Rule Hashing for Efficient Packet Classification in Network Intrusion Detection

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

***Abstract*—A rule-based intrusion detection system compares the incoming packets against rule set in order to detect intrusion. Unfortunately, it spends the majority of CPU time in packet classification to search for rules that match each packet. 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 becomes a major bottleneck in high-speed networks since each packet incurs many memory accesses with little locality. In this paper, we propose rule hashing for fast packet classification in intrusion detection systems. The rule hashing, combined with hierarchical rule trees, saves memory and reduce the number of memory accesses by allowing the whole working set to be accommodated in a cache in most of the time, and thus improves response times in finding matching rules. We implement our algorithm in Snort, a popular open-source intrusion detection system. Experimental results show that our implementation is faster than original Snort to deal with the same real packet traces while consuming an order of magnitude less memory.**

I. Introduction

An intrusion detection system (IDS) is an important security tool for network administrators to protect their networks. It enables them to monitor the networks by inspecting packets in real time and detecting malicious attacks. An IDS classifies packets using a rule (or signature) database in order to determine whether packets are malicious. A common approach to efficiently search for a matching rule 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 amount of traffic and threats, intrusion detection becomes very resource-intensive; with today's high-speed networks and large rule sets, an IDS often exhausts CPU time and memory. In particular, searching the rule database and finding rules that match incoming packets 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]. When constrained by lack of CPU time, an IDS may allow malicious packets to enter the network. Therefore, reducing CPU time consumption in packet matching is crucial for overall intrusion detection performance.

In Snort, a rule database occupies most of memory space. At first, Snort builds a tree structure called "rule tree" from thousands of rules. Snort then creates index structures consisting of indices, e.g. port numbers, and patterns of each rule. These index structures are used to find matching rules fast and efficiently. Depending on the protocol type of incoming packets, which index structure is used to find matching rules are determined. When a TCP or UDP packet arrives, Snort checks the source and destination ports in the header. If a matching rule is found in the index structures, Snort performs pattern matching against the payload of the packet. If the packet contains the pattern that Snort is looking for in its payload, Snort then performs full packet inspection using remaining fields of the matching rule stored in the rule tree. However, there are also rules with wildcards in port fields, and they are treated differently. Snort duplicates patterns of these rules in multiple indices so that they can be found in $O(1)$ time, at the cost of more memory consumption.

In this paper, we propose *Hash-Based Detection Engine (HBDE)* for fast and memory-efficient packet classification in network intrusion detection. Our approach generates a hash value using five protocol fields (source IP, destination IP, source Port, destination Port, flow status) of each rule. Each hash value is stored in a single hash table. When a packet arrives, the same hash function is applied to the incoming packet and the hash value generated from the packet is used to search in the hash table for the matching rules. Our approach searches for the matching rules multiple times against each packet to cover all possibilities (wildcard rules). Since There is no pattern duplication in the proposed approach, it saves memory space and thus reduce the number of memory accesses to read a set of finite strings for string pattern matching. Although it may need to perform string pattern matching multiple times against one packet while Snort performs string pattern matching at most twice per packet, we claim that our proposed approach is fast because the hash table resides most of the time in cache due to no pattern duplication. We implement HBDE in Snort and compare its performance with the original detection engine of Snort using real packet traces. Experimental results validates our claim because our implementation shows faster performance than original Snort to deal with the same real packet traces while consuming an order of magnitude less memory and incurring less processing time.

978-1-4244-2390-3/08/$25.00 ©2008 IEEE

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.

```
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

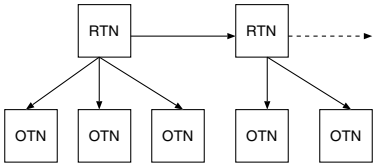


Fig. 2. A sample Rule Tree

The remainder of the paper is organized as follows. In section II we review background of Snort and its performance issue. In section III we describe the design and working procedure of *HBDE* in detail. Experimental results are presented in section IV. Other approaches to improve intrusion detection system's performance have been introduced in section V before we conclude in section VI.

II. Background

A network intrusion detection system (NIDS) is a device that monitors network traffic passively and match packets against rules. A NIDS detects malicious activities such as unauthorized accesses, port scans, and denial of service (DoS) attacks. Snort and Bro, popular open-source NIDSs based on rules, are capable of analyzing packets and identifying malicious attacks in real-time. If a suspicious behavior is detected, they generate alert message. Since Snort is known to perform better in previous experiments [4], we implement our algorithm in Snort and compare it with the vanilla Snort.

A. Intrusion Detection in Snort

Snort uses a simple language to define rules to describe network behaviors. Fig. 1 shows an example of a 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`". 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.

A straightforward way to check whether a packet matches any of the rules is to search the rule database in a brute-force manner: testing each rule against the packet one by one. It is easy to implement but time-consuming. To reduce the number of rules to examine, Snort builds a tree structure called "rule tree" as shown in Fig. 2 to store and organize all the rules. For each rule, the mandatory fields are stored in a rule tree node (RTN) and the option fields are stored in an option tree node (OTN). An OTN is associated with the corresponding RTN. If there are multiple rules that have the same mandatory fields, only a single RTN is created and OTNs share it. The detection engine of Snort builds indices for both source and destination port fields to allow fast access to TCP and UDP rules. It then searches its indices using the source and destination port numbers of each incoming packet to identify matching rules. If matching rules are found in the index structure, Snort performs string pattern matching between the matched rules and the payload of the incoming packet. If the string pattern matching is successful then all the remaining mandatory fields (protocol type, e.g. TCP, UDP, source/destination IP addresses) and optional conditions stored in the rule tree are checked. Since the indices are based on port numbers for TCP and UDP rules, the detection engine performs string pattern matching and full comparison only with a limited number of rules, which is critical for an IDS to inspect packets at wire speed.

B. 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 of the rule database is closely related to the internal structure of the IDS. Snort adopts index structures for faster access to the rule database. However, Dreger et al. [3] pointed out, Snort (as well as another open-source IDS, Bro) 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? To find out the answer, we measure the CPU consumption of each function in Snort. We compile Snort with the gprof [5] option so that Snort outputs the CPU times consumed by the called functions. We use two-week testing traces in 1998 and 1999 from the DARPA Intrusion Detection Evaluations [6] to measure the CPU time consumptions. We find out that string pattern matching is the most expensive operation in Snort. We also measure the elapsed time between the entrance and exit into the Snort detection engine. This measurement reveals that Snort consumes up to 70% of total CPU time in the detection engine. Therefore, reducing consumption of CPU time in packet matching inside the detection engine is critical for improving overall intrusion detection performance.

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.

Snort trades memory for speed; it duplicates patterns of rules in multiple locations in the index structures to deal with wildcards. Some of Snort rules use a wildcard (`any` in Fig. 1) in the port fields. Snort duplicates patterns of all these rules to all the other indices so that Snort can find the matching rules in $O(1)$ time per packet. Pattern duplications cause Snort to occupy a large amount of memory space to maintain all the set of strings. As a result, the processing time for finding matching rules against each packet increases because of memory access latency incurred from large memory occupancy. When the complete set of 8214 rules shipped with Snort are loaded, Snort consumes about 38.2 MB with the default string pattern matching algorithm, AC_BNFA. Snort provides several string pattern matching algorithms. These are AC_BNFA, ACF, ACS, ACB and ACSB. All of the algorithms are based on the Aho-Corasick algorithm [7]. The memory consumption and the speed for packet classification of each algorithm varies due to implementation difference. For example, AC_BNFA consumes small amount of memory by sacrificing the speed of packet processing.

The good part of pattern duplication is that Snort may perform string pattern matching at most twice per packet. Although our proposed algorithm may need to perform string pattern matching multiple times against one packet, we claim that our proposed approach is fast because it requires less memory access due to no pattern duplication which allows the whole rule set to be placed within cache.

III. Hash-Based Detection Engine

The goal of Hash-Based Detection Engine (HBDE) is two-fold: (i) to exclude rules that do not match a packet as early as possible without performing full payload inspection, and (ii) to reduce the amount of memory required to maintain the rule database. The latter not only decreases memory consumption but also improves speed because of the higher cache hit ratio.

A. Design of Hash-Based Detection Engine

In order to reduce the number of string pattern matching over payload, Snort needs to check more protocol fields than just port numbers so that Snort can reject more packets without entering string pattern matching phase. The main issue of the detection engine of Snort is that it duplicates patterns of rules. To maintain them, a large amount of memory space is consumed. If we can reduce the number of string pattern matchings as well as the number of memory accesses to read strings for such matchings, Snort will be able to deal with more packets in real time. However, there is a trade-off between the memory consumption by pattern duplication and the number of string pattern matchings that Snort performs. If we remove the pattern duplications, then the index structures will consume less memory. However, Snort must perform string pattern matching multiple times against each packet to cover all the rules. Suppose that we check port numbers as an index to find matching rules and patterns of each rule are not duplicated to other indices. Consider a packet with source port is 1111 and destination port is 80 arrives. Because of wildcards, we need to try the following four combinations of source and destination ports to find matching rules: (any, any), (1111, any), (any, 80), and (1111, 80). Thus, if there are rules with such port pairs in all four cases, Snort may need to perform string pattern matching over payload four times. With pattern duplications, however, rules with wildcards such as (any, any), (1111, any), and (any, 80) are all duplicated with an index (1111, 80), and thus only a single matching will be performed.

The time incurred in string pattern matching is dominated by the number of memory accesses. Let's consider two different implementations of Snort: Snort with pattern duplications *(PD)* and Snort without pattern duplications *(NPD)*. In this scenario, Snort uses port numbers as an index to find matching rules. In the worst case, *NPD* performs string pattern matching four times while *PD* performs only once. However, if the total memory requirement for the former is low enough so that we can keep whole working set in a cache, the additional memory accesses will not result in poor performance. In fact, it turns out to be faster. In our experiments, with a huge number of packets to process, the overall performance gain from *PD* to *NPD* is very high. The important point here is that we use more protocol fields, in addition to the port fields, to identify more packets as benign packets, especially before performing string pattern matching over payload. Even if Snort must perform string pattern matching, the total number of actual memory accesses, or cache misses, without pattern duplications would be smaller than with duplications because of no pattern duplications. The key factor in determining which additional protocol fields should be used depends on the fact that how many rules we can exclude in matching phase by examining those fields in a packet. Therefore, we first need to analyze Snort rules.

B. Analysis of Snort Rules

Rule headers contain necessary protocol fields that every rule must have and rule options contain a list of optional information that is mainly used for administrative purposes. If we use a protocol field in rule options, some rules may not have such a field. Thus, the first criterion to add a new protocol field is that all, or at least, most of the rules have that field. Otherwise, even if we add the field, our algorithm may not be able to exclude many rules by checking the newly-added field. The second criterion is that how many rules on average can be excluded by checking the newly added protocol field. In other words, how uniformly the values of each new protocol field are distributed. If a protocol field has three different values and each of them is used by one thousand rules, Snort can reject two-thirds of the rules by checking this protocol field. If a protocol field has four different values and each value is used by 750 rules, then Snort can reject more rules by checking this protocol field. Ideally, our algorithm should check only a small number of protocol fields, which can exclude most of the rules. Table I shows the analysis of the complete set of 8214 rules.

Table I includes only the protocol types, the IP address fields, the port fields, and the flow field. Rule headers consist

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.

TABLE I
ANALYSIS OF 8214 INTRUSION DETECTION RULES

	num of different values	most frequently used values	num of rules
Protocol types	4	TCP	7550
		UDP	490
		ICMP	135
		IP	39
Destination port	314	445	1574
		$HTTP_PORTS	1568
		any	1564
		139	1464
		$ORACLE_PORTS	291
Source port	196	any	7056
		$HTTP_PORTS	737
		1024	43
Destination IP	15	$HOME_NET	5519
		$EXTERNAL_NET	1220
		$HTTP_SERVERS	959
Source IP	11	$EXTERNAL_NET	6952
		$HOME_NET	1198
		any	28
Flow	4	FROM_CLIENT	6298
		TO_CLIENT	1182
		STATELESS	35

of rule action (`alert`), protocol types (`TCP, UDP`), IP addresses, port numbers, and direction operators (`->`). The protocol fields that we can use from them are protocol types, IP addresses, and port numbers. Rule action and the direction operators are not related to packet classification. Rule options consist of a variety of options such as content, flow, sid, etc., and the protocol fields in rule options that are related to packet classification are content and flow. Therefore, we select protocol types, IP addresses, port numbers, content, and flow for our algorithm.

The protocol type of most of the rules is TCP. The number of TCP rules is 7550 out of 8214 and the ratio of TCP rules in the whole rule set is about 92%. All of the TCP rules have the flow field to describe the status of TCP stream. The destination port field has many different values, and the four most common values are used by about 1500 rules. The rules that have one of these four values in the destination port account for 75 % of all the rules. Although the source port field has 196 different values, the value ANY accounts for about 86 % of all the rules. Contrary to port numbers, IP addresses do not have many different values. The main reason is that most of rules use variables, which start with "$" symbol, for IP address fields because the values of source and destination IP addresses vary depending on which network each end point belongs to. Thus, it is difficult to write specific IP addresses in the rules. On the other hand, applications typically use a predefined port number to communicate with each other. Thus, many distinct port numbers are used in the rules. The flow field has only four different types, most of which are either `FROM_CLIENT` or `TO_CLIENT`.

The question that now arises is how to utilize these protocol fields in building a new detection engine. Since our detection engine needs to check multiple protocol fields, it will take time in searching for matching rules. The aim of building a new detection engine is to handle a larger number of packets in real time than Snort does. Thus, to find matching rules quickly is also important.

C. Hash-Based Detection Engine

The goal of a new detection engine is to determine quickly whether string pattern matching is necessary for a given packet, assuming that there is no pattern duplication. In order to handle multiple protocol fields at once, we introduce hashing in our proposed detection engine. We propose a Hash function that converts a string consisting of the values in those protocol fields into a fixed-length numerical value. At first, HBDE computes a hash value against each rule using selected protocol fields. To do this, first HBDE generates 18-bit-long hash value from source, destination port and flow status. Then HBDE uses source and destination IP addresses to generates 16-bit-long hash value. Finally, HBDE uses both 18-bit-long hash value and 16-bit-long hash value to generate 32-bit-long hash value which is then stored in the hash table. In our experiment, hash collision does not occur with complete set of rule sets. If collision occurs, two rules that should have different hash values share the same hash value. The disadvantage of hash collision is that HBDE has to read extra "pattern" to examine the packet. When a packet arrives, HBDE generates a hash value using selected protocol fields from the packet header. Then HBDE searches the hash table to check whether there is any matching hash value exists in the table. Note that once HBDE computes a hash value for a given input, it can immediately determine whether it needs to perform string pattern matching against the packet or not.

IV. EXPERIMENTAL RESULTS

We implement HBDE in Snort 2.8.0. We use two-week testing traces in 1998 and 1999 from the DARPA Intrusion Detection Evaluations [6] for performance comparison between HBDE and the original detection engine of Snort. We use a

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.

testbed to carryout the comparison study. The Linux (2.6.15) testbed machine is equipped with a 3.00 GHz processor and 3 GB memory. We use the default configuration for both HBDE and Vanilla Snort. The experiments were repeated twenty times with each data set. The testbed machine reads each data set from the hard disk at the first run, but from the second run the content of the data set should be available in the disk cache. Therefore, we ignore the result of the first experiment to mitigate the effect of disk cache misses.

A. Comparison of Initialization Time and Memory Consumption

We conduct the first experiment to measure initialization time. During the initialization phase, Snort reads all the rules from the ruleset, parses each rule by protocol fields (e.g. TCP, UDP), stores the parsed protocol fields into RTN and OTN, and builds a rule tree and a detection engine. HBDE builds the hash table instead of the detection engine.

Fig. 3 shows the comparison of initialization time and memory consumption to maintain a set of strings using the complete set of 8214 rules. The default string pattern matching algorithm in Snort 2.8.0 is AC_BNFA. Prior to Snort 2.8.0, the default algorithm was ACF. Fig. 3(left) shows that HBDE consumes considerably less memory in all cases than snort does. This is because that there is no pattern duplication in HBDE while Snort duplicates the patterns of rules in the index structures. Due to the requirement of large amount of memory for maintaining the rule data set, Snort spends more time in initialization phase.

B. Packet-Processing Time

We conduct the second experiment to compare the packet-processing time of HBDE with that of Snort using AC_BNFA and ACF. Fig. 4(a) shows the comparison of packet-processing time in the case of AC_BNFA. The graph is plotted based on the pair of the processing time of HBDE and Snort. The diagonal line represents the points where the processing time of HBDE and Snort are equal. In other words, the points plotted above the line indicate that Snort is faster than HBDE and the points plotted below the line indicate that HBDE is faster. In Fig. 4(a), there are some points above the line which implies that Snort is faster than HBDE with some of the data sets. However, Even if Snort is faster than HBDE in those cases, since the distances between each point (above the diagonal line) and the diagonal line are short, the processing time between HBDE and Snort does not differ significantly. Comparatively, there are many points below the line plotted far apart from the diagonal. This clearly indicates that although Snort is faster in some few cases, HBDE performs better overall.

The reason that sometimes HBDE is slower than Snort is because that HBDE performs multiple string pattern matchings for a single packet which causes memory accesses. Although we expect that the low memory requirement in HBDE will allow the whole working set to accommodate in a cache, it may not be the case from time to time. Another important point to note is that if Snort uses AC_BNFA algorithm, Snort does not consume much memory space even if the complete set of 8214 rules is included as shown in Fig. 3 (left). Therefore, because of the small difference in memory consumption between HBDE and Snort, HBDE does not perform better in some data sets.

We also compare the processing time of Snort with HBDE using ACF. Fig. 4 (b) shows that the results are quite similar to Fig. 4(a). However, the number of points above the diagonal line is reduced and the overall performance of HBDE is better than that of Snort.

Fig. 3(left) shows that Snort with AC_BNFA consumes only 38 MB memory and Snort with ACF consumes more than 1 GB memory. On the other hand, HBDE using AC_BNFA consumes about 3 MB memory and HBDE using ACF only consumes about 70 MB memory. The question that arises from this results is that if the memory consumption of Snort and HBDE are similar then how it is possible that theie packet-processing times are different. Therefore, we are interested to find out whether memory consumption affects the packet-processing time of Snort and HBDE at all. In order to find the answer to this question, we use Snort with AC_BNFA and HBDE with ACF since their memory consumptions are similar. Fig. 4 (c) shows that there are still some points above the diagonal line, but in the most of the cases, the packet-processing time of HBDE is faster than that of Snort. Therefore, it is clear that although the memory consumption of Snort and HBDE is similar, HBDE performs better.

V. Related Work

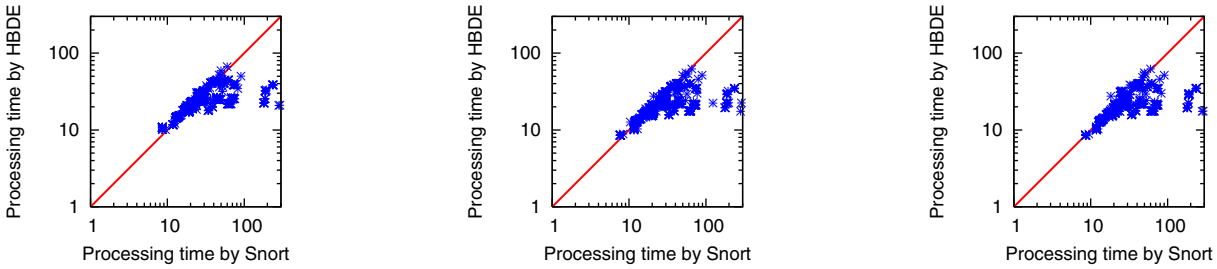
This section briefly reviews existing work to improve overall performance of intrusion detection. Most of the work have focused on the most time-consuming part of intrusion detection system, i.e., string pattern matching.

Researchers have proposed many string pattern matching algorithms for fast and efficient string pattern matching. One approach is based on software implementation [7]–[9]. The Aho-Corasick algorithm [7], implemented in Snort, matches a set of substrings against the payload of packets in $O(n)$. Wu-Manber [9] performs string pattern matching efficiently using multi-pattern optimization. The other approaches are based on hardware [10]–[12]. Although [10] used hashing, their work differs from ours because they used hashing of content string and our method use hashing of protocol fields. Some of the hardware implementations use FPGA to build a DFA/NFA and reprogram it whenever the pattern is changed. Sommar and Paxson used regular expressions for Bro and built a DFA [13] to enable users write rules more flexibly. They noticed that DFA may consume too much memory so Bro computes a new state in DFA whenever the DFA needs to transit into the state and the states that are not transited are removed to maintain the overall memory size small.

Another proposed approach takes traffic patterns into consideration. Most of string pattern matching algorithm are independent of traffic pattern and may end up with longer

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.

Fig. 3. Comparison of memory consumption (left) and initialization time (right) in different string pattern matching algorithms



(a) AC_BNFA (b) ACF (c) ACF and AC_BNFA

Fig. 4. Comparison of packet-processing time with different string pattern matching algorithm: Snort Vs. HBDE

matching time depending on actual traffic. WIND [4] implements workload-aware intrusion detection system. In this approach, an IDS collects the current traffics for a period of time, and then builds a rule tree based on the collected data. This approach improves the performance of Snort up to 1.6 times.

The contribution of our paper is to build a Hash-Based Detection Engine (HBDE) which can exclude rules that don't match a packet as early as possible without performing full payload inspection (fast packet classification) and reduces the amount of memory required to maintain the rule database (overall less memory consumption).

VI. Conclusion

The aim of building a rule tree is to reduce the number of rules that Snort must examine against each packet. However, the current detection engine is designed to duplicate patterns of rules to find a matching rule in $O(1)$. We propose rule hashing for fast packet classification with no pattern duplications. Rule hashing generates hash values using five protocol fields and stores them in the hash table. Instead of pattern duplications, our approach searches for the matching rules multiple times to cover all the rules against each packet. The amount of memory consumed by each index or hash value to maintain a set of finite strings and the time for searching for matching rules are the trade-off. Our approach saves memory space and reduces the number of memory access. However, our approach may perform string pattern matching multiple times against each packet. But this will not affect the overall performance of IDS because we expect that the low memory requirement in our approach will allow the whole working set to be accommodated in a cache in most of the time.

The experimental results show that the overall performance of packet processing of HBDE is better than the default detection engine of Snort using both string pattern matching algorithms AC_BNFA and ACF. Memory consumption for string pattern matching in HBDE is considerably less than the default detection engine of Snort in case of all the string pattern matching algorithms. The reason of such a less memory consumption is because of no pattern duplications.

References

[1] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conf. on System Administration*, Nov. 1999.

[2] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23, Dec. 1999.

[3] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational experiences with high-volume network intrusion detection," in *Proc. of the 11th ACM Conference on Computer and Comm. Security*, Oct. 2004.

[4] S. Sinha, F. Jahanian, and J. Patel, "WIND: Workload-Aware INtrusion Detection," in *Proc. of Recent Advances in Intrusion Det.*, Sep. 2006.

[5] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call grph execution profiler," in *Proc. of the 1982 SIGPLAN symposium on Compiler construction*, 1982.

[6] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 DARPA off-line intrusion detection evaluation," *Computer Networks*, vol. 34, no. 4, pp. 579–595, Oct. 2000.

[7] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Comm. of the ACM*, vol. 18, no. 6, Jun. 1975.

[8] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Comm. of the ACM*, vol. 20, no. 10, Oct. 1977.

[9] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," University of Arizona, Tech. Rep., 1994.

[10] 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 Trans. on Embedded Computing Systems*, vol. 3, no. 3, Aug. 2004.

[11] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," in *Proc. of the 12th International Conference on Field-Programmable Logic and Applications*, Sep. 2002.

[12] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. of 10th Annual IEEE Symp. on Field-Programmable Custom Comp. Machines*, Apr. 02.

[13] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signarures with context," in *Proc. of the 10th ACM Conference on Computer and Comm. Security*, Oct. 2003.

Authorized licensed use limited to: National Cheng Kung University. Downloaded on February 23,2010 at 08:39:03 EST from IEEE Xplore. Restrictions apply.