Journal of Network and Computer Applications 198 (2022) 103283

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Design and implementation of an intrusion detection system by using Extended BPF in the Linux kernel

Shie-Yuan Wang [*], Jen-Chieh Chang

Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan

ARTICLE INFO

Keywords:
Intrusion detection system (IDS)
eBPF
Snort

ABSTRACT

An intrusion detection system (IDS) checks the content of headers and payload of packets to detect intrusions from the network. It is an essential function for network security. Traditionally, an IDS, such as Snort, which is a widely used open source IDS, is implemented as a program running in the user space on a hardware server. Recently, with the availability of Extended BPF (eBPF) in the Linux kernel, efficiently checking and filtering arriving packets directly in the kernel becomes feasible. In this work, we design and implement an IDS that has two parts working together. The first part runs in the Linux kernel. Its uses eBPF to perform fast patterns matching to pre-drop a very large portion of packets that have no chance to match any rule. The second part runs in the user space. It examines the packets left by the first part to find the rules that match them. Using a modified version of the registered ruleset of Snort, experimental results show that the maximum throughput of our IDS system can outperform that of Snort by a factor of 3 under many tested conditions.

1. Introduction

The extended BSD Packet Filter (eBPF) (Anon, 2021e) is an in-kernel virtual machine that is widely used for packet filtering and system monitoring. It utilizes hooks in the kernel so that it can customize some behaviors of the kernel for different purposes. One of the hooks is the Express Data Path (XDP) (Høiland-Jørgensen et al., 2018), which allows an eBPF program to be attached to a network device driver. With XDP, one can use customized programs to process packets at a very early stage. eBPF offers several advantages. Firstly, compared to other kernel bypassing techniques like Data Plane Development Kit (DPDK) (Anon, 2021d), eBPF closely cooperates with the existing network stack. Thus, one can selectively use the data structures or functions provided by the network stack in eBPF to greatly save development time and effort. Secondly, eBPF can achieve high performance for data-plane applications. Since XDP can process packets before they enter the higher layers of the network stack, the processing overhead of XDP is relatively low. Finally, eBPF does not require dedicating full CPU cores to packet processing. In contrast, to use DPDK, the network driver must operate in the polling mode, which occupies full CPU cores and wastes lots of computing power when the traffic load is low. Since eBPF uses the interrupt-based mechanism, the CPU usage is proportional to the traffic load. In short, eBPF provides better performance and CPU power efficiency than existing network stacks for a variety of network applications.

Intrusion Detection System (IDS) is a network function that examines packets for detecting network attacks. An IDS often has to perform Deep Packet Inspection (DPI) on packets, which significantly impacts its throughput. For example, to detect and forward the packets containing URLs to a tool that can use machine learning methods to classify whether a URL is malicious or not (Johnson et al., 2020), DPI needs to be performed by the IDS. In a production environment where the performance is critical, hardware-based IDS is preferred because of their high throughput of processing packets. At the other end, software-based IDS has the advantages on the cost and scalability. Recently, as network function virtualization (NFV) (Mijumbi et al., 2016) and service function chains (SFC) (Thang and Park, 2020) become popular, IDS software gradually draws more attention. There have been several mature open-source IDS software such as Snort (Anon, 2021l) and Suricata (Anon, 2021n). However, currently the performance of most IDS software is still low. To mitigate this problem, we propose an approach that integrates an eBPF program running in the kernel and a program running in the user space.

This paper makes two major contributions. Firstly, we proposed an eBPF-based approach to design and implement our IDS system. Although there have been several tools written in eBPF, they only perform simple tasks such as parsing packet headers and then accepting or rejecting packets based on the 5-tuple information in the header. In contrast, in addition to filtering packets based on the information in the

[*] Corresponding author.
E-mail address: shieyuan@cs.nctu.edu.tw (S.-Y. Wang).

https://doi.org/10.1016/j.jnca.2021.103283
Received 30 May 2021; Received in revised form 26 October 2021; Accepted 15 November 2021
Available online 30 November 2021

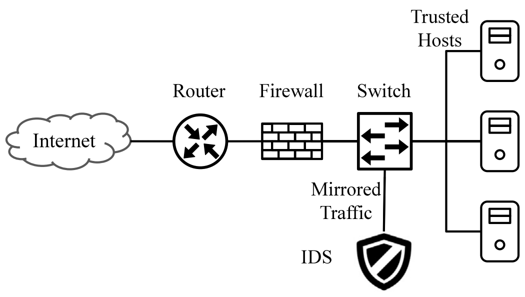

Fig. 1. A typical usage of IDS.

header, the eBPF program of our IDS system can match arbitrary patterns in parallel at any position of the packet payload. Secondly, using a modified version of the registered ruleset of Snort, our experimental results show that the maximum throughput of our system was almost three times that of Snort under many tested conditions.

The rest of the paper is structured as follows: Section 2 presents a typical usage of IDS. Section 3 compares related work. Section 4 is a brief introduction to eBPF, XDP, Aho–Corasick algorithm, and Snort. Section 5 presents the design and implementation of our IDS system. Section 6 presents the experimental setup and Section 7 presents the results of the performance evaluation. Finally, Section 8 discusses future work and Section 9 concludes the paper.

2. Network architecture

Fig. 1 shows a typical usage of IDS. In this example, it is assumed that network attacks come from the Internet. Starting from the left-hand side, the malicious traffic may go through the router and the firewall. If the traffic is not blocked by the firewall, it may reach the trusted hosts in the local area network (LAN). The IDS is attached to the switch and it does not block any traffic. All the traffic passing through the switch is mirrored a copy and the copy is sent to the IDS for examinations. If any copy matches certain patterns, the IDS will generate alerts to inform the network administrator.

3. Related work

Nowadays, network attacks occur frequently and have different motives (Abhishta et al., 2020). How to efficiently filter and block attack packets becomes very important. Both Snort (Anon, 2021m) and Suricata (Leblond and Manev, 2019) have the ability to filter packets with eBPF. However, they only parse up to the layer-3 header and neither of them can use eBPF to match patterns in the packet payload. Snort can specify a tcpdump-style filter expression by using the -F command-line option. The expression will later be converted into eBPF instructions. Similarly, eBPF is used for three tasks in Suricata: packet filtering, load balancing, and XDP-based flow bypassing. The major difference between our system and Suricata is that our system can use customized eBPF programs rather than simple expressions. This feature provides great flexibility for the users. Besides, our system can examine the payload of packets with eBPF. In contrast, Suricata does not perform pattern matching in the eBPF context. Thus, currently it is impossible to perform DPI in Suricata with eBPF. The authors in Baidya et al. (2018) implemented an eBPF-based DPI scheme to detect the types of video frames carried in the packets. However, it can only process fixed-format packets, which is inadequate for an IDS. ebpfH (Findlay, 2019) is a host-based IDS that uses eBPF. However, it detects system anomalies rather than network anomalies.

eBPF is widely used in many data-plane applications. The authors in Vieira et al. (2020) made a thorough survey of eBPF. This survey provides technical details as well as the areas that are utilizing eBPF. Many basic network functions have been rewritten with eBPF, such as switching (Viljoen and Kicinski, 2018), routing (Xhonneux et al., 2018), and firewalls (Miano et al., 2019a). Some advanced features of eBPF enable people to develop complex applications such as load balancing (Anon, 2021f), key–value storage (Lazri et al., 2019), application-level filtering (Anon, 2021b), and Distributed Denial-of-Service (DDoS) mitigation (Miano et al., 2019b). Specifically, researches of eBPF related to NFV are booming in recent years. In-KeV (Ahmed et al., 2018) is an in-kernel framework for building network functions. It can construct in-kernel service function chains (SFC) using the tail call feature of eBPF. In Miano et al. (2018), the authors discussed the limitations of eBPF and their experiences with eBPF. In Scholz et al. (2018), the authors measured the performance of using eBPF to filter packets based on the 5-tuple information in the packet header. Nowadays, since 5G networks are becoming popular, the authors in Parola et al. (2021) implemented an open-source 5G mobile gateway based on eBPF. The authors in Hong et al. (2018) used eBPF for Inter-VM (virtual machine) traffic monitoring. In Miano et al. (2021), the authors proposed a framework to implement eBPF-based network functions for microservices. Although there have been many existing network functions based on eBPF, none of them implements complex string searching algorithms such as Aho–Corasick (Aho and Corasick, 1975) algorithm to support DPI.

The authors in Miano et al. (2018) pointed out several key factors affecting the performance of eBPF, especially when implementing complex network functions. However, since eBPF is being continuously improved, some limitations have been lifted after the article was published. Although the authors in Scholz et al. (2018) analyzed the performance of eBPF as well, this paper only focused on an eBPF-based firewall, which is simpler than an IDS. The paper of Hohlfeld et al. (2019) provided a detailed analysis of the performance of XDP in virtual machine (VM) and hardware offloading environments and showed that XDP inside VMs suffered performance penalties.

Since the emergence of Snort, many researches have been focused on how to optimize it. String matching is the key component in an IDS and has a great impact on its performance. Snort adopts both Boyer–Moore (BM) (Boyer and Moore, 1977) and Aho–Corasick (AC) algorithms for string matching. The BM algorithm can perform a single pattern matching in $O(n+m)$ complexity, where n, m is the length of the text and the pattern, respectively. On the other hand, the AC algorithm can match multiple patterns by scanning the text only once. Obviously, the latter performs more efficiently than the former when the number of patterns is large. Although many research efforts have been spent on the optimization of Snort, so far very few of them utilize the power of eBPF.

4. Background

4.1. Introduction to eBPF and XDP

eBPF originates from the classical BPF (cBPF) (McCanne and Jacobson, 1993), which was proposed by Steven McCanne and Van Jacobson in 1993. The original use case of cBPF is packet filtering. Tcpdump (Anon, 2021h) is a great example of using cBPF. The command line expressions passed to tcpdump are first parsed and converted to cBPF assembly code and then executed by the BPF virtual machine inside the kernel. Although cBPF was a great success in the beginning, the limited features of it soon became an obstacle for future development. Linux introduced eBPF to replace cBPF in version 3.18 (Anon, 2021g). Compared to cBPF, eBPF enhances virtual hardware and high-level language support. The new ability of attaching eBPF programs to various trace points in the kernel greatly expands the application areas of it.

The express data path (XDP) is a special trace point inside the Linux kernel. Fig. 2 shows the architecture of eBPF and XDP. The driver maintains a ring buffer for storing incoming packets. Right after the

Table 1
The transition table of $he, she, his, hers$.

State	Transitions			Matches
0	h→1	s→7	fail→0	
1	e→2	i→5	fail→0	
2	r→3	fail→0		he
3	s→4	fail→0		
4	fail→7			hers
5	s→6	fail→0		
6	fail→7			his
7	h→8	fail→0		
8	e→9	fail→1		
9	fail→2			she, he

packets are copied from the Network Interface Card (NIC) hardware to the ring buffer, they will be fetched and processed by the BPF program. There are three possible outcomes for the packets: drop, receive local, or forward. If the packet is dropped, it will be removed from the ring buffer without further processing. If the packet is intended to be received locally, it will be passed to the upper layers of the network stack. Otherwise, the packet can be forwarded back to the NIC and re-transmitted. Once the packet is received locally, the BPF program can specify a CPU to perform further processing on it. This means that eBPF can steer packets for load balancing between multiple CPU cores. Finally, there should be a control application in the user space to load and configure the BPF program.

eBPF maps are generic data structures for storage of different data types. They are located inside the kernel and can be accessed by both eBPF programs and control applications in the user space. Because they are similar to shared memory, eBPF maps can act as the communication medium between eBPF programs and control applications. In addition to eBPF maps, eBPF programs can also send data to the user space by outputting perf events, which are data structures used by the Linux perf tool.

4.2. Introduction to the Aho–Corasick algorithm

The Aho–Corasick (AC) algorithm (Aho and Corasick, 1975) was proposed by Alfred V. Aho and Margaret J. Corasick in 1975. It is an efficient algorithm to locate all occurrences of a finite number of keywords in a string of text. The algorithm consists of two phases. One is constructing a state machine from the keywords, and the other is using the state machine to process the text string in a single pass. This algorithm is especially suitable for IDSes because they have to search the occurrences of multiple patterns in one string (i.e., the packet data) at high speed. Since our eBPF program implements the AC algorithm to pre-drop packets that have no chance to match any rule, in the following we briefly introduce it.

4.2.1. Phase one: Building the state machine

The first phase can be further divided into three steps: building the trie (i.e., prefix tree), adding fail links, and converting the state machine into a deterministic finite automaton (DFA). Here is a simple example drawn from the original paper. There are four patterns in total: $he, she, his, hers$. The process of building the trie is shown in Algorithm 1 and the result is shown in Fig. 3. All patterns are added to the trie one by one. The order of the patterns does not matter since the same sets of patterns will be converted to identical tries.

Algorithm 1 Build a trie from keywords

```
Input: A list containing all patterns $P$
Output: A trie $T$
1: procedure BuildTrie($P$)
2:     $T.root \leftarrow newNode$
3:     for each $P_i$ in $P$ do
4:         AddWordToTrie($P_i$)
5: procedure AddWordToTrie(p)
6:     $currentNode \leftarrow T.root$
7:     for each character $c$ in $p$ do
8:         if $currentNode.next[c] = NULL$ then
9:             $currentNode.next[c] \leftarrow newNode$
10:        $currentNode \leftarrow currentNode.next[c]$
```

Next, fail links are added to the trie constructed in the previous step. Algorithm 2 shows the algorithm for adding fail links. The fail link of the root points to itself, and those of the root's children point to the root, too. After that, fail links are added to all other states with a breadth-first search. After adding all fail links, the transition table is shown in Table 1. Now, let us define a new term called fail paths. The fail path of a state is a path that starts from the state and follows the fail links until reaching the root state. For example, the fail path of state 4 is 4 → 7 → 0. To find all matched patterns of a state, we have to traverse the fail path of the state. For instance, the fail path of state 9 is 9 → 2 → 0 and state 9 and state 2 match patterns she and he, respectively. Thus, state 9 matches both patterns.

Algorithm 2 Add fail links

```
Input: The trie $T$ from Algorithm 1
Output: The state machine $M$ with fail links
1: procedure AddFailLinks(T)
2:     $T.root.fail \leftarrow T.root$
3:     $queue \leftarrow empty$
4:     for each $child$ in $T.root.children$ do
5:         $child.fail \leftarrow T.root$
6:         $queue.push(child)$
7:     while $queue \neq empty$ do
8:         $node \leftarrow queue.pop()$
9:         FindFailLinkForNode($T$, $node$)
10:        for each $child$ in $node.children$ do
11:            $queue.push(child)$
12: procedure FindFailLinkForNode($T$, $node$)
13:    $node.fail \leftarrow NULL$
14:    $currentNode \leftarrow node.parent.fail$
15:    $c \leftarrow$ the character $\ni currentNode.next[c] = node$
16:    while $currentNode \neq T.root$ do
17:        if $currentNode.next[c] \neq NULL$ then
18:            $node.fail \leftarrow currentNode$
19:            break
20:        $currentNode \leftarrow currentNode.fail$
21:    if $node.fail = NULL$ then
22:        $node.fail =\leftarrow T.root$
```

The last step is turning the state machine into a DFA so that the transitions of all possible input characters can be performed in one single step. Algorithm 3 shows the process and Table 2 shows the DFA version of the transition table that is the output of the algorithm. Similarly, nodes are visited in a breadth-first order. In other words, the 256 possible transitions of each node are precomputed so that

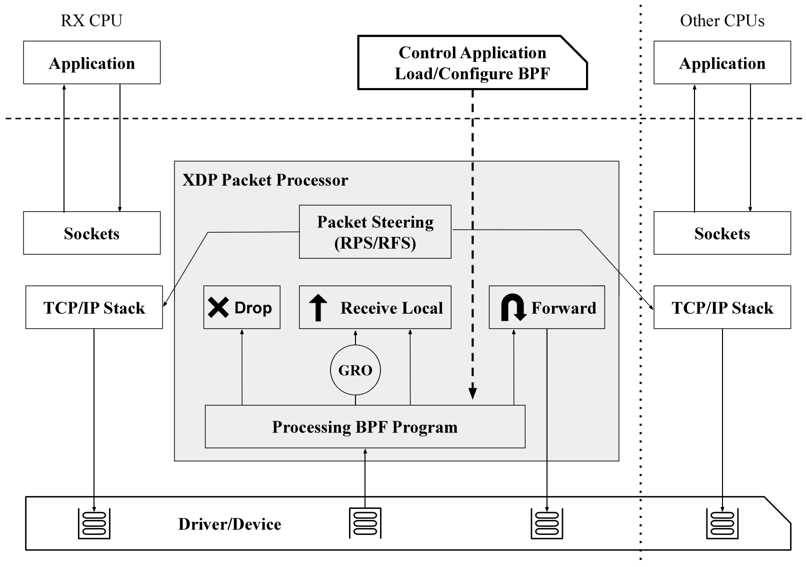

Fig. 2. The architecture of eBPF and XDP.

we can process an input character in one transition. (Note that since an character is 8-bit long, there can be 256 different characters.) In Table 2, the dot symbols represent characters other than the characters specified on their right sides. For example, state 0 goes to state 1, 7 for input character h, s, and characters other than h or s should go to state 0.

Algorithm 3 Compute the DFA

```
Input: The state machine $M$ from Algorithm 2
Output: A state machine $M$ with all transitions determined
 1: procedure COMPUTEDFA($M$)
 2:     $queue \leftarrow empty$
 3:     $queue.push(M.root)$
 4:     while $queue \neq empty$ do
 5:         $node \leftarrow queue.pop()$
 6:         for each $c \in [0, 255]$ do
 7:             if $node.next[c] = NULL$ then
 8:                 COMPUTENEXTMOVE($M, node, c$)
 9:         for each $child$ of $node$ do
10:             $queue.push(child)$
11: procedure COMPUTENEXTMOVE($M, node, c$)
12:     $currentNode \leftarrow node.fail$
13:     while $currentNode \neq M.root$ do
14:         if $currentNode.next[c] \neq NULL$ then
15:             $node.next[c] \leftarrow currentNode.next[c]$
16:             break
17:         $currentNode \leftarrow currentNode.fail$
18:     if $currentNode = M.root$ then
19:         if $currentNode.next[c] \neq NULL$ then
20:             $node.next[c] \leftarrow currentNode.next[c]$
21:         else
22:             $node.next[c] \leftarrow M.root$
```

4.2.2. Phase two: Pattern matching

Algorithm 4 shows the procedure used to match patterns in a string with the AC algorithm. The algorithm loops through each character of the input string. For each character, the algorithm first processes the matches in the current state and then performs a state transition. For instance, an input string "*shehe*" produces a series of transitions $0 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 1 \rightarrow 2$ and two matched patterns *he*, *she*. There are three factors that can influence the efficiency of matching. One is the length of the input string (n) since the for loop in Algorithm 4 Line 2 is bounded by it. The longer the length of the input string, the more iterations the loop will run. The second factor is the total number of occurrences of patterns in the input string (z). The algorithm will consume more time if more patterns are matched in the string. The last factor is the total length of the patterns (m). Each character in a pattern takes a constant amount of time to be output, so the complexity of outputting patterns is $O(m)$. To summarize, the total time complexity of the matching algorithm is $O(n + z + m)$.

Table 2
The DFA transition table.

State	Transitions					Matches
0	h→1	s→7	.→0			
1	e→2	i→5	h→1	s→7	.→0	
2	r→3	h→1	s→7	.→0		he
3	s→4	h→1	.→0			
4	h→8	s→7	.→0			hers
5	s→6	h→1	.→0			
6	h→8	s→7	.→0			his
7	h→8	s→7	.→0			
8	e→9	i→5	h→1	s→7	.→0	
9	r→3	h→1	s→7	.→0		she, he

Algorithm 4 Match patterns in an input string *str*

```
Input: The state machine $M$ from Algorithm 3, the input string
Output: Patterns matched in the string
1: procedure MATCHPATTERNS($M, str$)
2:     $state \leftarrow M.root$
3:     for each character $c$ in $str$ do
4:         for each $pattern$ matched in $state$ do
5:             Output $pattern$
6:         $state \leftarrow state.next[c]$
```

4.3. Introduction to Snort

Snort is an open-source IDS proposed by Martin Roesch in 1998 (Roesch, 1999). It can run under three different modes: the *inline*, *inline–test*, and *passive* modes. When Snort is in the *inline* mode, it acts as an intrusion prevention system (IPS). If packets hit some rules, they may be dropped in this mode. For the *inline–test* mode, packets are analyzed like in the *inline* mode, but they will not be dropped. For the

Table 3
The number of rules in each ruleset.

	Community (Default)	Registered	Emerging Threats
Total	3917	50 979	27 982
Uncommented	1087	12 630	18 493
Commented	2830	38 349	9489

passive mode, there are three configuration modes available: the sniffer, packet logger, and network intrusion detection system (NIDS) modes. For the sniffer mode, Snort just captures packets and displays them on the screen. For the packet logger mode, packets are saved to files. For the NIDS mode, Snort performs detailed analysis on the packets against a set of rules defined by the user. If any intrusion is detected, Snort will output alerts to the console or log files. In the paper, we used the NIDS configuration mode to compare the performance of our system and Snort.

4.3.1. The rulesets of Snort

Table 3 shows the three rulesets of Snort that were available when we conducted this research. The community (or default) ruleset is shipped with the Snort package. It can be downloaded from the official web site of Snort without any charge or registration. Unfortunately, it only contains about four thousand rules, which is unable to cover all common threats. The registered ruleset can be downloaded from the web site and it includes much more rules than the community ruleset. However, the user needs to register with the web site to get this ruleset. The authors of Snort suggest that users should use the registered ruleset for most environments because it also includes the community rules. The emerging threat ruleset is maintained by a third-party group. Since the number of rules in the emerging threat ruleset is fewer than that of the registered ruleset, we chose the registered ruleset as the base to evaluate the performance of our system and Snort.

By default, many rules are commented out in these rulesets. According to the official FAQ (Anon, 2021j) of Snort, users should not load all rules in a ruleset, or the performance may be greatly impacted. Within a ruleset, we can choose from four different configurations: connectivity over security, balanced, security over connectivity, and maximum detection. Each configuration includes only a subset of the ruleset and the unused rules are commented out. Some rules are commented out in the rulesets because they may have great impact on the performance, generate false alerts, or target at minor threats. From Table 3, one can see that for the registered ruleset, about 12 000 rules are used for most environments. Therefore, when we varied the number of rules for comparing the performance of our system and Snort, the maximum number of rules used was set to 12 000.

4.3.2. The rule format of Snort

Fig. 4 shows the rule format of Snort. Starting from the left side, the first part is the rule header. The first field of the rule header specifies the action that will be performed if the rule is matched. The possible actions include *alert*, *log*, and *pass*, which will generate an alert message, log the packet, and ignore the packet, respectively. If a packet is ignored, it will not be checked by other rules. Thus, the “pass” action functions like the ACCEPT action of iptables firewall rules used in the Linux kernel, and it is used for trusted hosts/network domains to speed up the matching process. The next field is the protocol type and its possible values include *tcp*, *udp*, *icmp*, and *ip*. Packet of protocols other than IP will be dropped directly. The rest of the header is the 5-tuple information in the packet header that this rule must match. Note that the source and destination ports are meaningful only for TCP and UDP rules.

The second part is the rule options. In this example, the *content* option indicates the exact match pattern, and the *msg* option specifies the message printed when the rule is matched. The payload detection rule options such as *content* and *pcre* can be specified multiple times in

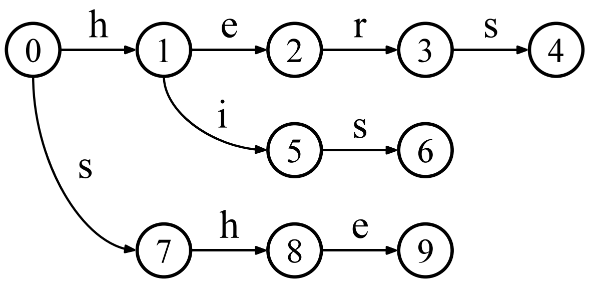
$$\begin{tikzcd}
0 \arrow[r, "h"] \arrow[dd, bend right, "s"'] & 1 \arrow[r, "e"] \arrow[d, bend right, "i"'] & 2 \arrow[r, "r"] & 3 \arrow[r, "s"] & 4 \\
 & 5 \arrow[r, "s"] & 6 & & \\
 & 7 \arrow[r, "h"] & 8 \arrow[r, "e"] & 9 &
\end{tikzcd}$$

Fig. 3. An example AC trie.

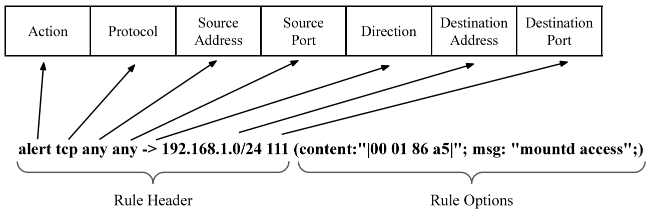

Fig. 4. The rule format of Snort.

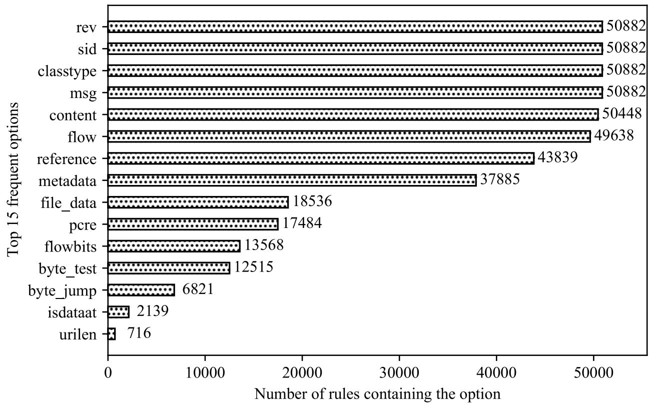

Fig. 5. The top 15 frequent options used in the registered ruleset of Snort.

a rule. All options that make up a rule must be true for the indicated rule action to be taken. The options of a rule can be considered to form a logical AND statement. Snort evaluates the options of a rule from left to right. If one option is not matched, Snort will not evaluate the following options. At the same time, the various rules in a Snort ruleset can be considered to form a large logical OR statement. Thus, if a packet matches any rule in a rule set, it will be processed by the action of that rule. On the other hand, if a packet matches no rule in the ruleset, no alerts will be raised.

4.3.3. The Snort options implemented in our system

Our system supports the Snort rule format, but it only supports a subset of its options due to the constraints of eBPF. Fig. 5 shows the top-15 frequent options used in the registered ruleset. In order to achieve a large coverage of the ruleset, we decided to support the options from the most frequent ones. Note that the options shown in Fig. 5 include many general rule options (Anon, 2021k). As these options do not influence the rule matching process, our system just bypasses these general options. To summarize, our system supports the *content*, *file_data*, *pcre*, and *byte_test* options. The *rev*, *sid*, *classtype*, *msg*, *reference*, and *metadata* options are bypassed, and the *flow*, *flowbits*, *byte_jump*, *isdataat*, and *urilen* options are unsupported by our system.

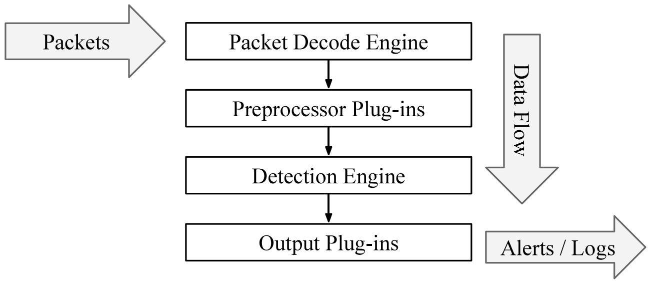

Fig. 6. The architecture of Snort.

4.3.4. *The architecture of Snort*

Fig. 6 shows how Snort processes packets. In the beginning, the packet decode engine first handles the packets. The information inside the packet headers is extracted and stored into internal data structures in this step. Then, the next module is the preprocessors. If there are multiple preprocessors, Snort will sequentially invoke them. Examples of commonly used preprocessors are *frag*3, *session*, and *stream*. *Frag*3 preprocessor can reassemble fragmented packets and *session* preprocessor can track the status of TCP or UDP flows. *Stream* preprocessor is responsible for tracking the information of TCP/UDP flows, too, but it is specialized for flow reassembly. Next, the packet is processed by the detection engine, which is explained in the following section. If the packet hits any rule, the output plug-ins will generate alarms.

4.4. *The detection engine of Snort*

The detection engine consists of two components: a fast pattern matching engine (FPM) and a rule tree matching engine. Any rule that has one or more *content* options in it has a fast pattern associated with it. Snort puts fast patterns into the FPM to facilitate the process of detection. By default, a fast pattern is the longest pattern defined by the *content* option in a rule. If the lengths of the patterns are the same, the first one appearing in the rule will be the fast pattern. In addition, the fast patterns can also be specified by rule writers with the *fast_pattern* option. Listing 1 is an example demonstrating how Snort picks fast patterns. For the first rule, the fast pattern is the longest one *she*. For the second rule, although the longest pattern is *which*, because the rule contains a *fast_pattern* option specifying the fast pattern explicitly, the fast pattern is *his* rather than *which*. For the third rule, both of the patterns have the same length, so the first pattern *where* is picked as the fast pattern.

```
alert tcp any any -> any any (content: "he"; content: "she";)
alert tcp any any -> any any (content: "his"; fast_pattern;
      content: "which";)
alert tcp any any -> any any (content: "where"; content: "which";)
```

Listing 1: Example of fast patterns

The fast pattern matching engine (FPM) is an important component of Snort. Packets will be first scanned by the FPM. If any fast pattern of the rules is present in the packet, the packet will then be processed by the rule tree matching engine. Otherwise, it will be ignored without any further processing. Rules are organized as rule trees in Snort to efficiently find the rule that matches a packet. Our system adopts similar designs and we will present the details of them in later sections. Note that if the payloads of packets are encrypted (Duong et al., 2020), since our system needs to compare the packet payloads against signatures to detect malicious packets, our system will not be able to detect attack packets in such a condition.

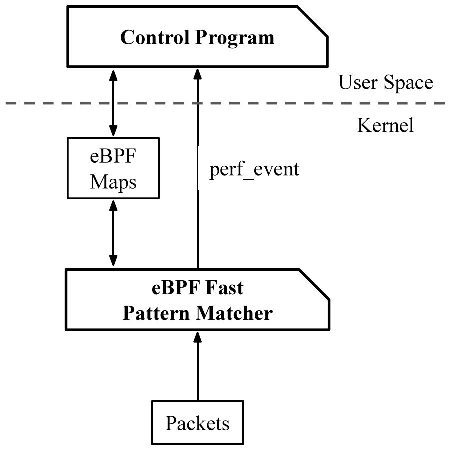

Fig. 7. The architecture of our system.

5. Design and implementation of our system

5.1. *System architecture*

As depicted in Fig. 7, there are two programs running together in our system. One is the eBPF FPM program (i.e., the eBPF fast pattern matcher) running in the kernel. Each packet received by the network driver will first be processed by the eBPF FPM to check if its payload contains any fast pattern from the ruleset. As explained previously, each rule in the ruleset has a fast pattern. (An example illustrating the usage of fast patterns will be shown in Fig. 16.) If no fast pattern of the ruleset appears in the packet, it is impossible that this packet will match any rule and thus the eBPF FPM will drop this packet without sending it to the control program for further examinations. As mentioned previously, both our system and Snort run in the NIDS configuration mode. Thus, the packets that they check actually are mirrored copies. Thus, dropping packets here means that these mirrored packets are considered safe packets and they can be safely dropped without affecting the network devices that send (or receive) them. On the other hand, if the packet contains at least a fast pattern from the ruleset, the packet will be sent to the control program by the *perf_event* structure for detailed analysis.

The other part of the system is the control program running in the user space. In the initialization stage, it parses rules to extract the information required by the eBPF FPM and then stores the information into the eBPF maps. It then handles the packets sent from the eBPF FPM to see if they would match any rule. If any rule is matched for a packet, it applies the action of that rule to the packet. Otherwise, it drops the packet. After initializing all necessary data structures and the eBPF program, the control program enters an infinite loop to continuously poll the perf events. Such an event includes the packet data and the information about matched fast patterns, which are essential for the control program to find the rule that matches the packet.

5.2. *Limitation of eBPF-based solutions*

Since eBPF is executed inside the kernel, eBPF programs must satisfy several constrains so that they will not crash the kernel. Before injecting the eBPF programs into the kernel, these programs should be checked by a verifier. The verifier will statically check several constrains including instruction counts, execution paths, and memory accesses.

Currently, all possible execution paths should contain less than one million instructions, and there should be less than 4096 instructions in a single eBPF program. To overcome this restriction, we used tail calls to run another program after the execution of one program and used bpf-to-bpf function calls to eliminate duplicated code and reduce the code length.

The eBPF verifier forces the eBPF program developer to check memory access validity before each memory access except the stack. In the XDP context, only three types of memory can be accessed: stack, eBPF maps, and packet data. For eBPF maps, our eBPF program ensures that the pointer to a map value is valid before uses it. For packet data, our eBPF program ensures that all pointers to it are within the bound of the packet. Since currently there is no automatic ways to do this check, we manually added checks before each memory access in our eBPF program.

Before Linux 5.3, the eBPF verifier did not allow any backward jump. This means that we cannot use loops in our eBPF program. The only workaround is fully unrolling all loops so that they will not contain backward jumps. However, as the number of loop iterations rises, the size of the eBPF program will soon exceed the limit set by the verifier. To overcome this problem, Linux introduced bounded loops in version 5.3. Bounded loops only iterate a fixed number of times. In other words, loops become legal if the number of iterations can be derived by the verifier. To use this feature, our eBPF program ensured that all loops run for constant iterations. Bounded loops not only can reduce the size of eBPF programs, but also can improve the cache hit rate as well as the performance. Since our system needs to loop through each byte of the packet, we adopted bounded loops to overcome the limitation of instruction count.

5.3. The eBPF program of our system

Fig. 8 is the top-level flow chart of our eBPF program. Starting from Fig. 8(1), because the incoming packets may be encapsulated within the Virtual Local Area Network (VLAN) header, we first strip the VLAN header until we find the Ethernet header. Then, we extract the *ethertype* of the packet. If the packet is not an IP packet, it will be dropped immediately (Fig. 8(2)). Once we determine that it is an IP packet, depending on the higher-layer protocol type, there are four possible cases. For TCP and UDP packets (Fig. 8(3)), the source and destination ports are used to find the starting state of the AC algorithm (Fig. 8(5)) for the packet. The detailed description of the *get_start_state* function is shown in Fig. 9. For ICMP packets and packets of other protocol types, since they do not have the source port or destination port, the starting state for these two cases is hard-coded into the program. The *start_pos* variable indicates the offsets of the place from where the program should start scanning. The values are calculated by summing the length of the L2, L3, and L4 headers. Take TCP as an example, the value is $14 + 20 + 20 = 54$. After knowing the starting state and *start_pos* for the packet, the program can perform the AC algorithm on the packet (Fig. 8(6)).

Fig. 9 explains how to calculate the start state of the AC state machine for a packet. We divide all TCP and UDP rules into different groups by the source/destination port pairs called port groups. (Later on, Fig. 15 will show an example.) Each port group has its own AC state machine. For simplicity, we store all AC state machines into one map called *aho_nodes_global*. Thus, the start state returned by the *get_start_state* function is actually an integer index to this map to locate the AC state machine for this packet. We adopt a two-level approach to map from the source/destination port pair to the corresponding AC start state. Starting from Fig. 9(1), the program first checks whether the level-one map (*l1_map*) has the source port (*src_port*) as a key. For Fig. 8(3), one can see that the value of *l1_map* can be *tcp_l1_map* or *udp_l1_map*. If *l1_map* has the key, we can safely get the value of *l1_map*[*src_port*], which points to the level-two map (*l2_map*) (Fig. 9(2)). Otherwise, the program will check if *l1_map* has the key of the *any* value (Fig. 9(3)). If so, *any* is used as the key to *l1_map* and *l2_map* will be *l1_map*[*any*] (Fig. 9(4)). In Snort rules, the source/destination port may be *any*, which means that the rule matches packets with any port number. If *l1_map* has neither *src_port* nor *any* as a key, the packet will be dropped since there will be no rule that can match this packet. Similarly, *l2_map* takes the destination port (*dst_port*) as the key and it will output the start state *start_state* (Fig. 9(6)) corresponding to the source/destination port pair of the packet.

Fig. 10 is the detailed steps performing the AC algorithm. The procedure starts from Fig. 10(1). The *pos* variable indicates the current position related to the beginning of the packet, which is initialized by *start_pos* calculated in Fig. 8(3) and (4). After that, the program will enter a bounded loop scanning the whole payload. There are two termination conditions for this loop. Fig. 10(2) is the first termination condition of the loop. The *MAX_SCAN_LEN* constant is hard-coded as 1520. This is because in most networks the Maximum Transmission Unit (MTU) is set to 1500. If we also take the Ethernet header into consideration, the total length of a packet will be 1514, which is still less than 1520. This static condition is necessary because the eBPF verifier needs to know the bound of this loop. The second termination condition is the length of the packet *pkt_len*. Since it is only known at run time, we cannot directly use it as the only termination condition.

If *pos* is less than *pkt_len*, we can safely access the packet data at this position (Fig. 10(4)). Note that the data byte is first converted into the upper cases since our system (like Snort) only performs case-insensitive matches. The *ac_state* variable has an attribute *has_match* (Listing 2) indicating whether there is any match in this state (Fig. 10(5)). If so, the program will then examine if this is the first match of this packet (Fig. 10(6)). If yes, the eBPF program will pass the current *ac_state* and the whole packet data to the control program (Fig. 10(7)). Otherwise, only *ac_state* will be sent to the control program (Fig. 10(8)). Obviously, we can always send both *ac_state* and the packet data no matter whether it is the first match of a packet or not. However, the program will have to copy the packet data to the control program for each match in the packet data and this will waste lots of computing resources. The reason why the packet data is copied to the control program is that if a packet matches a rule in the control program, the control program can display the content of the packet data to the network administrator for further manual examinations.

```
//ids_common.h
struct aho_node_t {
    int32_t has_match;
    int32_t next[256];
} ac_state;
```

Listing 2: The definition of AC states

After processing the match within the current *ac_state*, the program will perform a state transition (Fig. 10(9)). Since the AC automaton is always a deterministic finite automaton (DFA), we can pre-compute all possible transitions for each state. In the initialization stage, the control program computes all possible transitions and stores them into the *next* array in each AC state shown in Listing 2. Then in Fig. 10(9), the eBPF program can perform a state transition with only one line of code. Finally, the program goes back to the beginning of the scanning loop and run for another iteration.

5.4. The methods used by our system to overcome the limitations of eBPF

To implement the designs shown in Figs. 8–10, we used the following methods to overcome the limitations of eBPF. The first problem is the limitation of loops and the number of instructions. To analyze the entire payload of a packet, we have to loop through each byte of it. In the early versions of eBPF, all loops should be fully unrolled to prevent backward jumps. However, this will make an eBPF program easily exceed the number limit of instructions. Recently, bounded loops have been allowed and the following is an example showing how we converted a normal loop into a bounded loop. In Listing 3, we want to process each byte in a packet and use a variable *pos* to indicate the offset in the packet. However, this loop will be rejected by the eBPF verifier because the loop bound *pkt.len* is a variable.

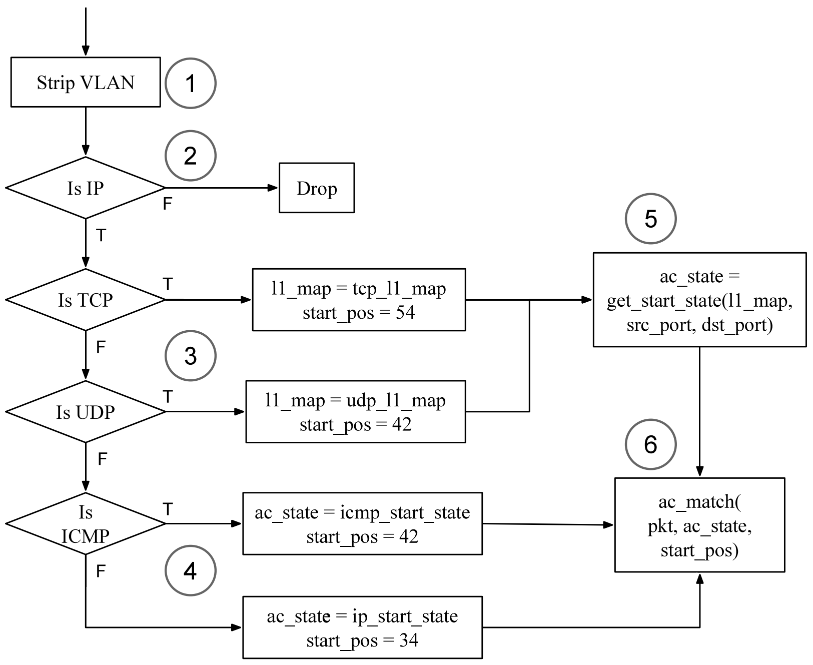

Fig. 8. The flow chart of our eBPF program.

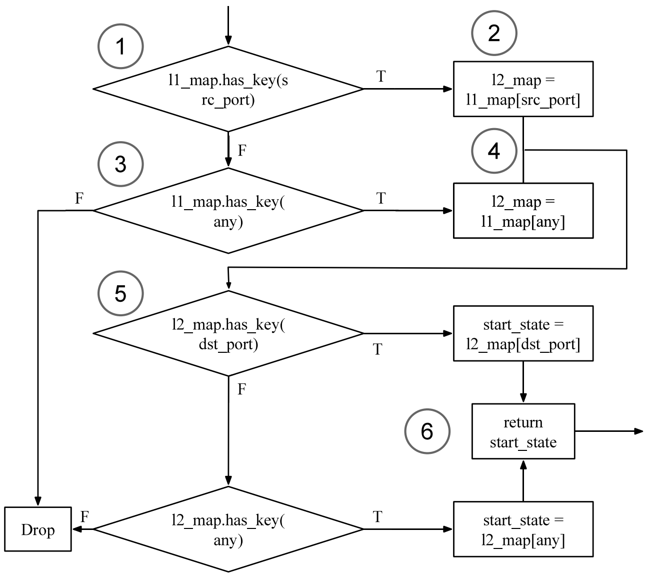

Fig. 9. The flow chart of the *get_start_state* function.

```
for(pos = 0; pos < pkt.len; ++pos) {
    //loop through each byte of pkt
}
```

Listing 3: A normal loop

To pass the check of the verifier, the loop bound should be a constant known at compile time. Listing 4 is the bounded version of the same loop. The loop bound is set to the maximum possible length of a packet, which is a constant value. However, since it is possible that the length of a packet is smaller than MAX_SCAN_LEN, we have to break the loop if *pos* exceeds the length of the packet. In this way, we not only can satisfy the constrain of bounded loops, but also can process packets of different lengths.

```
for(pos = 0; pos < MAX_SCAN_LEN; ++pos) {
    if(pos >= pkt.len)
        break;
    //loop through each byte of pkt
}
```

Listing 4: A bounded loop

Another problem is the limited types of memory that we can use. In eBPF, one can only use eBPF maps and stacks and cannot dynamically allocate memory from heaps. To overcome this restriction, instead of using heaps, we placed the dynamic data used by our eBPF program in eBPF maps. Fig. 11 shows all the maps used in the eBPF program. On the right of Fig. 11, the "SM" stands for "State Machine". As can be seen, we stored all AC state machines, which are divided by the port groups, into the same big eBPF map. The index to the starting element of a state machine in this map is the *start_state* returned from Fig. 9.

5.5. The control program of our system

5.5.1. The initialization procedure

The initialization procedure of our IDS system shown in Fig. 12 can be divided into two parts. The first part only focuses on parsing the rules into proper data structures, and the second part is in charge of configuring maps and buffer related to eBPF.

At first (Fig. 12(1)), the program reads a file containing all rules and parses the rules into a rule database (*rule_db*) for later uses. Then, all rules are classified into four categories according to the higher-layer protocols of them. For TCP and UDP rules, each category is further divided into different port groups, each of which contains a set of rules with the same source/destination port pair. After the classification is done, an AC state machine (ACSM) will be built for each port group.

In the second part (Fig. 12(2)), the program initializes all data structures used by the eBPF program and attaches the eBPF program to the NIC driver. Specifically, the C code of the eBPF program should first be compiled by the clang compiler and translated into an eBPF binary by the LLVM tool. The binary will then be verified and loaded into the kernel. During the verification, all constrains presented in Section 5.2 will be checked. If the eBPF program does not satisfy any of them, it

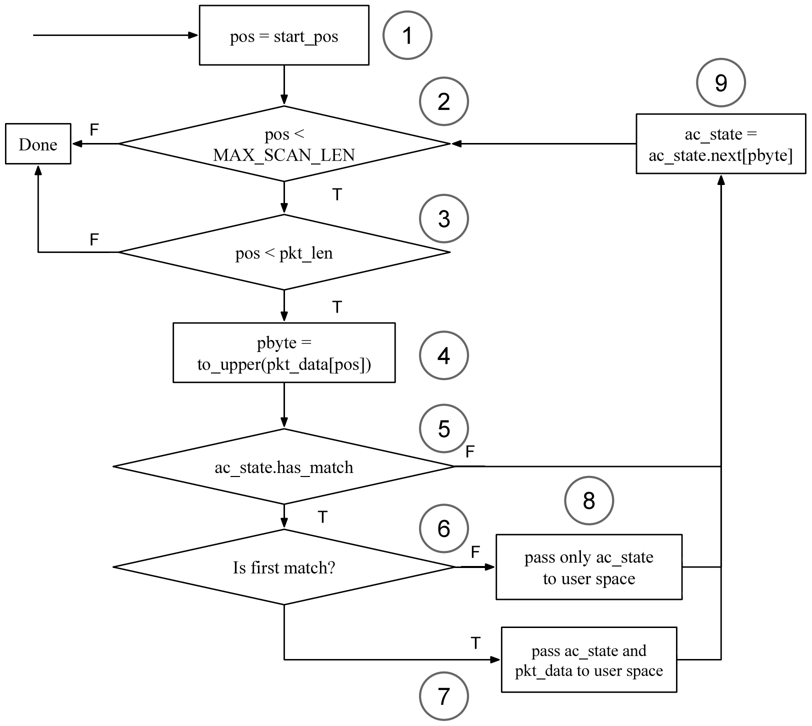
pos = start_pos
1
2
pos < MAX_SCAN_LEN
Done
F
T
3
pos < pkt_len
F
T
pbyte = to_upper(pkt_data[pos])
4
5
ac_state.has_match
F
T
6
Is first match?
F
T
8
pass only ac_state to user space
pass ac_state and pkt_data to user space
7
9
ac_state = ac_state.next[pbyte]

Fig. 10. The flow chart of the *ac_match* function.

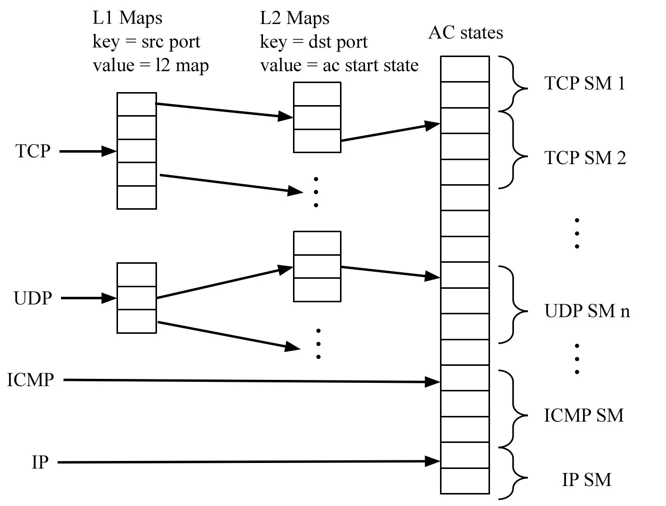

Fig. 11. All eBPF maps used in our system.

will be rejected by the kernel. After this step, the eBPF program is still not functional since it still lacks some important data structures. In the following step, the state nodes from all AC state machines are merged into one single array called the *aho_nodes_global* map. This array is used by the AC algorithm described in Section 5.3.

The fourth step of the second part initializes the buffer for perf events. The control program first allocates a buffer of perf events inside the kernel and then creates a file descriptor to monitor the status of this buffer. When a perf event is generated and written to the perf buffer by the eBPF program, the file descriptor related to it will become readable. Note that on a multi-core system, the perf buffer cannot be shared across multiple cores. Thus, we have to create one perf buffer for each core. Then, the control program attaches the eBPF program to the NIC driver. After this step, the eBPF program is fully functional. The last step is entering the loop to start polling events. Once an event is received, it will be processed by the handler described in Section 5.6.1.

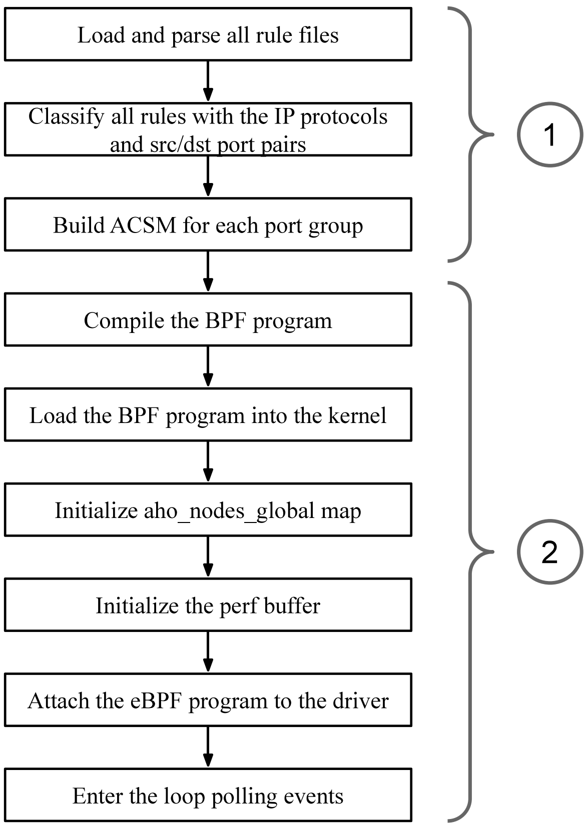
Load and parse all rule files
Classify all rules with the IP protocols and src/dst port pairs
Build ACSM for each port group
1
Compile the BPF program
Load the BPF program into the kernel
Initialize aho_nodes_global map
Initialize the perf buffer
Attach the eBPF program to the driver
Enter the loop polling events
2

Fig. 12. The initialization procedure of our IDS system.

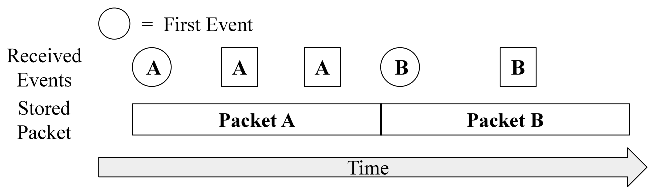

Fig. 13. The event sequence without any drop.

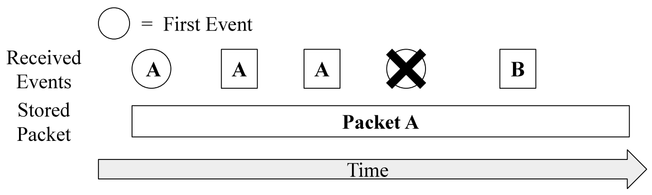

Fig. 14. The event sequence with drops.

5.6. The *perf_event* structure

There are two ways by which the eBPF program can pass data to the control program. One is eBPF maps and the other is perf events. The major differences between these two methods include the directions and the support for copying whole packets. For eBPF maps, both the eBPF program and the control program can read or write them, so the directions of sending data can be bidirectional. In contrast, perf events can only be sent by the eBPF program and received by the control program. In most cases, our system uses eBPF maps to communicate between the user space and the kernel since they are more flexible. However, regarding the support for copying whole packets, perf events are more suitable. If the eBPF program wants to send packet data to the user space with an eBPF map, the only way is copying all data bytes one by one to the map, which consumes a large amount of computing power. On the other hand, if we use perf events, we can perform the same task with only one function call to a built-in function provided by the kernel. This function runs on the CPU without involving the eBPF virtual machine and is very efficient. This is why our system uses perf events to pass the packet data.

Listing 5 shows the definition of the perf events. The *perf_event_sample* has two parts. The first part includes the header and size fields. These two fields are pre-defined by the kernel and cannot be changed. The following fields contain customized data defined by our system. The *pkt_context* structure contains information for rule matching and its definition is shown in the listing. In the *pkt_context_t* structure, *timestamp* and *is_first* are used for event handling explained in Section 5.6.1, *state* is the matched state of this packet, and *ip_proto* is the protocol type of this packet, which is used for rule matching. The *timestamp* field stores the time when a packet was received by the NIC driver. All events generated for a packet will carry the same timestamp of the packet. The last field *pkt_len* is the total length of the packet. Back to *perf_event_sampe* structure, the last field *pkt_data* is a pointer to the packet data. Note that as we have explained in Section 5.6.1, the perf event will not contain the packet data if it is not the first event of the packet. In other words, *perf_event_sample.pkt_data* is meaningless if *perf_event_sample.pkt_context.is_first* is *false*.

```
//bpf_ids.h
struct perf_event_sample {
    struct perf_event_header header;
    __u32 size;
    struct pkt_context_t pkt_context;
    __u8 pkt_data[4];
} __attribute__((packed));

//ids_common.h
struct pkt_context_t {
    __u64 timestamp;
    __u32 state;
    __u32 ip_proto;
    __u32 is_first;
    __u32 pkt_len;
};
```

Listing 5: The *perf_event* structure

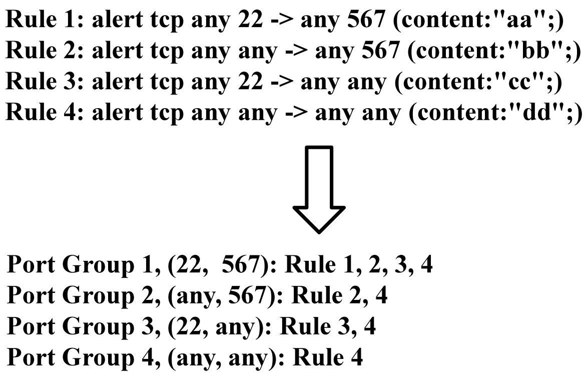
Rule 1: alert tcp any 22 -> any 567 (content:"aa";)
Rule 2: alert tcp any any -> any 567 (content:"bb";)
Rule 3: alert tcp any 22 -> any any (content:"cc";)
Rule 4: alert tcp any any -> any any (content:"dd";)
Port Group 1, (22, 567): Rule 1, 2, 3, 4
Port Group 2, (any, 567): Rule 2, 4
Port Group 3, (22, any): Rule 3, 4
Port Group 4, (any, any): Rule 4

Fig. 15. An example of grouping rules by their ports.

5.6.1. The event handler

The event handler processes the perf events sent by the eBPF program to the control program. Events generated by the eBPF program may be dropped by the kernel due to buffer overflow if the control program cannot handle them fast enough. Thus, we use the timestamp of packet to detect such a situation. For example, Fig. 13 shows a normal event sequence without any drop. A round-shaped event represents the first event generated by our eBPF program for a packet. In this case, both the first events of packet A and B are received by the control program. Thus, the stored packet will be A and then becomes B, which is the correct outcome. On the contrary, if the control program loses the first event of packet B (Fig. 14), the buffer in the control program will not be replaced with the data of packet B. Later on, when the control program receives another event of packet B, the time stamp of the event and that of the stored packet (which is packet A) will be different. To avoid generating wrong alarms, our system will drop such an event without processing it.

5.7. The port groups

As we presented in Section 5.3, TCP and UDP rules are grouped by their ports. The control program uses port groups to efficiently locate the AC state machine built for a specific port group. Fig. 15 shows a simple example. Suppose that we have four rules in total. If we receive a packet with the port pair (22, 567), all the four rules should be checked since they all match this port pair. Therefore, the first port group includes all the four rules. If the packet has a source port other than 22 and a destination port 567, only rule 2, 4 will be matched and thus port group 2 only contains rule 2 and rule 4. The situation of port group 3 is similar to that of port group 2. Finally, if the source port and the destination port are neither 22 nor 567, only rule 4 will be matched and thus port group 4 contains only rule 4.

Listing 6 shows the definition of port groups. We focus only on some important fields of them. The *src_port* and *dst_port* are the source and destination ports, and *rules* stores all the rules belonging to this port group. The attribute *acsm* is the AC state machine built for this port group and *global_start_state* is the starting state of it (which is shown in Fig. 11). The attribute *rule_tree* is another important data structure, which is explained below.

```
//pcrm.h
struct port_group_t {
    uint16_t src_port, dst_port;

    int nslots;
    int nrules;
    struct rule_t** rules;

    struct acsm_t* acsm;
    struct rule_tree_t* rule_tree;
    uint32_t global_start_state;

    int is_ref;
    struct port_group_t* ref_to_group;
};
```

Listing 6: The *port_group_t* structure

5.8. The rule trees

All the rules in a port group are organized into a rule tree and the control program uses the rule tree to efficiently find which rule matches a packet. Fig. 16 shows an example in which the four rules are in the same port group but the patterns of them may be the same or different. Each pattern is represented as a node in the tree and common patterns (nodes) can be shared by different rules. For example, both rule 1 and 2 have the pattern *aa*, so they share the same node of *aa*. The nodes in the second level represent the fast patterns of these rules (which are explained in Section 5.1). In this example, only three fast patterns are used to build the AC state machine.

When the control program receives a perf event, it first locates the corresponding port group with the matched state. For each port group, it then evaluates the rule tree by using the breadth-first search (BFS) method to check if the packet may match any rule. For instance, if the packet only contains the patterns *aa* and *bb*, in the second level node *dd* will be discarded first. Then, in the third level the two nodes with the pattern *cc* will be discarded since *cc* is not in the packet. At last, only the node *bb* in the third level is successfully evaluated, which means that the packet matches rule 1. Note that the evaluation of a rule tree may result in no rule matching the packet. For example, if a packet contains only the *aa* and *dd* patterns, then it will match no rule in this tree. On the other hand, because rules may overlap, it is possible that a packet may match multiple rules in the rule tree. However, since the BFS method will return the first rule that matches the packet and then stops, only one matched rule will be returned. This design is the same as that adopted by Snort.

5.9. A simple example illustrating rule matching in our system

In this section, we use an example to illustrate the whole process of matching rules in our system. The used ruleset is the same as shown in Fig. 16 and the incoming packets are shown in Table 4. All rules are TCP rules and only one port group is built.

The first packet that comes is a TCP packet. After the eBPF FPM scans its payload, two perf events will be generated. One matches the pattern *aa* and the other matches the pattern *bb*. The event with the pattern *aa* also contains the entire packet data since it is the first event of this packet. When the control program receives the *aa* event, it will copy the packet data to its buffer and evaluate the rule tree shown in Fig. 16. The *content* options specified in a rule impose no order requirement in the payload, and all patterns specified by the options of a rule must simultaneously exist in the payload of a packet for the packet to hit the rule. When the control program receives the first event, although this event means that *aa* is in the packet, since neither the *bb* event nor the *cc* event has been sent to the control program for the same packet, the control program can only use *aa* to evaluate the rule tree, which hits no rule. When the second perf event, which matches *bb*, is sent to the control program, the control program uses both the *aa* and *bb* patterns to evaluate the tree and this time rule 1 is matched and alerted. When the second packet comes, the eBPF FPM will only output one perf event since only *bb* is matched in the payload. The control program will use the *bb* pattern to evaluate the rule tree but no rule will be hit. When the last packet comes, which is a UDP packet, since there are no UDP rules in this example, the packet will be dropped by the eBPF FPM and no perf event will be generated.

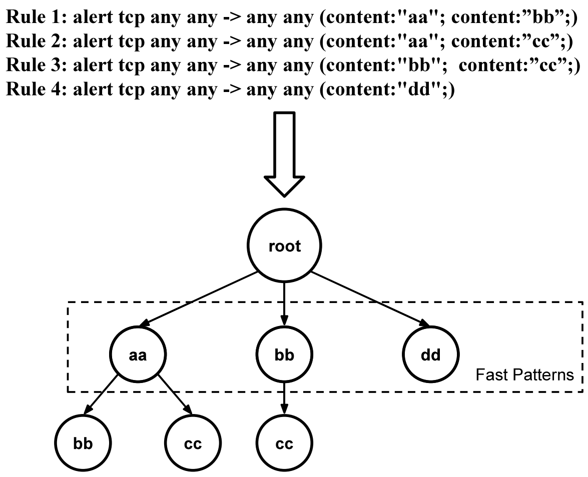
Rule 1: alert tcp any any -> any any (content:"aa"; content:"bb";)
Rule 2: alert tcp any any -> any any (content:"aa"; content:"cc";)
Rule 3: alert tcp any any -> any any (content:"bb"; content:"cc";)
Rule 4: alert tcp any any -> any any (content:"dd";)
root
aa
bb
dd
Fast Patterns
bb
cc
cc

Fig. 16. An example of building and using a rule tree.

Table 4
The incoming packets of an example.

#	Protocol	Src port	Dst port	Payload
1	TCP	22	10 004	aabb
2	TCP	80	10 001	bbdd
3	UDP	100	1000	aabb

5.10. Multi-core support of our system

In order to optimize the performance of our system, our eBPF program should utilize multiple CPU cores simultaneously. This feature relies on the multi-queue and Receive Side Scaling (RSS) support of the NIC hardware. In the past, most NICs have only one receive queue. As the bandwidth of networks increases, the single queue soon becomes the bottleneck of the network stack. Therefore, nowadays many NICs can dispatch packets to multiple queues in the kernel. However, the packets of a heavy flow still cannot be split across multiple queues for load balancing. To solve this problem, RSS has been proposed. Using RSS, multiple queues can be used and the number of queues is usually equal to the number of CPU cores on the machine. Take our testbed as an example, since there are 8 cores in the system, the kernel will create 8 receive queues for all NICs.

To support multiple cores in our system, some data structures in our eBPF program should be modified to prevent race conditions. For all eBPF maps used in the eBPF program, they are all read-only and thus they cause no problem. On the other hand, the perf events may be generated and sent to the control program by multiple cores. To solve this problem, we created one buffer for each core to store the perf events generated by it. As for the control program, since it is single-threaded, it will repeatedly poll each core for perf events and handle them sequentially.

Table 5
The specification of our testbed.

	Packet Generator	IDS Host
Model	N/A	Asus TS300-E10-PS4
CPU	Intel(R) Core(TM) i7-9700 CPU@3.00 GHz	Intel(R) Xeon(R) E-2134 CPU@3.50 GHz
Number of CPU Cores	8	8
RAM	DDR4 2666 MHz 32G	DDR4 2667 MHz 16G
NIC	Mellanox ConnectX-3 CX311A	
OS	Ubuntu 19.10	Ubuntu 20.04 LTS

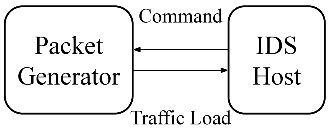

Fig. 17. The network topology of our testbed.

6. Experimental setup

6.1. Hardware and software

Fig. 17 shows the network topology of our testbed. Two 10 Gbps unidirectional optical links connect the IDS host and the packet generator. The IDS host runs the IDS as well as a control and monitoring program written in Python. This program sends commands to the packet generator to control the sending rate and attributes of the generated traffic. The packet generator is a host on which several packet generator programs run simultaneously to generate traffic. The commands and generated traffic are transmitted over separate links and thus they do not interfere with each other. The detailed specification of the testbed is shown in Table 5.

6.2. Modifying the registered ruleset

As we mentioned in Section 4.3.1, we used the registered ruleset as a base to evaluate the performance of our system and Snort. In the registered ruleset, the rules of the following four kinds are excluded. The first kind is the rules containing the options that are not supported in our system. Section 4.3.3 explains which options are not supported in our system. Next, rules that match almost all packets are excluded. During the performance evaluation, we found that some specific rules can overwhelm the eBPF FPM, especially the rules with very short fast patterns. For example, if the fast pattern of a rule is '\0', which is only one-byte long, almost all packets will match this pattern since bytes with the value zero are very common in network packets. Such rules are excluded in the ruleset used for performance evaluation. The authors in Valenza1 and Cheminod (2020) studied the anomaly problems in firewall rules and proposed a resolution, which can be used in our future work.

Third, rules requiring preprocessors of Snort are excluded. Preprocessors are optional plugins that can process packets before the pattern matching process starts. Most rules in the registered ruleset can function correctly without using any preprocessor. Therefore, we disabled all preprocessors in the default configuration except the one called $frag3$, which handles fragmented IP packets. Lastly, the rules without any $content$ option are excluded. In our system, there should be at least one $content$ option in a rule so that we can choose one of them as the fast pattern. If there is no fast pattern in a rule, the eBPF FPM will never hit the rule.

After excluding these rules, we call the resulting ruleset the modified ruleset. In Section 6.3, we explain how we used the modified ruleset to create rulesets of different numbers of rules for performance evaluation.

6.3. Choosing parameters and their default values

In the experiments, the generated traffic load has two components — the alert-triggering traffic and the background traffic. Alert-triggering traffic is the traffic that will match at least one rule in the used ruleset. On the contrary, background traffic is the traffic that will not match any rule in the used ruleset. We studied three important parameters to know their impact on the matching performance — the alert triggering traffic ratio (ATTR), the maximum packet length (MPL) of the background traffic, and the number of rules (NR) in the used ruleset. We conducted three sets of experiments. In each set of experiments, only two parameters were varied and all other parameters were set to their default values. To make our experiment settings more realistic, we chose reasonable values as the default values for these parameters. For ATTR, since the ratio of network traffic that will trigger alerts in real-life networks is low, we set its default value to 1%.

For MPL, since the MTU is 1500 bytes in most real-life networks, we set its default value to 1500. To make the packet length distribution of the generated background traffic as close to the real-life packet length distribution as possible, we took the real-life traffic data from the CAIDA traffic dataset (Anon, 2021a), whose packet length distribution is shown in Fig. 18, and used the same distribution to generate our background traffic, whose distribution is shown in Fig. 19. As can be seen, the two distributions are very similar. That is, most packets are either very large or very small. In Section 7, the MPL is used as a parameter, whose value was varied from 100 to 1500 with an interval of 200. To generate a background traffic whose MPL was L, in Fig. 19 we shifted the frequencies of the two largest packet lengths from right to left so that the packet length L has the same frequency as that of the largest packet length (i.e., 1500). Then, the background traffic was generated based on the adjusted packet length distribution.

As for NR, we chose 6000 as its default value. According to Table 3, there are about 4000 rules in the community ruleset. Therefore, we think that 6000 rules is a reasonable number of rules to be used in most environments. In Section 7, the number of rules in the used ruleset is used as a parameter, whose value was varied from 0 to 12 000 with an interval of 1200. To create a ruleset containing a specific number of rules, we selected the desired number of rules from the modified ruleset. The selection starts from the beginning of the modified ruleset. That is, when we wanted to create a ruleset with N rules, we selected the first N rules from the modified ruleset.

In addition to the three parameters, we also examined other attributes of the background traffic. Fig. 20 shows the ratio of different application protocols in the CAIDA dataset. Because most of the protocols are based on TCP and only about 10% of the traffic is based on UDP, the ratio of TCP and UDP traffic was set to 90% and 10% respectively in our generated traffic. In our experiments, we launched 30 concurrent flows to generate the background traffic.

6.4. Controlling the rate and attributes of generated traffic

In order to control the attributes of the generated traffic, we used the following steps to generate the background traffic. The traffic was generated by three steps performed by Python scripts. First, we used $iperf3$ to generate the traffic. To control the number of flows, we ran up multiple instances of $iperf3$. The instances were divided into two

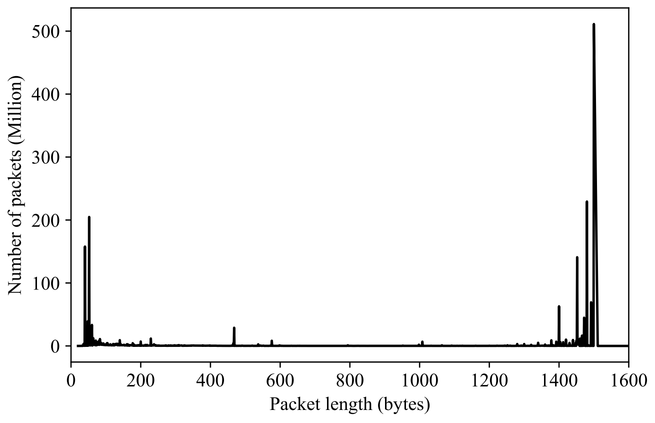

Fig. 18. The distribution of packet length in the CAIDA traffic dataset.

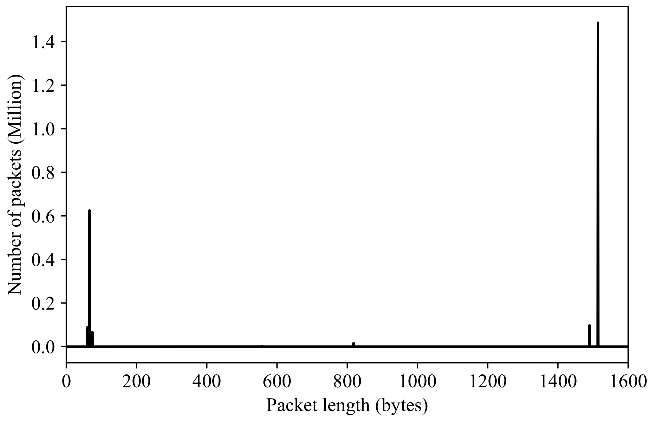

Fig. 19. The distribution of packet length in our generated background traffic.

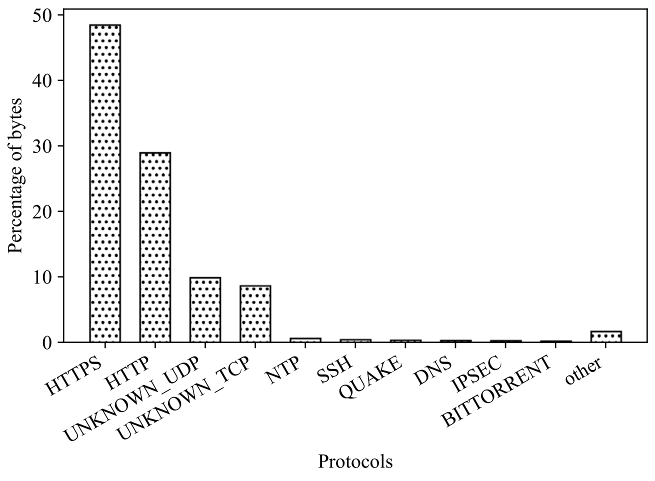

Fig. 20. The ratio of application protocols in the CAIDA dataset.

groups, one sent TCP traffic while the other sent UDP traffic. Using the $-b$ command-line option, we specified the bandwidth of each *iperf3* instance and controlled the ratio of TCP and UDP traffic. In the second step, we captured the traffic generated by *iperf3* with *tcpdump*. To generate the alert-triggering traffic, we used *pytbull* (Anon, 2021i) to generate the traffic and used *tcpdump* to capture the generated traffic. Both the background and alert-triggering traffic were separately saved

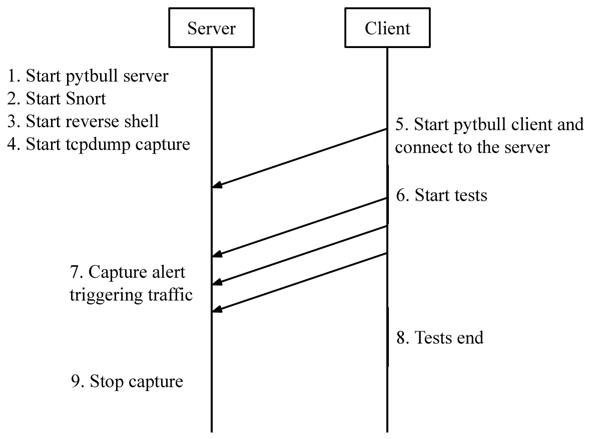

Fig. 21. The steps used to generate alert-triggering traffic.

as *pcap* files for later uses. Finally, the *pcap* files were replayed by the *tcpreplay–edit* tool. *tcpreplay–edit* is a variation of *tcpreplay*, which can modify and replay packets stored in *pcap* files. Using the $-M$ option of *tcpreplay–edit*, we specified different replay speeds to stress test the IDS. Using the $--mtu$ option, we could control the maximum length of the generated packets.

6.5. Generating alert-triggering traffic

The number of triggered alerts per unit of time will greatly impact the performance of an IDS. Therefore, we controlled the ratio of the traffic that triggers alert to see its impact. Pytbull is an open-source IDS/IPS testing framework. By default, it is shipped with about 300 tests grouped in 11 testing modules. Users can also write their own tests with Python easily. In our experiments, we used all of the tests provided by the official package to generate the alert-triggering traffic and the detailed steps are shown in Fig. 21. Pytbull adopts a server-client architecture to test an IDS. The server and client are two virtual machines provided by the Vagrant tool. Because the generated malicious traffic is sent from the client to the server, we installed an IDS and captured packets on the server. The captured packets were stored as a pcap file for later uses.

6.6. Measuring the maximum throughput

We used binary searches shown in Algorithm 5 to find the maximum throughput of an IDS without incurring any packet loss. That is, we want to find the maximum rate of the generated traffic at which the IDS can handle without causing processing backlog and packet dropping when the backlog queue overflows. The unit of the upper and lower bounds is Mbps. Since the link bandwidth in our testbed was 10 Gbps, we set the upper bound to 10 000. The *getLoss* function will run up the IDS and let the packet generator send packets to the IDS according to the used parameter values. Then, it will return the packet loss rate measured under the tested condition. Algorithm 5 Line 7 shows the termination condition for the binary search. When the difference between the upper bound and lower bound is less than 100 Mbps and the packet loss rate is 0%, the value of *speed* is returned as the maximum throughput without incurring any packet loss. Using 100 Mbps rather than 1 Mbps as the termination threshold can reduce the execution time of the algorithm. To compensate for the minor loss of accuracy, we performed the same measurement ten times and averaged their maximum throughput results to get the final result.

Algorithm 5: Search the maximum throughput of an IDS.

Input: ATTR, NR, MPL
Output: The maximum throughput of an IDS incurring no packet loss

```
 1: procedure BSMEASURE($attr, nr, mpl$)
 2:     $lower \leftarrow 0.0$
 3:     $upper \leftarrow 10000.0$
 4:     while $True$ do
 5:         $speed = (lower + upper)/2.0$
 6:         $loss \leftarrow getLoss(speed, attr, nr, mpl)$
 7:         if $loss \leq 0.0$ and $upper - lower < 100$ then
 8:             break
 9:         if $loss \leq 0.0$ then
10:             $lower \leftarrow speed$
11:         else
12:             $upper \leftarrow speed$
```

7. Performance evaluation

In the following, we present the performances of our scheme and Snort. The performance metrics include the CPU usage, packet loss rate, and maximum throughput. During evaluations, various test conditions were created by varying the values of the ATTR, MPL, and NR parameters, whose definitions have been explained in Section 6.3. To measure the CPU usage and packet loss rate of our scheme and Snort under different traffic load, we varied the load of the generated traffic from 0 to 10 Gbps, which is the bandwidth of the links used in our experiments.

7.1. Schemes under test

We compared the performance of two schemes — Snort and our eBPF-based system. We compiled Snort version 2.9.15.1 from the source code. This was the latest stable version of Snort when we were conducting this research. At that time, Snort 3.0 was still a beta version. The other scheme was our system, which was implemented with eBPF. We used the Linux version 5.3 kernel to implement our IDS system. This kernel was shipped with the Ubuntu 19.10 distribution. The major reason was that we relied on the bounded loop feature introduced in Linux 5.3 to implement the eBPF FPM. As we discussed in Section 5.2, although it was possible to implement the eBPF FPM without bounded loops, the performance overhead of the solutions was too high for our purpose.

To compare the performance of the two schemes, we first ensured that the two IDSes were run under identical conditions, including the hardware, traffic load, the configuration of OS, and the ruleset. For the hardware, we ran both IDSes on the same machine. Thus, the hardware resources and operating system available to them were the same. For the traffic load, all the packets were generated by using the procedure described in Section 6.4. The rules used by both IDSes were identical, too. In short, we have tried our best to ensure that the comparison was fair and meaningful.

7.2. Performance metrics

Three performance metrics were used in the performance evaluation. First, the CPU usage and packet loss rate of the two schemes under different traffic loads were measured. (Note that default values were used for the ATTR, NR, and MPL parameters.) The CPU usage was divided into three parts: the *system*, *user*, and *softirq* usages. The *system* usage is the percentage of time that the CPU runs in the kernel context. The *user* usage is the percentage of time spent in the user context. The *softirq* usage is the percentage of time spent in the *softirq* context. This is the place where the kernel processes received packets. Although this context is also in the kernel, the kernel maintains a separate counter to measure the time spent in the *softirq* context. The packet loss rate was the percentage of packets that were dropped due to processing backlog when the IDS cannot keep up with the generated traffic load.

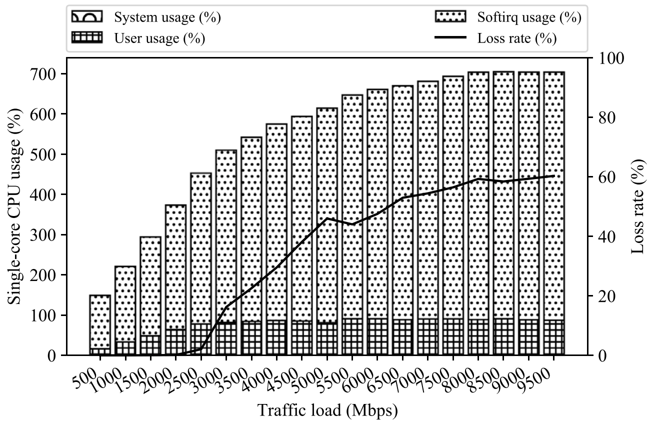

Fig. 22. The CPU usage and packet loss rate of Snort under different traffic loads.

Third, we measured the maximum throughput of the IDS without incurring any packet loss. We used Algorithm 5 to find this value. In each measurement, we measured the loss rate of the IDS under a 20-second traffic load. We found that 20 s was long enough to obtain stable results. The measurement of CPU usage and maximum throughput was each repeated 10 times. In the following figures, each presented performance data is the average of 10 results.

7.3. CPU usage and packet loss rate

Fig. 22 shows the CPU usage as well as the packet loss rate of Snort under different traffic loads. The horizontal axis is the traffic load. The vertical axis on the left is the single-core CPU usage in percentage, and the one on the right is the packet loss rate in percentage. Note that as the host running the IDS under test has eight cores, the single-core CPU usage can go up to 800% at the maximum. The CPU usage is drawn as bars while the loss rate is drawn as a line. There are two findings in this figure. First, the maximum throughput of Snort was bounded by a single CPU core. One can see that when the traffic load was higher than 2500 Mbps, the user usage had reached and then been bounded by 100%. This phenomenon can be explained as Snort is implemented as a user-space application using only one thread. Second, the maximum throughput of Snort without incurring any packet loss was about 2000 Mbps. Once the traffic load exceeded 2000 Mbps, one can see that the *softirq* usage continued to grow but the user usage remained at 100%. In other words, more and more packets were received by the kernel but Snort could not process them in time. Consequently, packets were queued in the kernel and finally dropped by the kernel when the queue was full.

The CPU usage and packet loss rate of our system is shown in Fig. 23. Compared to the CPU usage of Snort, there several findings here. First, the *user* usage of our system was tiny under all different traffic loads. In the experiment, every incoming packets were pre-checked by our eBPF program. Because our eBPF program uses the AC algorithm to quickly know whether a packet may have a chance to match any rule, a very high portion of incoming packets could be identified as safe packets and immediately pre-dropped by our eBPF program in the kernel. Thus, only a very small portion of incoming packets needed to be sent to the control program for further examinations. (Note that the ATTR value used in this experiment is its default value 1%.) Second, the loss rate of our system was lower than that of Snort in all cases. Since eBPF can use multiple cores simultaneously, the throughput of our eBPF

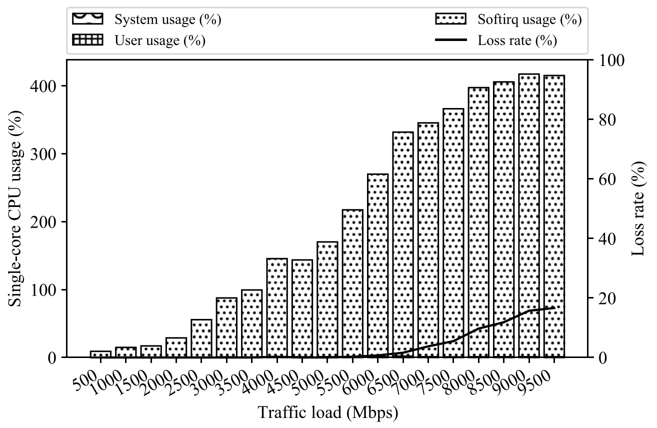

Fig. 23. The CPU usage and packet loss rate of our system under different traffic loads.

Table 6
The possible values for the parameters.

Parameter	Min	Max	Interval	Default
ATTR	1%	10%	1%	1%
NR	0	12 000	1200	6000
MPL	100	1500	200	1500

program can be increased as the traffic load increases. Therefore, one can see that the scalability of our system is better than that of Snort. Third, the $softirq$ usage of our system was less than that of Snort under the same traffic load. For example, when the traffic load was 9500 Mbps, the $softirq$ usage of our system was about 400% while that of Snort was about 600%. The reasons are as follows. As shown in Fig. 2, XDP is at the earliest stage of the network stack when receiving packets. Thus, if packets are pre-checked as safe packets and immediately pre-dropped by XDP, their complex processing at the upper layers of the network stack (note that this processing is still in the $softirq$ context) can be avoided.

7.4. Maximum throughput without incurring any packet loss

We chose three pairs of parameters for the maximum throughput measurements. Each pair of parameters consists of a primary parameter and a secondary parameter. The primary parameter is put on the x-axis of the figure while the secondary parameter is used to plot different lines each using a different value for the secondary parameter. The tested values for each of the three parameters are shown in Table 6. If a parameter is taken as the primary variable, all of its possible values were used for measurements. To leave enough space in the generated figure so that it is easier to compare different lines, if a parameter is taken as the secondary parameter, only three of its possible values were used for the measurements, which are its min, max, and default value.

In Fig. 24, we selected ATTR as the primary parameter and NR as the secondary parameter. There are two findings about this figure. First, the maximum throughput of our system was not affected by ATTR when NR was 0 or 6000. When NR was 12 000, the maximum throughput of our system slightly decreased when ATTR was higher than 6%. This phenomenon can be explained as when more rules were loaded, it was more likely that a packet might hit one of them. Since rule matching is a time-consuming process, generally the maximum throughput will decrease as the number of rules grows. Second, the maximum throughput of our system was higher than that of Snort in all cases. When no rule was used, our system could achieve almost the line rate of 10 Gbps while Snort could only achieve about 2500 Mbps. When NR was 6000 or 12 000, the maximum throughput of our system was almost three times that of Snort.

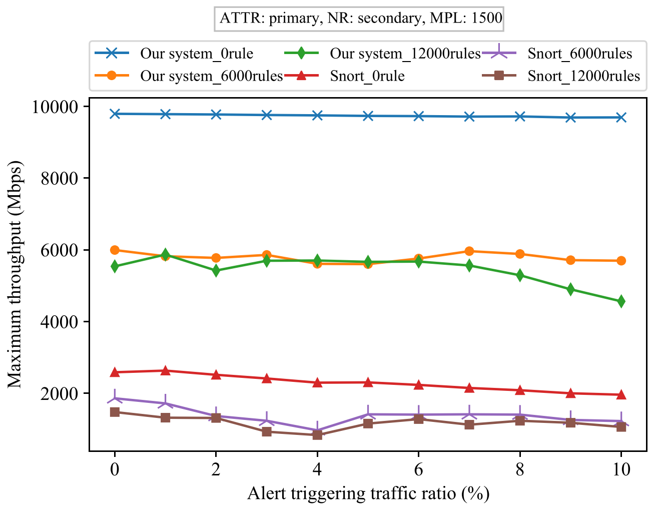

Fig. 24. The maximum throughput under different alert triggering traffic ratios and number of rules.

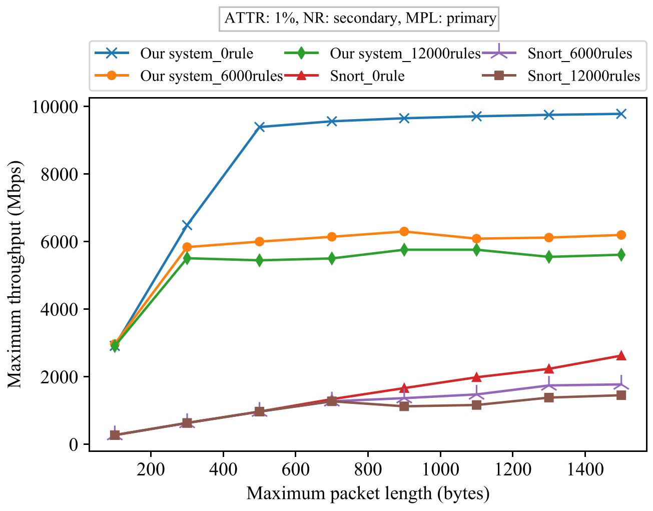

Fig. 25. The maximum throughput under different maximum packet lengths and number of rules.

In Fig. 25, the primary parameter was MPL while the secondary parameter was NR. One can see that given the same MPL, the maximum throughput of our system is much higher than that of Snort. One can also see that when the MPL decreases, the maximum throughputs of our system and Snort decrease as well. This phenomenon can be explained as the number of packets that can be processed by an IDS in each second is roughly fixed. As a result, when the MPL becomes smaller, the corresponding throughput (which is packet processing rate times packet length) will become smaller as well.

Finally, Fig. 26 selected NR as the primary parameter and ATTR as the secondary parameter. One can see that the trends in this figure are similar to those in Fig. 24. The maximum throughput of our system was always much higher than that of Snort in all cases and it only slightly decreased when NR was 12 000 and ATTR was 10%. This phenomenon can be explained as follows. When both NR and ATTR increased to large values, more packets hit the rules per second and this imposed more load to our control program. When the control program could not handle these packets in time, these packets were dropped and the maximum throughput decreased. From Fig. 26, one can see that our system outperformed Snort by almost a factor of three (6000/2000) regarding the maximum throughput in all tested conditions.

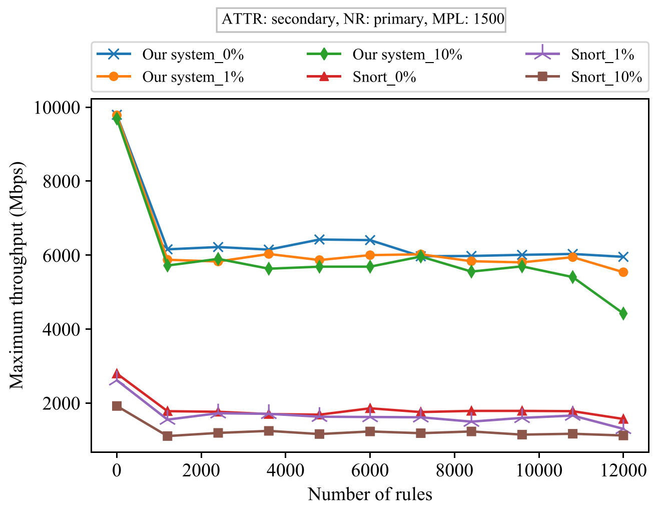

Fig. 26. The maximum throughput under different numbers of rules and alert triggering traffic ratios.

8. Discussion and future work

Due to its many constraints imposed to enforce kernel security, the current eBPF mechanism cannot be used to develop large eBPF programs for handling complex examinations or processing of packets. For this reason, currently our eBPF-based system can only support common features of Snort, rather than all features of it. As a result, to compare the performance of our system and Snort under the same test conditions, we had to modify the registered ruleset of Snort to create rulesets of different sizes that were composed of rules supported by both of our system and Snort. Since the performance comparison was made using the modified rulesets, the performance gains of our system over Snort should be cautiously interpreted in such a context.

As shown in the experimental results, our system could use all cores of an eight-core CPU while Snort could only use one core. However, currently the maximum throughput of our system is only three times that of Snort. This indicates that our system has not fully utilized the cycles of these eight cores. Regarding this problem, the control program of our system may be one performance bottleneck since currently it is single-threaded. In the future, we plan to make it multi-threaded.

Another place that can be improved is the generated traffic load. Because we want to control and vary the attributes of the traffic load, we generated synthesized traffic to evaluate the performance of our system and Snort. Although we have tried our best to simulate the real-life traffic on the Internet, it is still not realistic enough. There have been several datasets that are available for IDS evaluation. For example, UGR'16 (Anon, 2021o) and CTU-13 (Anon, 2021c) include both normal user traffic and malicious traffic. In the future, we plan to use more realistic traffic datasets to generate more realistic synthesized traffic.

Recently, Customer Edge Switching (CES) serves as an extension of the classical firewall functionality that can communicate with other security devices to establish whether network traffic should be considered as benign or malicious (Nowaczewski and Mazurczyk, 2020). Since our scheme is a firewall-based scheme, we think that our scheme can be applied to this area. Besides, because malwares are prevalent on the Internet and many of them have been identified with signatures (Marra et al., 2020), we think that our system can be used to help analyze and detect these malwares whe they are transmitted over networks.

9. Conclusion

Nowadays, an IDS is an essential function for network security. As the traffic load keeps increasing on the Internet, improving the throughput of IDS is highly desired. In this work, we used the eBPF mechanism to design and implement a high-performance IDS that has two parts working together. The first part is an eBPF program running in the Linux kernel to pre-check and then pre-drop a large portion of packets at an early stage. The second part is a program running in the user space that further examines the packets left by the first part to find which rules would match them. Using a modified version of the registered ruleset of Snort, experimental results show that the maximum throughput of our IDS system can outperform that of Snort by a factor of 3 under many tested conditions.

The eBPF mechanism has great potential to be used as a framework to implement a high-performance IDS in the Linux kernel. However, due to its many constraints imposed to enforce kernel security, the eBPF program developer requires advanced skills to fully achieve the performance gains enabled by eBPF. In addition, due to these strict constraints, currently several complex examinations and processing of packets used by Snort cannot be implemented by eBPF in the kernel. Despite the limitations of eBPF, however, eBPF still has great potential as it is still evolving. Many new features and improvements are continuously added in the new releases of the Linux kernel. Thus, the performance and functionality of our eBPF-based IDS system can continuously be enhanced in the future as well.

CRediT authorship contribution statement

Shie-Yuan Wang: Propose research ideas and funds, Design and implementation, Write the final paper. **Jen-Chieh Chang:** Design and implementation, Conduct experiments, Make figures, Write the draft paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was financially supported by the Center for Open Intelligent Connectivity from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan and the MOST project #109-2221-E-009-082.

References

Abhishta, A., van Heeswijk, W., Junger, M., Nieuwenhuis, L.J.M., Joosten, R., 2020. Why would we get attacked? An analysis of attacker's aims behind ddos attacks. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 11 (2), 3–22.

Ahmed, Z., Alizai, M.H., Syed, A.A., 2018. Inkev: In-kernel distributed network virtualization for DCN. SIGCOMM Comput. Commun. Rev. 46 (3), http://dx.doi.org/10.1145/3243157.3243161.

Aho, A.V., Corasick, M.J., 1975. Efficient string matching: An aid to bibliographic search. Commun. ACM 18 (6), 333–340. http://dx.doi.org/10.1145/360825.360855.

Anon, 2021a. The CAIDA dataset. (accessed on October 26, 2021) URL https://www.caida.org/data/passive/trace_stats/nyc-A/2019/?monitor=20190117-130000.UTC.

Anon, 2021b. Cilium. (accessed on October 26, 2021) URL https://cilium.io/.

Anon, 2021c. The CTU-13 dataset. (accessed on October 26, 2021) URL https://www.stratosphereips.org/datasets-ctu13.

Anon, 2021d. DPDK official web site. (accessed on October 26, 2021) URL https://www.dpdk.org/.

Anon, 2021e. eBPF official web site. (accessed on October 26, 2021) URL https://ebpf.io/.

Anon, 2021f. Katran load balancer. (accessed on October 26, 2021) URL https://github.com/facebookincubator/katran.

Anon, 2021g. Linux 3.18 change log. (accessed on October 26, 2021) URL https://kernelnewbies.org/Linux_3.18.

Anon, 2021h. The official web site of tcpdump. (accessed on October 26, 2021) URL https://www.tcpdump.org/index.html.

Anon, 2021i. Pytbull IDS/IPS testing framework. (accessed on October 26, 2021) URL http://pytbull.sourceforge.net/.

Anon, 2021j. Snort FAQ: Why are rules commented out by default?. (accessed on October 26, 2021) URL https://www.snort.org/faq/why-are-rules-commented-out-by-default.

Anon, 2021k. Snort general options. (accessed on October 26, 2021) URL http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node31.html.

Anon, 2021l. Snort official web site. (accessed on October 26, 2021) URL https://www.snort.org/.

Anon, 2021m. SNORT®users manual. (accessed on October 26, 2021) URL http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node16.html.

Anon, 2021n. Suricata official web site. (accessed on October 26, 2021) URL https://suricata-ids.org/.

Anon, 2021o. The UGR'16 dataset. (accessed on October 26, 2021) URL https://nesg.ugr.es/nesg-ugr16/.

Baidya, S., Chen, Y., Levorato, M., 2018. eBPF-based content and computation-aware communication for real-time edge computing. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 865–870.

Boyer, R.S., Moore, J.S., 1977. A fast string searching algorithm. Commun. ACM 20 (10), 762–772. http://dx.doi.org/10.1145/359842.359859.

Duong, D.H., Susilo, W., Trinh, V.C., 2020. Wildcarded identity-based encryption with constant-size ciphertext and secret key. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 11 (2), 74–86.

Findlay, W., 2019. Extended berkeley packet filter for intrusion detection implementations. (Ph.D. thesis). Honours Thesis Proposal, Carleton University.

Hohlfeld, O., Krude, J., Reelfs, J.H., Rüth, J., Wehrle, K., 2019. Demystifying the performance of XDP BPF. In: 2019 IEEE Conference on Network Softwarization (NetSoft), pp. 208–212.

Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D., 2018. The express data path: Fast programmable packet processing in the operating system kernel. In: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. In: CoNEXT '18, Association for Computing Machinery, New York, NY, USA, pp. 54–66. http://dx.doi.org/10.1145/3281411.3281443.

Hong, J., Jeong, S., Yoo, J.-H., Hong, J.W.-K., 2018. Design and implementation of eBPF-based virtual TAP for inter-VM traffic monitoring. In: Proceedings of IEEE 14th International Conference on Network and Service Management, CNSM'2018, pp. 1–8.

Johnson, C., Khadka, B., Basnet, R.B., Doleck, T., 2020. Towards detecting and classifying malicious URLs using deep learning. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 11 (4), 31–48.

Lazri, K., Blin, A., Sopena, J., Muller, G., 2019. Toward an in-kernel high performance key-value store implementation. In: 2019 38th Symposium on Reliable Distributed Systems (SRDS), pp. 268–2680.

Leblond, E., Manev, P., 2019. Introduction to eBPF and XDP support in suricata. URL https://www.stamus-networks.com/blog/2019/07/16/whitepaper-introduction-to-ebpf-and-xdp-support-in-suricata.

Marra, A.L., Martinelli, F., Mercaldo, F., Saracino, A., Sheikhalishahi, M., 2020. D-BRIDEMAID: A distributed framework for collaborative and dynamic analysis of android malware. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 11 (3), 1–28.

McCanne, S., Jacobson, V., 1993. The BSD packet filter: A new architecture for user-level packet capture. In: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings. In: USENIX'93, USENIX Association, USA, p. 2.

Miano, S., Bertrone, M., Risso, F., Bernal, M.V., Lu, Y., Pi, J., 2019a. Securing linux with a faster and scalable IPtables. SIGCOMM Comput. Commun. Rev. 49 (3), 2–17. http://dx.doi.org/10.1145/3371927.3371929.

Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.V., 2018. Creating complex network services with eBPF: Experience and lessons learned. In: Proceedings of IEEE 19th International Conference on High Performance Switching and Routing, HPSR'2018, pp. 1–8.

Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.V., 2018. Creating complex network services with eBPF: Experience and lessons learned. In: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), pp. 1–8.

Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., Sommese, R., 2019b. High-performance server-based ddos mitigation through programmable data planes.

Miano, S., Risso, F., Bernal, M.V., Bertrone, M., Lu, Y., 2021. A framework for eBPF-based network functions in an era of microservices. IEEE Trans. Netw. Serv. Manag. 18 (1), 133–151.

Mijumbi, R., Serrat, J., Gorricho, J., Bouten, N., De Turck, F., Boutaba, R., 2016. Network function virtualization: State-of-the-art and research challenges. IEEE Commun. Surv. Tutor. 18 (1), 236–262.

Nowaczewski, S., Mazurczyk, W., 2020. Securing future internet and 5G using customer edge switching using dnscrypt and DNSSEC. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 11 (3), 87–105.

Parola, F., Risso, F., Miano, S., 2021. Providing telco-oriented network services with eBPF: the case for a 5G mobile gateway. In: Proceedings of IEEE 7th International Conference on Network Softwarization, NetSoft'21, pp. 221–225.

Roesch, M., 1999. Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration. In: LISA '99, USENIX Association, USA, pp. 229–238.

Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., Carle, G., 2018. Performance implications of packet filtering with linux eBPF. In: Proceedings of the 30th International Teletraffic Congress, ITC 30, pp. 209–217.

Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., Carle, G., 2018. Performance implications of packet filtering with linux eBPF. In: 2018 30th International Teletraffic Congress (ITC 30). Vol. 01, pp. 209–217.

Thang, N.C., Park, M., 2020. Detecting malicious middleboxes in service function chaining. J. Internet Serv. Inf. Secur. 10 (2), 82–90.

Valenza1, F., Cheminod, M., 2020. An optimized firewall anomaly resolution. J. Internet Serv. Inf. Secur. 10 (1), 22–37.

Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Júnior, E.P.M.C., Vieira, L.F.M., 2020. Fast packet processing with EBPF and XDP: Concepts, code, challenges, and applications. ACM Comput. Surv. 53 (1), http://dx.doi.org/10.1145/3371038.

Viljoen, N., Kicinski, J., 2018. Using eBPF as an abstraction for switching, URL http://vger.kernel.org/lpc_net2018_talks/ebpf_for_switches.pdf.

Xhonneux, M., Duchene, F., Bonaventure, O., 2018. Leveraging eBPF for programmable network functions with IPv6 segment routing. In: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, pp. 67–72.

Dr. Shie-Yuan Wang is a full professor of the department of computer science at National Yang Ming Chiao Tung University (NYCU), Taiwan. He received his master and Ph.D. degrees in computer science from Harvard University in 1997 and 1999, respectively. His research interests include Internet of things, software-defined networks, programmable networks, and network security. He received the "Outstanding Information Technology Elite Award" of Taiwan government in year 2012. He authors the NCTUns/EstiNet network simulator and emulator, which is a famous tool widely used by researchers in the world. In year 2012, the EstiNet tool won the "Outstanding Information Technology Application and Product Award" of Taiwan government. In year 2014, Dr. Wang received the "President Award of Tokyo University of Science" from the President Akira Fujishima for his contributions in computer network researches. In year 2021, Dr. Wang received the prestigious 19th Y. Z. Hsu Scientific Paper Award from Y.Z. Hsu Foundation, Taiwan. Dr. Wang has published many high-quality journal and conference papers in the fields of computer networks such as IEEE Transactions on Networking and Elsevier Journal of Network and Computer Applications. He has served as general chairs and technical program co-chairs or members for many prestigious IEEE conferences such as ICC, GLOBECOM, PIMRC, VTC, ISCC, etc. Currently, he is serving as associated editor for ACM Computing Surveys.

Mr. Jen-Chieh Chang received his Master degree from the Institute of Computer Science and Engineering, National Chiao Tung University, Taiwan in 2020. His research interests include network security and system design and implementation. Currently, he is working for MediaTek Inc. as an engineer.