Proceedings of ICCTA2009

EXTRACTION OF FINGERPRINT FROM REGULAR EXPRESSION FOR EFFICIENT PREFILTERING

Xiaofei Wang [1], Junchen Jiang[2], Wei Lin[2], Yi Tang[2], Xiaojun Wang[1], Bin Liu[2]

[1] School of Electronic Engineering, Dublin City University, Ireland
[2] Department of Computer Science and Technology, Tsinghua University, Beijing, PRC
xiaofeiw@eeng.dcu.ie

Abstract

Deep packet inspection at high speed has become extremely important due to its application in a wide range of network applications, such as network security and network monitoring. Network intrusion detection system (NIDS) uses a collection of signatures of known security threats and viruses to scan the payload of each packet. Signatures are often specified in the form of regular expressions (regex), called patterns, which are traditionally implemented as finite automata. Deterministic Finite Automata (DFA) is fast, but requires prohibitive amounts of memory which limits their practical use. Instead of matching an incoming packet with each individual regex in a ruleset, we match the packet with a fixed substring, called fingerprint, of a regex first. Fixed string matching is faster and consumes less energy than regex matching. The fact is that if a packet does not match with the fingerprint of a regex, it will not match the regex itself. So fingerprints can be used in a prefilter engine to filter out those packets and do not match any of the fingerprints of the regex in a rule set, which represents normal non-malicious traffic. This actually reduces the number of regex rules being matched, which results in increased throughput of the NIDS. We present a *weight* scheme to extract a good fingerprint from a regex. A good fingerprint is the one that not only indicates the regex uniquely, but also occurs as less as possible in the matching procedure. We demonstrate how to use fingerprints for efficient prefiltering by means of Bloom filters in practice.

KeyWords: deterministic finite automata, prefilter, weight scheme, bloom filter, deep packet inspection.

1 Introduction

There is an increasing demand for network devices capable of examining the content of data packets in order to improve network security and provide application-specific services. Deep packet inspection (DPI) is widely used in intrusion detection systems for deterring, detecting and deflecting malicious attacks over the network. Nearly all intrusion detection systems have the ability to search through packets and identify contents that match against known attacks.

There are a wide variety of attacks. Thus each packet needs to be matched against thousands of attack signatures. For example, the SNORT [1] network intrusion detection system (NIDS) has 9182 rules as of October 2008, each contains attack signatures.

SNORT is a popular open-source intrusion detection system, with millions of downloads to date. It can be configured to perform protocol analysis, content searching and matching on real-time traffic to detect a variety of worms, attacks and probes.

The goal of the Linux L7-filter [2] is to detect the application layer protocols. Currently, the L7-filter application contains 111 application protocol signatures, contributed by researchers and developers world-wide.

Bloom filters have recently become popular within the networking community because they are suitable for high-speed implementations besides enabling novel algorithmic solutions to key networking problems, such as packet forwarding, traffic measurements and security checking. The primary use of a standard Bloom filter is for determining set membership: does an element x belong to a given set S? Its probabilistic nature makes it produce false positives; that is, it may declare that x belongs to S even when this is not the case. But in real applications, we always try to minimize the false positives as much as possible.

Because pattern matching is time consuming and malicious traffic only takes a small percentage of the whole traffic, it's not necessary to check every incoming packet with the whole pattern set, which may contains complicated regular expressions. Therefore, in order to speed up the pattern matching process and utilize the characteristics of the traffic, prefilter technology help create such an opportunity to speed up the pattern matching process by replacing most of regex matching with faster fixed string (fingerprint) matching.

In this paper, we describe the following two problems of prefilter engine and propose the corresponding solutions. Our contributions are:

- The introduction of a *weight* scheme to choose

978-1-4244-4817-3/09/$25.00 ©2009 IEEE

good fingerprint from the extracted substring candidates of a pattern.

- The introduction of splitting sliding windows corresponding to the bloom filters for most frequently occurring string lengths so as to make full use of the fingerprints more efficiently.

The rest of the paper is organized as follows. The related works is presented in section II, The problem descriptions are illustrated in section III. The methodology is provided in section IV and concludes the whole paper in section V.

2 Related works

In software implementations, Snort is an open source rule-driven NIDS. With millions of downloads to date, Snort is the most widely deployed NIDS, and has established itself as the de facto standard of the industry. Snort utilizes the signature, protocol, and anomaly based methods to perform deep packet inspection. At the core of Snort is a multi-pattern matching engine based on the Wu-Manber algorithm [3] as the prefilter that checks the payload of the packet against a given set of string patterns extracted from regular expression patterns. If one string pattern is matched, the corresponding regex pattern will be checked at the verification process. Limited by the software implementation, the throughput of Snort can only achieve several hundred Mbps.

In hardware implementation, Bloom filter and TCAM are used to implement the prefilter. Dharmapurikar et al [4] proposed a DPI architecture with multiple parallel Bloom filters. Bloom filter is a memory efficient hashing structure which can be used to query if an input string is a member of the pattern set. The query result can be false positive but not false negative. The query time is unrelated to the number of members in the set. The storage space is linear to the number of members in the set. Simple string sub-pattern is extracted from each pattern in the pattern set and divided into several groups based on the string length. One Bloom filter corresponds to a group of string-sub patterns. If an input string is matched by one of the Bloom filters, the input string will be further verified with the corresponding original integrated patterns. Bloom filter has two major drawbacks. Firstly, there should be one Bloom filter for each pattern length. The hardware cost can be prohibitive if the number of distinct pattern lengths is large. Secondly, this method suffers from the member set explosion problem when it is applied to regular expressions with character subclasses. Consider a *pcre* pattern that specifies 10 digits, "*\d{10}*". To detect this pattern, we need to enumerate all possible strings with 10 digits, i.e. the size of the member set is 10^{10}. The length of the bit vector becomes unmanageable.

TCAM has been widely deployed in Internet routers for IP address lookup and packet classification [5]. A TCAM cell can store one of the three values {0, 1, * (don't care)}. TCAM has an advantage in handling case-insensitive strings. The ASCII code of a lower case letter differs from that of the corresponding upper case letter at the 3rd most significant bit. Case-insensitive strings can be matched by setting the value of that bit to do not care in the TCAM entry. The word length of TCAM is limited (the most popular TCAM devices available in the market support word length of w=18 bytes). Strings longer than w bytes need to be divided into multiple segments. In the method of [6], the search engine maintains a processing window of w bytes that slides along the input buffer one byte per cycle. Auxiliary data structures, such as a partial hit list and a match table, are required to record the matching status of long strings. For 250MHz TCAM, the maximum throughput of the search engine is limited to 2Gbps.

Liu et al proposed an alternative two-stage approach that allowed the processing window to be shifted by multiple bytes per cycle [7]. The first stage is implemented in TCAM while the second stage in SRAM with an embedded comparator. The w-byte segment of a pattern $\{c_1, c_2 \ldots, c_w\}$ is stored in the TCAM in a staggered manner: $\{c_1, c_2 \ldots, c_w\}$, $\{*, c_1, c_2 \ldots, c_{w-1}\}$, $\{*, *, c_1, c_2 \ldots, c_{w-2}\}$, and so on. Characters in the processing window are compared to the staggered strings in parallel. The TCAM stage serves to identify potential matching patterns and the second stage will verify if the specific pattern can be found at the given location in the input stream. Liu's implementation only supports simple exact match in the second stage, which is obviously inadequate in handling *pcre* patterns found in the Snort rule database.

Although TCAM offers some advantage in handling case-insensitive strings, it falls short in dealing with *pcre* patterns with character subclasses. Let's consider the example where we have a string of 10 digits. A digit can be represented using two 8-bit patterns, 0011 0*xxx* and 0011 100*x*. To detect any combination of 10 digits, 2^{10} TCAM entries are needed. A small number of *pcre* patterns can easily consume all the TCAM storages.

3 Problem statement

In this section, we first discuss the approaches to use regular expression matching in packet payload scanning applications, then present the definition of fingerprint in prefilter engine area, and finally state the problem that we have addressed in this paper.

A regular expression describes a set of strings without enumerating them explicitly. For example, consider a regular expression: "*abc(ac|ad).*bd*". This pattern matches any packet payload that starts with *abc*, followed by *ac* or *ad*, and some arbitrary characters, and finally end with *bd*.

Regular expression is powerful, however, it is time

consuming to generate corresponding DFA and cost huge memory usage even we could transfer a regex to a DFA successfully. So some prefilter based applications have been proposed to enhance the packet payload scanning speed. The prefilter based application performs string matching on subparts of a signature, invoking the matching procedure for the full regular expression only when a subpart has been matched. Such as Snort that can achieve good performance with *content* filter scheme.

How to select fingerprint is the crucial part of prefilter engine because it has a direct relationship with the usefulness or not. Using unambiguous and unique fingerprint, we could get an agreeable throughput in NIDS. Contrarily, the prefilter matching systems become vulnerable and useless if using bad fingerprints.

Fingerprints are several explicit sub-strings from a certain regular expression. A good fingerprint is the one that not only indicates the regex uniquely, but also occurs as less as possible in the matching procedure.

For a set of regex patterns, we construct a "fingerprint DFA". When certain fingerprints are matched by "fingerprint DFA", the DFA of the original regular expression is looked-up to confirm whether there is a match in the input sequence.

There are three selection criteria for extracting a good fingerprint:

1) It is better uniquely identify the corresponding original signatures.

2) The chance of matching with multiple fingerprints is small.

3) The occurrence of fingerprint is low, in other word, the possibility of its corresponding regex accurate matching being triggered is small, so as to improve the effectiveness of prefiltering.

As the occurrence of different fingerprints of a pattern is varying from each other, we need to seek the fingerprints with the lowest matching possibility.

Here we define the scope of the problem as follows:

- For a given set of patterns, how to extract a good fingerprint for each of them?
- How to use the refined fingerprints effectively in the real applications such as Snort or ClamAV [8]?

Four fixed substrings could be extracted from the example pattern "*abc(ac|ad).*bd*", i.e., *abc*, *ac, ad* and *bd*. Each of them is a candidate fingerprint of the original pattern. Which fixed substring is selected as a fingerprint for a pattern is not decided independently from the other patterns in the ruleset.

The procedure is to extract fixed substrings for each pattern first; then the selection of a fixed substring as fingerprint for each string is decided jointly at the same time. The criteria are: 1). each pattern has a unique fingerprint, i.e. no two patterns shares the same fingerprint; 2). the fingerprints are efficient for use in a prefilter engine. For example if *abc* is a fixed substring for two patterns, then *abc* should not be chosen as fingerprint for any of the patterns, as if there is a match of *abc* in the prefilter engine, it is not clear which pattern should be used for exact matching.

Taking these factors into account, we present a *weight* scheme to evaluate the efficiency of the fingerprints. In next section, we present a weighting scheme and an algorithm to extract a good fingerprint for a pattern.

4 Methodology

L7-filter rule set does not have as many rules as Snort, so that it is unnecessary to use advanced data structures like bloom filter. There are only 111 rules in the newest version L7-filter according to the statistics released in Dec 2008.

There are 88 rules in L7-filter starting with '^' while 21 rules do not start with '^' but followed by simple strings. It means that the majority number of rules can be extracted to fixed contents. As an example, consider rule *poco*: */^\x80\x94\x0a\x01....\x1f\x9e/*, which can be extracted to "*\x80\x94\x0a\x01*" and "*\x1f\x9e*". Nearly 80% of the L7-filter rule set can be processed via this method.

Although prefilter engine enhances the performance of the overall matching system, we have to say, there still exist some rules that no suitable fingerprints could be extracted for using in prefilter engine. For example this rule: "*abc|[^de]{10}*", is split in to two parts by the *or* symbol '|'. We could not conclude that the incoming stream does not match the pattern if substring *abc* has not been found in it as it matches the second part. However it is hard to extract a fixed string as fingerprint for the second part. For accuracy, we consider this kind of rules as complicated rules and present the following architecture to guarantee all the situations have been considered.

As is shown in Figure 1, when incoming streams arriving at the prefilter engine for those patterns that have fixed substring as fingerprint, they are detected by the prefilter engine and split as two parts: matched traffic and unmatched traffic. The former one is to be send to the corresponding regex whose fingerprint has been matched in the Prefilter Engine (PE) to make an accurate matching. Similarly, the unmatched traffic after PE is send to the corresponding regexes which do not have fingerprints to make an accurate matching. If the regex is matched by regex accurate matching engine (AE), NIDS will issue an alarming message to users to indicate the incoming traffic is a malicious one. Otherwise it is a normal traffic.

4.1 Evaluate fingerprint via a weighting sheme

We set the default *weight* of each candidate fingerprint as the reciprocal of the numbers of

fingerprint candidates in the corresponding pattern. Take the previous pattern as an example, "*abc(ac|ad).*bd*", there are 4 fingerprint candidates, so the weight of each of the four fingerprint candidates is 1/4 = 0.25.

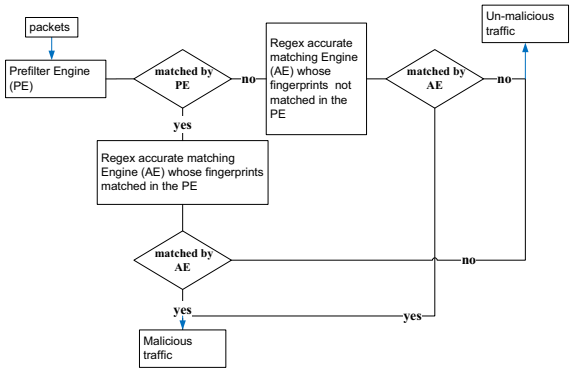

Figure 1 Architecture of prefilter engine

The Levenshtein distance [9] between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation refers to an insertion, deletion, or substitution of a single character.

For example, If *s* is "test" and *t* is "test", then *LD(s,t)* = 0, because no transformations are needed. The strings are already identical. If *s* is "test" and *t* is "tent", then *LD(s,t)* = 1, because one substitution (change '*s*' to '*n*') is sufficient to transform *s* into *t*. The greater the Levenshtein distance, the more different the strings are.

We denote the similarity as $1 - \mathrm{LD}(s, t) / \max(|s|, |t|)$, LD(s, t) is an abbreviation of the value of Levenshtein distance between string *s* and *t*, $|s|$ means the length of string *s*.

Pseudo-code of weight calculation and modification is given in Algorithm 1.

For each pattern P_i in rule set
  extract the candidates $S_{i,j}$ of P_i
  set weight $W_{i,j}$ for $S_{i,j}$ respectively
  add $S_{i,j}$ into static candidates rule set C' according to different groups
End

For any pair of candidate $S_{i,j}$ and $S_{m,n}$ in different groups of C'
  // $S_{m,n}$ is the n^{th} candidate of Pattern P_m

  $W_{i,j}$:= the weight of $S_{i,j}$
  $W_{m,n}$:= the weight of $S_{m,n}$
  calculate the similarity *s* between $S_{i,j}$ and $S_{m,n}$
  // ***abs[]*** is the absolute value function

  $W_{m,n} := abs[W_{m,n} - Avg(W_{m,n} + W_{i,j})*s]$ **if** $S_{m,n}$ contains $S_{i,j}$
  $W_{i,j} := abs[W_{i,j} - Avg(W_{i,j} + W_{m,n})*s]$ **if** $S_{i,j}$ contains $S_{m,n}$

  Update $W_{m,n}$, $W_{i,j}$ of C'
End

Algorithm 1 Pseudo-code of fingerprints evaluation

After the above procedure, we make a loop for each pattern to get the fingerprint candidate that has the biggest weight value in this pattern. The fingerprint result set can be considered a basic and direct fixed string rule set in prefilter engine.

We take the following example to illustrate the above algorithm. Given 4 real patterns $P_i(0 < i \le 4)$ of Snort:

P_1 = /^USER\s[^\n]?%[^\n]*?%/;*
P_2 = /^USER\s+y049575046/;
P_3 = /^Content-Type\s\x3a\s*image\x2fgif/;*
*P_4 = /^Content-Type\x3a\s*application\x2fsmi.*?<area[\s\n\r]+href=[\x22\x27]file\x3ajavascript\x3a/,*

For each pattern in a ruleset, extract fixed substrings as candidate fingerprints for the pattern. We could extract 4 corresponding groups substring candidates $S_{i,j}(0 < i \le 4; 0 < j)$ as following:

$$P_1 : \{S_{1,1} = USER\};$$

$$P_2 : \{S_{2,1} = USER,\ S_{2,2} = \mathrm{y049575046}\};$$

$$P_3 : \{S_{3,1} = \mathrm{Content\text{-}Type},\ S_{3,2} = \mathrm{image},\ S_{3,3} = \mathrm{gif}\};$$

$$P_4 : \{S_{4,1} = \mathrm{Content\text{-}Type},\ S_{4,2} = \mathrm{application},\ S_{4,3} = \mathrm{area},\ S_{4,4} = \mathrm{javascript}\}.$$

As defined, If there are N_i candidate fingerprints for pattern P_i, then each candidate fingerprint $FP_{i,j}$ *(j = 1 to N_i)* is assigned an initial weight $W_{i,j} = 1/N_i$:

$$P_1 : W_{11} = 1/1 = 1;$$

$$P_2 : W_{2,1} = W_{2,2} = 1/2 = 0.5;$$

$$P_3 : W_{3,1} = W_{3,2} = W_{3,3} = 1/3 = 0.3$$

$$P_4 : W_{4,1} = W_{4,2} = W_{4,3} = W_{4,4} = 1/4 = 0.25.$$

Then we add all the candidates and their weight of each pattern into set C' according to different groups:

$$C' : \{\{(S_{1,1}, 1)\},$$
$$\{(S_{2,1}, 0.5), (S_{2,2}, 0.5)\},$$
$$\{(S_{3,1}, 0.3), (S_{3,2}, 0.3), (S_{3,3}, 0.3)\},$$
$$\{(S_{4,1}, 0.25), (S_{4,2}, 0.25), (S_{4,3}, 0.25), (S_{4,4}, 0.25)\}\}$$

Iteratively we calculate the similarity *s* for any two pair candidates in different groups and modify their corresponding weight. Calculate a similarity *s* between $FP_{i,j}$ and each candidate fingerprints $FP_{m,n}$ of all other patterns in the ruleset, update $W_{i,j}$ and $W_{m,n}$ as follows:

C': { $\{(S_{1,1}, 0.17)\}$,
$\{(S_{2,1}, 0.23), (S_{2,2}, 0.5)\}$,
$\{(S_{3,1}, 0.26), (S_{3,2}, 0.07), (S_{3,3}, 0.27)\}$,
$\{(S_{4,1}, 0.12), (S_{4,2}, 0.11), (S_{4,3}, 0.2), (S_{4,4}, 0.10)\}\}$

We make a loop for each pattern to get the fingerprint candidate that has the biggest weight value in this pattern. As a result, the good fingerprint could be selected as: $S_{1,1}$ for P_1, $S_{2,2}$ for P_2, $S_{3,3}$ for P_3 and $S_{4,3}$ for P_4.

4.2 Splitting sliding windows

How to use the refined fingerprint set reasonable is the main problem next step. Some solutions have to be considered to utilize the above fixed strings efficiently.

Dharmapurikar et al. proposed a hardware architecture based on parallel Bloom filters for network packet inspection [4]. In order to perform the input streaming detection, the original architecture is designed to use multiple Bloom filters each of which detects strings of a unique length [10]. However, the more Bloom Filters we use the more processing time and resources it takes. To reduce the number of Bloom Filters, a few sliding windows are constructed to process the continuous input streams and flows by the *n* popular string lengths of these rules.

For instance, we take Snort rule sets as an example. Snort works by matching traffic patterns to its rules, stored in a rule set. The captured packets are passed through a packet decoder, which determines which protocol is in use for a given packet and matches the payload data against allowable behavior for patterns of that protocol. The content keyword is one of the important features of Snort. It allows the user to set rules that search for specific content in the packet payload and trigger response based on that data.

By extracting those unique content keywords from snort rule set, we can get string length distribution in Figure 2.

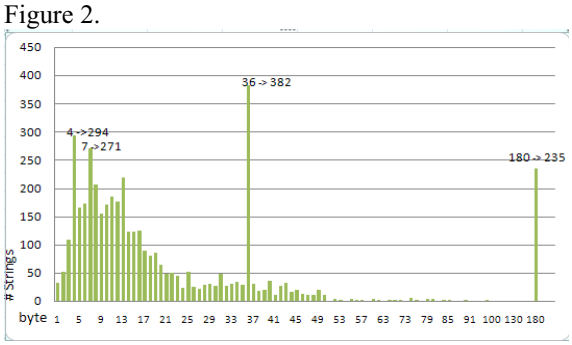

Figure 2. String length distributions in snort V2.8.4.

Obviously, there are four types of length which dominate the whole length distribution of patterns. Here we set *n* equals to 4. Thereafter only 4-byte BF, 7-byte BF, 36-byte BF and 180-byte BF will be constructed. To make efficient use of Bloom filters, strings are clustered with different lengths in a group, and strings are truncated to the length of the shortest string in the group. For instance, all the string length more than 7 but less than 36 will be truncated to 7 from the beginning of each string. Specially, content fields of less than 4 bytes will be rewritten to get rid of its ambiguous character. Furthermore, one special small test engine is added to detect the length which is less than 4 bytes in simple rules.

Similarly, as is shown in Figure 3, we can see that there is various pattern length distributions in ClamAV rule set. Here we use main.ndb rule set in ClamAV.

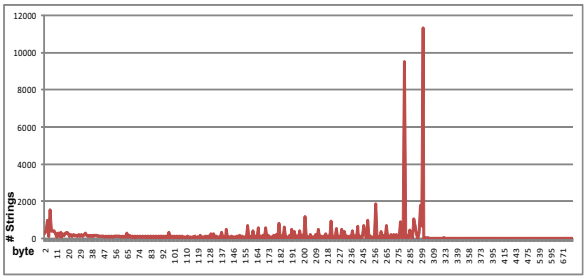

Figure 3 Pattern length distributions of *main.ndb* in ClamAV 0.95.1.

There are 58574 patterns in it while 69237 fixed strings after extraction according to wildcards. The candidates' number which extracted from patterns by wildcard is shown in Table 1.

Table 1 Selected DFA groups of different size

wildcard	# of patterns
*	4580
{}	6152
?	3325

Since we have more fingerprint candidates in ClamAV, it is unavoidable to meet conflicts when the string is truncated to small length. In this situation, we shift one character from left to right to test if there is an occurrence in the original fingerprint sets.

This prefilter engine could filter 22.8% un-malicious data. To analyze the factors that influence the filtering ratio, we record matched keywords during the process. Some simple characters, such as, 'a' and 'u' are found in the recorded result file. This is because multiple content fields exist in some rules, such as the *backdoor* in snort V2.8.4. These multiple content keywords, like: content: "from|3A|", content: "a" have significant influence on the filtering ratio. By rewriting or refining these rules, we could get an agreeable filtering ratio of approximate 62.5% of un-malicious data.

To make this architecture robust, we can train it via adjusting the settings of bloom filter. We know that the filtering ratio will be different for different input data. Then some bad traffic that has been detected previously can be used to train our prefilter engine.

We can also mix malicious part with normal trace to make this training.

5 Conclusions

We introduced the basic concept of prefilter and how it works in NIDS. A *weight* scheme has been presented to choose good fingerprint from the extracted substring candidates of a pattern.

We used fingerprints extracted with our algorithm in the patterns of Snort and did experiment with real trace. According to the statistics of our simulations, the filter ratio can reach up to 60% if the whole engine has been trained well. It has been showed that this engine could enhance the capabilities of general-purpose IDS solutions.

There are some issues to be thought about in future. For example, if a new pattern is added to a rule set, in which all patterns have their fingerprints been decided already, then when select a fingerprint for the newly added pattern, it would be inconvenient to update the existing fingerprints for the patterns in the original ruleset. So a solution is needed to be found to choose a good fingerprint for the newly added pattern without changing the fingerprints of the patterns in the original ruleset.

Acknowledgements

This work is supported by NSFC (60625201, 60873250), the Cultivation Fund of the Key Scientific and Technical Innovation Project, MoE, China (705003), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20060003058), 863 high-tech project (2007AA01Z216,2007AA01Z468), and Enterprise-Ireland.

References

[1] Snort system, http://www.snort.org

[2] L7-filter. http://l7-filter.sourceforge.net/, 2008

[3] S. Wu, U. Manber, “Fast text searching allowing errors”, Commun. of ACM, Vol. 35, No. 10, pp. 83-91, 1992.

[4] Dharmapurikar Sarang, Krishnamurthy Praveen, Sproull Todd S., et al. Deep packet inspection using parallel bloom filters. IEEE Micro, 2004, 24(1): 52-61.

[5] D. Pao, Y. K. Li and P. Zhou, “Efficient packet classification using TCAMs”, Computer Networks, Vol. 50, Issue 18, pp. 3523-3535, Dec. 2006.

[6] F. Yu, R. H. Katz, and T. V. Lakshman, “Gigabit rate packet pattern-matching using TCAM”, IEEE Int. Conf. on Network Protocols 2004.

[7] R.-T. Liu, C.-N. Kao, H.-S. Wu, M.-C. Shih, N.-F. Huang, “FTSE: The FNP-like TCAM Searching Engine”, IEEE Symp. on Computers and Communications, 2005.

[8] http://www.clamav.net/

[9] http://www.merriampark.com/ld.htm

[10] Sarang Dharmapurikar and John W. Lockwood,“Fast and Scalable Pattern Matching for Network Intrusion Detection Systems” *IEEE Journal on Selected Areas in Commuications,* vol. 24, no. 10,OCTOBER. 2006.