International Journal of Electronic Commerce Studies
Vol.8, No.1, pp.77-96, 2017
doi: 10.7903/ijecs.1483

ACHIEVING 100 GB/S URL FILTERING WITH COTS MULTI-CORE SYSTEMS

Surachai Chitpinityon
Kasetsart University
50 Ngamwongwan Road, Chatuchak, Bangkok 10900, Thailand
g5417550019@ku.ac.th

Surasak Sanguanpong
Kasetsart University
50 Ngamwongwan Road, Chatuchak, Bangkok 10900, Thailand
surasak.s@ku.ac.th

Supaporn Erjongmanee
Kasetsart University
50 Ngamwongwan Road, Chatuchak, Bangkok 10900, Thailand
supaporn.e@ku.th

Kasom Koht-Arsa
Kasetsart University
50 Ngamwongwan Road, Chatuchak, Bangkok 10900, Thailand
kasom.k@ku.ac.th

ABSTRACT

URL filtering is an essential tool used by Internet Service Providers (ISPs) and organizations to restrain clients from accessing non-secured or illegal web content. Designing a URL filtering method that achieves a high bit rate of 100 Gb/s and beyond for international ISPs is a challenging task. High-performance URL filtering with multi-gigabit rate capacity requires a fast URL matching algorithm and an enhanced packet processing technique. In this paper, we tackle these challenges by design and development of a software-based URL filtering system to support 100 Gb/s bandwidth. Our aim is to build a system that runs on a single commercial off-the-shelf (COTS) server with multi-core CPUs. We propose a compact URL representation using AVL tree and a multi-core/multi-thread filtering technique with session hijacking and fast packet processing framework. Performance measurements results show successful URL filtering operating at 100 Gb/s in a real network testbed.

Keywords: URL Filtering, Web Filtering, Session Hijacking, 100 GbE, COTS, AVL.

1. INTRODUCTION

URL filtering has become an important network middlebox to control web access and enforce security policy. In several countries, Internet Service Providers (ISPs) are enforced to deploy URL filtering to filter inappropriate websites such as sites containing violence, pornography, gambling, and illegal drug content[1]. In enterprise networks, URL filtering is used to restrict access to non-productivity or phishing websites that perfectly replicate web content like legitimate banking and e-commerce sites

Current URL filtering architecture can be classified into two main categories, i.e., *pass-through* and *pass-by*[2]. The pass-through (also called *inline* or *bump in the wire*) technique requires that the filtering engine presents itself as a traffic barrier. The filtering engine receives and validates a URL request against a pre-defined URL blacklist to decide whether to pass or to drop the request. In contrast, the pass-by technique allows all traffic to flow freely. Traffic is mirrored and inspected in a non-blocking fashion with no additional queueing delay. TCP sequence number prediction with a session termination technique is required to track and discontinue HTTP connections. When a client’s requested URL is matched against the database, the filtering engine injects packets to prematurely terminate the TCP-based HTTP connection before web content is transferred from the web server to the client.

Figure 1 shows an example of the pass-by URL filtering architecture used in our approach. This architecture is highly effective for filtering unencrypted HTTP traffic. For filtering encrypted HTTP traffic, however, the pass-through architecture is required because traffic decryption is necessary for inspection. Filtering encrypted traffic requires installation of trusted certificates on client machines, and it is not a focus of this paper.

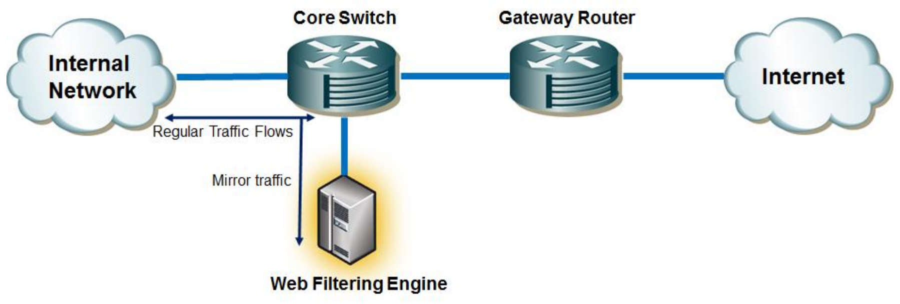

Figure 1. A pass-by URL filtering architecture

A single URL filtering machine for 10 Gb/s Ethernet (10 GbE) network is common in practice. But there is a new challenge for URL filtering now that several ISPs have begun to deploy the 40/100 GbE to meet increasing demands for high speeds and bandwidth driven by emerging applications. New software systems can gain benefits from advanced processor technology since the number of cores in commodity CPUs keeps increasing. Meanwhile, modern network interface cards (NICs) support multiple queues allowing cores to process packets concurrently in multi-gigabit rate. The evolution of *commercial off-the-shelf* (COTS) hardware with multi-core CPUs enables software-based applications to achieve hardware-level performance with a reasonable investment cost.

Today, URL filtering tools manage several millions of URLs entries stored in blacklists. The set of URL blacklists, called the *URL database* or for short *database*, borrows four basic database operations, i.e., *Create*, *Read*, *Update*, *Delete* (CRUD). However, the generic representation of URL strings used in traditional databases is inadequate to support real-time CRUD operations in multi-gigabit networks.

This paper tackles these challenges by developing a software-based URL filtering system to achieve 100 Gb/s bandwidth capability. Our approach is to design and to implement the URL filtering system that utilizes multi-core x86 based CPUs on a single COTS server. A design for URL representation with built-in compression property helps optimize memory usage and provide real-time high performance URL filtering. Compact URL representation with fast URL matching and multi-core CPUs with fast packet processing is an essential combination in the design. To the extent of our knowledge, the proposed system is the first demonstration of software-based URL filtering on COTS to handle full 100 Gb/s traffic.

The remainder of this paper is organized as follows: Section 2 discusses related works and existing approaches of URL filtering. Section 3 describes the high-level system and functionality of the URL filtering system. Section 4 presents session tracking based on the hijacking technique with timing analysis. Section 5 describes data structures supporting the URL representation and the URL matching algorithm. Section 6 presents performance measurements and experimental results from a real 100 Gb/s testbed. Section 7 concludes the paper and discusses future works.

2. RELATED WORKS

URL filtering extracts a URL from packet's payload and matches it against a set of URL blacklists. Real-time URL matching at multi-gigabit per second rates requires a fast matching algorithm with high-speed packet processing capacity. A software-only solution for real-time URL filtering using the standard NIC has poor performance under multi-gigabit networks, since in-kernel network stacks in operating systems have been designed for generic applications based on per-packet interrupts. A high rate of interrupts from the NIC to the CPU can potentially overload the CPU and result in packet loss.

Hardware-assisted accelerators using a Field-Programmable Gate Array (FPGA)[3, 4] and network processors[5] are known to overcome such software limitations. Both FPGA and network processors are designed to offload common tasks associated with upper layers, such as header parsing, pattern matching, and packet modification. Multi-threaded processing allows several packets to be processed in parallel, offering filtering performance improvement in both throughput and latency. The FPGA testbed in Garnica et al.[4] has been developed to handle 10 Gb/s traffic with the possibility to support 100 Gb/s traffic by estimation.

Recently, many literatures have extensively examined several software programs based on fast packet processing frameworks for high-speed packet I/O. The well-known frameworks are PF_RING ZC[6], netmap[7], and Intel DPDK[8]. Some commercial vendors offer proprietary frameworks with their NICs, such as OpenOnload by Solarflare[9] and Sniffer10G by Myricom[10]. Basically, the fast packet framework is a software driver that uses polling with a ring buffer to handle packets instead of interrupts. The default in-kernel network stack is bypassed, i.e., packets are directly processed by the software driver and passed to applications. Eliminating interrupt overhead helps free up the CPU and enables applications to handle network traffic at several 10 Gb/s rate. The frameworks are used as modern building blocks for developing software-based high-speed packet processing applications instead of using hardware-assisted approaches.

In addition, advanced development in multi-core CPUs and new PCI Express bus (PCIe) play a significant role in performance improvement for packet processing. Currently, a single PCIe3.0 bus with 16 lanes (signaling pairs) allows packet transfer from NICs to CPUs at speeds up to 126 Gb/s. By utilizing multi-core processors more efficiently under the framework, packet processing performance is substantially improved in both throughput (Mpps) and bandwidth (Gb/s).

URL matching is the core operation of URL filtering systems and requires an efficient URL representation to support fast queries and frequent updates for a large collection of URL datasets. The hashing method[11] offers a remarkable URL matching speed, but lacks prefix matching. The modified *Wu-Manber* algorithm with 32-bit CRC hashing proposed by Zhou, Song, and Jia[12] provides URL prefix matching and achieves about 80% URL compression rate. However, a legitimate URL not in the blacklist may be blocked due to a false positive match caused by a collision in the hash function.

Dealing with collisions in hashing can be addressed in various methods using separate chaining, open addressing, and coalesced hashing[13]. In an average case, the time complexity of insertion and searching the hash table is $O(1)$ constant time. In the worst-case scenario, the operations can degrade to $O(n)$ so that all input data are mapped to same index. Several hashing variations are proposed to improve the asymptotic results, for example Robin Hood hashing, hopscotch hashing, cuckoo hashing, Horton tables, and others[14]. Although collisions can be totally avoided by using a perfect hash function or minimal perfect hashing[15], such functions would impose substantially high computation time and space requirements.

Alternative to hashing, tree is a highly versatile data structure for URL representation in a hierarchical form. Self-balancing binary trees such as *AVL* (Adelson-Velskii and Landis) and *Red-Black* tree[16] are superior than unbalanced binary trees. For lookup-intensive applications, the AVL tree performs better than the Red-Black tree because it is more strictly balanced. Keeping balanced factor data at each node can improve the performance of AVL tree operations with a low computation overhead. Compared to hashing, an AVL tree guarantees a collision-free property with the $O(\log n)$ time for all cases. URL entries are consistently added or removed and the tree's logarithmic insertion and deletion time are preferable. Moreover, trees efficiently support highly complicated range queries and nearest-neighbor queries for hashing. By these criteria, AVL tree is the best-fit data structure to represent URLs in our proposal.

3. HIGH LEVEL SYSTEM DESIGN

In this section, we describe a design for a URL filtering system by focusing on core components and intercommunications among them. The proposed system consists of four main components, as illustrated in Figure 2. The next subsections describe detailed functionalities of each component.

3.1 URL Manager

The URL Manager is responsible for CRUD operations with the GUI. It receives an individual or batch URL with additional information (categories, filtering period, etc.) and keeps them in two databases: (1) URL database and (2) In-memory database. The URL database is long-term storage supporting high level management such as logging, report, and backup. We use MySQL to manage and store the URL database. The in-memory database keeps URLs in compression form using an AVL tree for fast URL matching.

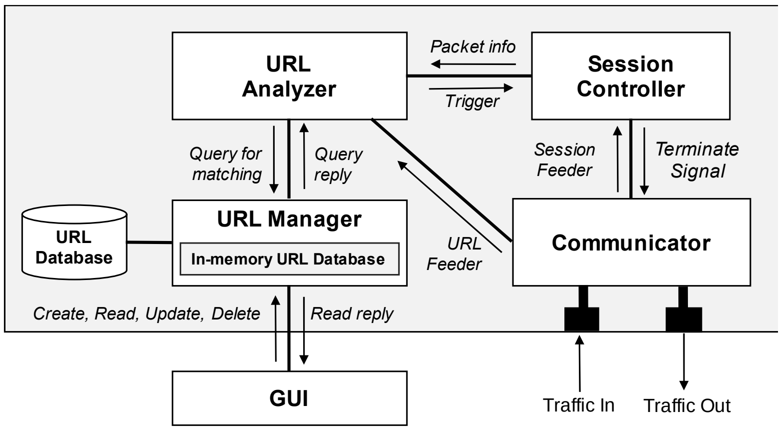

Figure 2. High level system design of the proposed URL filter

3.2 URL Analyzer

The URL Analyzer functions as an inspector and query engine to determine whether a requested URL exists in the in-memory database or not. Given an HTTP connection from a client to a server, the URL Analyzer receives traffic from the communicator and silently observes the TCP 3-way handshake operation between the two endpoints. After the handshake is completed and the connection is established, the URL Analyzer continuously inspects subsequent packets by looking for the "HTTP GET" keyword. Once it is found, the requested URL in the packet will be extracted and compared to the in-memory database. An inspection of the “HTTP GET” keyword helps cover all HTTP packets regardless the port numbers used in the request.

A URL query operation is regarded as a binary tree search operation. A result from a query either returns a *true* or *false* answer depending on whether or not a URL match is found. Neither a false positive nor a false negative answer is allowed in our query operation. For each matching URL, the URL Analyzer activates the Session Controller to start a filtering mechanism.

3.3 Session Controller

Technically, a client-server-based HTTP connection will be completely terminated when the client receives a TCP FIN (Finish) packet from the server and the server receives a TCP RST (Reset) packet from the client. The Session Controller performs this operation by sending a forged FIN and a forged RST packet with a corresponding sequence number to the client and the server, respectively. This technique is widely known as *Session Hijacking*[17]. Section 4 presents the design and analysis of the Session Hijacking technique.

3.4 Communicator

The Communicator acts as an interface for incoming and outgoing traffic by receiving packets from network interfaces and forwarding them to the URL Analyzer and Session Controller. It also passes forged FIN and RST packets to terminate HTTP connections.

4. SESSION HIJACKING

Session Hijacking is widely known as a *man-in-the-middle* (MITM) attack method. In this method, a third-party intercepts communication sessions and pretends to be one of the parties involved in the session. URL filtering needs to recognize every HTTP session by computing the TCP sequence numbers in each session.

4.1 Session Timing Analysis

Session Timing Analysis shows the timing diagrams of URL inspection and packet injections based on the Session Hijacking technique, as described in the following steps:

(1) After completing 3-way handshake, the client sends a URL request to the web server at t_0.

(2) URL filtering captures this request at t_1.

(3) URL filtering analyzes and matches the requested URL and tells the Session Controller to compute the corresponding sequence number. The Session Controller injects one FIN and one RST packet at t_2.

(4) The FIN packet reaches the client at t_3. The RST packet may reach the web server at a time later than the server's reply packet.

(5) The HTTP reply packet from the web server reaches the client at t_4, and the client responds with an RST packet back at t_5.

As shown in Figure 3, the web server receives two RST packets – the forged one from URL filtering and another one from the client. Both packets perform the same functionality, which is to terminate the connection on the web server. Technically, URL filtering does not need to send the RST packet. However, sending it at the earliest possible time may prohibit the server from replying with HTTP data back to the client to save bandwidth.

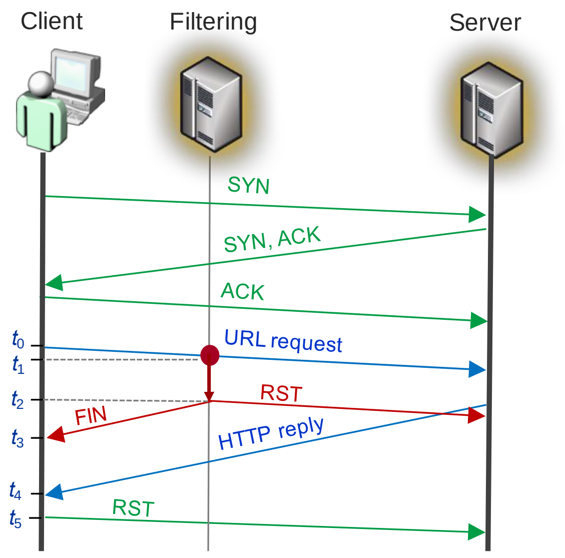

Figure 3. Session Hijacking timing diagram

4.2 Successful Filtering Conditions

As described in the previous section, a forged FIN and a forged RST packet are injected with the correct sequence number. The forged FIN packet forces the client to ignore all subsequent packets belonging to the current session, while the forged RST packet is aimed at stopping the server from relaying data to the client. To achieve successful filtering, the forged FIN packet must reach the client at t_3 before the server's HTTP reply packet reaches the client at t_4. This condition can be expressed by the following inequality:

$$t_3 < t_4 \tag{1}$$

Note that the length of time starting from sending an HTTP request packet at t_0 until the reply HTTP packet reaches the client at t_4 is called the Round-Trip Time (RTT). To represent the inequality of (1) in terms of RTT, we add t_0 to the inequality:

$$t_3 - t_0 < t_4 - t_0 \tag{2}$$

Where $\Delta T_i = t_i - t_{i-1}$, the $t_3 - t_0$ interval is obviously the summation of ΔT_1, ΔT_2, and ΔT_3; hence, we can rewrite inequality (2) as follows:

$$\Delta T_1 + \Delta T_2 + \Delta T_3 < t_4 - t_0 \tag{3}$$

The $\Delta T_1 + \Delta T_3$ is the RTT from the client to URL filtering (RTT_{CF}) and ΔT_2 is the URL processing time (T_{PROC}). In addition, $t_4 - t_0$ is the RTT from the client to the server (RTT_{CS}). Thus, inequality (3) can be rewritten as:

$$T_{PROC} + RTT_{CF} < RTT_{CS} \tag{4}$$

We will show in Section 6 that T_{PROC} is approximately only 1% of RTT_{CF}, and it can be neglected. Therefore, RTT_{CS} is always larger than RTT_{CF}, and a successful filtering condition is guaranteed.

5. URL DATA STRUCTURES

This section describes how to represent URLs with an AVL tree. Firstly, we introduce a basic node representation and the overall tree structure. Then, we show the space and time spent in URL processing.

5.1 Node Representation

We adopt the AVL tree implementation[18] to represent URLs with incremental encoding[19] for URL compression. Figure 4 illustrates a node structure for storing a URL. Each node contains the following five fields:

- **RefID**: Unique URL identifier referenced to its predecessor. The RefID will be incremented by one for each newly created node.
- **cPrefix**: Number of common prefix characters referenced with its predecessor.
- **DiffURL**: Uncommon tail URL string terminated with a null string.
- **Lchild**: Pointer to the left sub-tree.
- **Rchild**: Pointer to the right sub-tree.

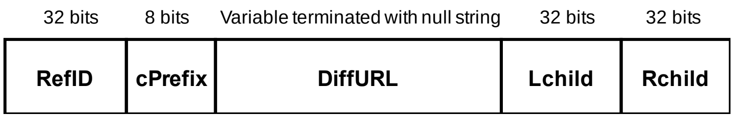

Figure 4. Representation of a URL as a node in an AVL Tree

The URL string will be encoded using the differences between successive data. A node is created and a URL is compressed through the following steps:

(1) The first incoming URL is assigned to be the root node. The **DiffURL** contains the complete URL string. The **cPrefix** is set to zero and the rest of the fields are set to null.
(2) The next URL will be compared with every node on the path starting from the root node and its predecessors to find the common prefix. The **RefID** of a newly added node will point to its predecessor. The **cPrefix** is set to the number of common characters and the remaining URL string is stored in the **DiffURL**.
(3) The **Lchild** or **Rchild** in the predecessor node is updated, depending on the branch of the sub-tree.
(4) Repeat step 2 until no URL remains.

Figure 5 shows an example representing four listed URLs. Searching (reading) a URL is available for both exact matching and prefix matching using standard tree traversal methods. Although additional encoding algorithms, such as Huffman, can be applied to strings in **DiffURL** to get a higher compression ratio, such algorithms will significantly decrease the search performance and should be avoided.

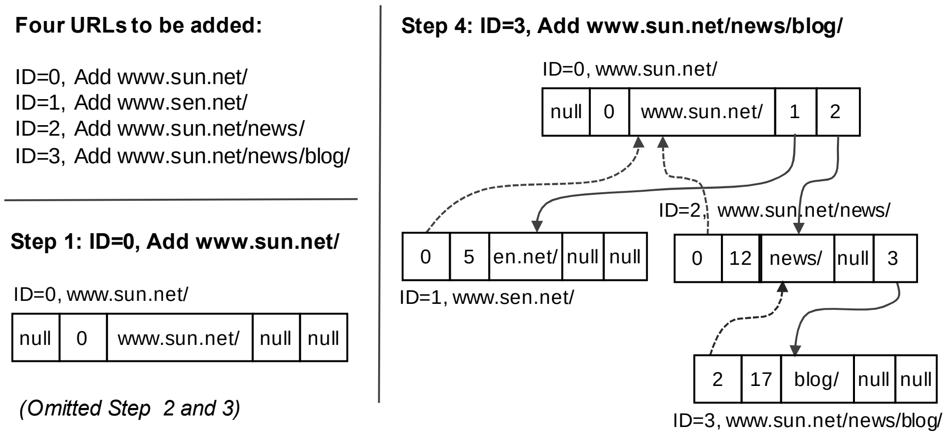
Four URLs to be added:
ID=0, Add www.sun.net/
ID=1, Add www.sen.net/
ID=2, Add www.sun.net/news/
ID=3, Add www.sun.net/news/blog/
Step 1: ID=0, Add www.sun.net/
ID=0, www.sun.net/
null | 0 | www.sun.net/ | null | null
(Omitted Step 2 and 3)
Step 4: ID=3, Add www.sun.net/news/blog/
ID=0, www.sun.net/
null | 0 | www.sun.net/ | 1 | 2
0 | 5 | en.net/ | null | null
ID=1, www.sen.net/
ID=2, www.sun.net/news/
0 | 12 | news/ | null | 3
2 | 17 | blog/ | null | null
ID=3, www.sun.net/news/blog/

Figure 5. An example of URL representation using AVL tree

5.2 Tree Representation

An AVL tree is represented by three arrays, as shown in Figure 6. The first array, TreeNode, contains a list of nodes in the AVL tree. It contains Lchild, Rchild, and their corresponding tree heights. All URL nodes are stored continuously using the CompressedURL array. Direct access to each URL node is referenced by the DataPtr array.

Note that each entry in **CompressedURL** is a slightly modified version of a URL node. The 32-bit **Lchild** and 32-bit **Rchild** (as shown in Figure 4) have been relocated from **CompressedURL** into **TreeNode.** Two nibbles, one from **Lchild** and one from **Rchild**, are borrowed to keep the tree height for re-balancing the rotation operation. Thus, the length of **Lchild** and **Rchild** are reduced from 32-bit to 28-bit pointers. The maximum number of URLs is equal to 2^{28}, or approximately 268 million URLs. This database size is practically sufficient for deployment to large-scale ISPs.

The reason behind this modification is to enhance memory usage and code optimization. Generally, a compiler efficiently optimizes code if the data is aligned in 2^N byte boundary. Moreover, **TreeNode** and **DataPtr** arrays can be viewed as a single array. This allows a URL string query via the **TreeNode** array to act as a regular searching tree and gives direct access to a node by a given URL ID through the **DataPtr** array.

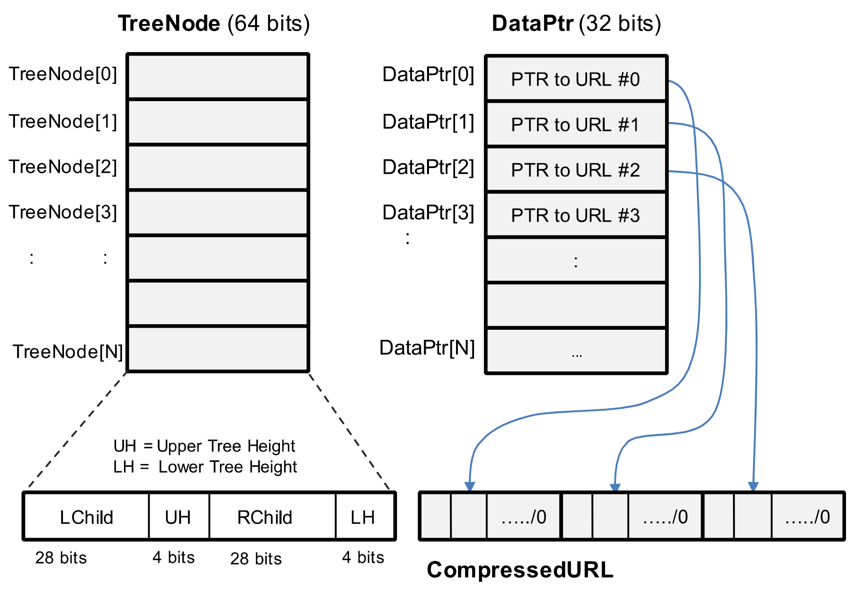

Figure 6. Global structure for URL representation

5.3 URL Storing

With incremental encoding as described in Section 5.1, a node at level *L* will store only non-common suffix strings compared to the previous node at level *L*-1. The total memory bytes required to store compressed URLs, S_{url},, can be computed by summation of the non-duplicated characters from all nodes in the tree and can be expressed as follows:

$$S_{url} = \sum_{j=1}^{n}(\sum_{i=d}^{m} Y_{ij}) + C \qquad (5)$$

where:

Y_{ij} is the number of characters at node *j* and position *I*;
m is the number of URL characters at node *j*;
n is the number of total URL database;
d is the number of non-common suffix character stored in a node and $d \in 1,2,3,\dots,m-1$; and
C is a constant represented all overhead of URL representation.

We ran an experiment on 10 million URLs. The average URL length before compression was equal to 67.49 bytes. The average URL length after compression, including all the overhead, was reduced to be 35.54 bytes. This yields a 52.65% compression ratio. Based on this ratio, compressing a total of 268 million URLs requires approximately 9 GB of memory space.

5.4 URL Matching

Based on URL representation in the AVL tree, the search operation is performed with O (log n) time complexity, where n is the number of nodes in the tree. Each URL matching process requires a character comparison while traversing each node of the tree. The computation time, T_{match}, for URL matching can be expressed as follows:

$$T_{match} = \sum_{j=1}^{\log(n)} \sum_{i=1}^{m} X_{ij} \tag{6}$$

where:

X_{ij} is the matching time for each character of a requested URL with the URL string at node j, and i is the compared character position;

m is the length of URL string at node j; and

n is the number of total URLs in the database.

6. PERFORMANCE MEASUREMENTS

We conducted two experiments to measure system performance. The first experiment measured URL processing time and compression throughput. The second experiment tested the filtering performance at 100 Gb/s synthetic traffic modified from real traffic.

6.1 The Testbed

The testbed was composed of five traffic generators and one filtering engine. Each generator was a DELL R230 server equipped with a Xeon E3-1220v5 4-core CPU running at 3.00 GHz and a dual-port 10 GbE Myricom NIC. A total of 10 ports of 10 Gb/s each generated the total 100 Gb/s of traffic. The filtering engine was a DELL R620 server equipped with dual Xeon E5-2643v2 6-core CPUs running at 3.5 GHz (total 12 CPU cores and hyper-threading off), 64 Gigabyte of total memory, and five dual-port 10 GbE Myricom NICs. The Sniffer10G was used as the packet processing driver in the filtering engine. The system ran on CentOS 6.5 Linux with kernel 2.6.32. Each CPU core was assigned to receive a 10 Gb/s traffic stream. The URL database contained 10 million URLs generated from a blacklist database (http://urlblacklist.com) combined with our own blacklist.

The traffic generators and URL filtering were connected back-to-back with fiber-optic cables as shown in Figure 7. Each traffic generator used the *tcpreplay* tool to replay its own *pcap-based* traffic repository captured from a university campus network. Each repository contains one million packets with a total 651 MB in size.

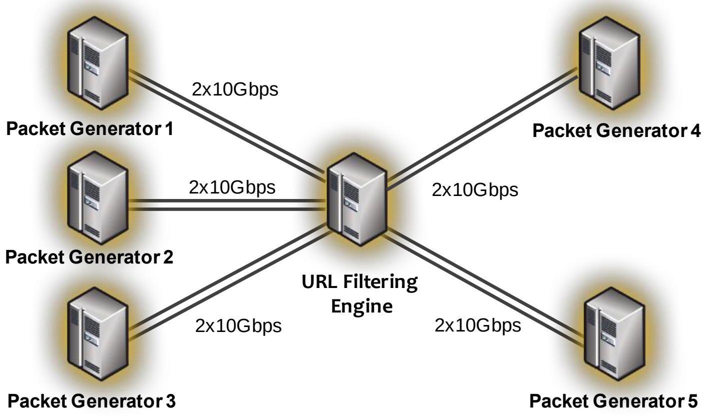

Figure 7. The testbed for 100 Gb/s URL filtering

6.2 Basic URL Processing Performance

We measured the time to create and search one URL from one to 10 million entries. In addition, we measured the compression throughput in terms of Mbps. The measurements were conducted on the filtering engine using a single core. Increased performance could also be achieved by parallelization of these operations.

Storing a URL is a procedure that creates a node and stores it in a compressed form in the AVL tree. Figure 8(a) shows the distribution of time needed to create the AVL tree until it completely contains 10 million URLs. It takes 4.95 μs on average to store one URL (Min = 1 μs, Max = 98 μs). Figure 8(b) shows the compression throughput as a function of the number of URLs. The compression throughput decreases since processing time grows with the tree size.

We tested the searching time by querying a URL entry by entry. Figures 9(a) and 9(b) respectively show searching time using URL ID and URL string. The search using URL ID runs very fast at 0.6 μs on average (Min = 0.1 μs, Max = 35 μs). On the other hand, searching with URL string (matching) requires more processing time. On average, the matching time (T_{Match}) is equal to 3.83 μs (Min = 1 μs, Max = 43 μs).

Referring to the inequality (4), we can express the processing time,

T_{PROC}, as:

$$T_{\text{PROC}} = T_{\text{Match}} + T_{\text{FIN-RST}} \tag{7}$$

The $T_{\text{FIN-RST}}$ is the time that the filtering engine spends to generate a FIN and an RST packet. From the measurement, $T_{\text{FIN-RST}}$ is less than 1 μs, which is relatively smaller than the averaged T_{Match}. Therefore, successful filtering mainly depends on T_{Match}., i.e., performance of the URL matching mechanism.

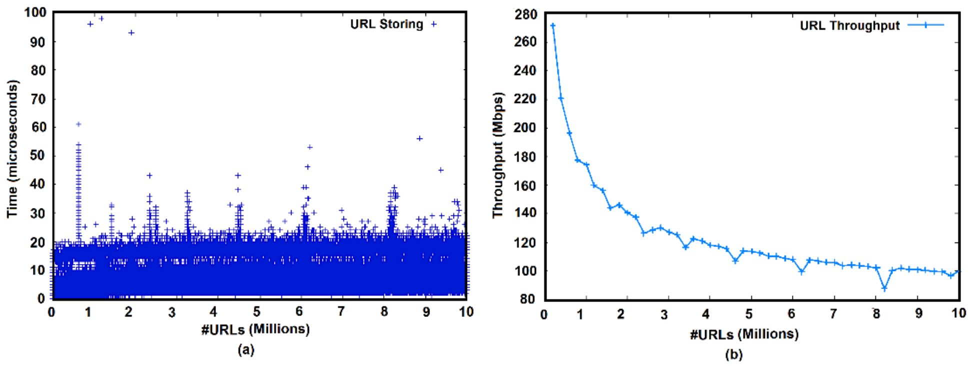

Figure 8. URL processing performance: (a) URL creation time and (b) URL compression throughput

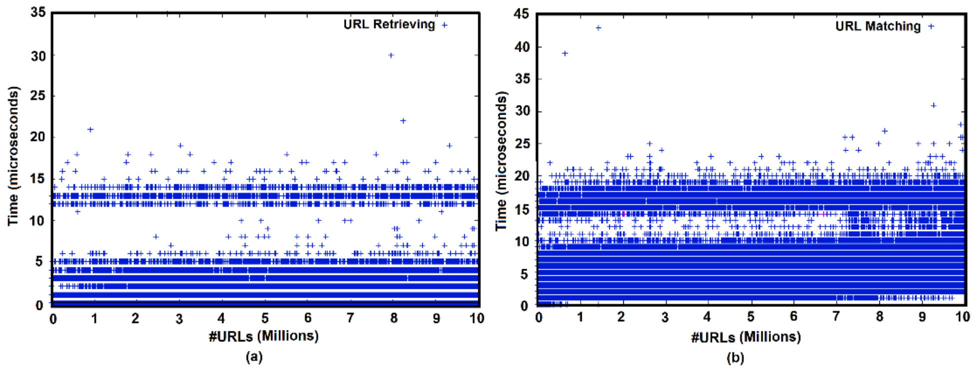

Figure 9. Search performance using (a) URL ID and (b) URL string

6.3 Filtering Performance

We measured the filtering performance in terms of (1) CPU utilization and (2) Filtering throughput as a function of the number of URLs. Each measurement was conducted with a 100 Gb/s traffic rate for two cases: (1)

Mixed traffic (MixT) and (2) HTTP-only traffic (HoT). In MixT, the pcap repository contained mixed traffic. In HoT, all non-HTTP traffic was filtered out. Since a higher consecutive HTTP request rate was generated, HoT creates a heavier load on the filtering engine than the MixT does.

Figure 10(a) shows that the filtering engine can sustain 100 Gb/s bandwidth for both MixT and HoT. In MixT, the system reached 70% of CPU utilization. In HoT, the system consumed 5-15% more CPU utilization.

Figure 10(b) shows the throughput test in terms of packets per second. The graph shows a flat line, since the throughput mainly depends on the traffic rate regardless of the number of URLs. Both MixT and HoT can sustain throughput at 31.1 Mpps and 33.58 Mpps, respectively. HoT gives slightly better packet throughput than MixT, since non-HTTP packets with large payloads were previously filtered out.

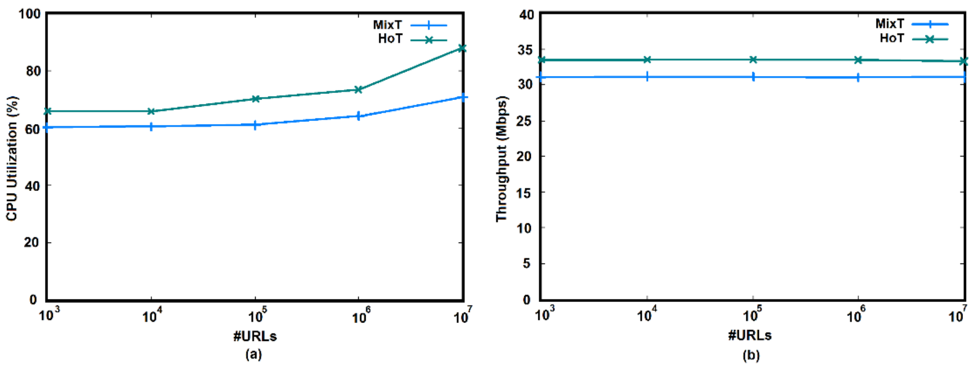

Figure 10. Filtering performance at 100 Gb/s: (a) CPU utilization and (b) Filtering throughput

To further investigate the filtering performance against the traffic rate, we measured CPU utilization as a function of the traffic rate ranging from 1 to 100 Gb/s for both the MixT and HoT cases. As shown in Figure 11(a), CPU utilization linearly increased up to 70% (MixT) and 85% (HoT) for 100 Gb/s.

We collected real RTT statistics between clients and servers from the university campus network. The RTT of 15,000 web servers were recorded and the first 1,500 servers were selected because the remaining values have a very large RTT, and they can be neglected. Figure 11(b) shows the ascendingly sorted RTT_{CS}, ranging between 12.15 and 200.57 *ms* (Maximum 900 *ms* is found when total 15,000 servers are considered). The result confirms our assumption described in inequality (4) that the RTT_{CS} is much larger than T_{PROC}+ RTT_{CF}.

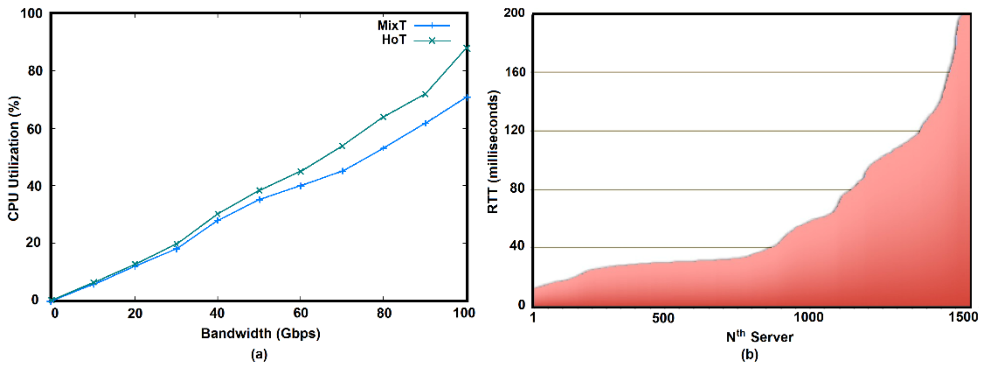

Figure 11. (a) CPU utilization as a function of bandwidth and (b) RTT between clients and servers

7. CONCLUSION

We proposed a compact URL representation scheme with fast URL matching. The outcome was a software-based URL filtering system that supports a 100 Gb/s data rate using an industry standard COTS server with multi-core processors. The design requirements must meet the following criteria : (1) Low latency, (2) Configurability, and (3) Economical cost. The proposed URL filtering method is transparent to clients using a non-blocking method and has zero latency. The system can handle up to 286 million URLs with flexible configurability. Nowadays, a standard 10 GbE NIC costs less than \$500. Using high-end Intel Xeon-based COTS hardware, a 100 Gb/s URL filtering system can be built for less than \$15,000. In the future, we are planning to implement the system with a single 100 GbE NIC and use the DPDK framework[8] as a packet processing core instead of using Sniffer10G.

Our system is designed to handle unencrypted HTTP traffic. Filtering encrypted traffic requires using a pass-through architecture and installing trusted certificates on the client side. This method has not been commonly deployed in public ISPs, except in small or enterprise networks. Moreover, a new transport service like Quick UDP Internet Connections (QUIC, pronounced '*quick*')[20] to speed up HTTP requests using UDP instead of TCP was recently proposed. QUIC is a UDP-based secure and reliable transport for HTTP/2.0. The Session Hijacking technique under the pass-by architecture will not work under QUIC, and the only other choice would be to implement a pass-through technique to handle QUIC; otherwise, ISPs must disable QUIC at their gateways and fallback to TCP-based HTTP requests.

8. REFERENCES

[1] J. Zittrain, and J. Palfrey, Internet filtering: The Politics and Mechanisms of Control. In J. G. Deibert et al. (Eds.), *The practice and policy of global internet filtering* (p26-56). Cambridge MA: MIT Press, 2008.

[2] M.T. Banday, and N.A. Shah, *A concise study of web filtering*. AIS Electronic Library (AISeL), 10(31), 2010. Retrieved on January 15, 2016, from http://aisel.aisnet.org/sprouts_all/352/.

[3] A. Goodney, S. Narayan, V. Bhandwalkar, and Y.H. Cho, Pattern based packet filtering using NetFPGA in DETER Infrastructure. *Paper presented at the 1st Asia NetFPGA Developers Workshop*, Korea, June 14, 2010.

[4] J. Garnica, S. Lopez-Buedo, V. Lopez, J. Aracil, and J.M.G. Hidalgo, A FPGA-based scalable architecture for URL legal filtering in 100GbE Networks. *Paper presented at the International Conference on Reconfigurable Computing and FPGAs*, Mexico, December 5-7, 2012.

[5] H. Chen, R. Liu, Y. Chang, Y. Huang, P. Wu, A. Yeh, and N. Huang, The design and implementation of network-processor based gigabit web filtering system. *Paper presented at the Taiwan Area Network Conference (TANET2002)*, Hsin-Chu, Taiwan, October 30, 2002.

[6] L. Deri, M. Martinelli, and A. Cardigliano, Realtime high-speed network traffic monitoring using ntopng. In N.F. Velasquez (Ed.), *Proceedings of the 28th USENIX Conference on Large Installation System Administration Conference (LISA14)* (p69-79), Seattle, Washington: The Advanced Computing Systems Association, 2014.

[7] L. Rizzo, Netmap: A novel framework for fast packet I/O. *Paper presented at the 2012 USENIX Annual Technical Conference* (*USENIX ATC12*), Boston, Massachusetts, June 13-15, 2012.

[8] DPDK, Data plane development kit for fast packet processing. Retrieved on January 10, 2016, from http://dpdk.org/.

[9] Openonload, The high performance network stack. Retrieved on January 10, 2016, from http://www.openonload.org/.

[10] Sniffer10G, Complete packet capture in a cost-effective package. Retrieved on January 10, 2016, from http://www.cspi.com/ethernet-adapters/ software/sniffer10g/.

[11] H. Yuan, B. Wun, and P. Crowley, Software-based implementations of updateable data structures for high-speed URL matching. *Paper presented at the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (*ANCS*), La Jolla, California, October 25-26, 2010.

[12] Z. Zhou, T. Song, and Y. Jia, A high-performance URL lookup engine for URL filtering systems. *Paper presented at the IEEE International*

Conference on Communications (*ICC*), Cape Town, May 23-27, 2010.

[13] R. Enbody, and H. Du, Dynamic hashing schemes. *ACM Computing Surveys*, 20(2), p850-113, 1988. http://dx.doi.org/10.1145/46157.330532.

[14] A. Breslow, D. Zhang, J. Greathouse, N. Jayasena, and D. Tullsen, Horton tables: Fast hash tables for in-memory data-intensive computing. *Paper presented at the 2016 USENIX Annual Technical Conference* (*USENIX ATC16*), Denver, Colorado, June 22-24, 2016.

[15] T. Zink, and M. Waldvogel, Efficient hash tables for network applications. *SpringerPlus*, 4(1), p1-19, 2015. http://dx.doi.org/10.1186/s40064-015-0958-y.

[16] R. Sedgewick, and K. Wayne, Algorithms 4th edition. Boston: Addison-Wesley Publishing, 2011.

[17] Z. Qian, Z. Mao, and Y. Xie, Collaborative TCP sequence number inference attack- how to crack sequence number under a second. In G. Danezis (Ed.), *Proceedings of the ACM conference on Computer and Communications Security* (p593-604). North Carolina: Association for Computing Machinery, 2012.

[18] K. Koht-Arsa. *High performance clustered-based web spider*. Master's Thesis, Department of Computer Engineering, Faculty of Engineering, Kasetsart University, 2003.

[19] P. Ferragina, and G. Manzini, Indexing compressed text. *Journal of the ACM*, 52(4), p552-581, 2005.

[20] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, How secure and quick is QUIC? Provable security and performance analyses. *Paper presented at the IEEE Symposium on Security and Privacy*, San Jose, May 17-21, 2015. http://dx.doi.org/10.1109/SP.2015.21.