RULE HASHING FOR EFFICIENT PACKET CLASSIFICATION IN NETWORK INTRUSION DETECTION

By

ATSUSHI YOSHIOKA

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2007

To the Faculty of Washington State University:

The members of the Committeee appointed to examine the thesis of ATSUSHI YOSHIOKA find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGEMENT

I would like to thank my adviser Min Sik Kim for his advice and guidance throughout my studies at WSU, and for providing financial support for education and research as a research assistant. I would also like to thank Dave Bakken and Murali Medidi for taking the time to be on my committee. Further I would like to thank all members of NRL.

RULE HASHING FOR EFFICIENT PACKET CLASSIFICATION

IN NETWORK INTRUSION DETECTION

Abstract

by Atsushi Yoshioka, M.S.
Washington State University
December 2007

Chair: Min Sik Kim

An intrusion detection system (IDS) 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 thesis, we propose rule hashing for fast packet classification in intrusion detection systems. The rule hashing, combined with hierarchical rule trees, saves memory by minimizing the number of redundant nodes in the graph, and thus improves response times in finding matching rules. We implement our algorithm in Snort, a popular open-source intrusion detection system, and compare the performance of our algorithm with that of Snort's detection engine using real packet traces. Experiments show that our implementation handles more packets than Snort does while consuming an order of magnitude less memory.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Intrusion detection systems (IDSs) are important security tool for network administrators to protect their network. IDSs work with other security tools such as firewalls and enable network administrators to monitor their networks, inspect packets in real time, and detect malicious attacks. In order to determine whether packets are malicious, IDSs use rule (or signature) for packet classification. 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 is 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, Snort and Bro, popular open-source IDSs, spent all the CPU time, consumed entire memory, and halted immediately when they were deployed under high-speed network environment [3]. In our experiments, Snort spent up to 50% of CPU time on traversing a DFA (deterministic finite automaton) for packet matching. When constrained by lack of CPU time, an IDS may allow malicious packets. Therefore, reducing CPU time consumption in packet matching is critical for overall intrusion detection performance.

When Snort starts up, Snort builds a rule tree by reading all the rules. Snort then builds TCP, UDP, ICMP, and IP Portlists that consist of the pairs of either a source port, a destination port, or an ICMP type and contents from the rule tree. Since some of Snort rules use a wildcard in their port fields, Snort duplicates all the rules specifying a wildcard in the source port field and the destination field to all the other nodes whose protocol type is the same so that Snort can find the matching rules in $O(1)$ time per packets. Therefore, because of rule duplications, Snort needs to

consume a large amount of memory space to maintain all the rules.

Whenever Snort receives a packet, Snort checks the protocol type of the packet and traverses the appropriate Portlist to inspect the packet. The main bottleneck of detection engine of Snort is a string-pattern matching; the majority of CPU time spent on the detection engine is because of string-pattern matching. The main reason is that Snort performs string-pattern matching against each packet by checking only port numbers.

When Snort receives a packet, Snort first searches for matching nodes in the Portlists and if there is a matching node, Snort performs string-pattern matching using the set of strings maintained by the matching node against the packet. It is clear that the more nodes exist in the Portlists the more likely each packet matches the different node. Therefore, Snort needs a large number of memory accesses to read a finite set of strings from the memory space for string-pattern matching and a large number of memory accesses result in low overall performance. This bottleneck offers the key to designing a fast intrusion detection: lower number of nodes in the Portlists and small amount of memory that each node consumes for maintaining a set of strings are desirable to reduce the number of memory accesses that Snort has to do and improve the performance of Snort.

In this thesis, we propose rule hashing for fast packet classification in network intrusion detection. Rule hashing generates a hash value using five protocol fields of each rule to reduce the number of redundant nodes in the Portlists. Snort checks the values of the source port and the destination of each packet independently to determine whether Snort needs to perform string-pattern matching against the packet. In our approach, we do not duplicate any rules. Instead of rule duplications, our approach searches for the matching rules multiple times to cover all the possibilities against each packet. The amount of memory consumed by each node to maintain a set of finite strings and the time for searching for the matching nodes are the trade-off. Our approach can save memory space and reduce the number of memory accesses to read a set of finite strings

for string-pattern matching, but it may also need to perform string-pattern matching multiple times against one packet although Snort only performs string-pattern matching at most twice against one packet. We implement our algorithm in Snort, a popular open-source intrusion detection system, and compare the performance of our algorithm with that of the detection engine of Snort using real packet traces. Experimental results show that our implementation handles more packets than Snort does while consuming an order of magnitude less memory.

The rest of the thesis is organized as follows. Chapter 2 gives an introduction to intrusion detection system required to understand this work. Chapter 3 details the design of rule hashing and how it works. Chapter 4 presents the experimental results and evaluation of rule hashing. Chapter 5 introduces some of the researches that have been done on intrusion detection system. Finally, Chapter 6 provides a summary of this work and the conclusions.

CHAPTER TWO

BACKGROUND

This chapter gives an overview of the general IDSs and details of Snort: Snort rule, how Snort maintains rules, and how Snort inspects packets. The goal of this chapter is to provide background of the rest of the thesis.

2.1 Intrusion Detection System

Types of IDS IDS can be classified into three types: host-based IDS (HIDS), network IDS (NIDS), and distributed IDS (DIDS). HIDSs are built in the host machine as software and monitors whether abnormal behaviors occur in the machine. NIDSs are devices that monitor network traffic from where they are deployed and match the traffic against rules. NIDSs detect malicious activities such as unauthorized accesses, port scans, DoS (Denial of Service) attacks and so on. DIDSs consist of multiple sensors and a centralized manager. Each sensor is an NIDS. By having multiple sensors distributed on a network, network administrators can get a broader view of what is occurring on their networks. The sensors report logs of network where they are deployed to the centralized manager.

Types of intrusion detection techniques There are two types of intrusion detection techniques: anomaly detection and misuse detection. Misuse detection systems contain signatures or rules known as malicious behaviors and match them against network traffic. The advantage of misuse detection is efficiency and false-positive rates are lower than those of anomaly detection. The disadvantage is that they can not detect unknown attacks. If new attacks are discovered, the administrators of misuse detection systems must add them to the signatures. Anomaly detection systems

have knowledge of the normal behavior of users and applications and search for anomalous behavior from the normal behavior. The advantage of anomaly detection systems is that they provide detections of previously unknown attacks. Since anomaly detection systems define what is normal, they can detect any attacks whether it is a part of the threat model or not. However, this advantage could cause high false-positive rates.

Rules Rules are descriptions of network state that network administrators look for. Misuse IDSs use rules as inputs, match them against packets and generate alerts and create log messages according to the results of matching.

Alerts and logs If an IDS detects a malicious packet, it generates an alert to notify the network administrators of it and a log message is saved in a file with the information of the pair of matched rule and the malicious packet.

Snort and Bro Snort [10] and Bro [9] are popular open-source IDSs based on rules. Snort and Bro work with libpcap, a multiplatform interface for low-level network monitoring, to acquire packets. Libpcap is a de facto library to acquire packets from the wire and is used by a number of application software. They are capable of real-time packet analysis and can be used to detect malicious attacks. If a suspicious behavior is detected, they log a message with information of the pair of the suspicious packet and the matched rule.

2.2 Snort rules language

Snort uses a simple, flexible, and powerful language for rules to describe network behavior. Figure 2.1 below is an example of Snort rule. This rule means that if there is an established TCP stream from *HOME_NET* to *EXTERNAL_NET* that contains the string *Volume SerialNumber* in its

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES di
rectory listing"; flow:established; content:"Volume SerialNumber"; classtype:bad-unknown;
sid:1292; rev:9;)
```

Figure 2.1: Snort Rule

payload, Snort should generate an alert and log a file entitled *ATTACK-RESPONSES directory listing*. *HOME_NET* and *EXTERNAL_NET* are the variables that users can define their own networks for *HOME_NET* and *EXTERNAL_NET*.

Snort rules can be classified into two portions: packet headers and a list of optional information. The first line in Figure 2.1 is the first portion. The second portion such as flow and content is optional information and is used for administrative purposes and to describe further network statuses to detect malicious behaviors.

2.2.1 Rule headers

Rule headers consist of six elements: rule action, protocol type, source IP address, source port number, destination IP address, and destination port number. These elements are mandatory and if a rule does not contain any of them, Snort treats the rule as an invalid rule and does not add the rule to the rule tree.

Rule Actions The first field in a Snort rule is the rule action which tells Snort what to do if a packet matches one of the rules. Snort provides eight actions: *alert, pass, drop, reject, sdrop (silent drop), log, activation*, and *dynamic*. *alert* and *pass* are the two common actions. *alert* generates an alert and logs the matched packet. *pass* simply ignores the packet and does not process the further packets which are identical to it.

Protocols The second field describes the protocol types. There are four protocol types that Snort in default can deal with: TCP, UDP, ICMP, and IP. IP means any of the other three protocol types. If there is a packet that Snort can determine neither TCP, UDP, nor ICMP, then Snort checks whether a matching IP rule against the packet exists or not.

IP addresses The third field and the sixth field represent the source IP address and destination IP address respectively. Snort allows users to use the dot-slash notation to describe a subnetwork. The set of subnetworks are also allowed. The value of IP address could be ANY, which indicates a wildcard value for IP address field.

Port numbers The source port and the destination port are in the fourth field and the seventh field respectively. As similar to the IP address fields, we can specify an individual port number or a range of port numbers. Multiple ranges of port numbers are allowed as well. Port numbers also could be ANY.

The direction operator The direction operator, $\rightarrow$, indicates the direction of the packet in which Snort is applying the rule. The left portion of IP address and port number defines the source host, and the right portion defines the destination host. We can also use the bidirectional symbol, $\leftrightarrow$, which tells Snort to consider the rule as two different rules: the first rule is that the pair of IP address and port number on the left side is the source host and right side is the destination host. The pairs of IP address and port number in the second rule is reversed; the left side is the destination host and the right side is the source host.

2.2.2 Rule options

The rest of the part following the rule headers is rule options. Their order does not matter. Rule options may consist of one or more options and each rule option is delimited by ";". Most of options are not related to packet analysis and they are just administrative options so that the load of administrators can be lighted. However, the content field is an exception. It is greatly related to packet analysis. The detail of the content field will be discussed in Chapter 3. In this section, some of the frequently-used rule options are discussed.

Msg *Msg* stands for Message. *Msg* is a rule title that tells Snort to log the rule and the matched packet.

Flow *Flow* is used to describe the flow stream of TCP packets. The following options can be used with *Flow*: *to_server, from_server, to_client, from_client, established*, and *stateless*. *to_server* and *from_client* are synonym and not allowed to use together. *to_client* and *from_server* are also synonym. If we are interested only in established TCP session, *established* is used to apply for the rules. *stateless* tells Snort to detect packets without considering a state of TCP sessions.

Content *Content* is very important option that provides a feature to search for a specific string in the payload of each packet. *Content* is described by either ASCII string or binary data as in the form of hexadecimal characters.

uricontent *uricontent* is very similar to *Content*. The difference is that *uricontent* looks for a string only in the normalized output of the URLs.

Depth *Depth* is used with *Content* to specify where in packets we want to look for to find whether *Content* is there.

Sid *Sid* stands for Snort ID. It is a unique number that every rule must have. When an alert is logged, *Sid* is used to identify rules.

Rev *Rev* describes a revision number for rules. Each rule might be updated so we can use *Rev* to distinguish among them. Whenever a rule is updated, *Rev* of the rule is incremented.

2.3 Rule tree

The easiest way to check whether a packet matches any of rules is brute-force search: checking each rule against the packet one by one. Brute-force search is easy to implement, but unfortunately not efficient. Thus, Snort builds a rule tree by reading all the rules to reduce the number of rules that Snort must examine. Snort parses each rule into two portions; the first portion, packet headers, is stored in a rule tree node (RTN) and the second portion, a list of optional information, is stored in an option tree node (OTN). OTNs are associated with a proper RTN. If the first portions of two rules are identical, the OTNs of the rules are associated with the same RTN.

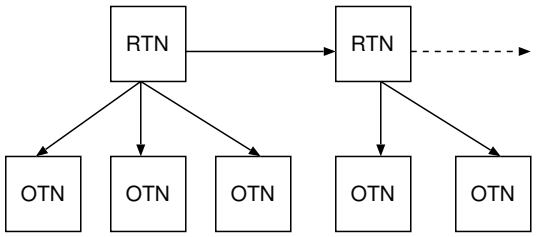

Figure 2.2: Rule tree

Figure 2.2 shows the structure of Snort's rule tree. The rule tree is fairy simple. An RTN

is a structure including multiple variables. Thus, when Snort searches for a matching RTN, Snort has to check whether all the variables in the RTN are matched against packets. Moreover, Snort must examine all the RTNs against each packet. Even if Snort finds a matching rule, Snort does not stop applying rules against the packet to cover all the possibilities of the rule tree. Therefore, this search method is also not efficient. For faster matching, Snort builds detection engine.

2.4 Detection engine of Snort

The detection engine is the most important part of IDS. It detects a packet if suspicious behavior is found in the packet. Snort and Bro uses rules to achieve this goal. If a packet matches any of rules, an appropriate action is taken. The detection engine is very time-critical since IDSs must inspect all the packets in real time. The workload of detection engine depends on the following factors: the computational power of the machine on which IDS is installed, the number of rules that IDS has, and the load of network where IDS is deployed. The computational power of the machine which IDS is running and the load of network are not directly related to the structure of IDS but the number of rules is greatly related to it since how to maintain rules and perform rule matching against packets depends on the structure of IDS. The number of rules is increased whenever new security threats are found so that IDS can deal with them. Thus, the number of rules keeps growing very fast. The speed of network is also increased and IDS is required to handle more than Gigabit network in some environments. Therefore it is desirable that the detection engine can maintain a large number of rules and efficiently perform rule matching against packets in such high-speed network environment.

In this section, we discuss two detection engines that are implemented in Snort: FastPacket Detection Engine (FPDE) and Portlist. Prior to Snort 2.7.2, the default detection engine was FPDE, but a new detection engine, Portlist, is introduced in Snort 2.8.0, which was released on October

9th 2007, and the default detection engine has been changed to Portlist. The new features of Portlist and how it works are also covered in this section.

2.4.1 FastPacket Detection Engine

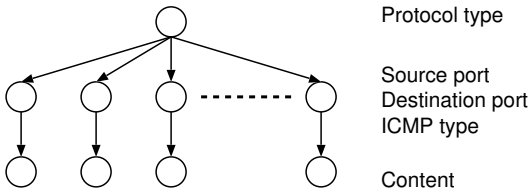

Figure 2.3: Structure of FastPacket detection engine

Figure 2.3 shows the structure of FPDE. FPDE consists of three levels; the first level represents a protocol type, the second level represents source port numbers, destination port numbers or ICMP types, and the third level represents contents. In this thesis, we denote the FPDEs whose second level represents source port numbers and destination port numbers as SRC FPDE and DST FPDE respectively. If protocol type is TCP or UDP, Snort uses source port numbers and destination port numbers in the second level. Thus, Snort creates two FPDEs for each of TCP and UDP: a SRC FPDE and a DST FPDE. In addition, Snort creates one more FPDE called Generic FPDE for TCP and UDP. Generic FPDE consists of two levels and does not have the notion of port numbers. Rules using ANY in the field of both source port and destination port are stored into Generic FPDE. If the protocol type is ICMP, Snort uses ICMP types in the second level instead of source port numbers or destination port numbers. FPDEs are created as Figure 2.4 shows. First of all, Snort traverses the rule tree and reads all the rules one by one. Whenever Snort reads a rule, Snort checks its protocol type and chooses the corresponding FPDE(s) to insert the rule. Suppose that the protocol type of a rule is TCP and the values of the source port and destination port in the rule are

1234 and 80 respectively. Snort then checks whether the node corresponding to the value of port 1234 exists in SRC FPDE. If the node exists, Snort inserts the content of the rule into the content node associated with the node. If the node does not exist, Snort creates the node corresponding to the value of port 1234 and the content node associated with the port node and inserts the content of the rule into the content node. Similarly, the content of the same rule is inserted into the DST FPDE.

Snort stores the content of the rule in the appropriate FPDE according to the algorithm shown in Figure 2.4. After reading all the rules, Snort duplicates all the contents in the Generic FPDE of TCP rules and UDP rules to respective SRC FPDEs and DST FPDEs. Storing contents are done by a string-searching algorithm. In default configuration, the Aho-Corasick algorithm [1] is used to manage contents and operates string-pattern matching against each packet. Note that the algorithm to insert rules into FPDEs leads to a large number of redundant nodes. We will discuss this issue in the next subsection.

If there are TCP rules with value v in the destination port, Snort can reject all the other TCP rules that do not have the value v or ANY in the destination port against any packets destined to port value v. As opposed to the rule tree, Snort can reject many rules by just checking sour port, destination port, or ICMP type.

All the packets are first sent to a FPDE. If a matching rule exists in the FPDE, the packet is checked whether the packet is matched with all the information in the rule tree of the matching rule. In this manner, FPDEs enable Snort to find a rule that might match the packet without checking all the information in each rule.

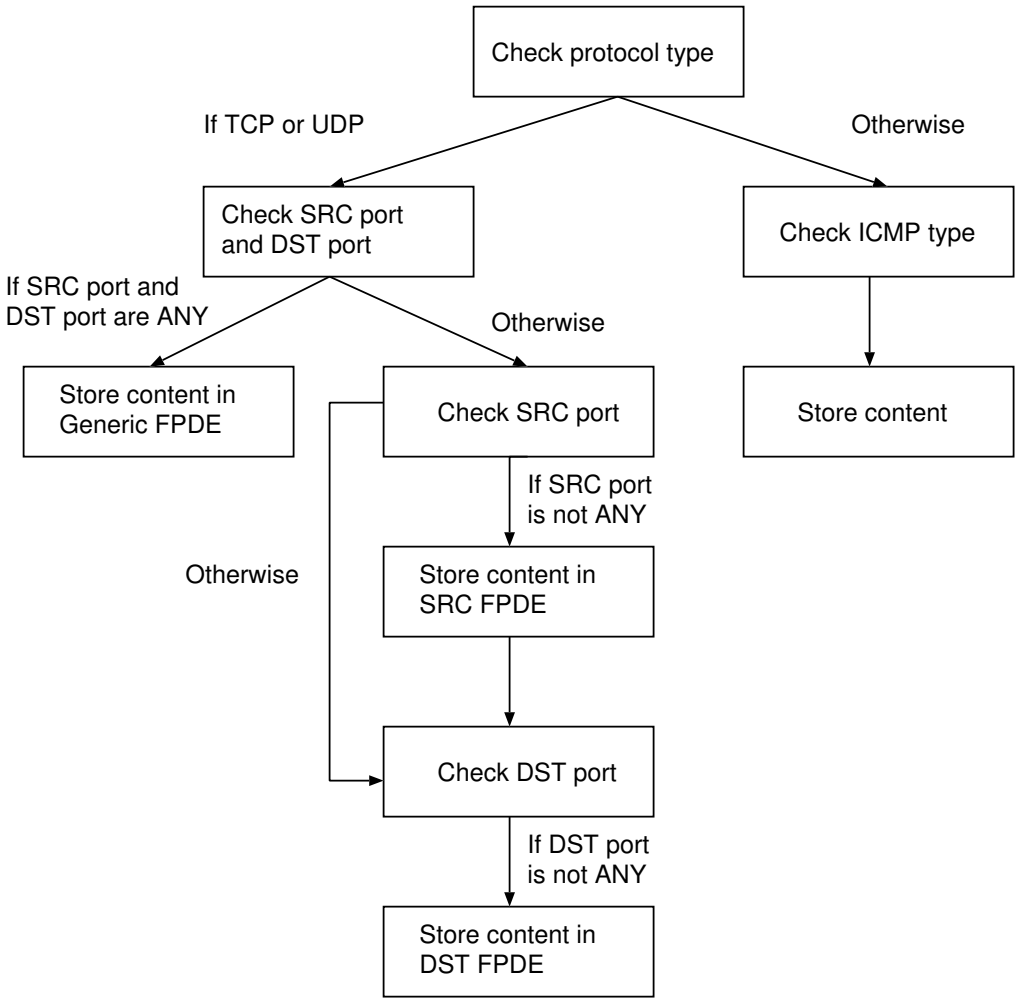

Figure 2.4: algorithm of FastPacket detection engine

2.4.2 Portlist

One of the important features introduced in Snort 2.8.0 is Portlist. Portlist is a new detection engine of Snort that gives users more flexibility to write Snort rules. For example, FPDE only allows users to write either an individual port number or a range of port numbers. If users want to specify two different port numbers that are not consecutive, users had to write two rules. However, Portlist accepts port lists as the name indicates so users can specify multiple individual port numbers and different ranges of port numbers in a single rule. Portlist is used as a detection engine in Snort 2.8.0 in default configuration instead of FPDE. The way of Portlist to parse rules and store the parsed protocol fields is similar to FPDE. The algorithm of FPDE shown in Figure 2.4 is the same as what Portlist does.

2.5 The issues of FPDE and Portlist

One of the issues in the detection mechanism of Snort is that it duplicates rules. The algorithm of FPDE and Portlist duplicate the content of each rule that specifies ANY in its source port and destination port "the number of different source port nodes $\times$ the number of different destination port nodes" times. Since these duplications increase the memory consumption of each content node, when Snort uses FPDE or Portlist for packet classification, Snort needs to copy large amount of memory for string-pattern matching. As a result, the processing time for finding matching rules against each packet is increased. When the complete set of 8214 rules are included in Snort, Snort consumes about 38.2 MB with default string-pattern matching algorithm, AC_BNFA. Snort provides several string-pattern matching algorithms so that users can choose the best algorithm for their purposes. All of the algorithms are based on the Aho-Corasick algorithm [1], but because of the different ways of implementation, the memory consumption and the speed for packet classifi-

cation of each algorithm vary. AC_BNFA consumes small amount of memory by sacrificing the speed of packet processing. Before Snort 2.8.0, the default string-pattern matching algorithm was ACF whose speed of packet processing is faster than AC_BNFA, but it consumes more than 1 GB for maintaining the 8214 rules.

Another issue is that Snort does string-pattern matching at most twice because of redundant content nodes. For TCP and UDP rules, Snort maintains the content of each rule by using the source port number and the destination port number separately. Thus, the source port and the destination port of each TCP and UDP packet is checked independently. It is clear that source port and destination port should be checked hierarchically so that Snort does string-pattern matching only when both source port and destination port are matched.

Dreger et al. [3] revealed that Snort and Bro did not work at all under high-speed network environment. In such environment, Snort and Bro immediately consumed all the CPU time and the entire memory space and then halted. This result brings us a question: which function in IDS is the bottleneck. In other words, which function consumes the most CPU time. Figure 2.5 shows the CPU time consumed by a function in Snort that performs string-pattern matching to search for the matching rules. To measure the CPU consumption for string-pattern matching, we compiled Snort with gprof option so that Snort outputs all the CPU times consumed by the used functions. We used two-week testing traces in 1998 and 1999 from the DARPA Intrusion Detection Evaluations [7] to measure the CPU consumptions. Figure 2.5 shows that string-pattern matching consumes up to 50% of total CPU time. This function consumes the most CPU time among all the functions in Snort except a function used for building the data structures such as the rule tree and Portlists. Therefore, to improve the speed of packet processing, it is needed to reduce the number of memory accesses that Snort must does for string-pattern matching.

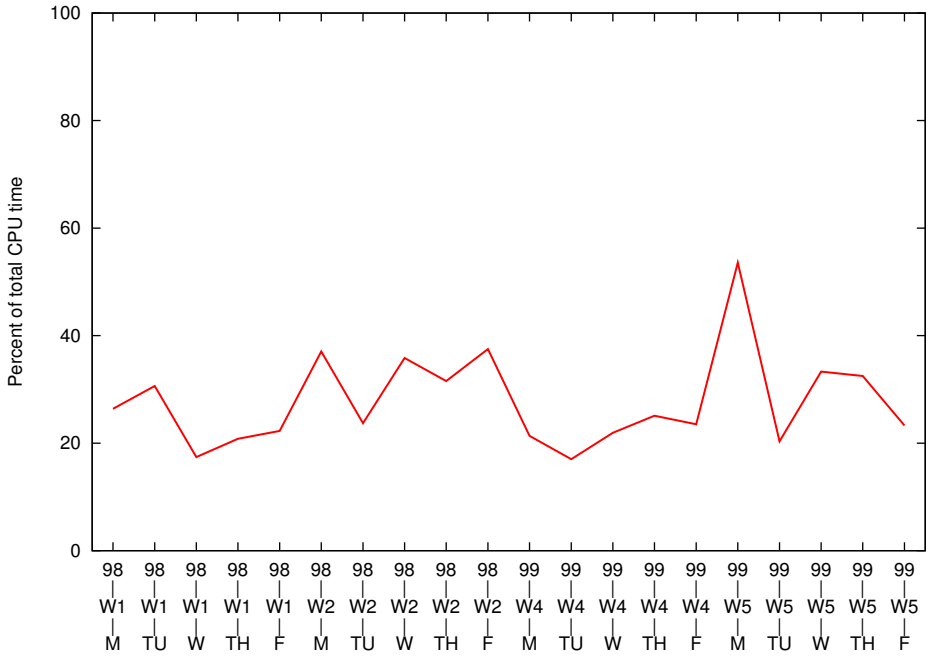

Figure 2.5: CPU time for string-pattern matching

CHAPTER THREE

HASH BASED DETECTION ENGINE

This chapter describes aspects of the design of Hash based detection engine (HBDE) by evaluating Portlist and Snort rules. HBDE is implemented in Snort 2.8.0 and used as a detection engine instead of Portlist. In other words, when Snort receives a packet, Snort first rejects some of the rules by using HBDE, and then does the full packet inspection using all the matching rules by the features already implemented in Snort. The purpose of this chapter is to give a proper understanding of how HBDE works and why HBDE is better than the current detection mechanism, Portlist. HBDE is the main contribution of this thesis.

3.1 Design of Hash based detection engine

As shown in Figure 2.5, the string-pattern matching is an expensive operation in intrusion detection. From this result, we can say the following requirements to improve the packet-processing time of Snort:

- A small number of times that Snort performs string-pattern matching to determine whether packets are benign or not.
- A small number of memory accesses to read a set of finite strings maintained by a content node when Snort performs string-pattern matching.

To reduce the number of times that Snort performs string-pattern matching, Snort needs to check more protocol fields in addition to port numbers so that Snort can reject more packets before string-pattern matching. The main issue of FPDE and Portlist is that both of them duplicate the rules is ANY in the source port and the destination port to all the other content nodes whose

protocol type are the same. Thus, Snort needs to consume a large amount of memory space to maintain all the rules. If we can achieve both a small number of times that Snort performs the string-pattern matching and a small number of memory accesses to read a set of finite strings for string-pattern matching, the number of packets that Snort can deal with in real time would be increased. However, the amount of memory consumed by each content node to maintain a set of finite strings and the number of string-pattern matching that Snort performs are trade-off. If we remove the rule duplications, then each node will consume less memory. However, the problem is that Snort must perform string-pattern matching multiple times against each packet to cover all the rules. Suppose that we check the port numbers before the string-pattern matching and do not duplicate any rules. Now a packet whose source port is 1111 and destination port is 80 arrives at Snort. This packet matches the following combinations of source and destination ports: (any, any), (1111, any), (any, 80), and (1111, 80). Thus, in the worst case, Snort must perform string-pattern matching four times.

The time for string-pattern matching is dominated by the number of memory accesses. Suppose that Snort with rule duplications and Snort without rule duplications receive a packet. Further suppose that Snort with rule duplications performs string-pattern matching once against the packet and Snort without rule duplications performs string-pattern matching four times against the packet. If the number of memory accesses for four times string-pattern matching is the same as the number of memory accesses for single string-pattern matching, their computational times are almost equivalent. Needless to say, the latter case is faster than the former case and since Snort must handle a huge number of packets, the overall performance of the latter case becomes far better than the former case. The important point here is if we use more protocol fields in addition to the port fields to determine more packets as benign packets without string-pattern matching, the total number of memory accesses without rule duplications would be smaller than with duplications

since Portlist checks only the source port numbers and the destination port numbers separately. The key is how many rules we can reject by adding new protocol fields. Therefore, we first need to analyze Snort rules.

3.2 Analysis of Snort rules

The goal of this section is to find new protocol fields that we can use for our algorithm so that our algorithm can reject more packets by checking the new protocol fields. As we discussed in Section 2.2.1 and Section 2.2.2, rule headers are necessary protocol fields that every rule must have and rule options are list of optional information that is mainly used for administrative purposes. That is, if we use the protocol fields in rule options, some of rule may not have such protocol fields. Thus, the first criterion to add new protocol fields is that most or all of the rules have new protocol fields. Otherwise, even if we add new protocol fields, our algorithm can not reject many rules by checking the newly-added fields. The second criterion is that how many rules can be rejected by checking those protocol fields. In other words, how uniformly the values of each new protocol field are distributed. If a protocol field has three different values and one thousand rules use each of them, Snort can reject two thousand rules by checking this protocol field. If a protocol field has four different values and 750 rules use each of them, then this time, Snort can reject more than two thousand rules by checking this protocol field. Therefore, it is ideal that our algorithm checks a few protocol fields but most of or all the rules are rejected. Table 3.1 shows the analysis of complete set of 8214 rules.

Table 3.1: Analysis of complete set of 8214 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

Table 3.1 includes only the protocol types, the IP address fields, the port fields, and the flow field. Rule headers consist of rule action, protocol types, IP addresses, port numbers and direction operators and 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, and 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 rules is TCP. The number of TCP rules is 7550 out of 8214 and the ratio of TCP rules among all the rules 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 that start with "$" symbol for IP address fields because the values of source IP address and the destination IP address 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, application software typically uses a specific number to communicate with each other. Thus, many individual port numbers are used in the rules. The flow field has only four different types and most of them are either *FROM_CLIENT* or *TO_CLIENT*.

The question that now arises is how to use these protocol fields to build a new detection engine. Since our detection engine checks multiple protocol fields to reject as many rules as possible before string-pattern matching, it would take time to search 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.

3.3 Hash based detection engine

This section gives the details of how to build a new detection engine using the selected protocol fields. What we want to achieve by building a new detection engine is to quickly determine whether string-pattern matching should be performed against the packets without rule duplications. Thus, we introduce hashing for our detection engine to handle multiple protocol fields at once. A hash value is computed by the hash function that converts a string into a fixed-length numeric code. If our detection engine can compute a hash value for a given input, then our detection engine can immediately determine whether it needs to perform string-pattern matching against packets. The question which we must consider next is that which protocol fields we can use for hashing. In other words, how to design the hash function so that it can compute the same hash values for the packets and their matching rules. If a packet has a matching rule, then the hash function must compute the same hash value for the packet and the matching rule. The most important point to note is that Snort allows users to use ANY and negations for port fields and IP address fields. Snort also allows users to use a range of numbers for the port fields and a set of subnetworks for the IP address fields.

3.3.1 Analysis of flow field for hashing

The flow field does not have the notion of negation and ANY and has only four different types: *From_server*, *From_client*, *stateless*, and *established*. In general, *From_server* and *From_client* are used with *established*, but they are also used alone. Thus, there are five combinations of these three types: *established, From_server, From_client, established+From_server*, and *established+From_client*. Every TCP rule matches one of these five combinations, stateless, or nothing. As of October 2007, all the TCP rules have the flow protocol field, but in the future, TCP rules

without the flow protocol field would be created. Furthermore, since the protocol type of IP rules could be TCP, UDP, and ICMP rules, the IP rules do not have the flow field so we also need to consider the case that the flow field does not exist in the rules. Since there are only seven possibilities in the flow field, we can use the flow field for hashing.

3.3.2 Analysis of port fields for hashing

The problem in the port fields is how to deal with ANY and the negation symbol in the port fields. Snort also allows users to use a range of port numbers, but a range of port numbers can be said as a subset of ANY.

In the case of ANY, all the port numbers are logically *TRUE*. If the negation symbol is used with a port number, packets that do not match the port number are logically *TRUE*. If we apply hashing to port numbers, the port numbers are converted to numeric codes. Thus, ANY and the negation symbol are also treated as normal characters by the hash function. However, packets do not have the notion of ANY and the negation symbol so we can not use these notions to compute hash values. Therefore, if a rule uses ANY and/or the negation symbol in its port fields, our detection engine ignores the port filed with ANY and/or the negation symbol; the hash function does not use the port fields with ANY and/or the negation symbol to compute hash values.

Suppose that a rule uses the negation symbol for its source port field and destination field. Against this rule, since the hash function does not consider the port fields, all the packets are logically *TRUE* in terms of the port fields. However, this is wrong because if a packet has the port number in the source field that is the same value with the negation symbol in the source field of the rule, then the packet must be logically *FALSE*. I noted a little earlier that the aim of our detection engine is to determine whether our detection engine needs to perform string-pattern matching against packets as quickly as possible without rule duplications. The detection mechanism of

our detection engine is wrong at this point but later, if the packets have any matching rules, full inspection is performed against the packets using the matching rules. That is, the port numbers in the packets are carefully examined at this point. Therefore, as similar to a range of port numbers, our detection engine can also treat the port numbers with the negation symbol as ANY.

Then a new problem arises here. Although every packet has certain port numbers in the source port field and the destination port field, some of rules do not have the values in their source port field and/or destination port field because of ANY. Thus, as we discussed in Section 3.1, we need to consider the combinations of the values of the source port field and the destination field to cover all the matching-rule possibilities. Table 3.2 shows the combinations of all of the four possibilities. X and Y in the table mean certain values of port number. This table indicates that against each packet, we must compute four hash values in the worst case. If the hash value of X-ANY and ANY-Y exist in the hash table, then we also need to consider the hash value of X-Y. If either the hash value of X-ANY or ANY-Y does not exist in the hash table, then we do not have to consider the combination of X-Y. Therefore we can use the port fields for hashing.

Table 3.2: Combination of the values of source port and destination port

value of source port	value of destination port
X	ANY
ANY	Y
ANY	ANY
X	Y

3.3.3 Analysis of IP address fields for hashing

In the case of IP address fields, the main problem is how to deal with the slash notation. Although rules use the slash notation to denote prefix of source IP address and destination IP address, every packet has 32-bit-long IP addresses. In other words, with respect to IP addresses, prefix matching

is necessary and it is difficult to use hashing. Since rules are predetermined when Snort starts up, it is possible to extract all the subnet masks used in the rules. However, in the worst case, the combination of subnet masks in source IP field and destination IP field is $24 \times 24 = 576$. It is not acceptable to consider the 576 possibilities against each packet. The key to use hashing for IP addresses is the number of combinations of the source IP address and the destination IP address. As we have seen in Table 3.1, the number of different values in the source IP address field and destination IP field are fifteen and eleven respectively. Table 3.3 shows the ratio of the three most used combinations of the source IP address and the destination IP address. The summation of ratios of these three combinations is 93.3 %. Thus, we only consider the following four combinations: \$HOME_NET $\times$ \$EXTERNAL_NET, \$EXTERNAL_NET $\times$ \$HOME_NET, \$HOME_NET $\times$ \$HTTP_SERVERS, and the others.

Table 3.3: Ratios of combination of source IP address and destination address

SRC IP	DST IP	num of rules	Ratio
\$HOME_NET	\$EXTERNAL_NET	5513	67.1 %
\$EXTERNAL_NET	\$HOME_NET	1188	14.5 %
\$HOME_NET	\$HTTP_SERVERS	959	11.7 %

Against each packet, we check whether the source IP address and the destination IP address of the packet match the combination of \$HOME_NET $\times$ \$EXTERNAL_NET, \$EXTERNAL_NET $\times$ \$HOME_NET, or \$HOME_NET $\times$ \$HTTP_SERVERS. If there is a matching combination, HBDE performs string-pattern matching against the packet. Regardless of this result, HBDE always considers the case other than these combinations of IP addresses. Therefore, we can not use the IP addresses for hashing.

3.3.4 Detection mechanism of HBDE

As discussed in Section 3.3.1, Section 3.3.2, and Section 3.3.3, we can use the flow field and the port fields for hashing but the IP address fields are not. Thus, we build a two-step detection mechanism. In the first step, the hash function computes a hash value for each packet using the port fields and flow field of the packet and checks whether the hash value is in the hash table. In the case of ICMP packets, we use ICMP types instead of the port fields. If the same hash value does not exist in the hash table, it means that no matching rules exist so HBDE stops inspection against the packet. If the same hash value is in the hash table, then HBDE checks IP addresses of the packet as discussed in Section 3.3.3. HBDE performs string-pattern matching at most twice: the first time is against the rules maintained by the matching combination of IP addresses, and the second time is against the rules maintained by the IP addresses other than the three combinations. Since HBDE generates four different hash values against packets in the worst case, HBDE performs string-pattern matching at most eight times. The evaluation of HBDE is discussed in Section 3.4.

3.3.5 Hash function

The hash function computes hash values for given rules or packets as follows. Figure 3.1 shows the flow of the hash function. First of all, the hash function divides the source port and the destination port into four portions respectively. That is, each portion is 4-bit long. If the value of source port or destination port is ANY, the hash function treats ANY as the value of 0. Then the hash function does XOR operation of the first portion of source port and the fourth portion of destination port, the second portion of source port and the third portion of destination port, the third portion of source port and the second portion of destination port and the fourth portion of source port and the first portion of destination port. Next, the hash function combines the four results into one 16-bit long value. The hash function then combines 4-bit long flow status. Therefore the hash value is 20-bit

long numerical value. If the hash value is computed from a rule, the hash function checks whether the same hash value is already inserted into the hash table and if it is not inserted, the hash function inserts the hash value into the hash table.

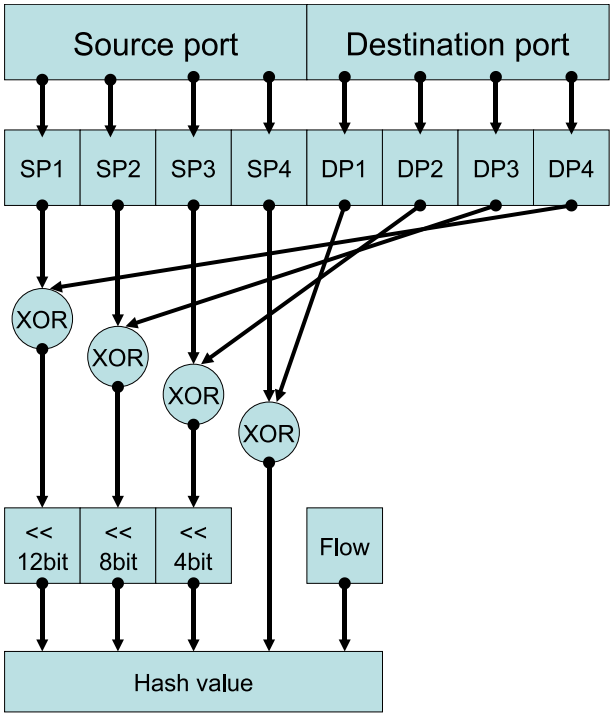

Figure 3.1: Flow of hash function

3.4 Evaluation of HBDE

In this section, HBDE is evaluated with respect to the number of memory accesses in the worst case. The first step of HBDE is compared with Portlist. The underlying goal of HBDE is to inspect packets fast. To achieve this goal, in HBDE, the multiple protocol fields are checked before string-pattern matching so that HBDE can reject all or most of the rules. Even if HBDE finds one or

more matching rules and has to perform string-pattern matching multiple times, since HBDE does not duplicate any rules and the amount of memory that the matching rules maintained is smaller than that of Snort, the processing time for the packets should be faster than Snort. As discussed earlier, the memory consumption of each content node is not related to the number of rules that share the same content node but how different the content of each rule is. For example, suppose that three rules share the same content node. Let the content of each rule be ABC, ABD, and ABE respectively. They share the same prefix "AB" so the content node only needs to store the five letters: A, B, C, D, and E. However, if the content of each rule is ABC, DEF, and GHI, then the content node needs to store the nine letters. Moreover, even if a content node maintains only one rule, if the content of the rule is ABCDEFGHIJ, the content node has to store the ten letters. Therefore we make the following assumptions:

- Every rule has a content.
- Every rule has the same length of content.
- No rules have the common prefixes in their contents.

These three assumptions help us to make our model simple. Suppose that the number of rules is n. There are four combinations of source port and destination port, and each rule belongs to one of them. They are is defined as follows: $F_1(X, ANY)$, $F_2(ANY, Y)$, $F_3(ANY, ANY)$, and $F_4(X, Y)$. Function F_i represents the frequency: the fraction of rules belonging to each case. Thus, the summation of F_i is the following:

$$\sum_{i=1}^{4} F_i = 1 \tag{3.1}$$

Let the length of content that each rule has be L. Let us now consider how the processing time can be expressed by these variables. Since processing time is dominated by memory access time, we simply compute the number of memory accesses in the worse case by Portlist and HBDE.

Portlist extracts the value of port fields if the protocol type of packets is TCP or UDP. Then Portlist checks whether the SRC port-rule map has the content node corresponding to the value of source port and the DST port-rule map has the content node corresponding to the value of destination port. If one or both of the corresponding content nodes exist, Snort performs string-pattern matching using associated content nodes. It is clear that the worst case scenario is when both SRC port-rule map and DST port-rule map have a corresponding content node.

Before computing the processing time in the worst case, we must draw attention to the number of duplications that Portlist makes. In the case of F_1 and F_2, Snort duplicates the content of each rule to SRC port-rule map and DST port-rule map respectively. Thus, the number of duplication is 1. In the case of F_3, Snort duplicates each rule to all the content nodes in SRC port-rule map and DST port-rule map. Thus, the number of duplication is the summation of number of content nodes in SRC port-rule map, the number of content nodes in DST port-rule map and 1. In the case of F_4, Snort duplicates the content of each rule to both SRC port-rule map and DST port-rule map. Thus, the number of duplication is 2.

In the worst case, all the rules have the same value of source port and destination port. In this case, the SRC port-rule map and DST port-rule map have only one content node respectively, and all the contents in the SRC port-rule map and DST port-rule map are associated with the content node respectively. From the assumption, all the rules do not have the common prefixes in their contents so the memory consumption of each content node can be calculated as follows:

$$\text{Memory consumption} = \text{Frequency} \times n \times L \times (\text{the number of duplications}) \tag{3.2}$$

We can calculate the number of rules that belongs to each port combination by Frequency $\times$ n. Thus, the summation of memory consumption will be Frequency $\times$ n $\times$ L. Since F_3 and F_4 duplicate the contents to the other content nodes multiple times, we also need to multiply the summation of memory consumption by the number of duplications. As a result, the worst case memory consumption can be formulated as follows:

$$\begin{aligned} &\text{The worst case memory consumption} = \\ &F_1 \times n \times L + F_2 \times n \times L + \\ &F_3 \times n \times L \times (F_1 + F_2 + F_3 + 1) + \\ &F_4 \times n \times L \times 2 \end{aligned} \tag{3.3}$$

Since the amount of memory can be considered as the number of memory accesses that Portlist needs to read the contents, the number of memory accesses in the worst case of Portlist is equivalent to the formula 3.3.

Next, we derive the CPU time of the first step of HBDE in the worst case. As shown in Table 3.2, in the worst case, HBDE must perform the string-pattern matching four times. As opposed to Portlist, each rule exactly matches one of them in HBDE and Snort does not duplicate the content of any rule to the other content nodes. Therefore, the number of memory accesses that Snort must do to read the content in the worst case can be derived by the following formula:

The number of memory accesses in the worst case $=$

$$F_1 \times n \times L + F_2 \times n \times L + F_3 \times n \times L + F_4 \times n \times L \tag{3.4}$$

Similar to the worst case of Portlist, the amount of memory derived from the formula 3.4 is equal to the number of memory accesses in the worst case.

Subtracting the formula 3.4 from the formula 3.3, we get the following.

Difference of the number of memory accesses $=$

$$F_1 \times n \times L + F_4 \times n \times L \times (F_1 + F_2 + F_3) \tag{3.5}$$

The formula above shows that the first step of HBDE is better than the approach of Portlist in the worst case. HBDE does further inspection at the second step, but when HBDE gets into the second step, the amount of memory that the remaining rules maintains is smaller than the amount of memory that Portlist reads in the worst case.

CHAPTER FOUR

EXERIMENTAL RESULTS

We implemented hash based detection engine in Snort 2.8.0. To compare the performance of the HBDE with the detection engine of Snort, we use two-week testing traces in 1998 and 1999 from the DARPA Intrusion Detection Evaluations [7]. The testbed machine is Xeon 3.00 GHz with 3GB memory running Linux 2.6.15. Both HBDE and Snort run with the default configuration. With each dataset, we repeat the experiments twenty times and ignore the result of the first experiment to reduce the effect of disk caching. The testbed machine needs to read the each dataset at the first time, but from the second time, the content of the dataset is stored in disk cache and the machine can read the same dataset faster.

4.1 Initialization time and memory consumption

The first experiment is about initialization time. During the initialization, Snort reads all the rules, parses each rule by protocol fields, stores the parsed protocol fields into RTN and OTN, and builds a rule tree and port-rule maps. HBDE builds the hash table instead of port-rule maps.

Table 4.1: Comparison of initialization time and memory consumption in different algorithms

	Memory Consumption (MB)		Initialization Time (seconds)	
Algorithm	Snort	HBDE	Snort	HBDE
AC_BNFA	38.2	2.8	9.5	1.4
ACF	1086.3	70.71	113.0	4.8
ACS	503.3	17.4	114.8	4.9
ACB	702.0	31.7	114.2	4.8
ACSB	586.0	18.8	112.8	4.8

Table 4.1 shows the comparison of initialization time and memory consumption to maintain

a finite set of strings between Snort and HBDE using the complete set of 8214 rules. Snort provides several string-pattern matching algorithms so that users can choose the algorithm that is the best fit for their environments. The default algorithm is AC_BNFA in Snort 2.8.0. Prior to Snort 2.8.0, the default algorithm was ACF. All the string-pattern matching algorithms are based on Aho-Chorasick algorithm [1] but their ways of implementation are different. From this table, we can say that HBDE consumes considerably less memory with all of algorithms than snort does. The main differences between Snort and HBDE are that HBDE does not duplicate the rules to the other nodes while Snort duplicates the rules and builds redundant nodes. Due to the larger amount of memory for maintaining contents, Snort spends more time in initialization. The result of this experiment shows that HBDE helps Snort greatly to not allocate unnecessary memory space for string-pattern matching.

4.2 Packet-processing time

The second experiment is to compare the packet-processing time of HBDE with that of Snort using AC_BNFA and ACF. Figure 4.1 shows the comparison of packet-processing time in the case of AC_BNFA. The graph was 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 equivalent. In other word, the points plotted above the line means that Snort was faster than HBDE and if the points are below the line, HBDE was faster. In the Figure 4.1, there are some points above the line so with some of the datasets, Snort was faster than HBDE. However, Even if Snort was faster than HBDE in those cases, since the distances between each point and the line are short, there are not much difference in the processing time between them. On the other hand, there are many points below the line that are plotted apart from the line. Although there are a few that Snort is faster, HBDE is better from the point of view of overall performance.

The reason that HBDE is slower than Snort is that the number of memory accesses in HBDE is larger than in Snort because of multiple-time string-pattern matching. For example, suppose that a packet arrives at Snort and 100 rules matches it. Let the length of strings that the matched content node maintains be 100. In this scenario, Snort must read the 100-byte-long string to perform string-pattern matching. Suppose that HBDE needs to perform string-pattern matching twice since the 100 rules are maintained by two different content nodes. Even if the two content nodes maintain 50 rules each, total number of memory accesses would be larger depending on the prefix that each set of 50 rules share. 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 Table 4.1. Therefore, because of the small difference of memory consumption between Snort and HBDE, HBDE can not achieve higher performance in some datasets.

From Snort 2.8.0, the default algorithm is changed from ACF to AC_BNFA. The reason would be because in the case of ACF, Snort consumes too much memory. Although the speed of packet processing is slower than ACF, AC_BNFA reduces the memory consumption considerably. We also compare the processing time of Snort with HBDE using ACF. Figure 4.2 shows that the results are quite similar to Figure 4.1. However, the number of points above the line is reduced and the overall performance of HBDE is better than that of Snort.

However, although HBDE using ACF only consumes about 70 MB memory, Snort using ACF consumes more than 1 GB memory. Then the question is that if the memory consumption of Snort and HBDE are similar how their packet-processing times are different. This question is important to find out whether memory consumption affects the packet-processing time of Snort and HBDE. Thus, we use Snort with AC_BNFA and HBDE with ACF since their memory consumptions are similar. Figure 4.3 shows the result of this experiment. There are still some points above the line, but in the most of cases, the packet-processing time of HBDE is faster than that of Snort.

Consequently, if the memory consumption of Snort and HBDE is similar, the overall performance of HBDE is better that of Snort.

Table 4.2 shows the comparison of the number of execution times of string-pattern matching using different datasets. The number of execution times of string-pattern matching of Snort is smaller than the number of packets in all the datasets. In contrast, the number of execution times of string-pattern matching of HBDE is larger than the number of packets when we use the datasets of week 4 and week 5 in 1999. In addition, the number of execution times of string-pattern matching of HBDE is always larger than Snort. The main reason that HBDE performs string-pattern matching such a huge number of times is because HBDE performs string-pattern matching against each packet at most eight times. Although HBDE performs string-pattern matching larger number of times, the overall packet-processing time of HBDE is still better than that of Snort. This result indicates that each content node in HBDE consumes only small amount of memory for maintaining a set of strings so at each string-pattern matching, HBDE needs to access memory small number of times.

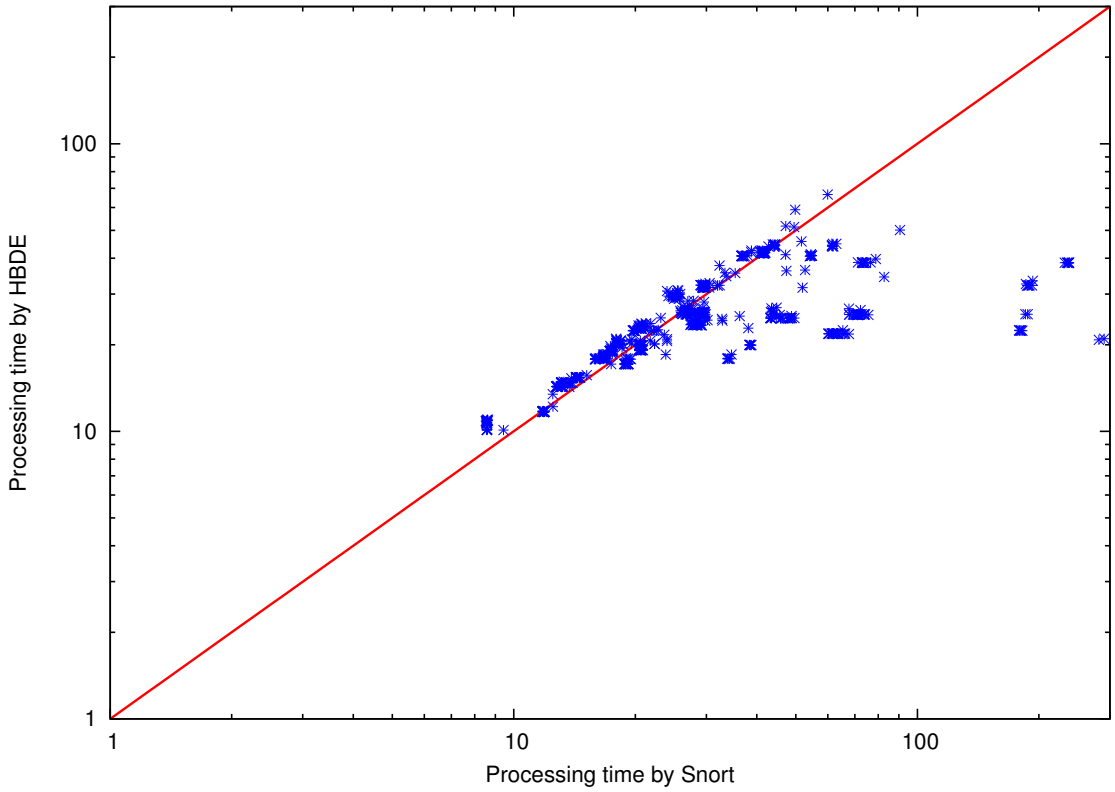

Figure 4.1: Comparison of packet-processing time in the case of AC_BNFA

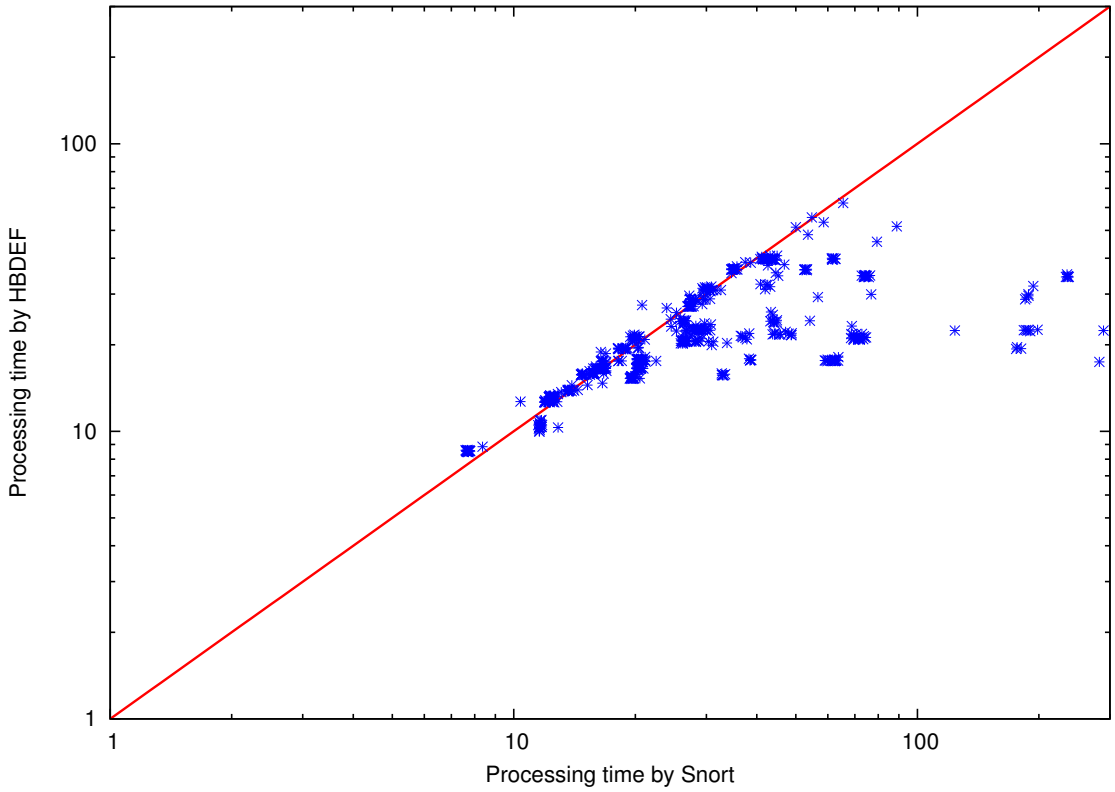

Figure 4.2: Comparison of packet-processing time in the case of ACF

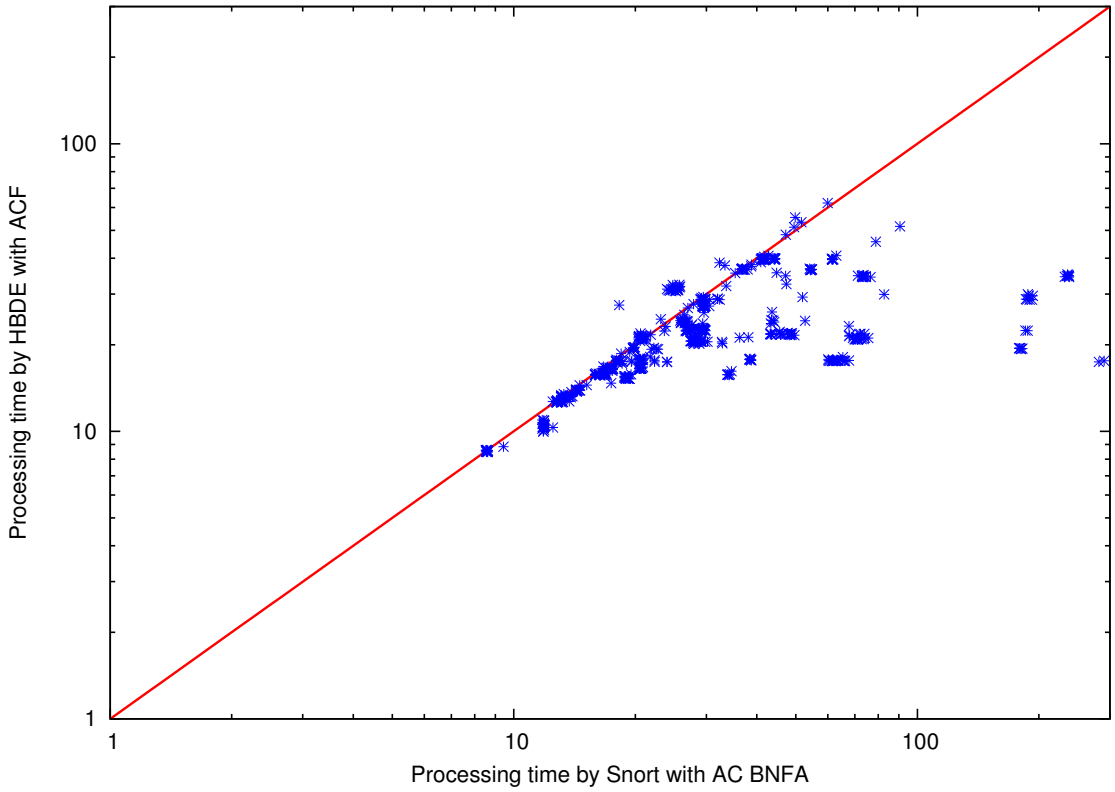

Figure 4.3: Comparison of packet-processing time Snort using AC_BNFA with HBDE using ACF

Table 4.2: Comparison of number of execution times of string-pattern matching

dataset	num of packets	Snort	HBDE
w1mon	2377402	1117777	1984574
w1tue	2567077	477142	1369108
w1wed	1908195	401861	1156515
w1thu	2401708	500387	1329443
w1fri	2236041	580894	1518625
w2mon	2948773	608695	1599320
w2tue	2066882	530647	1511123
w2wed	2233225	748439	1931642
w2thu	2798993	984520	2418688
w2fri	2233225	748439	1915443
w4mon	2959275	1935796	4587526
w4tue	1344608	720646	1964992
w4wed	3170937	1898058	4661633
w4thu	4090497	2543802	6098092
w4fri	3328634	2223365	5071032
w5mon	3728424	2688642	6000145
w5tue	6043656	5286539	13816440
w5wed	3564219	2294622	5286460
w5thu	5658038	3129746	7423971
w5fri	6276907	3253796	8393075

CHAPTER FIVE

RELATED WORK

Because of high-speed network and the number of rules, IDSs have problems to handle a huge amount of traffic in real time. Dreger et al. [3] revealed that Snort [10] and Bro [9] did not work at all under high-speed network environment. In such environment, Snort and Bro immediately consumed all the CPU time and the entire memory space and then halted. The main reason is that the significant number of string-pattern matching. Researchers have suggested using regular expressions so that users can easily write rules. Sommar and Paxson used regular expressions for Bro and built a DFA [13]. 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.

Researchers have proposed many fast string-pattern matching algorithms. One approach is based on software implementation [1,2,15]. The Aho-Crasick algorithm [1], which is implemented in Snort, matches a set of substrings against the payload of packets in $O(n)$. Wu-Manber [15] performs string-pattern matching efficiently using the multi-pattern optimization. The other approaches are based on hardware [4, 5, 8]. Some of the hardware implementations use FPGA to build a DFA/NFA and reprogram it whenever the pattern is changed.

The Distributed IDS [6, 12, 14] is the fundamental approach to solve the problem of detection speed. Since each sensor in the DIDS monitors network from where it is deployed and works as an independent device, our approach also helps DIDS to handle high-speed networks.

Another proposed approach is to take account of traffic patterns. Most of string-pattern matching algorithm are independent of traffic pattern and may end up with longer matching time depending on actual traffic. WIND [11] implements workload-aware intrusion detection. 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.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

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, Portlist, is designed to duplicate rules to find a matching content node in $O(1)$. We propose rule hashing for fast packet classification with no duplications in intrusion detection. Rule hashing generates hash values using several protocol fields to reduce the number of redundant nodes in Portlists. Instead of duplications, our approach searches for the matching rules multiple times to cover all the possibilities against each packet. The amount of memory consumed by each content node to maintain a set of finite strings and the time for searching for the matching nodes are the trade-off. HBDE can save memory space and reduce the number of memory access although HBDE may perform string-pattern matching multiple times against each packet.

The experimental results show that the overall performance of packet processing of HBDE is better than that of Snort using both AC_BNFA and ACF. The memory consumption for string-pattern matching in HBDE with the complete set of rules is considerably less than that of Snort with all the string-pattern matching algorithms. The reason of such a less memory consumption is because of no rule duplications. In other words, rule duplications greatly affect the memory consumption of Snort.

Since the most of rules have the same values in their port fields, IP address fields, and flow field, hash values are not uniformly distributed in the hash table and many rules share the same hash values. However, there are no protocol fields that we can use for rule hashing. If we have additional information that helps us to distribute hash values uniformly, the performance of HBDE would be improved. One of the possibilities is traffic pattern. The current string-pattern matching

algorithm is independent of traffic pattern and may end up with longer matching time depending on actual traffic. Therefore, if HBDE takes account of traffic pattern as well as rules to generate hash values, since the detection engine is optimized to the current traffic pattern, HBDE might be able to deal with the packets more efficiently. In this case, HBDE is desirable to be able to regenerate hash values while Snort is running in order to deal with the change of traffic pattern.

BIBLIOGRAPHY

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), June 1975.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10), October 1977.

[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*, October 2004.

[4] 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*, September 2002.

[5] B. L. Hutchings, Franklin R, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[6] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed network. In *Proceedings of the 2002 IEEE symposium on Security and Privacy*, May 2002.

[7] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000.

[8] 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*, 3(3), August 2004.

[9] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23), December 1999.

[10] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System administration*, November 1999.

[11] S. Sinha, F. Jahanian, and J. Patel. Wind: Workload-aware intrusion detection. In *Proceedings of Recent Advances in Intrusion Detection*, September 2006.

[12] S. R. Snapp, J. Bretano, G. V. Dias, T. L. Goan, L. T. Heberlein, C.-H. Ho, K. N. Levitt, B. Mukherjee, S. R. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (distributed intrusion detection systems)-motivation, architecture,and an early prototype. In *Proceeding of the 14th national computer security conference*, October 1999.

[13] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signarures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.

[14] R. Sommer and V. Paxson. Exploiting independent state for network intrusion detection. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[15] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical report, University of Arizona, 1994.