# A Scalable Bloom Filter based Prefilter and Hardware-oriented Predispatcher

Xiaofei Wang [1], Wei Lin[2], Yi Tang[2], Ashwin Lall[3]， Bin Liu[2] and Xiaojun Wang[1]
[1] School of Electronic Engineering, Dublin City University, Ireland
[3]Georgia Institute of Technology, Atlanta, USA
[2] Department of Computer Science and Technology, Tsinghua University, Beijing, PRC
Email: xiaofeiw@eeng.dcu.ie

## I. 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 with 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 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 [1]. It can be configured to perform protocol analysis and 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 Linux L7-filter 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.

In this paper, we propose a scalable bloom filter based prefilter and a hardware-oriented predispatcher. The key features of this mechanism are:

- The introduction of four sliding windows corresponding to 4 bloom filters whose string lengths are most frequently occur.
- It is suitable for hardware implementation on FPGAs and ASICs.
- A general procedure for training prefilter engine to make it become more and more robust.

This work is supported by NSFC (60573121, 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).

## II. METHODOLOGY

### A. *Snort prefilter and sliding windows*

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 figure1.

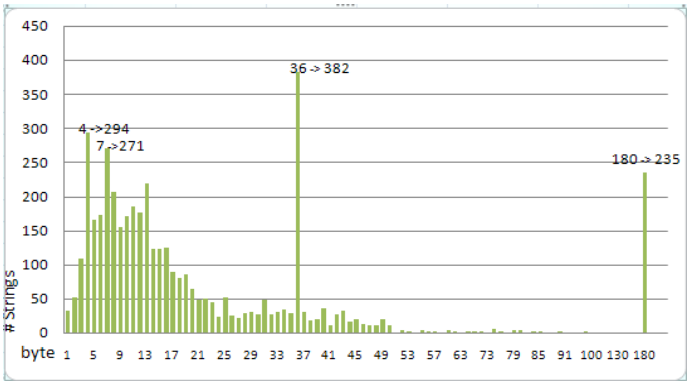


Figure 1 String length distribution in snort V2.8.4.

Dharmapurikar et al. proposed a hardware architecture based on parallel Bloom filters for network packet inspection [3]. 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 [4]. However, the more Bloom Filters we use the more processing time and resource it takes. To reduce the number of Bloom Filters, four sliding windows are constructed to process the continuous input streams and flows by the 4 popular string lengths of these rules. 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.

978-1-4244-3968-3/09/$25.00 ©2009

B. *L7-filter predispatcher and hardware implementation*

Differently, 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 [2]. Nevertheless, the rules would be still extracted to some simple string manually or automatically from complicated *regex* which can be divided into three types according to different situations.

1. Simple rules

There are 88 rules start 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 will be extracted to "\x80\x94\x0a\x01" and "\x1f\x9e" obviously. Nearly 80% of L7-filter rule set can be processed via this method.

2. Special rules

As is shown in Figure 2, pattern *skypeout* matches any packet payload that starts with hexadecimal letter '\x01' (\x02, \x03 etc.), followed by eight or fewer arbitrary characters, and finally the same character at the beginning.

It will generate 5610 states in NFA or 4584 states in DFA. Obviously, generated DFA in traditional approaches will consume huge storage. Contrarily, it will be significantly simplified if implemented by a routine in hardware logic. The meaning for this rule is that if the first character of an input stream appears again within the distance of ten characters from the first character, the input stream will be reported to be matched by this rule. In fact, only one register plus one counter are needed to record the first character of an input stream and to check if any one of the following ten characters is equal to the first character recorded in the register. The routine will set the hit flag to true if there is a match. Otherwise, the routine will stop checking the current input stream and wait for the next input stream.

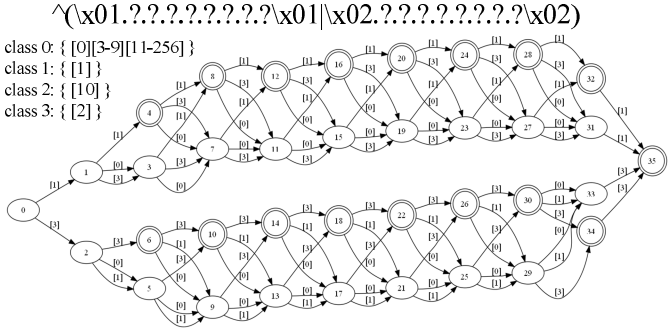


Figure 2 DFA generated with pattern *skypeout*

3. Complicated rules

As another example, consider rule *imesh*. The *imesh* rule is split into four segments by "|". The routine will report a match when any one of the four segments is matched by the input stream. The starting positions of the four segments are the same, which begin from the first character of an input stream. The first segment will be matched if the first four characters of an input stream are "post", followed by K (K>=0) characters belonging to [\x09-\x0d-~], followed by "<PasswordHash>", eliminating a certain number of characters, and followed by "</PasswordHash><ClientVer>". The other three segments are handled similarly by the corresponding routines. These routine are executed in parallel and report a match of the *imesh* rule occurred separately.

Taking advantage of these three types of rule sets, we present the following predispatcher engine. When the first packet is approaching, we send it to the fixed-contents rule sets which consist of Simple rules and Complicated rules first. If matched, this packet will be pushed to the corresponding candidate-rules to do accurate matching. Simultaneously, this packet will be sent to the Special rule sets without using predispatcher engine. Predispatcher reduces the number of possible matching rules through extracting specific content fields, reducing the numbers of exact matching operations as well as the energy consumption.

C. *Scalablity*

This prefilter engine could filter 22.8% unmalicious 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 unmalicious 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.

## III. Evaluation and Conclusion

We presented a scalable bloom filter based prefilter and a hardware-oriented predispatcher pattern matching mechanism for content filtering applications, which are scalable in terms of speed, the number of patterns and the pattern length. Our prefilter algorithm is based on a memory efficient multi-hashing data structure called Bloom filter.

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.

## IV. References

[1] Snort. http://www.snort.org/, 2008.

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

[3] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52-61, Jan. 2004.

[4] 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.