual P4 switches. First, virtual P4 switches provided in clouds are not necessarily trustworthy for users. If they are managed by untrusted cloud administrators, they could mount various attacks against P4 programs loaded by users. If eavesdropping on P4 programs themselves or in-VM information obtained by P4 programs, they could breach the confidentiality of the P4 programs or user VMs. If tampering with P4 programs to perform unintended packet forwarding, they could also breach the integrity of the P4 programs. Furthermore, the availability is lost if P4 programs are not executed for all packets in virtual P4 switches.

Second, P4 programs loaded by users could negatively affect virtual P4 switches if they are defective or malicious. If they consume large amounts of resources such as CPU and memory in virtual P4 switches, there could be a delay in forwarding all packets. If vulnerabilities exist in P4 programs or the P4 runtime executing them, the control of virtual P4 switches may be taken over. In fact, there are several reports of P4 vulnerabilities [4]. Vulnerable P4 programs fall into infinite loops and discard all subsequent packets, direct packets that do not correctly specify destination ports to specific ports, or leak information from previously forwarded packets under certain conditions.

Third, P4 programs can access only information contained in packet headers and payloads. To enable P4 programs to use in-VM information, it is necessary to extend P4 so that P4 programs can obtain information from user VMs. One possible solution is to add an API for network communication with user VMs. However, it is difficult to achieve sufficient performance of packet forwarding in virtual P4 switches because of communication overhead. To reduce this overhead, it is possible to add an API for VM introspection [10], which directly accesses the memory of user VMs and analyzes the data structure of the operating system (OS). Unfortunately, this method is not applicable if user VMs are running as confidential VMs [11], [12], whose memory is encrypted by processors. In this case, P4 programs cannot obtain information in the memory of user VMs.

The threat model in this paper is as follows. We do not trust clouds, including virtual switches and administrators. However, we assume that TEE hardware and the software running in confidential VMs are trusted and free of vulnerabilities. We consider attacks in which clouds eavesdrop on and tamper with users' P4 programs and information obtained from user VMs. Clouds could also mount attacks against the communication path for P4 programs to obtain information from user VMs. Furthermore, they could bypass the execution of users' P4 programs. In addition, we consider attacks in which users' P4 programs adversely affect virtual switches, such as resource exhaustion.

III. P4 Shield

P4 Shield enables clouds' virtual switches to securely execute each user's P4 programs using dedicated VMs called *P4 VMs*, which are prepared per user. When a virtual switch receives a packet related to a user VM, it first passes the packet to the corresponding P4 VM via shared memory, as shown in Fig. 2. Then, the P4 VM executes P4 programs and returns a decision on whether to forward or discard, and, if necessary, a rewritten packet data, to the virtual switch. When P4 programs need information inside the user VM for the

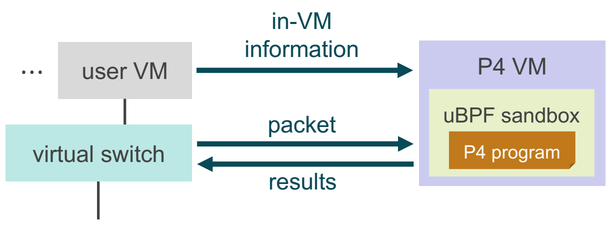

Fig. 2. The system architecture of P4 Shield.

decisions, they obtain the necessary information from the user VM using shared memory. Finally, the virtual switch forwards or discards the packet.

P4 Shield protects users' P4 programs from clouds by running P4 VMs as *confidential VMs*. Confidential VMs are protected by trusted execution environments (TEEs), e.g., AMD SEV [5] and Intel TDX [6], provided in recent processors. Since the memory and register states of P4 VMs are encrypted, clouds cannot eavesdrop on P4 programs and in-VM information obtained by P4 programs. Since their integrity is also maintained, clouds cannot tamper with P4 programs. Unlike the traditional virtual P4 switches, virtual switches are protected from users' P4 programs by executing no P4 programs inside the virtual switches. Even if P4 programs exhibit abnormal behavior, P4 Shield confines that negative impact in P4 VMs by the isolation of VMs.

Since P4 Shield can execute multiple P4 programs in one P4 VM, it uses *uBPF sandboxes* [7] to isolate P4 programs from each other. uBPF is a userspace implementation of eBPF [13], which is a framework used in Linux and Windows to safely run programs loaded into the OS kernel. P4 Shield compiles a user's P4 program into uBPF bytecode and loads it into the uBPF runtime running in a P4 VM. At this time, the uBPF runtime verifies the bytecode so that unsafe instructions are not executed. Simultaneously, it compiles the bytecode into native code using just-in-time (JIT) compilation to execute the P4 program efficiently. Since an isolated execution environment is provided for each uBPF bytecode, P4 programs do not affect each other or virtual switches.

To enable P4 programs to obtain information from user VMs, P4 Shield uses the shared memory between a P4 VM and each user VM. While communication via shared memory is already more efficient than network communication, P4 Shield further eliminates real-time interaction. A user VM stores information needed by P4 programs in the shared memory periodically. P4 programs can then obtain that information from the shared memory immediately. The shared memory is not encrypted by processors because encrypted memory cannot be shared between VMs. Therefore, clouds can eavesdrop on or tamper with information in the shared memory. To prevent such attacks, a user VM encrypts information and stores it with its message authentication code (MAC) in the shared memory. A P4 VM decrypts the information in the shared memory and verifies its integrity. These VMs securely exchange the cryptographic keys necessary for this self-protection in advance.

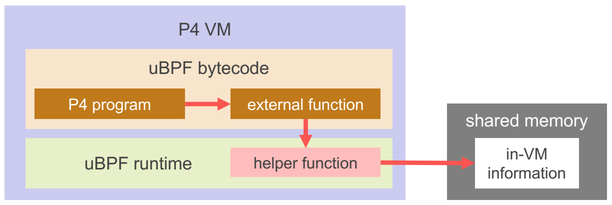

Fig. 3. Obtaining in-VM information via external functions.

P4 Shield allows P4 programs to access in-VM information in the shared memory by providing dedicated *external functions*, as illustrated in Fig. 3. Since the external function is included in the specification of the P4 language, P4 Shield does not need to extend the specification. The P4 external functions are compiled into uBPF bytecode, which is linked to a P4 program that is also compiled into uBPF bytecode. However, they cannot access the shared memory where in-VM information is stored due to uBPF sandboxes. Therefore, the P4 external functions invoke a *helper function* provided by the uBPF runtime extended for P4 Shield. Then, the helper function accesses the shared memory and returns the specified in-VM information to P4 programs via the P4 external functions.

To enable users to verify whether P4 programs are executed properly, P4 VMs record the execution logs of P4 programs in the shared memory with user VMs. Clouds' virtual switches could avoid executing users' P4 programs by performing packet forwarding without passing packets to P4 VMs. To detect such attacks, user VMs periodically obtain the execution logs stored in the shared memory and compare them with the statistics of packet transmission and reception. If the execution logs do not match the statistics, it is possible that P4 programs are not executed properly. Since clouds could tamper with the execution logs in the shared memory, P4 VMs store the MAC computed from the execution logs in the shared memory. Then, user VMs compare them with the re-computed ones to detect tampering.

IV. Implementation

We have implemented P4 Shield using Open vSwitch (OVS) 3.2.1 [8], the Data Plane Development Kit (DPDK) 22.11.7 [14], the uBPF runtime [7], and confidential VMs with AMD SEV running on KVM.

A. *Extension of OVS*

P4 Shield extends the datapath of OVS to send packet data to P4 VMs and receive the execution results of P4 programs from them. The datapath is the forwarding plane that processes and forwards network packets. OVS provides several datapaths such as the userspace datapath, the kernel datapath, and the DPDK datapath. We have implemented P4 Shield in the userspace datapath and the DPDK datapath. This paper focuses on the implementation in the DPDK datapath because the userspace datapath is less efficient. DPDK enables OVS to bypass the OS kernel and process packets only in

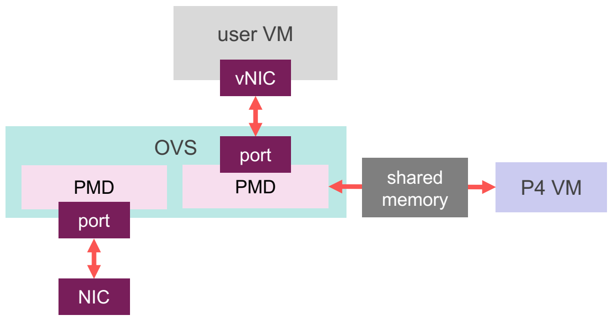

Fig. 4. The extended DPDK datapath in OVS.

userspace, as depicted in Fig. 4. It can reduce the overhead of interrupts and system calls and achieve efficient packet processing. In DPDK, a poll mode driver (PMD) occupies a necessary number of CPU cores and polls the transmit (TX) and receive (RX) queues of physical and virtual NICs.

When OVS receives a packet from a physical or virtual NIC using a PMD thread, it copies the first 64 bytes of the packet to the shared memory established with a P4 VM. OVS finds an appropriate P4 VM according to the source or destination IP address of the packet. Since OVS can receive up to 32 packets in batches, it passes all the packet data to P4 VMs at once. Then, it waits for the P4 VMs to return the execution results of P4 programs by polling the shared memory using a PMD thread. Finally, it re-creates a new batch of packets according to the received decisions and forwards it.

B. Extension of the uBPF Runtime

P4 Shield extends the uBPF runtime running in P4 VMs to execute P4 programs using in-VM information in coordination with OVS. It loads uBPF bytecode generated from a P4 program into the uBPF runtime in advance. After initialization, the uBPF runtime polls the shared memory between OVS and the P4 VM and waits until OVS stores packet data in the shared memory. When receiving packet data, it executes the uBPF bytecode with the packet data. The packet data in the shared memory is directly passed to the bytecode without data copies. After finishing the execution, the uBPF runtime stores the results in the shared memory to return them to OVS.

The uBPF runtime provides a new helper function named `read_vm_info()` to the uBPF bytecode. This helper function accesses in-VM information stored in the shared memory established between a P4 VM and a user VM. Since in-VM information is encrypted with AES-GCM by a user VM, the helper function decrypts it and verifies the authentication tag. The helper function takes an offset from the top of the shared memory as an argument and returns the value stored in that memory region. The uBPF runtime identifies the user VM to be accessed by the source or destination IP address of the packet header passed from OVS. P4 Shield defines which in-VM information is stored in each offset of the shared memory. The uBPF runtime assigns a unique number to each helper function and registers the mapping between the number and the function pointer. When uBPF bytecode executes the

```
extern bit<64> get_tcp_mem();

control pipe(inout Headers_t hdr, ...) {
  bit<64> tcp_mem;
  bit<32> threshold;
  bit<32> used_bytes;

  apply {
    if (hdr.ipv4.protocol == IP_PROTO_TCP &&
        (hdr.tcp.flags & TCP_SYN_MASK) != 0 &&
        (hdr.tcp.flags & TCP_ACK_MASK) == 0) {
      tcp_mem = get_tcp_mem();
      threshold = tcp_mem[63:32];
      used_bytes = tcp_mem[31:0];
      if (threshold < used_bytes)
        mark_to_drop();
    }
  }
}
```

Fig. 5. An example of a P4 program using an external function.

`call` instruction by specifying the number as an operand, instead of a target address, the uBPF runtime searches for the corresponding function pointer and invokes the helper function.

C. P4 External Functions for In-VM Information

P4 Shield provides external functions dedicated to P4 programs to obtain in-VM information stored in the shared memory. In P4, an external function can be used to execute a program defined outside a P4 program, e.g., a program written in the C language. It is declared with the `extern` keyword and is invoked like normal functions written in P4. It takes arguments if necessary and returns some value. Each P4 external function invokes the helper function, `read_vm_info()`, provided by the uBPF runtime to access the shared memory. It specifies the offset corresponding to the necessary in-VM information for the helper function.

Fig. 5 shows an example of a P4 program using an external function named `get_tcp_mem()`. After the P4 program analyzes a packet header in the parser block, it executes the control block named `pipe`. The control block takes the data structure of a parsed packet header as an argument. Then, the P4 program first checks whether the packet is for a TCP connection request. Specifically, it checks whether the protocol is TCP in the IPv4 header and whether the control flag includes SYN but not ACK in the TCP header. If so, the P4 program executes `get_tcp_mem()` and obtains information on TCP memory in a user VM as a 64-bit value. From the obtained value, it extracts the upper 32 bits as the threshold and the lower 32 bits as the amount of used TCP memory. If the amount of used TCP memory exceeds the threshold, the P4 program executes `mark_to_drop()` and directs the packet to be discarded.

P4 Shield compiles a P4 external function written in C into uBPF bytecode using the C compiler, clang. The generated bytecode is linked to the uBPF bytecode that a P4 program is compiled into. To compile a P4 program, P4 Shield first translates it to a C program using the uBPF-targeted compiler named p4c-ubpf, which is included in the P4 reference compiler named p4c [15]. We extended this compiler to generate the definition of the function pointer for the helper function added to the uBPF runtime. The value of this function pointer is set to the number assigned to the helper function, instead of the address of a function in C. Next, it compiles the obtained C program into uBPF bytecode. The invocation of the helper function is compiled into the `call` instruction with the value of the function pointer as its operand.

D. Shared Memory for Confidential VMs and OVS

P4 Shield uses two types of shared memory. One is between OVS and a P4 VM, and the other is between a P4 VM and a user VM. To create a shared memory object, P4 Shield runs the ivshmem server [16] on the host. This server is provided by QEMU, which is a device emulator for VMs. Then, P4 Shield configures VMs to provide a virtual PCI device for accessing shared memory. To enable access to the virtual PCI device in the userspace of VMs, it loads the ivshmem-uio driver [17] into the Linux kernel and provides the userspace I/O (UIO) device.

OVS maps the shared memory object onto its process address space. It writes packet data to the shared memory and reads the execution results of P4 programs from the shared memory. In a P4 VM, the uBPF runtime maps the UIO device for accessing the same shared memory onto its process address space. At this time, the ivshmem-uio driver clears the C-bit of the page table entries for that memory region so that the shared memory is not encrypted. In addition, the uBPF runtime maps another UIO device for accessing the shared memory established with a user VM. It reads information stored by the user VM from that shared memory. In the user VM, a userspace tool maps the UIO device for accessing that shared memory. It writes information needed by P4 programs to the shared memory. Also, the Linux kernel remaps the memory region of the virtual PCI device for accessing the shared memory onto its address space. Using this remapped memory region, the kernel also writes in-VM information.

V. Experiments

We conducted several experiments to show the effectiveness of P4 Shield. We first confirmed that P4 Shield could perform advanced packet filtering using in-VM information. In addition, we measured the communication performance of a user VM to investigate the overhead introduced by P4 Shield. We used two servers shown in Table I and connected them to a 10 Gigabit Ethernet switch. In each server, we ran a P4 VM and a user VM.

TABLE I
The experimental environments.

	Host 1		Host 2	
CPU	Intel Core i7-12700		AMD EPYC 7713P	
memory	64 GB		256 GB	
NIC	Intel X540-AT2		Broadcom 57416	
OS	Linux 6.8			
hypervisor	QEMU-KVM 6.2.0			
	P4 VM	user VM	P4 VM	user VM
virtual CPU	6	4	20	12
memory	4 GB	1 GB	32 GB	32 GB
guest OS	Linux 5.15		Linux 6.8	

A. Advanced Packet Filtering

For advanced packet filtering in a virtual switch, we used a P4 program that discarded new packets for TCP connection requests when available TCP memory was expected to run out in the target user VM. We explained part of this P4 program in Section IV-C. In the user VM, the Linux kernel manages the amount of used TCP memory and configures three thresholds: low, pressure, and high. It does not limit memory allocation for TCP while the amount of used TCP memory is less than the low threshold. If it becomes larger than the pressure threshold, the kernel starts to suppress memory consumption for TCP. When the used TCP memory exceeds the high threshold, TCP packets are discarded. Unlike this default policy in Linux, the used P4 program denies only new TCP connections when the used TCP memory exceeds the custom threshold.

In the user VM, a userspace tool periodically stored information on TCP memory in the shared memory established with a P4 VM. Specifically, it read data from `/proc/net/sockstat` and wrote a custom threshold and the value of used_bytes to the shared memory. A remote client continuously sent SYN packets to a server running in the user VM via the virtual switch and requested new TCP connections. The results show that the client could connect to the server while the used TCP memory was less than the threshold. When the amount of used TCP memory exceeded the threshold, only the requests of new TCP connections were rejected by the virtual switch, while existing TCP connections were maintained.

B. Communication Performance

We measured the latency and throughput of TCP and UDP communications in P4 Shield using netperf 2.7.0 [18]. The netperf client in a remote host sent packets to the netperf server in the user VM. We used the P4 program described in Section V-A, but we slightly modified it because netperf establishes only one TCP connection for the measurement. The modified P4 program obtained in-VM information for all the packets, not only for SYN packets in TCP. To prevent any packets from being discarded, the user VM stored a sufficiently large threshold as in-VM information. For comparison, we used traditional OVS without P4 support. In this experiment, we used Host 1.

Fig. 6 shows the 90th percentile latency when we specified a message size of one byte to the client. Compared with traditional OVS, the latency increased by 23% in TCP and 15% in UDP. This is the overhead of passing a packet to the P4 VM, executing the P4 program, and obtaining in-VM information. Fig. 7 shows the throughput of TCP and UDP communications when we increased the message size up to

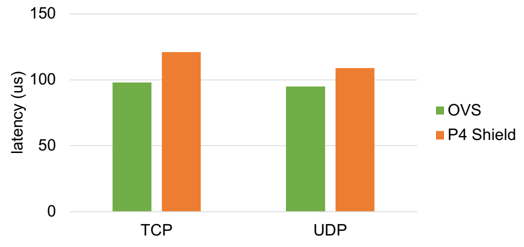

Fig. 6. The 90th percentile latency in P4 Shield.

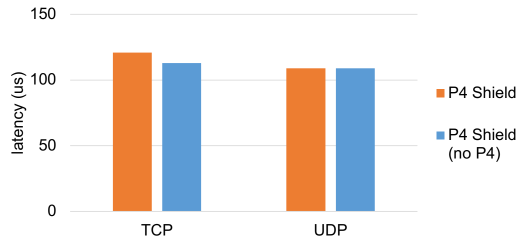

Fig. 8. The impact of P4 processing on the latency.

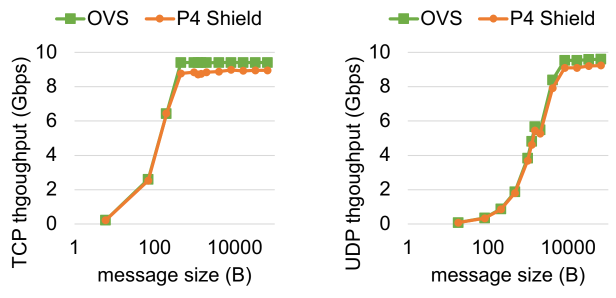

Fig. 7. The throughput of TCP and UDP communications in P4 Shield.

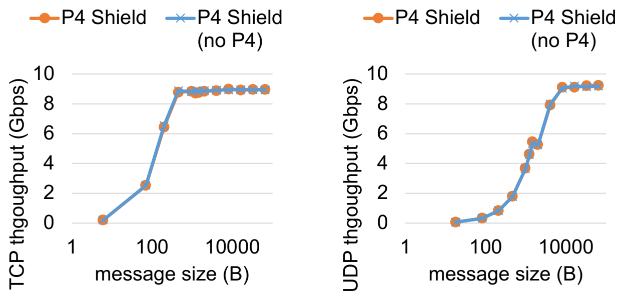

Fig. 9. The impact of P4 processing on the throughput.

about 64 KB. The throughput degradation was only 5% in TCP and 4% in UDP on average. This relatively small effect on throughput degradation can be attributed to parallel processing in the virtual switch, which hid the overhead of the P4 switch.

Next, we examined the impact of the execution of the P4 program on communication performance. For comparison, we did not load the P4 program in the P4 VM. Instead of the execution results of the P4 program, the P4 VM always returned a decision of forwarding to the virtual switch. Fig. 8 shows that the execution of the P4 program increased the latency by 7% in TCP. However, the latency in UDP was not affected by the P4 program. Similarly, the throughput was almost the same, as shown in Fig. 9.

Finally, we investigated the breakdown of the execution time of the P4 program. Fig. 10 shows the medians of each processing time. P4 processing, other than the execution of the external function, accounts for 68% of the total time. The decryption of in-VM information performed in the external function accounts for 27%. However, the total execution time was only 421 ns and much smaller than the increase in latency. This means that the communication overhead between OVS and the P4 VM dominates the overhead of P4 Shield.

C. Communication Performance with Confidential VMs

We measured the latency and throughput of TCP and UDP communications when we ran the P4 VM and the user VM as confidential VMs with AMD SEV. In this experiment, we used Host 2. As shown in Fig. 11 (left), the latency increased by 46% in TCP and 53% in UDP. These increases were much larger than the results in Host 1. To examine whether this was caused by the overhead of confidential VMs, we measured the latency when we ran the P4 VM as a normal VM. Since the latency did not decrease, as in Fig. 11 (right), the overhead of using a confidential VM for the P4 VM was negligible. We also ran the user VM as a normal VM, but it showed negligible differences in latency. This means that the overhead of P4 Shield comes from the other differences between Host 1 and Host 2.

Fig. 12 shows the throughput, indicating that even traditional OVS did not achieve satisfactory performance. Even when the client increased the message size, the throughput failed to approach 10 Gbps. For TCP, the throughput of traditional OVS was 4% higher than that of P4 Shield on average, but that of P4 Shield was higher in many message sizes. For UDP, the throughput of P4 Shield was almost always higher. The throughput steeply decreased when the message size became larger. These trends were similar when we ran the P4 VM and the user VM as normal VMs. From these results, we conclude that Host 2 could not achieve sufficient performance using OVS for some reasons.

D. Security Analysis

Clouds could attempt attacks to eavesdrop on and tamper with users' P4 programs or in-VM information obtained by the P4 programs. In P4 Shield, P4 programs are isolated in P4 VMs, whose memory and registers are protected by the TEE. This prevents clouds from breaching the confidentiality and integrity of users' P4 programs and user VMs. Note that the TEE cannot protect the shared memory used for passing information from user VMs to P4 VMs. To guarantee the confidentiality of that communication, information is encrypted in user VMs and decrypted in P4 VMs by themselves in P4 Shield. The integrity is verified using the MAC.

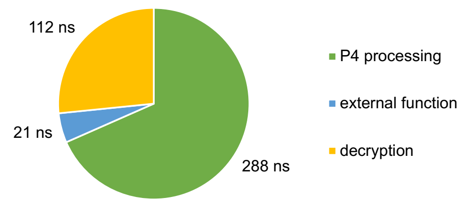

Fig. 10. The breakdown of the execution time of the P4 program.

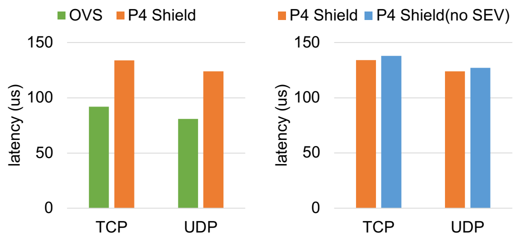

Fig. 11. The latency in P4 Shield with confidential VMs.

Users could attempt attacks against virtual P4 switches by loading malicious P4 programs if vulnerabilities exist in the P4 runtime running in virtual switches. They could also execute P4 programs that consume a large amount of resources in virtual switches to cause delays in packet forwarding. In P4 Shield, users' P4 programs are confined to P4 VMs and isolated from virtual switches. Therefore, the abnormal behavior of the P4 programs cannot affect virtual switches. In addition, P4 Shield prohibits illegal access and infinite loops using the load-time verifier of the uBPF runtime running P4 programs.

Clouds could easily disable users' P4 programs by not executing them in virtual switches. As a result, they could forward packets to be discarded to user VMs or discard packets to be forwarded. In P4 Shield, P4 VMs record the execution logs of P4 programs to detect this attack. Then, user VMs can detect the possibility that P4 programs are not properly executed by comparing the logs with the statistics of packet transmission and reception. P4 Shield protects the shared memory used for passing the logs from P4 VMs to user VMs by encryption and the MAC.

VI. Related Work

P4rt-OVS [2] is a virtual P4 switch based on Open vSwitch. P4 programs are executed in the virtual switch whenever a packet arrives. They are also compiled into uBPF bytecode using p4c-ubpf. Unlike P4 Shield, the uBPF runtime is embedded into the virtual P4 switch. uBPF protects the virtual switch from the abnormal behavior of P4 programs to some degree. However, the virtual switch itself is compromised if the uBPF runtime contains vulnerabilities. In addition, P4rt-OVS assumes that administrators load P4 programs into the virtual switch. It does not allow P4 programs to be loaded by users or to use information inside users' VMs.

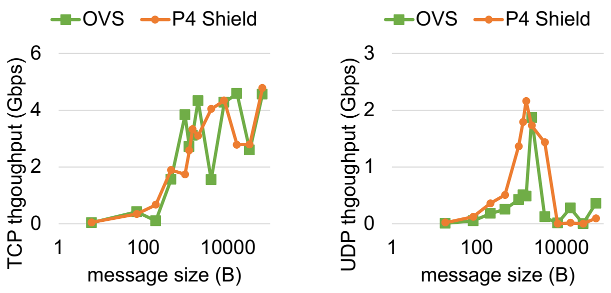

Fig. 12. The throughput in P4 Shield with confidential VMs.

SGX-Box [19] enables the secure monitoring of encrypted traffic inside middleboxes using Intel SGX, which is one of the trusted execution environments (TEEs). Unlike AMD SEV, SGX enables applications to create protected memory regions called enclaves. SGX-Box securely decrypts packets using session keys maintained inside an SGX enclave and executes in-network functions. To handle encrypted traffic, SGX-Box provides a high-level programming language called SB lang. SB lang hides the details of the analysis of the TLS handshake, packet reconstruction, decryption, and re-encryption from developers. It is similar to the P4 language and assumes that network administrators load programs written in SB lang into the SGX enclave.

EndBox [20] enables the secure execution of middlebox functions, e.g., firewalls and intrusion detection, on the client side. It uses Intel SGX to protect middlebox functions. An EndBox client running on the client side maintains a key for encrypted communication inside an SGX enclave. Then, it connects to an EndBox server in the organization or provider that the client belongs to using a virtual private network (VPN). Since client applications cannot communicate without going through the EndBox server, the use of EndBox is enforced.

Several proposals exist for fine-grained packet filtering using information inside VMs. VMwall [9] runs in the management VM of Xen and analyzes the memory of user VMs using VM introspection [10]. Then, it searches for the process that transmits or receives a packet from the port number included in the packet header. If the process name is not in the whitelist, VMwall discards the packet. xFilter [21] is similar to VMwall, but its critical component runs in the hypervisor for efficiency. xFilter obtains information using VM introspection and dynamically generates filtering rules. The in-VM information used in these systems can also be used by P4 programs in P4 Shield.

VII. Conclusion

This paper proposed P4 Shield for enabling the secure execution of users' P4 programs using in-VM information in P4 VMs. In P4 Shield, virtual switches pass received packets to P4 VMs to execute P4 programs and perform packet forwarding according to their execution results. P4 Shield protects users' P4 programs from clouds by running P4

VMs as confidential VMs and protects clouds' virtual switches by confining users' P4 programs in uBPF sandboxes created in P4 VMs. After user VMs store the necessary information in shared memory, P4 programs obtain it using P4 external functions via the uBPF helper function. We have implemented P4 Shield on Open vSwitch and confirmed its usefulness and communication performance.

One of our future work is the improvement of communication performance in the server equipped with an AMD EPYC processor, which supports confidential VMs. Since the performance was much higher in the PC with an Intel Core processor, we need to find performance bottlenecks in that server. Additionally, it is necessary to investigate which information should be made available to P4 programs and what APIs should be provided.

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[2] T. Osiński, H. Tarasiuk, P. Chaignon, and M. Kossakowski, "P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4," in *Proc. IFIP Networking Conference*, 2020, pp. 413–421.

[3] Open Programmable Infrastructure Project, "IPDK Networking Recipe (P4 Control Plane)," https://github.com/ipdk-io/networking-recipe.

[4] M. Dumitru, D. Dumitrescu, and C. Raiciu, "Can We Exploit Buggy P4 Programs?" in *Proc. Symp. SDN Research*, 2020, pp. 62–68.

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

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

[7] IO Visor Project, "Userspace eBPF VM," https://iovisor.github.io/ubpf/.

[8] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proc. Symp. Networked Systems Design and Implementation*, 2015, pp. 117–130.

[9] A. Srivastava and J. Giffin, "Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections," in *Proc. Recent Advances in Intrusion Detection*, 2008, pp. 39–58.

[10] 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.

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

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

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

[14] Intel Corp., "DPDK – The Open Source Data Plane Development Kit," https://www.dpdk.org/.

[15] P4 Compiler Project, "P4C," https://github.com/p4lang/p4c.

[16] The QEMU Project Developers, "Inter-VM Shared Memory Device," https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[17] C. Macdonell, "ivshmem-uio," https://github.com/shawnanastasio/ivshmem-uio.

[18] R. Jones, "Netperf Benchmark," https://hewlettpackard.github.io/netperf.

[19] J. Han, S. Kim, J. Ha, and D. Han, "SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module," in *Proc. Asia-Pacific Workshop on Networking*, 2017, pp. 99–105.

[20] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza, "EndBox: Scalable Middlebox Functions Using Client-side Trusted Execution," in *Proc. Int. Conf. Dependable Systems and Networks*, 2018, pp. 386–397.

[21] K. Kourai, T. Azumi, and S. Chiba, "Efficient and Fine-grained VMM-level Packet Filtering for Self-protection," *Int. J. Adaptive, Resilient and Autonomic Systems*, vol. 5, no. 2, pp. 83–100, 2014.