mKIPS: A Lightweight Modular Kernel-Level Intrusion Detection and Prevention System

Yuan-Zheng Yi[1] and Mei-Ling Chiang[2,*]

[1,2] Department of Information Management, National Chi Nan University, Taiwan, R.O.C.
s105213529@mail1.ncnu.edu.tw[1], joanna@mail.ncnu.edu.tw[2,*]

Abstract. With many research results and the development of related tools, user-level intrusion detection and prevention systems (IDPS) have been widely used to defend systems against network attacks. However, there are still bottlenecks in their high packet drop rate and low detection efficiency under heavy network traffic. In contrast, kernel-level IDPS has a higher packet detection rate and higher efficiency, whereas kernel-level design faces many challenges. The system designed with the monolithic architecture has high performance. The dynamically loadable module architecture design has higher flexibility and scalability; however, the increased operating costs lower system performance.

This paper explores the modular architecture of kernel-level IDPS that can expand or reconfigure system functions through dynamic plug-in modules and maintain the system's stability and high performance. We have developed a lightweight, high-efficiency, scalable, and highly modular kernel-level IDPS named mKIPS. This modular architecture divides the system into several kernel modules, in which functional components can be dynamically inserted or removed during runtime to adapt to changing demands. Therefore, administrators can control the IDPS's packet processing by mounting modules of different versions and functions for their needs. Besides, mKIPS dispatches packets to various cores for processing through software and hardware functions by properly setting the IRQ affinity and using Receive Packet Steering technology. As a result, the load of each core can be more balanced to utilize the multicores. Experimental results show that our mKIPS can achieve a high detection rate and efficiency.

Keywords: Intrusion Detection and Prevention System, Multicore Systems, Kernel Level, Linux Kernel.

1 Introduction

As network attack events occur frequently, providing system security for information systems is an important issue. The intrusion detection and prevention system (IDPS)[1-5] has been proven effective against information security attacks.

IDPSs can be classified into several categories according to deployment, functionality, and detection methods. The network-based IDPS is deployed within the network structure to monitor network traffic in real-time, which analyzes network packets to detect and prevent intrusions. The host-based IDPS is deployed on the host node of the

information system, which analyzes the activity of the host to identify potential intrusions or unauthorized activities. The detection method can be misuse detection, which checks the signature by comparing the network traffic or host behavior against a predefined set of signatures to detect known threats. Another detection method can be anomaly detection, which monitors the system's operation to detect anomalies that could indicate an attack or suspicious activity. IDPSs can be implemented and run at the kernel level or user level.

Snort[6] and Suricata[7] are the famous open-source network IDPS widely used in related fields and research. However, our practical experience and previous research [8] observed that Snort operating at the user level could not handle packet inspection under heavy network traffic, and its packet drop rate is relatively high. Compared with Snort and Suricata operating at the user level, kernel-level IDPS can directly intervene in the kernel's processing flow of network packets and detect threats as soon as packets are received or sent. Furthermore, it can avoid the operating cost of copying packets to the user level for inspection, waiting for the kernel scheduler's scheduling to execute user-level IDPS, and switching the protection domain back and forth between kernel mode and user mode, significantly reducing the impact on network performance.

Although kernel-level IDPS has the advantages of high efficiency and packet detection rate, developers need to have an in-depth understanding of operating system (OS) operations because it is within the kernel and directly interacts with kernel operations. Furthermore, because it operates at the kernel level, if there is a problem with the IDPS system design or a program bug is generated due to the negligence of the developers, it is easy to degrade the stability of the OS or directly cause a kernel panic. In addition, the kernel-level IDPS is highly dependent on the OS kernel. If the source code of the OS kernel is greatly changed, the kernel-level IDPS may need to be modified accordingly. These factors make it challenging to design and implement a kernel-level IDPS.

Due to the increasing maturity of virtualization technology, its use is becoming increasingly common. Furthermore, data centers virtualize the original physical server farms due to their many advantages. Therefore, how to perform intrusion detection and defense in a virtualized environment is also the focus of our research [8]. As a result, we have developed VMM-IPS [8] operating at the kernel level. It implements a reaction mechanism to respond to attacks in terms of intrusion detection functions and blocks the possibility of subsequent attacks by interrupting the attacker's connection.

From our practical experience [8] and related research [9], it is observed that kernel-level IDPS has better operating performance and better detection rate than user-level IDPS. Nevertheless, in contrast, every network packet is inspected because it is involved in the kernel's processing of network packets. Therefore, when a large number of packets come in, if the kernel-level IDPS is blocked in the kernel processing flow, it will affect the system's stability when encountering a performance bottleneck. Therefore, if it is necessary to maintain stable operating performance, providing the admission control mechanism can maintain the system's quality of service.

On the other hand, some technologies for evading detection and defense systems (IDS Evasion) [10] have been developed, and the methods are constantly being updated. With the development of defense tools, network attacks continue to evolve, and every time a new attack method is discovered, new defense tools, technologies, or systems

must be added. However, developing new systems is not easy, and the newly added functions may also be conflicts with the original system, coupled with the replacing developers, making the development of the new system difficult. How to effectively expand system functions to adapt to changing needs has become the research focus.

In view of the increasing diversification of network services and the importance of network security, we began to study the design and development of an IDPS that is scalable, highly modular, and located at the kernel level. We study the structure and dynamic plug-in modules with expandable system functions, as well as the modules for detecting IDS evasion technology, to make the defense function of the IDPS more complete so that the operation or system development can be more flexible and convenient.

In this paper, we explore the modular architecture of kernel-level IDPS. The critical work includes designing a lightweight modular architecture that can expand or reconfigure system functions through dynamic plug-in modules. The goal is to allow the system to maintain stability and good performance with a high detection rate even under heavy network traffic. So we developed a lightweight, high-efficiency, scalable, and highly modular IDPS run in the Linux kernel [11], named mKIPS.

At the same time, we also explored the processing flow of the Linux kernel in receiving and sending network packets under multicore systems. The Linux system is based on interrupt and subsequent processing for packet processing. If the same core is responsible for packet processing, although it can improve its cache usage, the multicore's parallel processing performance is not fully utilized. The Linux networking stack provides technologies [12-14] for the parallel processing of network packets under multicore systems, which can improve system performance by distributing packets to cores for processing through software and hardware setting. We employ these technologies and correctly set the IRQ affinity to distribute the processing of packets to different cores to avoid the performance bottlenecks caused by the centralized processing of packets on the same core. Experimental results show that the proposed mKIPS can have a very high detection rate and efficiency even under heavy network traffic.

2 Related Work

Snort[6] and Suricata[7] are well-known open-source network-based IDPS. The development of Snort is relatively mature. Snort has a large and active developer community and is widely used in related research. Since version 3.0, Snort has been developed from scratch with a new software framework, especially in multithreading, automatic configuration, and cross-platform support. It adopts misuse detection technology, operates at the user level, and captures live network packets from the kernel through the packet capture library - libpcap[15]. First, the Sniffer module of Snort determines the packet type and performs statistical analysis. Then the Preprocessor module performs operations such as decoding and reassembling the packet content. After that, the Detection Engine module compares the packet payload with the rules of the attack signature database. Finally, the Output module determines the packet's response method and returns the control to the original processing flow. Snort's attack signature database is constructed based on the threat reports by the Cisco Talos Intelligence Group[16].

Like Snort, Suricata[7] is also a user-level network-based IDPS initially developed using a multithreaded software framework. It is a relatively lightweight IDPS, and its detection engine also uses misuse detection technology. Its attack signature database is also built based on the threat reports by the Cisco Talos Intelligence Group. Even so, the attack signature databases of Snort and Suricata have unique features in their design, which limits their compatibility.

Many studies on IDPSs work on making IDPSs more effective, efficient, complete, and with more applications. Gaddam and Nandhini [17] analyzed various IDPSs against various types of attacks in different environments. They then proposed an architecture to improve Snort's detection rate and reduce the packet drops under heavy traffic. In the study [18] of Yuan et al., Snort is used to form a distributed IDPS. By building multiple detection nodes, the detection performance is improved. However, there may be problems with repeated detection of packets. The research suggests that distributed IDPS should strengthen the communication ability between nodes.

Shah and Issac [19] reported that Suricata could have a lower packet drop rate than Snort under high network traffic but consumed higher computational resources. Whereas, Snort had higher detection accuracy. They explored using machine learning (ML) technology to improve the efficiency and detection rate of IDPS. This study developed a Snort adaptive plug-in that implements ML algorithms to determine attacks, and this module runs parallel with Snort's original detection engine. Shah and Bendale [20] surveyed related research on using AI technology in IDPS and anomaly detection.

Chin et al. [9] proposed the kernel-level IDS built under the SDN network and discussed the advantages and challenges of building the IDS at the kernel level. They pointed out that the kernel-level IDS obtains better performance than the user-level IDS, and network packets can be immediately examined when packets are received or sent. The disadvantage is that the kernel-level IDS does not have a rich library of functions available, and developers must have complete knowledge of the kernel to implement the system. They implemented a kernel-level IDS and constructed it in the SDN network, and tried to ensure that the network function of the system would not be affected when there were problems with the functional components of the IDS.

The Linux networking stack provides technologies for parallel processing of network packets and improving performance in multicore systems by distributing packets to different cores for processing through software and hardware, such as Receive Side Scaling [12-14], Receive Packet Steering [12-14], Receive Flow Steering [12-14], Accelerated Receive Flow Steering [12-14], and Transmit Packet Steering [13,14]. However, they require experienced administrators to correctly enable and perform settings.

3 System Design and Implementation

3.1 System Overview

The proposed kernel-level IDPS named mKIPS is lightweight and modular. It is implemented as a set of kernel modules that can be dynamically inserted/removed into/from the Linux kernel during runtime. Its implementation employs the netfilter [21], a

packet-filtering framework built into the Linux kernel, to intercept all network packets entering or leaving the system. The netfilter framework allows developers to register functions to the hook point to intervene in the kernel's packet processing flow. Therefore, the mKIPS module is registered on the PRE_ROUTING hook point of the netfilter. When the packet enters the system, the packet can be sent to the mKIPS for inspection. After completing the packet inspection, a corresponding response will be given according to the detection result. If outcoming packets need to be inspected, the mKIPS module must also be registered on the LOCAL_OUT hook point of the netfilter.

3.2 Lightweight Modular Architecture

This research aims to strike a balance between the system performance and the flexibility of the system architecture. Because too much processing will degrade the system performance, which affects the packet detection rate under heavy network traffic. Therefore, we develop a lightweight dynamic modular architecture, which can avoid excessive processing due to the design providing high flexibility.

The modular architecture of mKIPS divides the system into different Linux kernel modules. As shown in Figure 1, the calling module is called a demand module, and the called module is called a supply module. In the implementation, the demanding function pointers point to the target functions implemented by other modules, and functions of the mKIPS modules can communicate with each other. The demand module must declare function pointers and implement an empty function that only returns the default value. The demand module exports the access authority through Linux kernel macro EXPORT_SYMBOL. The supply module needs to declare an extern modifier to obtain the access authority of the demand module and implement the supply function.

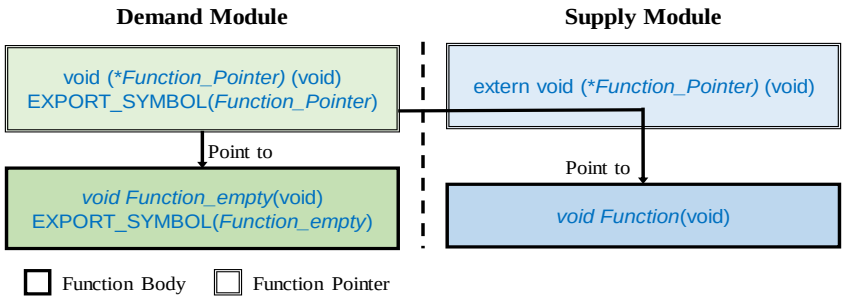


Fig. 1. Modular architecture design

When the demand module is inserted into the kernel, its demand function pointer will point to the empty function. When the supply module is inserted into the kernel, the demand function pointer will be changed to point to the implemented function of the supply module. Finally, when the supply module is removed from the kernel, it will redirect the demand function pointer to the empty function again. Under this framework, the mKIPS' kernel modules can be dynamically inserted or removed during system runtime. The administrator can also control the packet processing flow of IDPS by

inserting modules of different versions and functions. This lightweight modular architecture has less impact on the system operations and performance.

Under this lightweight modular architecture, the mKIPS system is divided into three kernel modules: the Manager module responsible for intervening in the kernel to process network packets, the Detection module for inspecting packet content, and the Response module for defending against attacks and giving responses. The modular architecture and system processing flow are shown in Figure 2.

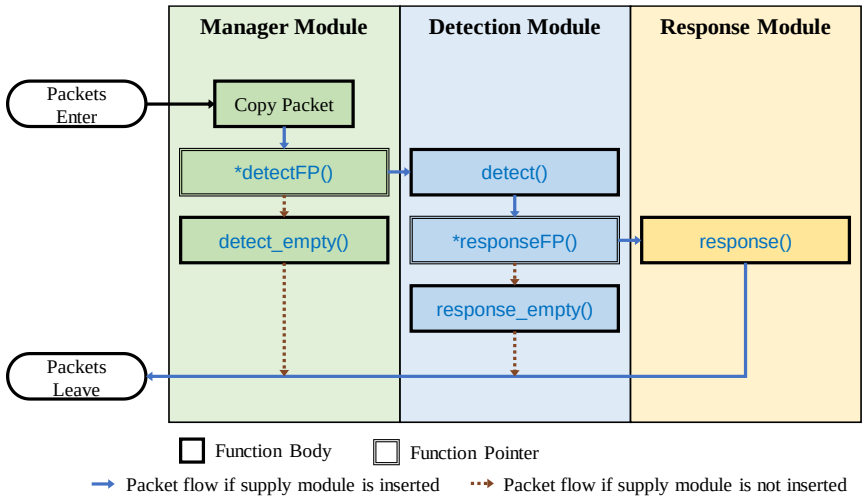


Fig. 2. mKIPS modular architecture and system processing flow

As a network packet enters the system, the kernel invokes the Manager module to begin the packet inspection flow. It then executes the function component (i.e., detect()) of the Detection module to detect attacks through the invocation of the function pointer (i.e., *detectFP()). The Detection module manages the rule database that stores detection rules and compares the packet payload with the detection rules. When an attack is detected, it will call the function component (i.e., response()) of the Response module to respond to attacks through the invocation of function pointer (i.e., *responseFP()). It will then call the corresponding response method to deal with the attack according to the type of attack. Finally, the control returns to the kernel's original packet handling flow to continue subsequent processing.

3.3 Detection Module and Detection Rules

The Detection module implements misuse detection and uses the detection rules in the signature database for threat detection. It compares the payload of the network packet with the attack signature database to determine the type of external attack. Our detection rules are derived from Snort[6]. After converting Snort's detection rules into our dedicated detection rule format, these rules are imported to construct all the AC Trees used for threat detection when the mKIPS system is initialized. The Detection module uses

the AC-BM algorithm [22] for fast string comparison, which combines the advantages of the Boyer-Moore [23] and Aho-Corasick [24] string comparison algorithms.

The mKIPS-specific rule format is shown in Figure 3. Each rule includes the response method, transport layer protocol type, TCP/IP information, warning message, and attack string. The design of the attack signature database built based on this is shown in Figure 4. The attack signature database is constructed as rule trees, and the corresponding rule tree is established according to the type of transport layer protocol. Each rule tree comprises a TCP/IP Tree Node containing TCP/IP information and several Rule Tree Nodes consisting of a response mechanism, warning message, and attack string, as shown in Figure 5.

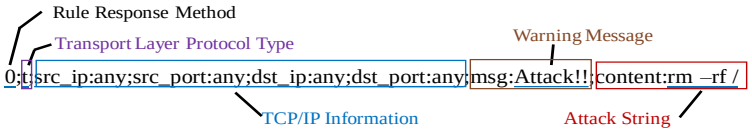


Fig. 3. The mKIPS-specific rule format.

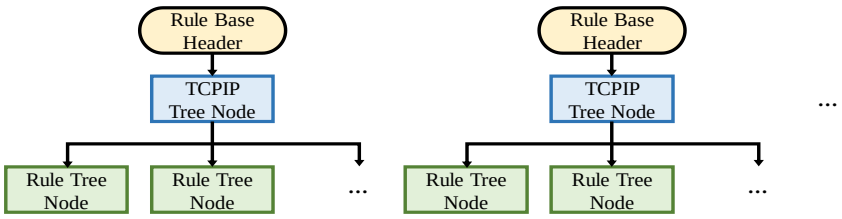
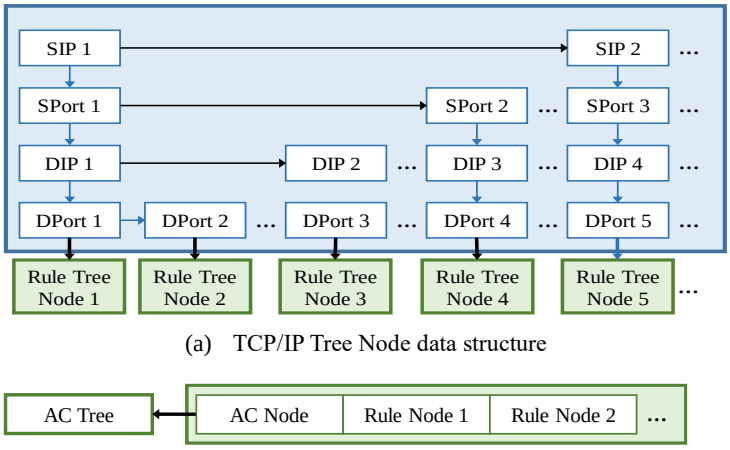


Fig. 4. The structure of the attack signature database.



(a) TCP/IP Tree Node data structure

(b) Rule Tree Node data structure

Fig. 5. The structure of TCPIP Tree Node and Rule Tree Node

The TCP/IP Tree Node structure shown in Figure 5(a) is composed of several SIP (Source IP), Sport (Source Port), DIP (Destination IP), and DPort (Destination Port).

The SIP will point to the SIP node on the right, the SPort node on the lower layer, and so on. The DPort at the end will point to the right DPort node and the Rule Tree Node that matches the TCP/IP information.

Figure 5(b) shows the Rule Tree Node structure consisting of an AC Node and several Rule Nodes. The AC Node records the number of Rule Nodes and points to the AC Tree jointly constructed by all Rule Nodes. Moreover, each Rule Node records the response mechanism, warning message, and attack string in a single rule.

When the Detection module examines a packet, it will start visiting from the corresponding rule tree according to the TCP/IP information of the packet. First, it will visit SIP, SPort, DIP, and DPort in the TCP/IP Tree Node. Then, the AC-BM algorithm is executed to search for attack strings by comparing the packet payload with the AC Tree of Rule Tree Nodes conforming to that TCP/IP Node.

3.4 Response Module and Reaction Mechanisms

When the Detection module detects a malicious packet, it will record the matching Rule Node and send it to the Response module to execute the response mechanism. The response mechanism that conforms to the rule is recorded in the Rule Node.

The data structure of the response mechanism is shown in Figure 6, and each response method occupies 1 bit as a switch. Four response methods are implemented: Alert (display warning message), Drop (discard packet), Reset Connection (interrupt the TCP/IP connection between the attacking end and the receiving end), and Logfile (record threat information). This design allows administrators to combine response mechanisms to form the required response mechanism for different detection rules. For example, the administrator can set the response mechanisms for a specific rule as Reset Connection and Logfile. The Unused bits are reserved for future expansion.

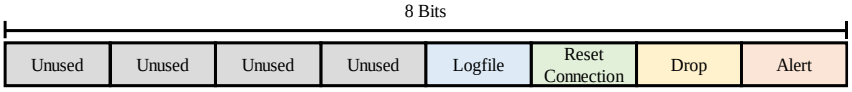


Fig. 6. Response mechanisms

3.5 Receive Packet Steering (RPS) Enabling

In addition to the lightweight and modular design, to allow the system to maintain stability and high performance when the network traffic is heavy, we study the Linux kernel's handling of network packets and utilize mechanisms to distribute packets to different cores for processing under a multicore system. For example, the administrator must correctly set the IRQ affinity setting to dispatch network packets to different cores to fully utilize the multicores performance under Linux.

Receive Packet Steering (RPS) [12-14] is a technology the Linux kernel selects for subsequent processing when Softirq is triggered. Since low-level NICs usually do not implement the function of hardware distribution of packets, resulting in the performance bottleneck problem of centralized processing of packets on a single core. Therefore, the Linux kernel triggers Softirq in the final stage of top-half interrupt execution

and distributes packets to different cores for processing. RPS bases on the NIC queues to set the affinity setting that can trigger Softirq on a specific core. The default setting does not turn on the RPS, that is, to prioritize triggering Softirq on the local core. Therefore, we use RPS to distribute packets to different cores for processing through software and hardware setting so that the loading of each core can be balanced as much as possible to make the most use of multicore system performance.

4 Experimental Results

This section evaluates the system performance of the proposed kernel-level IDPS named mKIPS and compares the effectiveness of the experimental system using mKIPS and user-level IDPS (i.e., Snort[6]).

4.1 Experimental Environment

In order to measure the impact of IDPS on system performance, this research takes the native system without IDPS as the base system and compares the performance of running the Web server in three system environments, including the base system, the system runs Snort IDPS, and the system runs the mKIPS IDPS. The same detection rules (i.e., snortrules-snapshot-2983) are used to compare the performance fairly. A total of 5432 rules are selected, each containing only one attack string for detection.

In the experimental environment we constructed, a server provides Web services, and an IDPS running on another machine operates in bridge mode and is connected to the router and the Web server. The client's request to the Web server will be sent to IDPS first and then forwarded to the Web server after being examined by IDPS. Likewise, the Web server's response to the client will also be sent to IDPS first and then forwarded to the client after being examined by IDPS. Five clients were used to generate a large number of requests to the Web server to obtain Web data for measuring the system performance. As more packets are received or sent by the Web server, more packets are inspected by IDPS. Therefore, IDPS's performance does affect system performance. The detailed software and hardware specifications are shown in Table 1.

Table 1. Software and hardware specifications of the experimental environment

	Clients	Web Server	IDPS
Processor	Intel i5-3470 3.2GHz	Intel i7 -7700 3.6GHz 4C8T	Intel i7-9700 3GHz 8C8T
Memory	Kingston 2G DDR3-1333 * 2	Kingston 16G DDR4-2666 * 2	Kingston 16G DDR4-2666 * 2
NIC	RTL8111/8168/8411	I219V	EXPI9301CTBLK * 2
OS	Ubuntu Server 18.04.1 LTS	Ubuntu Server 18.04.1 LTS	Ubuntu Server 18.04.1 LTS
Kernel	Linux Kernel 4.15.0	Linux Kernel 4.15.0	Linux Kernel 4.15.0
Benchmark	ApacheBench 2.3	-	-
Web Server	-	Apache 2.4.29 [26]	-
IDPS			mKIPS / Snort 2.9.15.1

ApacheBench [25] is a performance testing tool measuring Web server performance. Each client used ApacheBench during the experiment to measure the Web server performance by sending 100,000 requests with 1,000 concurrent connections. A total of 5,000 concurrent connections and 500,000 requests were sent. Each experiment was tested ten times, and the average value was calculated.

4.2 Experimental Results

We first measured the Softirq distribution for sending and receiving packets under different RPS settings. The default setting is off RPS. Table 2 shows that in the experimental environment, with the default off RPS setting, sending and receiving packets are wholly concentrated on Core 5 to trigger Softirq. After using the RPS technology, sending and receiving packets can be effectively distributed to each core for processing.

Table 2. Softirq distribution for sending and receiving packets under different RPS settings.

	RPS Configuration	
	Default (Disable)	RPS Enable
Core 0	0	896168
Core 1	0	880016
Core 2	0	880984
Core 3	0	893637
Core 4	0	894027
Core 5	6802114	890352
Core 6	0	882078
Core 7	0	904947
Total	6802114	7122207

We then measured system performance under different RPS settings. The experimental results of the transfer rate measurement are shown in Figure 7. The results show that using the RPS mechanism can solve the performance bottleneck problem. When RPS is not enabled, due to inspecting each packet for threat detection, mKIPS will cause 52.73-53.35% performance loss compared with the base system. The base system stands for the native Linux system without running any IDPS. When RPS is turned on, since sending and receiving packets can be effectively distributed to each core for processing, system performance with running mKIPS can be significantly improved. Besides, in the bridging environment, the sending and receiving of packets between the Web server and clients for IDPS are to receive and then send packets, and the kernel of the IDPS simply forwards packets. Running mKIPS will not significantly affect the overall system performance when the RPS setting is enabled.

The experimental results of the detection rate measurement in Figure 8 show that even if the RPS setting is turned on, the user-level Snort still cannot perform packet detection well under such heavy network traffic, and the packet drop rate reaches 66.44-69%. Whereas the kernel-level mKIPS can have a very high detection rate and performance. Therefore, it can effectively utilize the function of IDPS to protect the system.

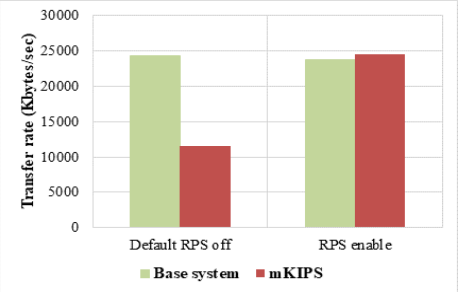


Fig. 7. Transfer rate comparison.

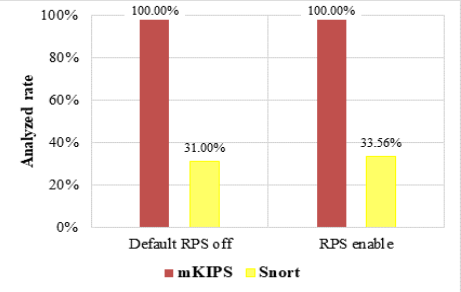


Fig. 8. Packet analyzed rate comparison.

5 Conclusions

We have designed and implemented a lightweight modular kernel-level IDPS named mKIPS. It adopts signature-based detection, inspecting each network packet to find malicious patterns in known attacks. mKIPS is implemented as a loadable kernel module that can be dynamically loaded into the Linux kernel during run time. Its modular architecture can support dynamic addition/deletion/replacement of functional components. Its implementation employs the netfilter framework, and all packets entering or leaving the system can be examined with in-place packet inspection. Furthermore, this work distributes packets to different cores for processing through software and hardware setting to make the most use of multicore performance.

Compared with user-level IDPS, mKIPS operating in the kernel can detect threats immediately after receiving packets. It needs not the overhead of copying packets to the user buffer for inspection. Furthermore, it does not need to wait for the scheduling to execute user-level IDPS and switch the protection domain back and forth between kernel mode and user mode for processing. These significantly reduce the impact on system performance. Experimental results show that mKIPS incurs less overhead on system performance and effectively ensures system safety with a high detection rate.

References

1. H. J Liao, C. H. R. Lin, Y. C. Lin, and K. Y. Tung, “Intrusion detection system: a comprehensive review,” Journal of Network and Computer Applications, Vol. 36, Issue 1, 2013, pp. 16-24.
2. A. Patel, M. Taghavi, K. Bakhtiyari, and J. C. Júnior, “An intrusion detection and prevention system in cloud computing: A systematic review,” Journal of Network and Computer Applications, Vol. 36, Issue 1, 2013, pp. 25-41.
3. C. Modi, D. Patel, H. Patel, B. Borisaniya, A. Patel, and M. Rajarajan, “A survey of intrusion detection techniques in Cloud,” Journal of Network and Computer Applications, Vol. 36, Issue 1, pp. 42-57, 2013.
4. U. Kumar and B. N. Gohil, “A Survey on Intrusion Detection Systems for Cloud Computing Environment,” International Journal of Computer Applications, Vol. 109, No. 1, pp. 6-15, 2015.

5. J. D. Araújo and Z. Abdelouahab, “Virtualization in Intrusion Detection Systems: A Study on Different Approaches for Cloud Computing Environments,” International Journal of Computer Science and Network Security, Vol.13, No.11, pp. 135-142, 2013.
6. Snort, https://www.snort.org, last accessed 2023/04/30.
7. Suricata, https://suricata-ids.org, last accessed 2023/04/30.
8. M. L. Chiang, J. K. Wang, L. C. Feng, Y. S. Chen, Y. C. Wang, and W. Y. Kao, “Design and Implementation of a Lightweight Kernel Level Network Intrusion Prevention System for Virtualized Environment,” 13th International Conference on Information Security Practice and Experience, 13-15 Dec, 2017, Melbourne, Australia, Dec. 2017.
9. T. Chin, K. Xiong, and M. Rahouti, “Kernel-Space Intrusion Detection Using Software-Defined Networking,” Security and Safety, 5(15):pp. 155-168, Jul. 2018.
10. T. H. Cheng, Y. D. Lin, Y. C. Lai, and P. C. Lin, “Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems,” IEEE Communications Surveys and Tutorials, Vol. 14, No. 4, pp. 1011-1020, 2012.
11. The Linux Kernel Archives, http://www.kernel.org, last accessed 2023/04/30.
12. Linux Network Scaling: Receiving Packets, https://garycplin.blogspot.com/2017/06/linux-network-scaling-receives-packets.html, last accessed 2023/04/30.
13. Scaling in the Linux Networking Stack, https://www.kernel.org/doc/Documentation/networking/scaling.txt, last accessed 2023/04/30.
14. Performance Tuning Guide, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index, last accessed 2023/04/30.
15. libpcap, https://www.tcpdump.org, last accessed 2023/04/30.
16. Cisco Talos Intelligence Group, https://www.talosintelligence.com, last accessed 2023/04/30.
17. R. T. Gaddam and M. Nandhini, “Analysis of Various Intrusion Detection Systems with a Model for Improving Snort Performance,” Indian Journal of Science and Technology, Vol 10(20), DOI: 10.17485/ijst/2017/v10i20/108940, May 2017.
18. W. Yuan, J. Tan, and P. D. Le, “Distributed Snort Network Intrusion Detection System with Load Balancing Approach,” Proceedings of the International Conference on Security and Management (SAM), Athens, 2013.
19. S. A. R. Shah and B. Issac, “Performance comparison of intrusion detection systems and application of machine learning to Snort system,” Future Generation Computer Systems, Vol. 80, pp. 157-170, Mar. 2018.
20. S. Shah and S. P. Bendale, “An Intuitive Study: Intrusion Detection Systems and Anomalies, How AI can be used as a tool to enable the majority, in 5G era,” 2019 5th International Conference on Computing, Communication, Control and Automation (ICCUBEA), Pune, India, Sept. 19-21, 2019, pp. 1-8, doi: 10.1109/ICCUBEA47591.2019.9128786.
21. Netfilter, https://www.netfilter.org, last accessed 2023/04/30.
22. C. J. Coit, S. Staniford, and J. McAlemey, “Towards faster string matching for intrusion detection or exceeding the speed of Snort,” in Proceedings of DARPA Information Survivability Conference & Exposition II, Vol. 1, 2001, pp. 367-373.
23. R. S. Boyer and J S. Moore, “A fast string searching algorithm,” Communications of the ACM, Vol. 20, No. 10, Oct. 1977, pp.762-772.
24. A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” Communications of ACM, Vol. 18, No. 6, June 1975, pp. 333-340.
25. ApacheBench, https://httpd.apache.org/docs/2.4/programs/ab.html, last accessed 2023/04/30.
26. Apache Server, https://httpd.apache.org, last accessed 2023/04/30.