

npf—A Simple, Traffic-Adaptive Packet Classifier Using On-line Reorganization of Rule Trees

Shariful Hasan Shaikot and Min Sik Kim
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington 99164–2752, USA
Email: {sshaikot,msk}@eecs.wsu.edu

Abstract—Packet classification is one of the crucial components of application such as firewalls, intrusion detection, and differentiated services. For example, an intrusion detection system (IDS) classifies packets either as benign or malicious and alerts the network administrator when hostile traffic is detected. Since existing IDS spend the majority of CPU time in packet classification, an IDS fails to detect malicious packets under high load. Many ideas have been proposed to make the packet inspection faster so that an IDS spends less time in packet classification. However, because of the increasing number of security threats and vulnerabilities, the number of rules often exceeds thousands, requiring more than hundreds of megabytes of memory. As a result, an IDS spends longer time to classify packets since each packet incurs many memory accesses, and thus the throughput of an IDS is limited by memory bandwidth. The problem can be mitigated by exploiting locality in traffic patterns. In this paper, we propose npf, a fast and traffic-adaptive packet classifier which dynamically reorganizes the internal data structure based on the traffic pattern. Unlike existing approaches requiring a separate, off-line reorganization phase, npf performs reorganization on-line with little overhead, resulting in higher throughput without compromising accuracy. Experimental results on our test bed show that npf outperforms a traditional packet classifier by spending an order of magnitude less time per packet in order to classify the packet.

I. INTRODUCTION

Even with the most advanced protection computer network systems are still not 100% secure. Consequently, Network intrusion detection systems (NIDS) becomes an important part of any network security architecture to discover and react to computer attacks. They provide a layer of defense which monitors network traffic for predefined suspicious activity or patterns, and alert system administrators when potential hostile traffic is detected. In order to detect suspicious activity, IDS monitor packets on the network passively and perform packet classification by comparing them against a database of rules (rule-set) that characterize the pattern of malicious threats. However, searching the rule database and finding rules that match incoming packets (i.e. packet classification phase) consume the majority of CPU time. For instance, open-source IDSs such as Snort [1] and Bro [2] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environment [3]. Since IDS spend the majority of CPU time in packet classification, IDS often fails to classify packets in high speed network if the underlying packet classification technique is not optimized

and consequently malicious packet enters into the network. Packet classification algorithms use two dominant resources, memory and time. Many ideas [4]–[9] have been proposed to speed up the packet inspection phase (i.e. packet classification time) so that IDS does not consume long time to classify packet and does not fail to detect malicious packet in high speed network. However, because of the increasing number of security threats and vulnerabilities, the number of rules often exceeds thousands requiring more than hundreds of megabytes of memory. As a result, an IDS spends longer time to classify packets since each packet incurs many memory accesses. Cheap DRAM can be used to fulfill the need for large memory. However, speed requirements does not allow to use this approach. In order to improve the classification time, most of the existing packet classification algorithms [10]–[13] exploit the characteristics of rule-set. Our research focuses on optimizing packet classifier’s performance by exploiting the locality in the traffic pattern. This optimization can contribute to the overall performance improvement of an IDS.

In this paper, we propose the design and implementation of npf, a fast and traffic adaptive packet classifier which utilizes the traffic pattern to improve the classification speed. In order to exploit the locality in traffic pattern, npf dynamically reorganizes its internal data structure (rule tree) so that the search time for frequently visited rules are reduced. As a result, using npf inside the detection engine, an IDS can detect malicious packets in high speed networks without compromising accuracy. Our approach generates an interval-based rule tree (IBT) for each of the four protocol fields (source IP, destination IP, source port, and destination port) for each rule in the rule set. npf uses IBT in order to organize rules for faster access. Our approach views the IP address and port number as a range of addresses or interval of numbers. For example, 16 bit port field has the range of 0 to $2^{16} - 1$ or in other words interval [0, 65535]. We choose a binary tree as underlying data structure in order to create the IBT for each protocol fields. In addition of keeping the pointer information to the left child and right child, we also keep the interval, list of ruleIDs and popularity counter in the tree node. When a packet arrives, we extract the packet header information and then traverses the four different IBT in order to find a matching node in each IBT. After the tree traversal step we obtain four different list of ruleIDs by collecting the list of ruleIDs stored

in each matching node. We perform intersection operation \cap amongst the 4 list of ruleIDs to find out the common ruleID. The intersection operation finally selects either a single ruleID or \emptyset . If the result of $\cap \neq \emptyset$, we collect the corresponding ruleAction of the common ruleID. This ruleAction determines the fate of the packet—“benign” or “malicious”. In case of $\cap = \emptyset$ at any step in npf execution, the fate of the packet is determined by the default ruleAction set by the administrator.

In order to make npf traffic adaptive, we include the notion of popularity in tree nodes. If any particular range of IP addresses or port numbers is frequently seen in the incoming traffic, we apply the heuristic of moving up the node associated with that range or interval towards the root of the tree to reduce the search time even further. We implement npf in the NetBSD operating system [14]. Experimental results show that npf spends less time in classifying a packet either as benign or malicious. As a result, using npf an IDS can process more packet in high speed network without compromising the detection accuracy.

The remainder of the paper is organized as follows. Related work is summarized in Section II. In Section III, we review the background of traditional IDS and its performance issue. In Section IV, we describe the design and working procedure of npf in detail. Experimental results are presented in Section V before we conclude in Section VI.

II. RELATED WORK

The Packet classification/filtering optimization has been studied extensively in the research literature. In order to improve the performance of packet classifier, specialized data structures, and heuristics based packet classification algorithms have been proposed. However, most of the current packet classification techniques do not consider the pattern of the incoming traffic in optimizing their search data structures.

Srinivasan et al. [12] build a table of all possible field value combinations (cross-products) and pre-compute the earliest rule matching each cross-product. Search can be done quickly by doing separate lookups on each field, pasting the results together into a crossproduct, and indexing into the crossproduct table. Unfortunately, the size of this table grows unexpectedly with the number of rules.

The need for speed can be finessed by pipelining. Gupta and McKeown [13] proposed a heuristic approach called Recursive Flow Classification (RFC). One advantage of RFC is that the various lookup stages can be pipelined, so in a hardware implementation the classifier can have a very high throughput. However, increased pipeline depths add expense and this approach does not scale easily to medium or large number of rules because it takes excessive storage.

The works exploiting the traffic pattern for optimizing the performance of packet classification was discussed in [15]–[18]. The technique presented in [15] was limited to only one field (routing prefix) that has a given frequency distribution and in [16], the technique was not able to scale with the number of fields if to be used for IDSs. The technique

in [17] used whole rules without exploiting the traffic patterns over separate fields. However, Adel et al [18] utilized traffic pattern over separate fields to obtain adaptive methods that can accommodate varying traffic statistics for achieving a high throughput. However, their data structure reorganization was done in offline which may reduce the throughput resulting in mediocre performance.

The simplest classification algorithm is a linear search through the rules of the classifier. For example, OpenBSD’s packet filter/classifier PF filters packet based on its rule set stored in the configuration directory. The implementation technology (usage of “pseudo device driver” for communication between user space and kernel space) and environment (NetBSD OS) of npf is quite similar to PF. However, the packet classification algorithm and the underlying data structure used by npf in order to organize rules differ significantly with PF and as a result we observe a significant performance improvement in npf.

III. BACKGROUND

Packet classification is the mechanism that enables the routers or IDS to classify the traffic and process them according to their needs. The importance of packet classification continues to grow. Both the core and the edge of the Internet are growing in speed. As a result, demand for fast and efficient packet classification technique is also increasing in order to provide QoS and security. Packet classification is important for application such as Intrusion Detection System (IDS) for taking security-related actions (e.g., dropping packets sent from a certain subnet). Snort [1] and Bro [2], popular open-source rule based IDSs use packet classification inside the detection engine to distinguish the benign traffic from the malicious traffic.

A. Intrusion Detection in Traditional IDS

A traditional rule-based intrusion detection system compares the incoming packets against rule set in order to detect intrusion. A common approach is to build a graph such as rule trees or finite automata for a given rule set, and traverse it using a packet as an input string. Because of the increasing number of security threats and vulnerabilities, the number of rules often exceeds thousands requiring more than hundreds of megabytes of memory. Exploring such a huge graph is a major bottleneck in high-speed networks since each packet incurs many memory accesses and as a result IDS spend longer time to classify packets.

B. Introducing Snort

Snort [1] is a popular open source, cross-platform, lightweight network intrusion detection tool that can be deployed to monitor small TCP/IP networks and detect a wide variety of suspicious network traffic. Upon detecting malicious traffic, Snort generates alarm and sends messages to the network administrator to aid them to take proper action. Snort can detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and much more.

```

alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES directory listing";
flow:established; content:"Volume SerialNumber"; classtype:bad-unknown; sid:1292;)

```

Fig. 1. Example of a Snort rule

C. Intrusion Detection in Snort

Snort uses a simple language to define rules that characterize the pattern of the malicious activities. Fig. 1 shows an example of a typical Snort rule. Each rule consists of five mandatory fields and numerous option fields. The mandatory fields include protocol type (e.g., TCP, UDP), source/destination IP addresses and port numbers, all of which are part of a packet header. Snort interprets keywords enclosed in parentheses as “option fields”. Commonly used option fields are “content” (Search the packet payload for the a specified pattern), “msg” (Sets the message to be sent when a packet generates an event) etc. For example, a packet matches the mandatory fields of the rule in Fig. 1 if it belongs to an established TCP stream from HOME_NET to EXTERNAL_NET regardless of its port numbers (any). Once such a packet is identified, its payload is searched for the content string “Volume SerialNumber”, which is specified as an option field. If a packet that matches all the fields in the rule is detected, Snort generates a message with a label “ATTACK-RESPONSES directory listing”. HOME_NET and EXTERNAL_NET are variables defined by the administrator, representing the IP address prefixes of the local and external networks, respectively.

npf uses Snort rule-set as its classification policy.

D. Performance Issues

The performance of an IDS depends on many factors including the computational power of the machine, the network load, and the size of the rule database. The computational power and the network load are external factors, which an IDS cannot control. On the other hand, efficient management/organization of the rule database is closely related to the internal structure of the packet classifier used by the IDS. Dreger et al. [3] pointed out, IDS exhausts CPU time and memory as network speed increases. This result raises a question: which function in IDS is the bottleneck? In other words, which function consumes the most CPU time? In our previous study [19] we found out that packet classification phase is the most expensive operation in IDS. Our measurement revealed that IDS consumes up to 70% of total CPU time in the detection engine. Therefore, reducing consumption of CPU time in packet classification inside the detection engine of IDS is critical for improving the overall performance of IDS.

IV. PROPOSED APPROACH

We describe the design and working procedure of npf in detail in the following sections. The high level design of npf is shown in Fig. 2. The main difference between npf and the traditional packet classifier lies on npf’s ability to dynamically adapt to the pattern of the incoming traffic to improve the packet classifier’s performance. In traditional packet classification, the packet classifier classifies packets based on

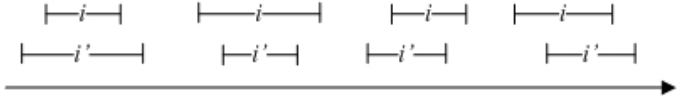


Fig. 3. The interval trichotomy for two closed intervals i and i' . If i and i' overlap, there are four situations: in each $low[i] \leq high[i']$ and $low[i'] \leq high[i]$

a rule set and being unaware of the traffic pattern. However, npf goes one step further. In addition of performing its main duty—“Packet classification”, npf intelligently detects the pattern of the incoming traffic and then reorganizes its internal structure according to the traffic pattern in order to reduce the classification time even further. This self adjustment based on traffic pattern is one of the underlying reasons of npf’s better performance.

Our proposed design mainly consists of 2 phases. In phase I, we read the database of rules (right side of Fig. 2) that characterize the pattern of malicious threats and then build the IBT (inside kernel box of Fig. 2) for each of the four protocol fields (source IP, destination IP, source port, and destination port). In phase II, we traverse all IBTs to find out the type of the incoming packet—malicious or benign.

A. Phase I: Construction of Interval based Rule Tree IBT

npf uses a special data structure called “Interval based Rule Tree (IBT)” in order to efficiently organize the rules in the rule-set. In order to construct the IBT, we consider 4 protocol fields: Source IP, Destination IP, Source Port, and Destination Port from Snort rule. We build 4 different tree structures IBT for these 4 different protocol fields. We name the tree structures as “Interval based Rule Tree (IBT)” because everything (IP address space and port number space) is considered in terms of interval or range. For example, 16 bit port field ranges from 0 to 65535. Thus we say that port field has interval $[0, 65535]$. We can represent an interval $[t_1, t_2]$ as an object i , with fields $low[i] = t_1$ (the low endpoint) and $high[i] = t_2$ (the high end point). We say that the interval i and i' overlap if $i \cap i' \neq \emptyset$, that is, if $low[i] \leq high[i']$ and $low[i'] \leq high[i]$. Any two intervals i and i' satisfy the *interval trichotomy*; that is, exactly one of the following three properties holds [20]:

- 1) i and i' overlap
- 2) i is to the left of i' (i.e., $high[i] < low[i']$)
- 3) i is to the right of i' (i.e., $high[i'] < low[i]$)

Fig. 3 shows the all possible interval trichotomy for two closed intervals i and i' .

If the interval specified by two rules overlap (partially or fully), the higher priority rule will overwrite the lower priority rules in the IBT. We evaluate the priority of the rule using the following condition—“shorter the length of the

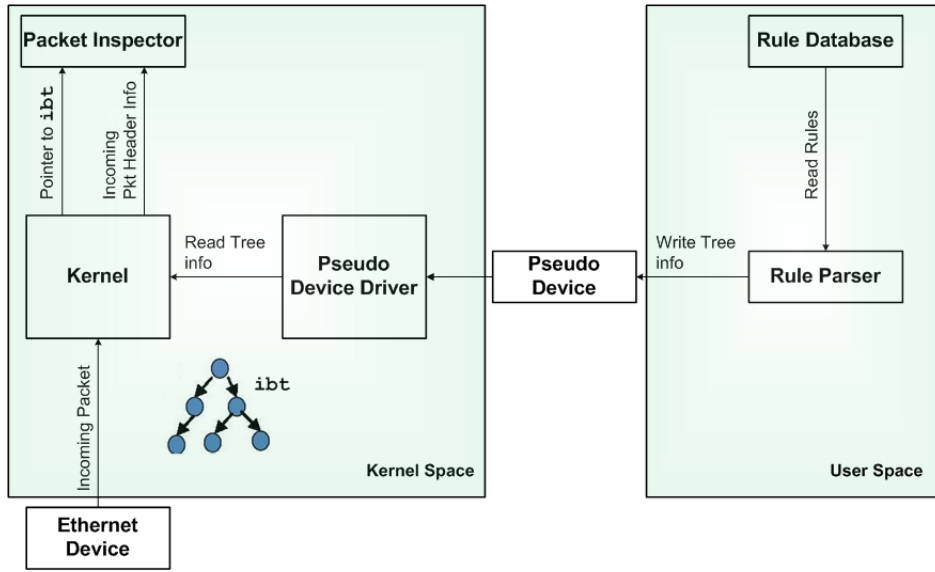


Fig. 2. High level view of npf

TABLE I
A SMALL SNORT RULE SET CONTAINING ONLY 2 RULES

ruleID	Snort rules
1	alert tcp any !600:610 → any 80 (msg: "Test Rule 1")
2	alert tcp any :630 → any 1024: (msg: "Test Rule 2")

interval, higher the priority". For example, a rule with interval [600, 610] will have higher priority than the rule with interval [0, 65535] because the interval length of the former rule is shorter than that of the latter rule.

Each node x of the tree contains $\langle \text{begin range, end range, popularity, \{list of ruleID\}} \rangle$ tuple. `begin range` is the low end point of the interval, `end range` is the high end point of the interval. For example, in case of 16 bit port field that has interval [0, 65535], `begin range` = 0 and `end range` = 65535. `popularity` defines the frequency of the interval [begin range, end range] seen in the incoming traffic. `\{list of ruleID\}` stores the ruleIDs that contains the interval [begin range, end range] in rule set. The IBT is created based on the *binary search tree property* and *max-heap property*. Let x be a node in IBT. If y is a node in the left subtree of x , then $\text{end_range}[y] < \text{begin_range}[x]$. If y is a node in the right subtree of x , then $\text{end_range}[x] < \text{begin_range}[y]$. An in-order tree walk of IBT lists the intervals in sorted order by low end point or `begin range`.

The *max-heap property* mandates that for any non-root node the popularity must be less than to the popularity of its parent. Thus, the root node is the maximum-popular node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

We initialize the popularity of each interval node based on the number of its descendants. Then, on each instance of

a matching node we increase its popularity and compares its popularity to its parent's popularity in order to make sure the IBT maintains max-heap property. On violation of max-heap property, we promote the higher popular nodes towards the root of the tree and demote the lower popular nodes towards the leaf of the tree. Consequently, frequently matched nodes would be more likely to be near the root of the tree, causing searches for them to be faster [21]. The IBT is organized in such a way that all the nodes encompass the entire protocol field space, for example, in case of port field, it encompasses from 0 to 65535.

Fig. 5 describes the steps involved in IBT construction. For the sake of simplicity, we omit the popularity counter from the node in the tree. The node contain `begin range`, `end range` and `list of ruleID`. We construct IBT for destination port of Snort rule set (Table I). Fig. 5(a) is the default node in the IBT at the beginning of IBT construction. Here, `begin range` = 0, `end range` = 65535 and `list of ruleID` = {0}. `ruleID 0` refers to the default `ruleAction`. When we parse the `ruleID 1` from Snort rule set (Table I), the destination port is 80. Since it is a non-range value, this rule will have the following information for destination port: `begin range` = 80, `end range` = 80 and `list of ruleID` = {1}. When we traverse the current IBT (Fig. 5(a)), we find out that interval [80, 80] overlaps with the interval [0, 65K] (interval trichotomy 1). Note that interval [80, 80] has the higher priority than the interval [0, 65K] since the length of interval [80, 80] = 0 is shorter than the length of interval [0, 65K] = 65K. Therefore, interval [80, 80] will overwrite the current node and the intermediate IBT (Fig. 5(b)) is constructed. We provide a marking system to the node in IBT (Fig. 5(b)) for ease of reference. Note that this is not part of our design. Now, we parse the `ruleID 2` from Snort rule set. The destination port here is "1024:" which means

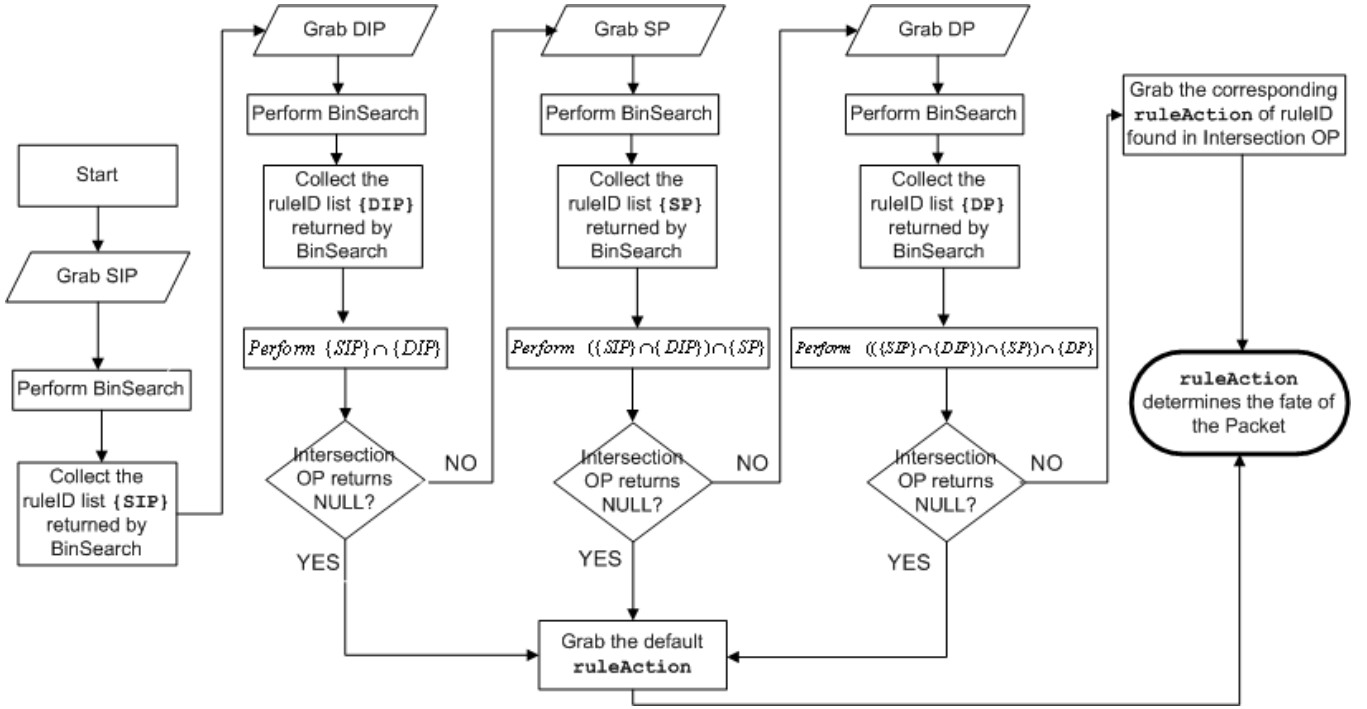


Fig. 4. Flowchart of npf. “SIP” means Source IP, “DIP” means Destination IP, “SP” means Source Port and “DP” means Destination Port. “BinSearch” refers to Binary Search. $(\{SIP\} \cap \{DIP\})$ means Intersection operation between the ruleID list of SIP and DIP

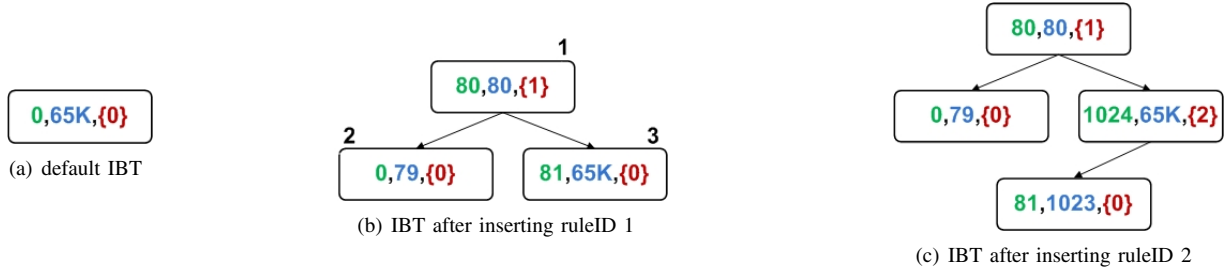


Fig. 5. Construction of Interval based Rule Tree IBT

≥ 1024 . It implies that ruleID 2 has the interval $[1024, 65K]$. We traverse the current IBT (Fig. 5(b)) to find out its position in order to insert it into the tree. At node 1, we find out that interval $[80, 80]$ is to the left of the interval $[1024, 65K]$ (interval trichotomy 2). So we make a right branch traversal. At node 3, we find out that interval $[81, 65K]$ overlaps with the interval $[1024, 65K]$. Since the interval $[1024, 65K]$ has the higher priority than the interval $[81, 65K]$, it overwrites the node 3 and IBT (Fig. 5(c)) is constructed. Without loss of generality, using this procedure the IBT can be constructed for each of the four protocol fields regardless of the number of rules in the rule set.

B. Phase II: Packet Classification

The search procedure of npf is given as flow chart in Fig. 4. During classification, the IBT tree is traversed based on the packet header, and a matching node is identified. When a packet arrives at Network Interface Card (NIC), we extract the value of the 4 protocol fields from the packet header (SIP, DIP,

SP and DP). We traverse IBT to find out a matching node that contain the interval. For example, when we traverse source IP IBT to find a matching node for IP address SIP, we look for the node such that $SIP \in \text{interval of the node}$. We do this search operation for a matching node in IBT for other three protocol fields as well. Traversing the trees eventually gives 4 matching nodes. We collect the corresponding list of rules (ruleID list) associated with the matching nodes. As a result, we get 4 different sets of ruleID list from four different IBT. We then perform Intersection Operation (\cap) amongst these four ruleID lists in the order of $\{SIP\} \cap \{DIP\}$, $(\{SIP\} \cap \{DIP\}) \cap \{SP\}$ and $((\{SIP\} \cap \{DIP\}) \cap \{SP\}) \cap \{DP\}$. Here, $\{SIP\}$ means the ruleID list extracted from the matching node while searching for SIP in source IP IBT. The intersection operation finally selects either a single ruleID or \emptyset . If the result of $\cap \neq \emptyset$, we collect the corresponding ruleAction of the matching ruleID. This ruleAction determines the fate of the packet—“benign” or “malicious”. In case the result of $\cap = \emptyset$ at any step in npf execution, the

fate of the packet is determined by the default `ruleAction` set by the Administrator.

Note that in order to make `npf` traffic pattern adaptive, we update the popularity counter for the matching node (i.e. a specific interval) during tree traversal operation. We then compare the matching node’s popularity with its parent node’s popularity. If the *max heap property* is violated (i.e. popularity of child node $>$ popularity of parent node) then we apply the heuristic to promote the higher popular nodes towards the root of the tree. As a result, searching time for the frequently visited rules is reduced further which yields significant improvement in packet classification time of `npf`.

C. `npf`’s Packet Classification Demonstration

In this section, we demonstrate the packet classification procedure with the help of a simple example. Table I is a very simple Snort rule set that contain only 2 rules. For the sake of simplicity and brevity we keep the number of rules in this rule set small. Rule 1 will generate an alarm if a packet is from any source port other than the interval 600 – 610 and destined for port number 80 regardless of its source and destination IP addresses. On the other hand, Rule 2 will generate an alarm if a packet is from any source port less than 630 (because of `:` operator) and destined for any port number greater than or equal to 1024 regardless of its source and destination IP addresses. The four IBT’s (Fig. 6) are constructed according to the procedure mentioned in section IV-A.

TABLE II
PACKET HEADER INFORMATION

source IP	69.166.48.145
destination IP	69.166.49.30
source port	49000
destination port	80

Now suppose a packet arrives at NIC whose header information is listed in Table II. At first we traverse the source IP IBT (Fig. 6(a)). Since this tree has only one node the `ruleID` list that we obtain from this tree is $\{SIP\}=\{1,2\}$. We do the same for destination IP IBT and obtain $\{DIP\}=\{1,2\}$. Now we search source Port IBT in order to find a matching node for source port 49000 and we obtain $\{SP\}=\{1\}$ because 49000 lies between the interval $[630, 65K]$ at the right most leaf node of (Fig. 6(c)). In the same way, we do search destination Port IBT in order to find a matching node for destination port 80 and we obtain $\{DP\}=\{1\}$. When we perform the intersection operation $((\{SIP\} \cap \{DIP\}) \cap \{SP\}) \cap \{DP\}$, the final result of $\cap = \{1\}$. We now collect the `ruleAction` of `ruleID 1`. According to Table I, `ruleID 1` is “alert”. Hence, this packet will be classified as a malicious packet by `npf`.

Let us consider that another packet arrives with the same header information except the source port equals to 620. Performing search in all IBT’s will yield the following `ruleID` list: $\{SIP\}=\{1,2\}$, $\{DIP\}=\{1,2\}$, $\{SP\}=\{2\}$, and $\{DP\}=\{1\}$. The intersection operation $((\{SIP\} \cap \{DIP\}) \cap \{SP\}) \cap \{DP\} \neq \emptyset$. However, $((\{SIP\} \cap \{DIP\}) \cap \{SP\}) \cap \{DP\} = \emptyset$. There-

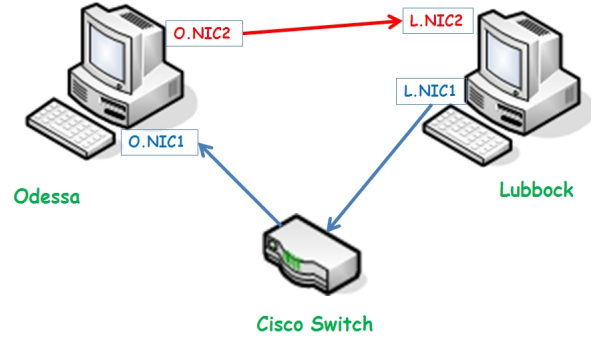


Fig. 7. A simple view of the Experimental Test bed

fore, the fate of this packet will be determined by the default `ruleAction`.

If the network administrator decides to accept all packets that match the default rule in the rule set then this packet will be classified as “benign” packet otherwise this packet will be classified as “malicious”.

V. EXPERIMENTAL RESULTS

We implement the proposed approach in the NetBSD kernel, by writing a new packet filter using NetBSD’s `pf` framework [22].

A. Experimental Test bed

We use a test bed to carry out the experiment and perform comparison study. The experimental test bed contains two hosts and a switch as shown in Fig. 7.

Both hosts, Lubbock and Odessa, have two network interface cards (NIC): L.NIC1, L.NIC2 and O.NIC1, O.NIC2 respectively. The mix of malicious and benign traffic is generated at Lubbock and are sent to the O.NIC1 through L.NIC1. Both our packet filters `npf` and OpenBSD’s PF are installed at Odessa, which has a 3 GHz CPU and 2 GB memory, and they passively monitor packet arriving at O.NIC1. If the packet is benign, IDS forwards the traffic through O.NIC2 to L.NIC2 otherwise it blocks the traffic and reports to the system administrator. The traffic mix was captured using `tcpdump` at different machines of our lab and then replayed back from Lubbock using `tcpreplay` [23].

B. Performance evaluation

In this section, we report on experimental studies on the performance of `npf`. We compare the performance of `npf` with OpenBSD’s packet filter PF because their functionality is same —“classify packet”. We tried to make the generated traffic mix as random as possible and we repeat the experiments for multiple times to obtain the average of all the data.

1) *Performance metrics*: We define the following metrics to analyze the performance of our proposed approach.

- *Packet Classification Accuracy*: Amount of accurateness in packet classification
- *Time spent per packet*: Average time spent per packet by the packet classifier in order to classify the packet

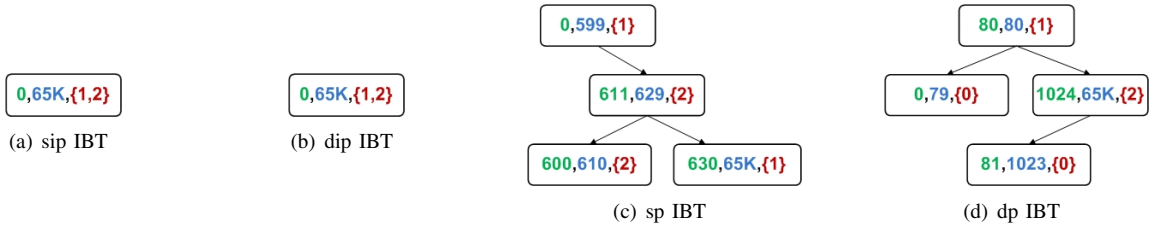


Fig. 6. Interval based Rule Tree IBT for four different protocol fields. “SIP” means Source IP, “DIP” means Destination IP, “SP” means Source Port and “DP” means Destination Port. The legend of the node is as follows: begin of interval, end of interval, {list of rule id}

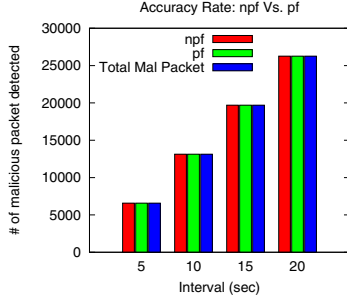


Fig. 8. Packet classification accuracy of npf and PF

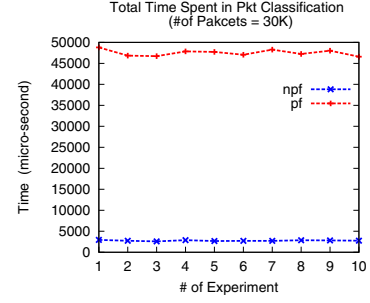


Fig. 9. Total time spent by npf and PF to classify packets

We use these metrics to evaluate and quantify the performance of our proposed design npf under different scenarios and configurations in the following section.

2) *Comparison of Packet Classification Accuracy*: We conduct the first experiment to measure the accurateness in classifying packets both by npf and PF.

Fig. 8 shows that the accuracy of packet classification of both npf and PF using a set of 1000 rules in the rule database. The interval length in X-axis shows the duration of packet generation. According to Fig. 8 both npf and PF detects malicious packet with 100% accuracy which implies their classification accuracy. In order to find out the accuracy of classification, we keep accounting on how many malicious packets we inject in the traffic mix so that we can use this information as ground truth. We modify the PF code in such a way that it keeps the information about the number of packets classified as malicious. We do the same for npf as well. We compare these values with our ground truth data and then plot the graph.

3) *Packet-Processing Time*: We conduct the second experiment to compare the packet-processing time of both npf and PF.

Fig. 9 shows that total time spent by npf and PF for classifying 30000 packets while the rule set contains 1000 rules. The time recorded in unit of microsecond. The processing time between npf and PF does differ significantly. While PF spends more than 47500 microseconds on average, npf only spend 2700 microseconds on average. The npf outperforms PF with more than 90% reduction in packet classification time.

Fig. 10 shows the average time spent per packet by npf while the rule set contains 1000 rules. We do not observe any significant fluctuation and hence we can conclude that on

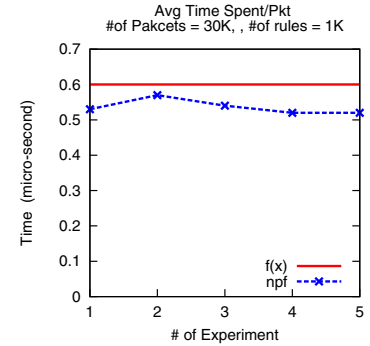


Fig. 10. Average time spent per packet by npf to classify packet

average npf does not spend more than 0.6 microsecond ($f(x)$ in Fig. 10) if the rule set contains 1000 rules. As a result, npf is able to classify more than 1.5 million packets/sec.

Fig. 11 shows that PF spends 33 microseconds per packet in order to classify it where as npf spends less than 1 microsecond, specifically 0.54 microsecond on average.

The reason npf classifies packets faster is because of the advantage that we get from the dynamic reorganization of its internal data structure of IBT which enables npf to retrieve information necessary for packet classification by expending less amount time. In particular, when we observe that a specific IP address or port number is frequently seen in the incoming traffic, we promote the node associated with the interval that include that particular IP address or port number towards the root of the IBT. Consequently, traversal time of the tree IBT is reduced even further next time when that IP address or port number is seen yielding significant reduction in packet classification time over the long period. Also, when we

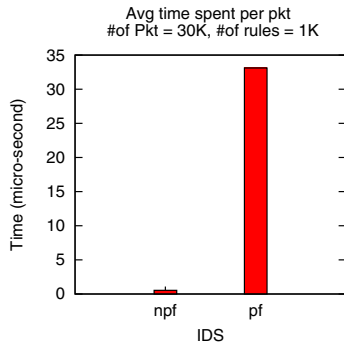


Fig. 11. Comparison of average time spent per packet by npf vs. pf to classify packet

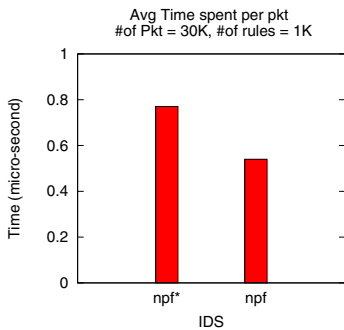


Fig. 12. “npf*” means traffic pattern independent npf and “npf” means traffic pattern adaptive npf

perform intersection operation (\cap) amongst the list of ruleIDs, we decide the fate of the packet by default ruleAction if at any step of intersection operation, the result of $\cap = \emptyset$. It even enables npf to bypass the tree traversal operation for some protocol fields. Note that bypassing tree traversal operation does not compromise the npf’s classification accuracy as because if \cap operation results to \emptyset , it is guaranteed that the fate of the packet solely depends on the default ruleAction set by the administrator.

We also carry out an experiment to see the impact of utilizing the traffic pattern in improving the performance of npf over npf*. Here npf is traffic pattern adaptive packet classifier and npf* is a naïve packet classifier (i.e. traffic pattern independent). Fig. 12 shows that npf outperforms npf* by spending approximately 30% less time per packet in packet classification.

VI. CONCLUSION

We presented a design of fast and traffic pattern adaptive packet classifier npf which exploits the locality in the traffic pattern in order to improve its average performance. In order to utilize the traffic pattern, npf intelligently reorganizes its internal data structure IBT so that the search time for frequently visited rules are reduced. Since packet classification is the core mechanism used in the detection engine of any IDS, we believe that using npf, an IDS will be able to cope up with the high link speed in detecting malicious traffic. However, the

dynamic reorganization of IBT introduces some maintenance overheads such as the max-heap computation, dynamic tree rotation etc. As a future work, we will investigate more thoroughly the tradeoffs between the performance gain of npf and the overheads associated with the online reorganization of rule tree IBT.

REFERENCES

- [1] *Snort User Manual 2.8.3*, The Snort Project, Sep. 2008, available as http://www.snort.org/docs/snort_manual/2.8.3/snort_manual.pdf.
- [2] V. Paxson, “Bro: A system for detecting network intruders in real-time,” *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.
- [3] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, “Operational experiences with high-volume network intrusion detection,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, Oct. 2004, pp. 2–11.
- [4] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.
- [5] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.
- [6] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” University of Arizona, Tech. Rep. TR-94-17, May 1994.
- [7] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, “A fast string-matching algorithm for network processor-based intrusion detection system,” *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 614–633, Aug. 2004.
- [8] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, “Granidt: Towards gigabit rate network intrusion detection technology,” in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Sep. 2002.
- [9] B. L. Hutchings, R. Franklin, and D. Carver, “Assisting network intrusion detection with reconfigurable hardware,” in *Proceedings of 10th Annual IEEE Symp. on Field-Programmable Custom Comp. Machines*, Apr. 02.
- [10] S. Singh, F. Baboescu, G. Varghese, and J. Wang, “Packet classification using multidimensional cutting,” in *Proceedings of ACM SIGCOMM 2003*, Aug. 2003, pp. 213–224.
- [11] F. Baboescu and G. Varghese, “Scalable packet classification,” *IEEE/ACM Transactions on Networking*, vol. 13, no. 1, pp. 2–14, Feb. 2005.
- [12] V. Srinivasan, S. Suri, and G. Varghese, “Packet classification using tuple space search,” in *SIGCOMM ’99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM, 1999, pp. 135–146.
- [13] P. Gupta and N. McKeown, “Algorithms for packet classification,” *Network, IEEE*, vol. 15, no. 2, pp. 24–32, Mar/Apr 2001.
- [14] “NetBSD,” www.netbsd.org.
- [15] P. Gupta, B. Prabhakar, and S. Boyd, “Near-optimal routing lookups with bounded worst case performance,” vol. 3, Mar 2000, pp. 1184–1192 vol.3.
- [16] H. Hamed, A. El-Atawy, and E. Al-Shaer, “Adaptive statistical optimization techniques for firewall packet filtering,” in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006, pp. 1–12.
- [17] E. Cohen and C. Lund, “Packet classification in large ISPs: design and evaluation of decision tree classifiers,” *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 73–84, 2005.
- [18] A. El-Atawy, T. Samak, and E. Al-Shaer, “Using online traffic statistical matching for optimizing packet filtering performance,” in *Proceedings of IEEE INFOCOM 2007*, May 2007.
- [19] A. Yoshioka, S. H. Shaikot, and M. S. Kim, “Rule hashing for efficient packet classification in network intrusion detection,” in *Proceedings of the 17th International Conference on Computer Communications and Networks*, Aug. 2008.
- [20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. MIT Press and McGraw-Hill, 1990.
- [21] R. Seidel and C. R. Aragon, “Randomized search trees,” *Algorithmica*, vol. 16, no. 4/5, pp. 464–497, 1996.
- [22] *NetBSD Kernel Developer’s Manual*, The NetBSD Foundation, Apr. 2008, available as <http://man.NetBSD.org/man/+ANY+NetBSD-5.0>.
- [23] “tcpreplay,” <http://www.tcpreplay.synfin.net>.