Randomness Testing of Non-Cryptographic Hash Functions for Real-time URL Based Internet Filtering Applications

By

Tahir Ahmad

2011-NUST-MS-CCS-32

Supervisor

Dr. Usman Younis

Department of Electrical Engineering

A thesis submitted in partial fulfillment of the requirements for the degree of Masters in Computer and Communication Security (MS CCS)

In

School of Electrical Engineering and Computer Science,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(October 2013)

Approval

It is certified that the contents and form of the thesis entitled "**Randomness Testing of Non-Cryptographic Hash Functions for Real-time URL Based Internet Filtering Applications**" submitted by **Tahir Ahmad** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Usman Younis**

Signature: ____________________

Date: ____________________

Committee Member 1: **Dr. Adnan Khalid Kiani**

Signature: ____________________

Date: ____________________

Committee Member 2: **Dr. Abdul Ghafoor**

Signature: ____________________

Date: ____________________

Committee Member 3: **Dr. Ali Mustafa Qamar**

Signature: ____________________

Date: ____________________

Abstract

In this thesis, a real-time URL based filtering system capable of URL filtering and blocking from domain level to sub folder level is proposed. It can be used as a standalone solution and can be integrated at any network level. The modular approach provides scalability by stacking hardware boxes to meet the current and future bandwidth requirements. The total delay caused by our proposed system is less then 1ms which is highly desirable for real-time filtering systems.

The core component of real-time URL based filtering system is URL lookup and storage engine. In this work, hash tables are used to develop the URL lookup and storage engine. The performance of hash table is dependent upon hash function used for implementation of hash table. To gain the space and time advantage using Hash Tables for real-time application over other constant space-set data structures following approaches are used: 1) The use of hash functions for compressing variable length URL strings to fixed size integers and dynamic allocation of memory 2) To compensate the collision problem associated with hash functions by performing statistical analysis and implementation of various non-cryptographic hash functions to identify their random nature and hash table resize operation depending upon the load factor. The performance analysis is performed mainly using statistical studies on the sequences generated using five widely used non-cryptographic hash functions: 1) CRC, 2) Adler, 3) FNV, 4) DJBX33A, and 5) Murmur. The comparative analysis of tested non-cryptographic hash functions shows that the Adler hash function is not suitable for hash table implementation, whereas, the rest of non-cryptographic hash functions exhibit similar and better randomizing features which make them an attractive choice for hash table implementation. The results of these statistical studies have been verified by the implementation of hash table using these non-cryptographic hash functions. The implementation results show that the average number of probes for Adler based hash table varies between 1.25 and 2.75 for different load factors and hash table sizes; whereas, for the rest of non-cryptographic hash functions the average number of probes in a hash table is 1, which is highly desirable for real-time network applications. Thus proving that 1) CRC, 2) FNV, 3) DJBX33A, and 4) Murmur non-cryptographic hash functions are good choices for hash table based implementation for real-time storage and lookup of uniform resource locators.

Analysis and trace-based experiments are employed to explore the properties of our proposed hash table based URL filtering System. The results show the false positive rate of 0.0 up to 160 Mbps and increases up to 0.29 at 300 Mbps. The URL lookup rate is 100% at 160 Mbps and drops gradually to 71% when data rate approaches 300 Mbps. The 0% packet drop ratio is observed up to 160 Mbps and gradual increase as the data rate increases i.e. around 30% at 300 Mbps. The URL storage rate of up to 90% at 1Mbps and drops only 10% when data rate rises to 90Mbps. The initial loss of 10% URLs is due to storage of URLs on secondary media for classification of suspicious and benign URLs. The URL storage in our proposed architecture is in passive-mode, so it doesnt affect the real-time performance of our URL filtering system.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Tahir Ahmad**

Signature: ____________________

Acknowledgment

Above all, I am extremely thankful to Almighty Allah, for His countless blessing bestowed on me throughout my life. Without His will and blessing nothing would have been possible.

My profound gratitude to my thesis supervisor Dr. Usman Younis, for his continuous encouragement, enlightening advice, and most willing help extended by him throughout my research work.

I am also thankful to Dr. Abdul Ghafoor, Dr. Ali Mustafa Qamar and Dr. Adnan Khalid Kiani for their support and time. Special thanks to ITS Departments Mr. Ajmal Farooq (System Administrator) and Mr. Abdel Wahab Khan (Network Administrator) for helping in data capturing at SEECS edge router and providing access logs of squid proxy server of SEECS.

I also want to mention the names of my friends, Muhammad Hanif, and Muhammad Faheem for their unconditional support. I am grateful to them. Last, but by no mean the least, my heartfelt thanks to my wife, little Eisha and sweet parents for their unwavering support and encouragement throughout my Master studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Today internet is the main source of information. However, all the information is not meant for everyone and there is also a lack of information flow regulations. This gives rise to problem like sensitive information in wrong hands and as a consequence can mislead the user. For example, pornographic material is harmful for kids, and information which may facilitate crimes is dangerous. All these issues are considered in the domain of information security for which information filtering is required, and is achieved using Internet Filtering. The spectrum of Internet Filtering comprises of,

- Cyber Safety
- Censorship
- Malware Protection

In Pakistan, if some offensive material needs to be banned, there is no centralized mechanism in place. Pakistan Telecom Authority (PTA) which is internet governing body in Pakistan forward request to Internet Service Providers (ISPs) as they are in compliance with regulations imposed by PTA. The ISPs then block the requested URL/web address by manually adding them to their block list. Each ISP has to manually undergo this blocking process. At times they slip through the cracks, which results in under blocking or over blocking. There is a need to automate the URL filtering and blocking process. ISPs in Pakistan donate money as a licensing agreement to National Information and Communication Technology (ICT) R&D Fund. They therefore requested ICT R&D to indigenously develop an Internet filtering and blocking system. This turns as a motivation for my thesis to take a first step towards building a real-time URL-based filtering system.

1.2 Internet Filtering Techniques

From technical perspective, the two common technique used for internet filtering are,

- Packet level filtering
- Higher layer filtering such as URL based filtering

In packet level filtering, the filtering can be done at Network Layer by just looking at the IP addresses or at Transport Layer by filtering the packets on port bases. This is the most efficient level approach towards filtering but at the cost of collateral damage. Many domains resolve to a single IP address.

Thus blocking based upon IP addresses will cause over blocking, which makes even legitimate domains inaccessible.

The other approach towards filtering using higher protocol layers is capable of performing web content filtering. This can be done using web proxies and deep packet level firewalls. Both schemes have similar purpose but are different in terms of architecture. Web Proxies are ideally suited for small network like universities and organizations. It provides anonymity to the users sitting behind the proxy. The filtering schemes in case of web proxy can be categorized as Index based filtering and Analytic based filtering. The prior perform filtering based on blacklist/ whitelist basis. The later perform filtering based upon the web content like keywords and images. This approach at times causes the problem of over blocking and under blocking due to its inability to classify the images in context of content. The Deep packet level firewall performs filtering directly on the data packets. The filtering can applied on the basis of packet header termed as stateful packet inspection. The other approach towards filtering packets in deep packet level firewall is based upon packet payload termed as deep packet inspection (DPI). It provides finer controls and helps in detection of malwares in pages with active contents. Attacks such as phishing and drive by download can be prevented using DPI approach.

1.3 Deployment Scenarios for Internet Filtering

Internet filtering techniques can be deployed at various levels in internet architecture. The users of internet are normally connected to ISP, which is connected to backbone via backbone service provider. Thus different filtering techniques can be deployed at different levels. Following are the possible deployment scenarios,

- Internet filtering between internet user and the ISP
- Internet filtering at the ISP level
- Internet filtering between the ISP and backbone service provider
- Internet filtering at the backbone service provider (BSP), to filter illegal internet coming from outside world.

It is important to avoid single point of failure while deploying internet filtering at ISP and BSP level. Most of internet traffic today is generated dynamically by social media websites, thus making the task of content filtering of both webpages and packets a challenging task. The challenges include dealing with various distributed contents, variation in access mode like non-HTTP such as peer to peer traffic and also to deal with encrypted traffic.

1.4 Contribution of Thesis

The practical solution towards filtering will be a hybrid scheme as shown in Figure 1.1. In the above discussed internet filtering techniques, only IP

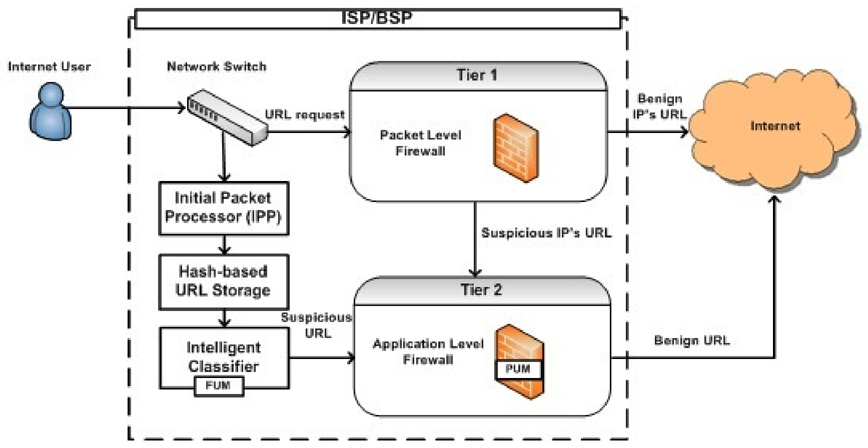


Figure 1.1: Network Architecture of Proposed System

.

level packet filtering has the capability to be deployed at high data link such as BSP. The other internet filtering schemes suffer with longer latencies. In hybrid scheme, the filtering can be performed in a two tier filtering system. At tier 1 the filtering is based upon IP level packet filtering. Most of the legitimate traffic is allowed to bypass the tier 2 of filtering system. Only packets with suspicious IP addresses or port numbers are forwarded to tier 2 of the filtering system. The tier 2 of filtering system is DPI based. It makes the filtering decision on basis of higher layer (layer 7) protocol information. The filtering decision can be either based on header information of protocol or on the payload of the protocol.

This work is focused on the tier 2 of internet filtering system. The decision at this tier is made by looking at the header information of Hyper Text Transfer Protocol (HTTP). HTTP works on top of TCP session. Once a TCP session is established between the host and webserver, the user request i.e. the Uniform Resource Locator (URL) of a webpage hosted on that webserver is forward using HTTP GET request. URLs stay for a longer duration in the system as compared to IP information, therefore, the classification based on URLs remains valid for longer period. The core of URL filtering system is a blacklist database containing all prohibited URLs, and a URL lookup engine. The challenges faced are the management of blacklist database, real-time query execution, and finally decision making to allow or deny the requested URL.

The main challenge in designing a URL-based filtering is the character encoding. URLs are merely character strings of variable lengths, which make it a challenging task to store URL entries in a memory efficient way, and also to perform the requested URL lookup in a minimal time. Hash functions are widely use for compression and conversion of URLs to a fixed size, and contribute towards improving memory and computing requirements.

The scope of this Research includes the study of various non-cryptographic hash functions (NCHF). NCHF due to their computational efficiency are widely used in many computing applications [5, 6, 14, 18, 23, 31]. These includes hash table implementation in which NCHFs make it possible to perform lookup operation in O(1) time irrespective of the table size [17, 21, 15]. Additionally, the performance of a hash table depends upon the random output of the hash function which is employed in its implementation. Randomizing functions are generally good NCHFs, but not the vice versa. Therefore, the performance of NCHFs as randomizing functions have been investigated in this work. Exhaustive test runs, identified in National Institute of Standards and Technology (NIST) statistical suite, have been executed on five NCHFs used in hash table implementation. Important features which determine the quality of a NCHF are collision resistance, distribution of outputs, avalanche effects, and speed [12, 19, 14, 13, 1]. Additionally, all of these features are data-dependent [29]. It is required of a NCHF to produce the output which follows a uniform distribution [19, 14]. The function must generate each possible output value with equal probability and it must be independent of the input distribution [19].

The focus of this work is to evaluate the randomness of uniform distribution of mapping done by NCHFs for input uniform resource locators (URLs). In our proposed hash table based approach for URL storage and Lookup, the entries are stored as a key-value pair. We are setting URL as a key, and the hash of a key provides the address of a slot where entries are required to be stored. An associated problem with the hash functions is the collision which occurs when the calculated hash of two distinct keys map to a same slot. This is resolved by finding an alternate slot for the storage of collided entry, called probing. The strength of our proposed URL storage and lookup lies in the random outcome of a hash function. A suitable hash function provides unique hashes which minimizes the probing and thus ensures O(1) average time for URL storage and lookup.

A hash function used in real-time network applications can be categorized as Checksum, cyclic redundancy check (CRC), and NCHF. We have analyzed following 32-bit hash functions,

- CRC-32 hash function (CRC hash function) [10, 20, 14].
- Adler-32 hash function (Checksum hash function) [10, 22].
- DJBX33A Hash function (non-cryptographic hash function) [6].

- Fowler-Noll-Vo(FNV) hash function (non-cryptographic hash function) [6, 4].
- and, Murmur hash function (non-cryptographic hash function) [1, 6].

1.5 Thesis Organization

The rest of the thesis is organized as follows; Chapter 2 presents the literature review and highlights the trends in approaches to design internet filtering and blocking systems. In Chapter 3, we discuss the objectives of this thesis along with data sets used for evaluation of system. Chapter 4 presents Research Methodology adopted in this research work and shows the architecture and deployment scenario of our proposed real-time URL based internet filtering system. In chapter 5, we discuss the underlying data structures of our proposed system. Chapter 6 presents the randomness testing of non-cryptographic hash functions for real-time network applications. In Chapter 7, we share the results and discussion of trace based experiments. Chapter 8 concludes the thesis and gives recommendations for future work.

Chapter 2

Literature Review

2.1 Internet Filtering

The idea of internet filtering is not new but there is a clear change in trends in the approach to achieve filtering. In mid 2000, the trend was a total hardware based solution but as the constraints of processing and memory are reduced the trend shifted towards software based solution, which enables internet filtering systems to achieve much finer controls. The proposed systems in 2005 are both hardware based. In [16], a fast URL lookup and storage engine is presented for content-aware multi-gigabit switches. The task of Uniform resource Locator (URL) lookup and storage is performed using Content Addressable memory (CAM). It is a hardware based implementation of associative array. When a data word is provided as input, CAM searches its memory and if entry exits in storage database, it returns the address of that data word. The advantage of using CAM for URL lookup and storage is the constant lookup time for each search operation. The proposed fast scalable URL lookup mechanism that uses CAM perform lookup results every single clock cycle by applying pipeline technology and the system can perform 100 million lookups per second for 100 MHz processing speed. The system has prefix and exact matching capability. In [25], a Network processor based gigabit multiple-service switch is proposed. URL lookup mechanism in this multiple-service switch uses CAM. It uses IQ2000 network processor and classiPI co-processor for URL lookup.

Since 2010, the trend in the area of internet filtering is shifted from total hardware based solution to more software dependent platforms. Gao Fuxiang et al. in 2010 [9] proposed a web access monitoring system. Web access monitoring is performed on basis of URL analysis and decision making on basis of blacklist database consisting of a list of suspicious URLs. URL lookup operation is performed using hash functions based fast matching algorithm. The system can perform exact URL matching but is incapable of performing prefix matching. Zhou et al. in 2010 [30] proposed a matching algorithm, which acts as URL lookup and storage engine. The storage efficiency is achieved using URL compression algorithm and for URL lookup multiple string matching based algorithm are used. The URL compression approach helps to achieve higher compression rates than its counterparts like UBF Guard and hash chain methods. The URL lookup and storage engine is capable of comparing 100,000 URLs in 0.26 seconds. The drawback of this system is the number of false positive, which occurs due to the use of CRC32 hash function in URL compression algorithms.

Thomas et al in 2011 [26] presented Monarch an email-based spam filtering system. It is capable of performing real-time URL spam filtering. The system is based on Whitelisting approach. At proxy a whitelist containing good domains is maintained. Monarch uses machine learning approach for classification of URLs in suspicious and benign. This classification is based upon the collected features. The system has a latency of 5ms to parse a URL. The accuracy of the system is 90.78% with 0.87% false positives. Feng et al. in 2011 [8] proposed a network based URL filtering system. Classification is based upon blacklisting approach. Generally internet filtering systems are gateway based. At higher data rates these gateway based filtering systems become the network bottleneck. In this work, the load of gateway based URL filtering system (GUF) is reduced by shifting control to Network based URL filtering system (NUF). It adds an additional overhead in terms of latency in range of 100 to 500ms. Local caching of search results is done at GUF, to avoid the overhead of latency. Multi-level bloom filters helps in reducing memory requirement of NUF gateway to 90.9% and providing low operation latency at the same time.

Garnica et al. in 2012 [11] proposed an FPGA based URL legal filtering. It is a hardware-software based solution for URL filtering at high data rates such as backbone service providers (BSPs). It is based on two level filtering. At level 1, an FPGA based hardware implementation is used to perform filtering on IP packet level filtering. At level 2, a software based lookup method is used. Other FPGA based solutions are totally hardware based and suffers from false positives; here false positives are handled at software level. The proposed system is capable of operating at 10 GbE and scalable up to 100 GbE. The two tier architecture for internet filtering helps in avoiding legitimate traffic to be tested at level 2, thus saving processing time. Only traffic from suspicious IP addresses existing in the blacklist database are forwarded for further inspection.

Yuan et al in 2013 [28] proposed a Multi-pattern matching algorithm for URL filtering system. The multi-pattern matching algorithm enables it to perform as a high-speed filtering system. A data set of 10 Million URLs is used to evaluate the performance of proposed system. The empirical results show that the system can operate up to 100Mbps with zero false positive. To avoid hash collisions two phase hash functions are used.

2.2 Randomness Testing of Non Cryptographic Hash Functions (NCHF) for Hash table based implementation

Search Table based data structures are widely used in many computing applications. They can be implemented using linear lists, binary search trees and hash tables. Among these data structures the hash table based approach is ideally suited for real-time network applications. Minimal effort is required to manage them to achieve its average case performance i.e. O(1), while linear lists have best case performance of O(n) and binary search trees have best case performance of Olog(n). This best case performance for binary search tree is achievable only for balance binary search tree. A lot of effort is required to maintain a balance binary search tree. But in case of hash tables, if a little care is taken in selection of hash function and design of hash table, it can guarantee an average case performance of O(1).

De Oliveria et al. in 2009 [3] highlighted the use of hash tables for real-time network applications. They first define limits of worst case scenario for hash table. Once the worst case limits are identified, they define a probability level to mark the occurring events as irrelevant which occurs outside the limits of worst case scenario. It is also proposed that hash collisions which are associated with the use of hash functions can be resolved using open addressing and chaining. The work concluded the possibility to adopt a probabilistic approach for the worst case behavior of hash tables.

Yuan et al. in 2010 [27] proposed a hash table based approach for high-speed URL matching. Comparative analysis between Hash tables and Deterministic Finite Automata (DFA) techniques is performed. It is noted that hash table consumes less memory then DFA and a significant throughput gap is being monitored between hash table and DFA based approaches. It is suggested that hash tables are by far superior then DFA based technique for URL matching applications such as URL blacklisting, URL based forwarding and URL shortening services.

Cesar Estebanez et al. in 2013 [7] presented performance of the most common non-cryptographic hash functions (NCHF). The importance of NCHF to provide extremely efficient searching of items in unsorted sets. It is highlighted that quality of NCHFs is dependent upon its collision resistance, Distribution of output, Avalanche effect and speed. The performance analysis is done for most common NCHF in literature including FNV, Murmur and DJBX33A. In our work, we have also analyzed these NCHF for implementation of Hash tables.

Weiling Chang et al. in 2010 [2] performed randomness testing of compressed data. Random number generators have great importance in many computing applications. The randomness testing are performed using NIST statistical test-suite and DIEHARD test provided by Dr. Marsaglia. NIST have developed 15 tests to evaluation of randomness of binary sequence generated by some random number generator. These test have been documented in NIST special publication (SP) 800-22, "A statistical test suite for random and pseudo random number generators for cryptographic applications" [24]. Diehard Test Suit has 18 tests for evaluation of randomness of binary sequence generated by some random number generator. The problem with Diehard test suite is lack of concrete criteria which makes it very difficult to pass these tests. In our performance evaluation none of our hash function passes DIEHARD test suite but we do have binary sequences passing few test from NIST statistical test suite.

Since, hash table based data structure for real-time network applications is a popular research area and performance of hash functions is data dependent. The randomization aspect of non-cryptographic hash functions for URL based string inputs is not studied yet; this thesis provides a comprehensive analysis in randomness terms of various Non-cryptographic hash functions for URL based string inputs.

Chapter 3

Problem Statement

3.1 Objectives

The current web filtering and blocking system deployed at Internet Service Provider (ISP) in Pakistan are incapable of dealing with millions of black-listed URL entries. Recent complete blockage of Youtube and Facebook services highlight the inability of internet governing body in Pakistan i.e. PTA to deal with specific web content. There is a need of a national web filtering and blocking system to be deployed at ISP level. This thesis presents a real-time URL based filtering system capable of URL filtering and blocking from domain level to sub folder level. It can be used as a standalone solution and can be integrated at any network level. The proposed sysetem is based on a modular approach, which provides scalability by stacking hardware boxes to meet the current and future bandwidth requirements. The total delay caused by our proposed system is less then 1ms which is highly desirable for real-time filtering systems. A hash table based approach is used for efficient URL lookup and storage to meet the needs of high data rates links. For selection of hash function suitable for real-time network application, a comparative study of various non-cryptographic hash functions is performed.

3.2 Datasets

The research is based upon a quantitative approach. Analysis and trace based experiments are being employed to explore various non-cryptographic hash function suitable for hash table implementation and performance evaluation of our hash table based URL lookup and storage engine. Following datasets are used for testing purpose,

3.2.1 Dataset No. 1

This dataset is used for selection of hash function to be used for hash table based URL filtering and blocking system. A total of 2744529 URLs are captured at the squid proxy server of School of Electronic Engineering and Computer Science (SEECS), NUST, Islamabad at various times and saved in six different files given in Table 3.1.

3.2.2 Dataset No. 2

This dataset is used for evaluation of our proposed URL Storage and lookup engine. It comprises of 24 hours of internet traffic captured at edge router of School of Electronic Engineering and Computer Science (SEECS), NUST, Islamabad. A total of 288 PCAP files are captured using TCPDUMP each of 300 seconds. The already captured PCAP files are further analyzed using our application and results are being confirmed using wireshark network packet

Table 3.1: The collected data sets which have been provided as an input to the hash functions.

Sr. No.	**Dataset Name**	**URL Entries**	**URL Components**
1	url_al	929583	4
2	url_al1	38115	3
3	url_al2	47390	3
4	url_al3	24629	3
5	url_a4	9453	3
6	url_al5	1695359	3

capturing tool. We then selected one PCAP file from each hour of the day, so a total of 24 PCAP files are used evaluation purpose. Table 3.2 shows the details of these PCAP files.

Figure 3.1 shows the variation in data rate at various times of the day. The maximum data rate of captured pcap files is 90 Mbps and the peak hours are 10 AM to 6 PM.

Figure 3.2 shows the number of packets for already captured pcap files. It can be seen that the number of packets are in proportion to the variation in data rate at various hours of the day.

Figure 3.3 shows the total number of URL request for each of PCAP file. A total of 50165 URL requests are made. The numbers of URLs are neither dependent on data rate nor on total number of packets.

Figure 3.4 shows the average number of URL components for the requested URLs. The average URL components for these 50165 URLs are 3.654.

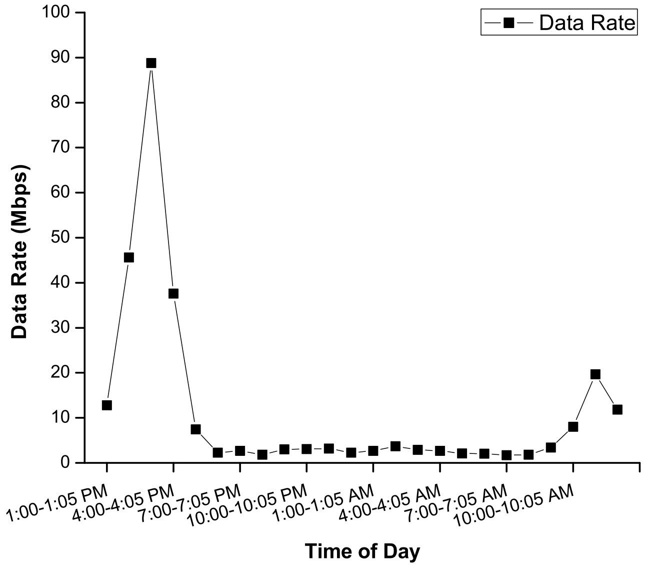


Figure 3.1: Data Rate vs Time of Day

.

3.3 Testing Scenario

The testing of our proposed hash table based URL filtering and blocking system is performed in following two phases,

- Performance evaluation of various non-cryptographic hash functions to be used in hash table based URL storage and lookup engine.
- Performance testing of URL storage and lookup engine.

In phase 1, the performance evaluation of various non-cryptographic hash functions is done using Dataset No. 1. NIST statistical test-suite is used for randomness testing of outputs of these NCHFs. The URLs in Table 1 are combined together in a single URL file i.e. url_list consisting of 2744529 URLs and passed to each NCHFs as shown in Table 3.

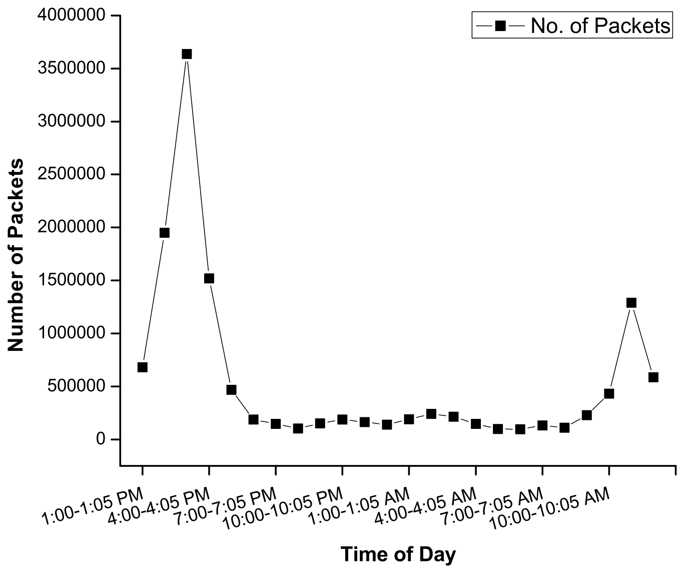


Figure 3.2: Number of Packets vs Time of Day

.

The list of unique URLs (url_list) is given as input to each hash function and the hashed outputs are stored in ASCII format. Each output file is further divided into fifteen sub files containing 106 bytes, in order to repeat the statistical tests fifteen times using distinct data.

The results of the statistical test are being verified by implementation of hash table based URL lookup and storage engine using each NCHF.

In phase 2, the performance evaluation of URL lookup and storage engine is done using dataset No. 2. Each pcap file is replayed to Ethernet port using TCP REPLAY at various data rates ranging from 1 Mbps to 300 Mbps and is captured from Ethernet port using our application. Following performance metrics are calculated,

- URL lookup rate

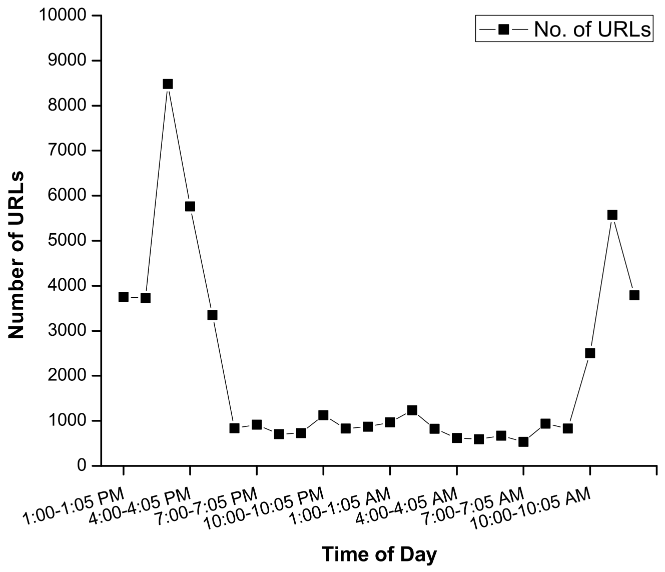


Figure 3.3: Number of URLs vs Time of Day

.

- URL storage rate
- Packet drop ratio
- False positive rate

3.4 Areas of Application

This research is applicable to many areas of application such as,

- Regulatory bodies like PTA in Pakistan
- Academia
- Core Networks

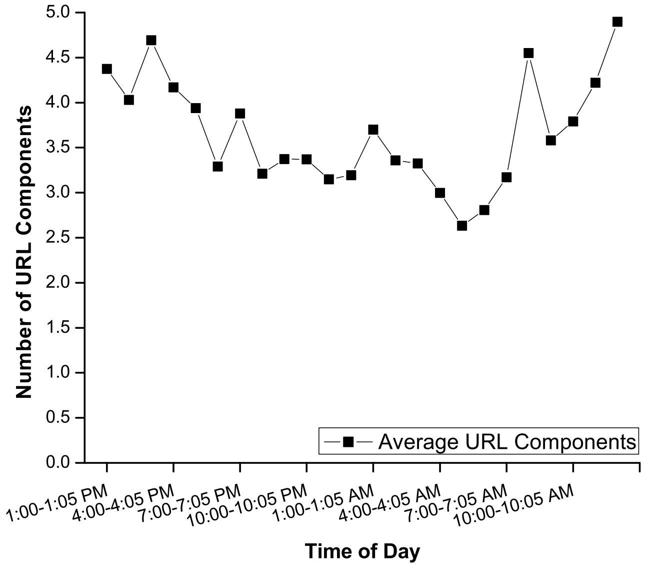


Figure 3.4: Number of URL Components vs Time of Day

.

- Distributed Networks
- Energy saving in terms of efficient bandwidth utilization

Table 3.2: Details of captured PCAP files.

PCAP Files	**Time of Day**	**No. of Packets**	**No. of URL Requests**
1	1:00-1:05 PM	679627	3753
2	2:00-2:05 PM	1950814	3724
3	3:00-3:05 PM	3636460	8479
4	4:00-4:05 PM	1519061	5761
5	5:00-5:05 PM	467662	3350
6	6:00-6:05 PM	188404	835
7	7:00-7:05 PM	146584	913
8	8:00-8:05 PM	103657	707
9	9:00-9:05 PM	150705	728
10	10:00-10:05 PM	186457	1124
11	11:00-11:05 AM	163032	833
12	12:00-12:05 AM	138778	871
13	1:00-1:05 AM	190122	968
14	2:00-2:05 AM	241201	1232
15	3:00-3:05 AM	215253	826
16	4:00-4:05 AM	147450	621
17	5:00-5:05 AM	98124	594
18	6:00-6:05 AM	94795	672
19	7:00-7:05 AM	131267	537
20	8:00-8:05 AM	110739	939
21	9:00-9:05 AM	228526	830
22	10:00-10:05 AM	431454	2503
23	11:00-11:05 AM	1288929	5575
24	12:00-12:05 PM	586292	3790

Chapter 4

Research Methodology

The aim of this research is to develop a real-time URL filtering system, which can work in parallel with existing architecture, which is based upon packet level firewall. The system provides in depth filtering based on application layer information. The proposed system is capable of performing URL lookup operation with zero false positive at high data rates. The system is based upon a modular design which makes the system scalable. The system has capability to manage over 1 billion URLs in its blacklist database. Apart from using the already available blacklist database maintained internationally, the system has the capability to generate its very own blacklist database. The system is being tested with the actual dataset and 100% success rate is achieved for data rates up to 160 Mbps.

Figure 4.1 shows the deployment scenario of our real-time URL filtering system. Apart from traffic routing decisions, no architectural changes are required in the existing network. A user connects to the internet via internet service provider (ISP). The network traffic from user side is gathered at a network switch at ISP level and it passes through a packet level firewall before connecting it to internet. Our system adds an extra level of security to the existing network. By using the tapping port of network switch at ISP level the traffic is forwarded to our real-time URL filtering system. This is a passive mode operation and doesn't affect the actual flow of internet traffic at ISP level. At the real-time URL filtering system level, the packets are received by the initial packet processor (IPP). The job of IPP is to perform deep packet inspection of these packets and extracting the HTTP GET requests i.e. uniform resource locators (URLs). These URLs along with respective IP addresses are forwarded to Hash-based URL storage, which is based upon full URL matching (FUM) algorithm. The job of FUM is to store the URLs along with respective IP addressing in a memory efficient way without duplication. The list of unique URL entries along with IP addresses is forwarded to intelligent classifier (IC). Based on the lexical and host based features the URLs are classified into suspicious and benign URL entries. The list of suspicious URL entries is forwarded to layer 7 firewall and the associated IP addresses are forwarded to packet level firewall at the ISP level. Both tasks of updating of Layer 7 firewall and packet level firewall are performed in passive mode. Only layer 7 firewall is part of active mode. When a user makes a URL request it arrives at the network switch. The request is forwarded to packet level firewall at the ISP level. If the IP addresses associated with that URL entry is not blacklisted the request is forwarded to internet but if the IP entry exist in the blacklist database the request is forwarded to second level of filtering i.e. Layer 7 firewall. It is based on partial URL matching (PUM) algorithm. PUM provides layer 7 firewall to perform exact URL matching and prefix URL matching. If the URL entry exists in blacklist database of layer 7 firewall the URL request marked as suspicious URL and is rejected. If no match is found the URL request is marked as benign and forwarded to respective web server.

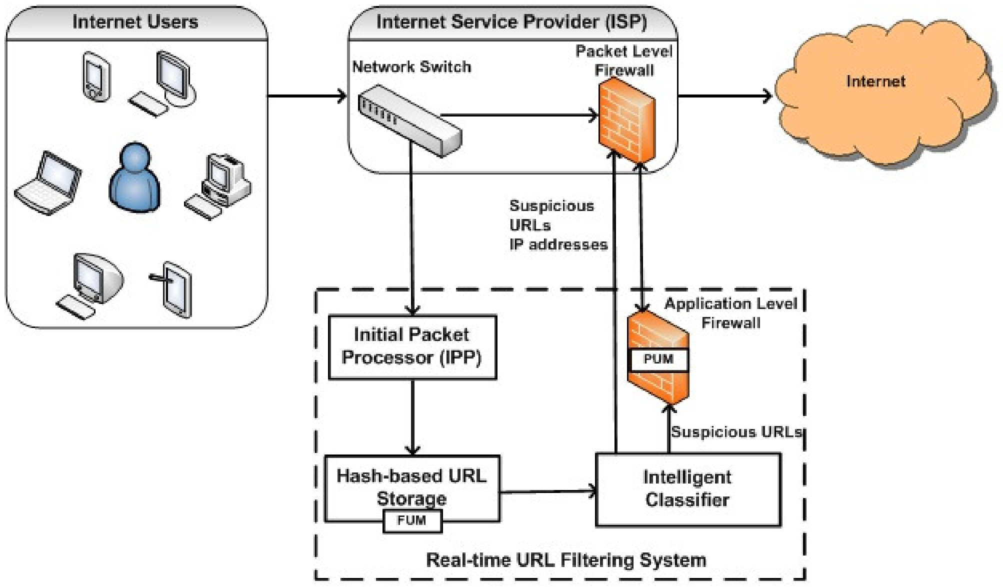

Internet Users
Internet Service Provider (ISP)
Network Switch
Packet Level Firewall
Internet
Suspicious URLs IP addresses
Application Level Firewall
PUM
Initial Packet Processor (IPP)
Suspicious URLs
Hash-based URL Storage
FUM
Intelligent Classifier
Real-time URL Filtering System

Figure 4.1: Deployment Scenario of Real-time URL filtering system

.

4.1 System Requirements

4.1.1 Programming Language

The system has been developed using Python which is a widely used high level programming language. Python version 2.7 is used in our implementation.

4.1.2 Operation System

The system has been developed and tested on Linux kernel version 3.2 in Ubuntu 12.04 LTS operating system.

4.1.3 Modules

Pcapy

It is python module for interfacing with the famous packet capturing library i.e. libpcap. It provides the capability to capture packets on the network. Pcapy version 0.10.5 is used for this implementation.

Deep packet (dpkt)

It is a python module for packet creation and parsing. It provides the capability to perform simple packet creation and parsing of packets on the network along with definition for TCP/IP stack. Dpkt version 1.7 is used for this implementation.

Scapy

It is a python module for packet manipulation. It provides with capability to forge as well as decode packets of various protocols. It can capture packets on the network and can also retransmit the forge packets on the wire. Scapy version 2.2.0 is used for this implementation.

Zlib

It is a python built in module for data compression. We have used the following functions from zlib data compression module,

- CRC32-This function computes CRC (cyclic redundancy check) checksum of input data.

- Adler32-This function computes the Adler-32 checksum of input data.

Pyhash

This python module provides functions for calculation of non-cryptographic hashes of input data. Pyhash 0.4.2 is used in our implementation. Following two functions are used from pyhash module,

- Murmur Hash-A fast 32-bit non-cryptographic hash function.

- FNV (Fowler-Noll-Vo) Hash-A fast 32-bit non-cryptographic hash function.

Chapter 5

Data Structure

This section describes our proposed data structure for full uniform resource locator (URL) matching (FUM) and partial uniform resource locater matching (PUM). Searching a URL entry in a big collection of URLs is similar to a search table or dictionary. The related operations of our proposed data structure are: 1) URL Insertion 2) URL Deletion and 3) URL Lookup. Following is the discussion of our proposed data structures and related operations,

5.1 Hash Table based Data Structure

FUM and PUM are search table type data structure. A search table data structure is like an associated array consisting of key value pairs. It can be implemented in many different ways: 1) Linked list, 2) Binary Search Trees and 3) Hash Tables. In this work, we have implemented search tables using Hash Tables.

5.1.1 Description

As discussed, search table in our work are implemented using hash tables. Hash table consists of slot and each slot has a unique index value. The key from key value pair is a unique identity. The key is mapped to the slot by using a hash function. Since key is a unique entry, the Hash table only store unique entries of URLs. Thus avoiding duplications, which is much desirable in our work.

A hash function is a basic component of hash table based implementations. Hash functions ensure the average lookup performance of O(1) but at cost of hash collisions. A hash collision in our case happens, when two distinct URL entries results in the same hash value. This is resolved using open addressing, which resolves hash collisions using probing. The probing can be performed in linear and random order. In case of linear probing the URL entries will be clustered in the hash table thus resulting in worst case scenario. To avoid clustering, the colliding URL entries are resolved using random probing. The selection of alternate slot is performed in pseudo random order. Hash collision resolution is performed in two ways: 1) Hash value comparison and 2) Key comparison. So if even two distinct URL entries results in same hash value, they can even than be identified in the hash table.

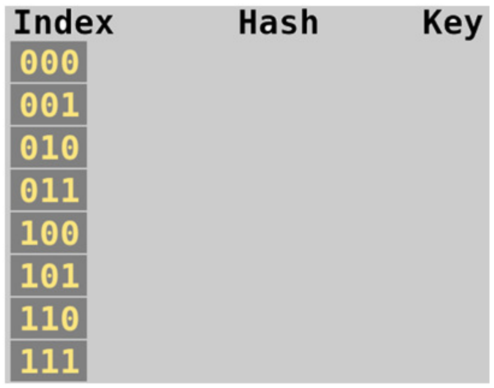
Our hash table implementation uses dynamic allocation of memory. When initialized only 8 slots are allotted and is resized depending upon the load factor. The load factor is computed as,

$$\text{Load factor} = \frac{\text{Number of URL entries in Hash Table}}{\text{Total size of Hash Table}} \tag{5.1}$$

This results in efficient usage of memory and making the data structure suitable for real-time applications. To keep the resize operation memory efficient, it follows the following conditions,

If URL entries in hash table $<$ 50K; Then hash table size $\times$ 4

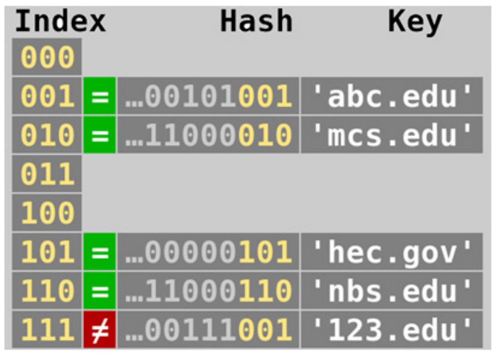
If URL entries in hash table $>$ 50K; Then hash table size $\times$ 2



Index	Hash	Key
000		
001		
010		
011		
100		
101		
110		
111		

Figure 5.1: View of initialized Hash Table

.



Index		Hash	Key
000			
001	=	…00101001	'abc.edu'
010	=	…11000010	'mcs.edu'
011			
100			
101	=	…00000101	'hec.gov'
110	=	…11000110	'nbs.edu'
111	≠	…00111001	'123.edu'

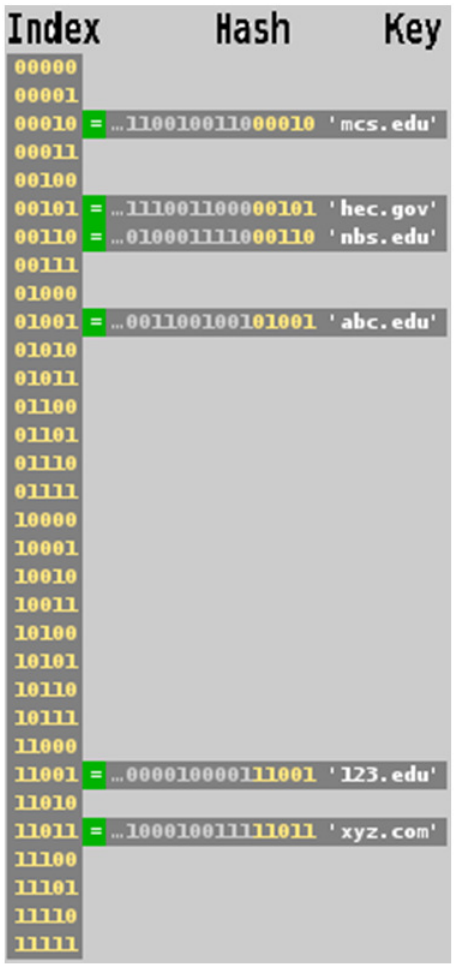
Figure 5.2: Hash Table at Threshold Level

.

Hash table is initialized with 8 slots as shown in Figure 5.1. It has empty key slot and index position are from 0 to 7. The hash table slots are provided on first come first serve basis. The key order is quite sensitive to hash table history. As seen in figure 5.2, the truncated bits of two keys ”abc.edu” and ”123.edu” are similar i.e. ”001”. The slot index as ”001” is allocated to ”abc.edu” as it comes first. The other key ”123.edu” with similar truncated bits is stored at an alternate location i.e. ”111”. As seen in Figure 5.2, the first one is marked as green, as it is stored at actual location and second one is marked as red, because it is stored at alternate location.

The hash table is resize when it is two third filled. When the load actor reaches 0.66 the resize operation takes place. Another URL entry in hash table will exceed the load factor from the threshold value. This is the time when a resize operation is required. As shown in Figure 5.3, a resize operation as per resize rule takes place. Since the Number of URL entries in hash table are less then 50K, the current size of hash table i.e. 8 is multiplied by 4, the new size of hash table is 32. To address 32 slots 5 LSB bits are required. Now the truncate operation will truncate the hash value of input URLs to 5 bits.

The collision rate after a resize operation drops dramatically. The improved performance is due to addition of extra bits for slot calculation. The extra bits add more randomness and reduce collisions. The hash table gradually gets more crowded as more URL entries are added but suddenly normalizes once a resize operation is performed.



Index	Hash	Key
00000		
00001		
00010	= …110010011000010	'mcs.edu'
00011		
00100		
00101	= …111001100000101	'hec.gov'
00110	= …010001111000110	'nbs.edu'
00111		
01000		
01001	= …001100100101001	'abc.edu'
01010		
01011		
01100		
01101		
01110		
01111		
10000		
10001		
10010		
10011		
10100		
10101		
10110		
10111		
11000		
11001	= …000010000111001	'123.edu'
11010		
11011	= …100010011111011	'xyz.com'
11100		
11101		
11110		
11111		

Figure 5.3: Hash Table after resize operation

.

5.2 Operations

Our hash table based data structure used for storage and lookup of URLs supports the following operations,

5.2.1 URL Storage

Hash table based storage is simply a list. Each entry in the list is combination of Hash, Key and Value. The Hash helps in find the slot for storage of URL entry. It is computed by taking the hash of key. A 32-bit hash function is used for this purpose. All 32-bit entries are not used for index calculation, only least significant bits (LSB) sufficient for addressing the current size of hash table. The URL storage operation is performed in following steps,

Step-1: Compute the hash value, using hash function of the URL entry. Step-2: The computed hash value is a big number; it is truncated based on the current size of hash table. Step-3: The truncated hash value is the slot address for storage of URL entry. If the entry is vacant the input URL entry is stored in it. In case the slot is already occupied, a recurrence function is used to find an alternate location. The recurrence function makes use of unused upper bits to add more randomness. The process is repeated until an empty slot is reached.

Table 5.1 shows the maximum storage capacity of our hash table based implementation. A total of over 2 billion entries can be stored in this table. The number of truncated bits for various hash table sizes is also shown in table 5.1. The load factor for resize operation can be fixed as per storage and efficiency needs. In our implementation the load factor is fixed as 0.666.

5.2.2 URL Lookup

URL lookup operation is actually a bit more complicated then hash, truncate and look. It is more like, keep looking until an empty slot is reached. The URL entry might be at the end of long list of collisions. When hash table

Table 5.1: Various Hash Table sizes.

Hash Table Size	**TruncatedBits**	**LoadFactor(0.666)**
8	3	5
32	5	21
128	7	85
512	9	341
2048	11	1365
8192	13	5461
32768	15	21843
131072	17	87373
524288	19	349490
2097152	21	1397962
8388608	23	5591846
33554432	25	22367384
134217728	27	89469537
536870912	29	357878150
2147483648	31	1431512600

experience collisions, lookups becomes expensive. All URL lookup operations are not the same. Some might finish at first slot and some loops over several slots. Thus the individual performance of hash table may vary but the average performance is excellent i.e. O (1).

5.2.3 URL Deletion

Our hash table based implementation supports URL deletion. The deleted URL entry cant just be left empty. It might be part of some long chain of collisions. In that case that URL entry will never be traced. To avoid such scenario, a slot of deleted entry is marked as dummy key. It will be cleared from the hash table once a resize of hash table is performed on reaching a threshold value set according to the load factor.

5.3 Full URL Matching (FUM) Data Structure

This section describes the FUM data structure. It is the underlying data structure for Hash-based URL storage (HUS). The job of HUS is to store the URL, IP pairs extracted by the initial packet processor (IPP). The Storage is done on secondary storage device. The URLs and IP addresses in the stored list of URL, IP pairs is processed by intelligent classifier (IC) for classification of URL in benign and suspicious URLs. It can perform classification based upon lexical features of URLs and host based features of respective IP addresses.

FUM data structure is similar to hash table based data structure discussed in previous section. Its supported algorithms are all the same. The requested URL entry is stored as key and respective IP address as the value of key, value pair. The storage of entries is done in URL, IP pairs on secondary storage device. The IO operations for secondary storage are expensive in terms of processing time. Hence making URL storage as bottleneck of our proposed real-time URL based filtering system. Since HUS performs in passive mode, it doesn't affect the inline filtering and blocking feature of our system.

The performance of HUS can be improved by using Machine Learning approach for classification of URLs in real-time based upon lexical and host-based features. In that case, the storage of URL on secondary storage will not be required and they can only be stored in temporary storage and processed. An alternate approach can be time based storage or URLs from temporary storage to secondary storage.

5.4 Partial URL Matching (PUM) Data Structure

This section describes the PUM data structure. It is the underlying data structure of Application Layer Firewall. It works in active mode. URLs associated with suspicious IP addresses are forwarded to Application level firewall for URL based filtering. It checks for existence of request URL in blacklist database. If the entry exists in blacklist database, the connection is refused otherwise request is forwarded to respective web server. The main challenges at this level are: 1) URL Storage 2) URL lookup.

Hash table based implementations are ideally suited for exact matching like in case of full URL matching. It is complicated for prefix matching. A hash of URL results in a big number, making it impossible to compare individual component of URL. To achieve that, each URL is broken into component URLs and is stored in hash table in a tree type structure. A tree type data structure is build on top of hash tables.

5.4.1 URL Storage

When it comes to hash tables, we are trading space for time. But we can't afford space in real-time application. This issue is resolved using tree type structure. Which is very much memory efficient. The URLs are broken in components and decomposed into tree structure. We don't have to bother about creating a balance binary tree, because we are storing the tree in search table form i.e. key, value pairs.

In case of FUM, the value slot is used for storage of respective IP address but here it is used to build a nested hash table. The key, value pairs are treated as pairs of parent and child node. The parent node will always be a unique URL; the repetition can be done at child nodes. This makes it suitable for storage of decomposed URL components. URL mainly consists of two parts: 1) domain and 2) paths (there can be zero or more paths). The domain will act as parent node. For each domain, we will be storing a tree structure. This key structure will be implemented as nested hash tables. While the domain act as parent node, the child will be the path associated with that domain and it will be the value part of (key: value) pair.

The URL storage database has a tree type structure for every domain. When a URL is received, it is decomposed in to URL components. The existence of domain is checked in database. If the domain entry is not listed, a new domain entry is made. The domain entry i.e. ”domain0” in Example 5.1 is not listed; it is added to the database. The domain entry act as a key value and path associated with that domain, which is ”path1” is added as a value entry against ”domain0”. For a URL request with same domain but different path address, the already existed domain tree is updated for the new path address. As shown in example 5.1, URL1 is already in database, only the path address of URL2 is not listed. Another value entry i.e. ”path2” is added to already listed ”domain0” key. It is also worth mentioning that a key is a unique entry, there can be zero, single or multiple values associated with each key. When a URL request having prefix components already existed in database they are not duplicated only the new components are added. As shown in Example 5.3, the first two URL components of URL1 and URL2 are listed in database. The third URL component is added by nesting approach. The already existed prefix path is initialized as a key, value pair and is populated. In case of URL1, the prefix component 1 is taken as a key and the component 3 of URL1 is added as the value entry. Same approach is followed for URL2. Now if a URL entry arrives and its domain is not listed. A new domain entry is added as a key entry and any path component associated with this domain is added as value entry. As shown in Example 5.4, the domain of URL3 is not listed, a new domain entry is added and path associated with that domain is also added as value entry.

Example 5.1

URL= domain0/path1

(key0: value) = (domain0: path1)

Example 5.2

URL1= domain0/path1, URL2=domain0/path2

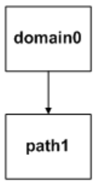


Figure 5.4: Initial URL Request

.

(key0:key00:value)=domain0:path1:path2

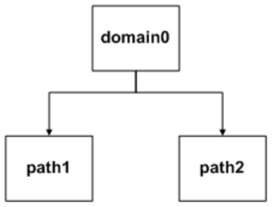


Figure 5.5: Same Domain with different single path addresses

.

Example 5.3

URL1= domain0/path1/path2, URL2=domain0/path2/path3

key0:key00,value,key01,value=domain0:path1:path2,path2,path3

Example 5.4

URL1= domain0/path1/path2, URL2=domain0/path2/path3, URL3=domain1/path1

key0:key00,value,key01,value, key1:value = domain0:path1:path2,path2,path3, domain1:path1

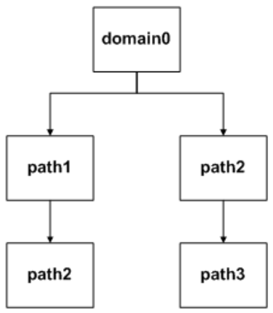


Figure 5.6: Same Domain with different multiple path addresses

.

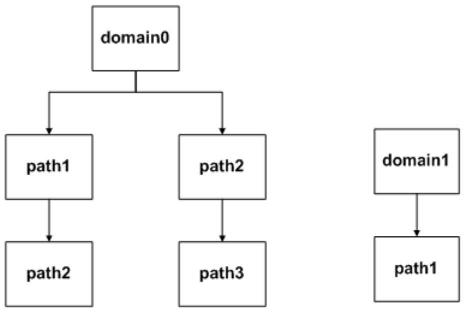


Figure 5.7: Different domains and multiple path addresses

.

5.4.2 URL Lookup

URL lookup operation in Application level firewall has prefix matching capability. This is achieved by using tree based approach for URL storage which is implemented in hash tables as nested hash tables. URLs can be searched to any number of levels. In prefix matching, if the prefix of request URL is listed in blacklist database the URL request is dropped.

When a URL is forwarded to application level firewall for lookup, it is first decomposed in to URL components. Depending on the number of URL components, the request is processed by first searching for the domain in the blacklist, the process continues until end of URL component is reached. The matching process is performed in sequential order for URL components. If a prefix match is found or all URL components are matched, the URL request is rejected. If no match is found URL request is treated as benign URL request.

URL request lookup are dependent on the entries in blacklist database. As shown in Example 5.5, the received URL request is first decomposed into URL components and the lookup operation is initialized in sequential order. For the blacklist database 1, when the URL request is received, it decomposes the URL in respective components. It first look for first component i.e. domain component in blacklist database. Since there is no entry for this domain the URL request is marked as benign URL. In case of blacklist database 2, it has only entry for the domain, which means that all the traffic for this domain is blocked. Since a match is found for domain part in blacklist database with value field empty, no further lookup is required so the URL request is blocked. In case of blacklist database 3, it has an entry for domain part and one component of URL path. When URL lookup operation is performed, the match is found for prefix of URL request, so the URL request is treated as suspicious URL. In case of blacklist database 4, a match is found for all components of URL and the URL request is treated as suspicious URL.

Example5.5

URL request: domain/path1/path2/path3

Blacklist database 1

key0:key00,value,key01,value, key1:value

=domain0.com:path1:path2,path2,path3

Blacklist database 2

key0:key00,value,key01,value, key1:value

=domain0.com:path1:path2,path2,path3, domain:None

Blacklist database 3

key0:key00,value,key01,value, key1:value

=domain0.com:path1:path2,path2,path3, domain:path1

Blacklist database 4

key0:key00,value,key01,value, key1:value

=domain0.com:path1:path2,path2,path3, domain:path1:path2:path3

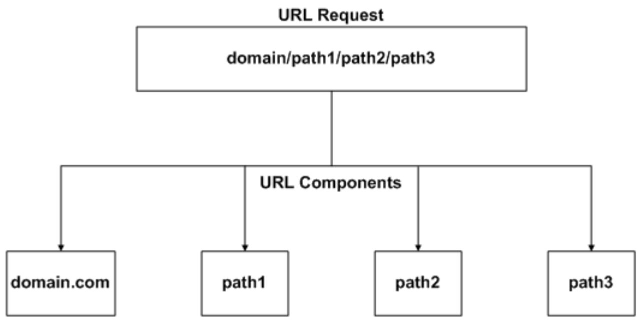


Figure 5.8: URL request decomposition

.

Chapter 6

Randomness Testing

A set of statistical tests are proposed by National Institute for Standards and Technology (NIST). The NIST statistical framework is based upon the null hypothesis H_0, i.e., the sequence under test is random. On the other hand, the alternate hypothesis H_a is that the sequence being tested is not random. Type-I error occurs for a random data if the alternative hypothesis is true, i.e., for a random sequence, H_0 is rejected, and H_a is accepted [24].

A statistical test, which is performed in order to assess the randomness of a hash function, results in P-values. Each P-value is a probability that the sequence under test is more random than the sequence which would have been generated by a prefect random number generator. A P-value of 1 means that the sequence under test is completely random; however, a P-value of 0 means that the sequence under test is not random at all. A significance level α is defined as the probability of concluding that a random sequence under observation is not random, i.e., the probability of Type-I error. For a fixed significance level, a certain percentage of P-values are expected to indicate failure. In our case, we have fixed the significance level to 0.01, which means that there is about 1% chance that the sequences under test might fail. A sequence passes a statistical test whenever the P-value $\geq \alpha$, and fails otherwise.

6.1 Empirical Results

The NIST suite is developed to test the randomness of a binary sequence produced by a random number generator, which in our case is the hash function. The NIST test suite focuses on various types of non-random behaviors that can exist in a binary sequence. The required and the used length of an input bit sequence for each test is given in Table 6.1. In our case, each of the five tests has been executed on one thousand distinct m-bit sequences, which have been individually obtained from fifteen ASCII files of the size 10^6 bytes containing hashed output.

Table 6.1: Lengths of the bit sequences which have been employed in various statistical tests.

Test No.	**Name of Test**	**Required length of input bit sequences (n) (bits)**	**Used length of input bit sequences (bits)**	**Recommended sample size (m) (bits)**	**Used Sample size**
1	Frequency Test	n≥100	100	m≥100	1000
2	Runs Test	n≥100	100	m≥100	1000
3	Cusums Test	n≥100	100	m≥100	1000
4	Longest Run Test	n≥128/6272	128	m≥100	1000
5	Spectral Test	n≥1000	1000	m≥100	1000

The number of samples produced by a specific hash function is termed as the sample size. It is required to use a sample size as per NIST recommendations for the evaluation based upon P-value proportions. In our case, as shown in Table 6.1, we have used the sample size of one thousand bits, i.e., m = 1000. If for a given test, P-value is greater than or equal to 0.01, the test will be successful and sequence under test should be considered as random.

In order to interpret the outcome of a statistical test, following two approaches as per NIST standard have been employed [24]:

1. Proportion of sequences passing a test

   Given the statistical results of a test performed, compute the proportion of sequences that have passed. The resultant P-value of a specific test is compared with a statistical threshold value (T-value), defined as:

   $$\text{T-value} = (1-\alpha) - 3\sqrt{\frac{\alpha(1-\alpha)}{m}} \tag{6.1}$$

   If P-value is greater than or equal to the threshold value, test is considered to be successful. For the significance level $\alpha = 0.01$, and m = 1000, the obtained T-value is 0.9805607. If at least 980 sequences out of 1000 sequences pass the test, the test is considered as successful. However, if the the test results are below the threshold levels, relative comparison among the output of hash functions are performed.

2. Uniform distribution of P-values

   The second approach proposed by NIST provides further in-depth analysis of the performed statistical tests. Uniformity of the hash functions is ensured by a uniform distribution of P-values. This can be visually illustrated using a histogram, in which the interval between zero and one is divided into ten sub-intervals, and the P-values that lie within each sub-interval are counted and displayed.

6.1.1 The Frequency (Monobit) Test

This test focuses on the frequencies of zeros and ones in the entire binary sequence [24]. The total number of zeros and ones in a sequence must be equally distributed for a truly random sequence is assessed as for a truly random sequence the number of zeros and ones are approximately the same. The P-value in this test is calculated based on the cumulative value Sn, which is the sum of the bits in input bit sequence. It can have both positive and negative values. A large positive value of Sn indicates more number of ones in the input bit sequence, and a large negative value of Sn indicates more number of zeros.

In Figure 6.1, we can see that the proportion of sequences passing frequency test is almost below the threshold value T-value = 0.9805607, which is calculated using equation 1 based upon the level of significance. This shows that frequencies of zeros and ones in entire bit sequence under test are not uniform. The required percentage of minimum sequences passing frequency test is 98 %, however, the percentage of sequences passing this test has been less than the set value.

If we look at the relative comparison of these hash functions, then, apart from Adler hash function, rest of the hash functions have similar statistical properties. The percentages of sequences passing the test for these hash

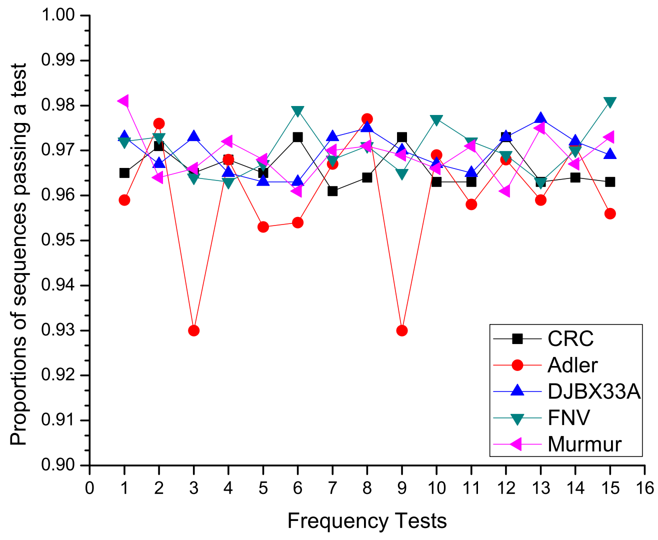


Figure 6.1: Proportion of sequences passing Frequency Test

.

functions (CRC, DJBX33A, FNV, and Murmur) lie between 96 % and 98 %, as shown in Figure 6.1.

Figure 6.2 shows the histogram of P-values for Frequency Test of all the hash functions. It is evident from the histogram that distribution between zero and one is non-uniform. In sub-interval 8, we have no frequency count for all hash functions. The standard deviation of all the hash functions for frequency Test is on the higher side (i.e. ranging between 43 and 46). DJBX33A hash function has least standard deviation i.e. St. Dev = 43.271 and Adler hash function the most standard deviation i.e. St. Dev = 45.811.

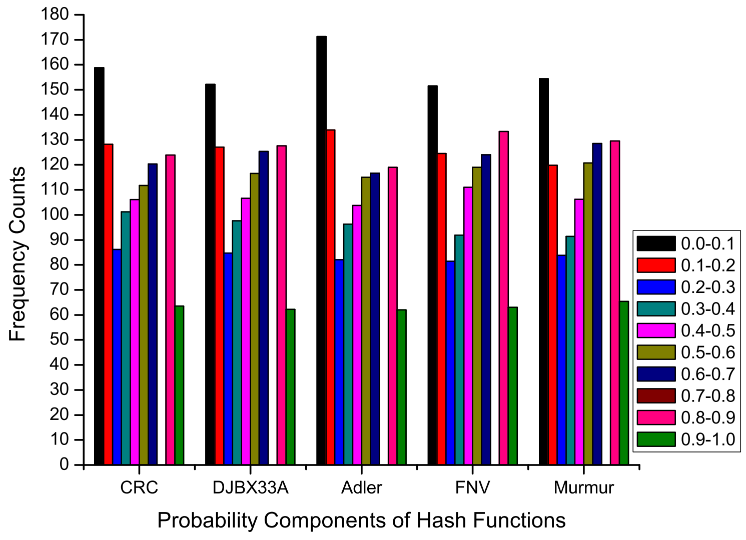


Figure 6.2: Histogram of P-values for Frequency Test

.

6.1.2 The Runs Test

This test looks for the runs of zeros and ones of various lengths across the random sequence [24]. The rate of oscillation between and ones and zeros is monitored to determine the nature of randomness. The reference distribution for this test statistic is "chi-squared" χ^2 distribution.

An input bit-stream with a high degree of oscillations results in a higher P-value, and vice versa. It is observed in Figure 6.3 that the Adler hash function has a slower oscillation rate, and hence, in some tests the passing proportion lies below the threshold limit. The rest of the hash functions (CRC, DJBX33A, FNV, and Murmur) have a fast variation between zeros and ones, and hence, a higher P-value. The percentage of sequences passing Runs test for these hash function lie above the threshold value.

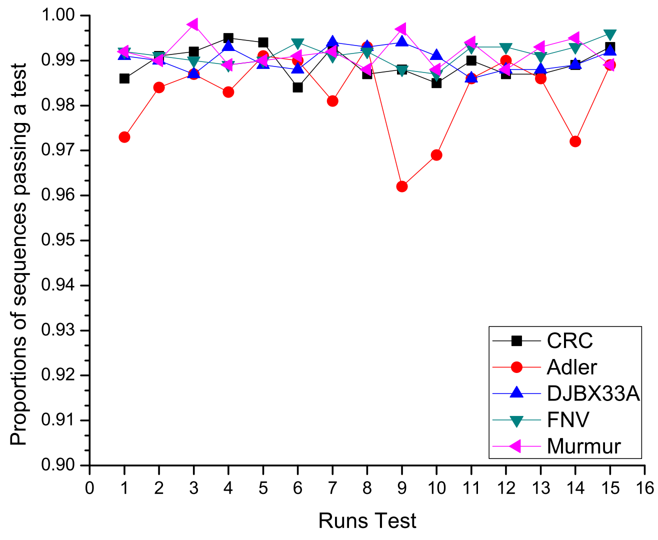


Figure 6.3: Proportions of sequences passing Runs Test

.

Figure 6.4 shows the histogram of P-values for Runs Test of all the hash functions. It is evident from the histogram that distribution between zero and one is quite uniform. The standard deviation of all the hash functions for Runs Test ranges between 8 and 13. CRC hash function has least standard deviation i.e. St. Dev = 8.170 and Adler hash function the most standard deviation i.e. St. Dev = 13.428.

6.1.3 Test for the Longest-Run-of-Ones in a block

In this test, the input bit sequence is divided into M-bit blocks, and for each block the test is executed [24]. The purpose of this test is to determine whether the length of the longest run of ones, within the test sequence, is consistent with the length of the longest run of ones which is expected of a

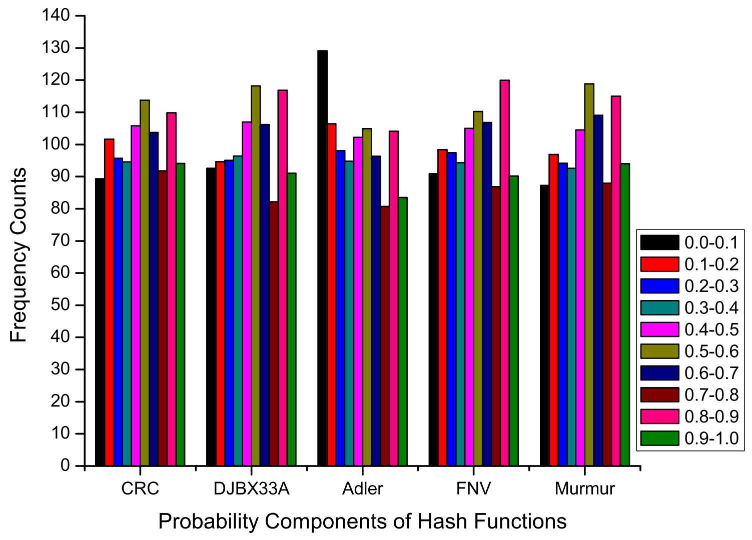


Figure 6.4: Histogram of P-values for Runs Test

.

random sequence.

The reference distribution in this test is chi-squared distribution. The P-value is calculated based upon the value of χ^2 (obs), which is the measure of how well the observed longest runs length, within M-bit blocks, matches the expected longest length. The larger value of χ^2 (obs) indicates that the tested sequence has clusters of ones, thus resulting in smaller P-value.

In Figure 6.5, we can clearly see that Adler hash function has lesser proportion of sequences passing this test. This indicates that output of Adler hash function has larger clusters of ones. The performance of other hash functions (CRC, DJBX33A, FNV, and Murmur) is observed to be quite similar.

Figure 6.6 shows the histogram of P-values for Longest-Run-of-Ones in

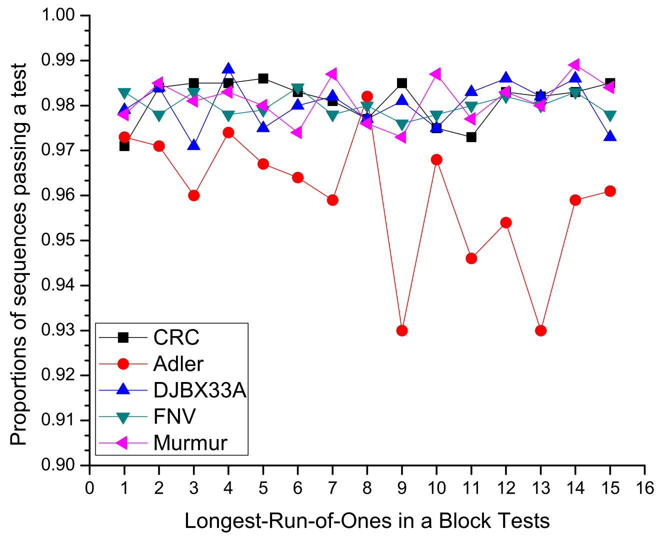


Figure 6.5: Proportions of sequences passing Longest Run of Ones in a block Test

.

a block test of all the hash functions. It is evident from the histogram that distribution between zero and one is not quite uniform. The standard deviation of all the hash functions for this test ranges between 24.406 and 53.935. FNV hash function has least standard deviation i.e. St. Dev = 24.406 and Adler hash function the most standard deviation i.e. St. Dev = 53.935.

6.1.4 The Discrete Fourier Transform (Spectral) Test

The purpose of this test is to detect the periodic features in a test sequence that would indicate a deviation from the assumption of randomness [24].

The reference distribution for this test is normal distribution. The P-value

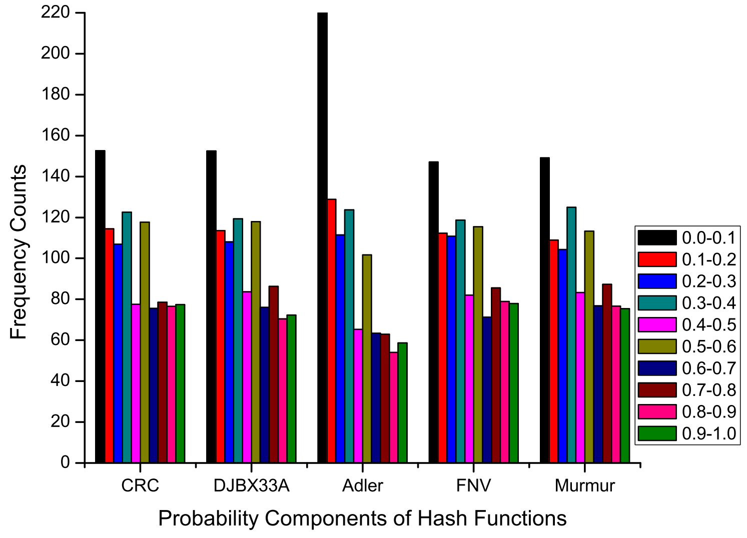


Figure 6.6: Histogram of P-values for Longest-Run-of-Ones in a block Test

.

in this test is calculated based upon the normalized difference d between the observed and the expected number of frequency components that are beyond 95 % threshold [1].

A smaller value of normalized distribution d is desirable, which means that there is acceptable number of peaks above the threshold value. It has been observed that the proportion of sequences generated by Adler hash function passing this test is lower as compared to other has function, figure 6.7. This is caused by a large number of peaks below the threshold, resulting in a higher normalized distribution d and a smaller P-value. The rest of the hash functions under observation (CRC, DJBX33A, FNV and Murmur) have similar periodic features which can be seen in Figure 6.7.

Figure 6.8 shows the histogram of P-values for Discrete Fourier Trans-

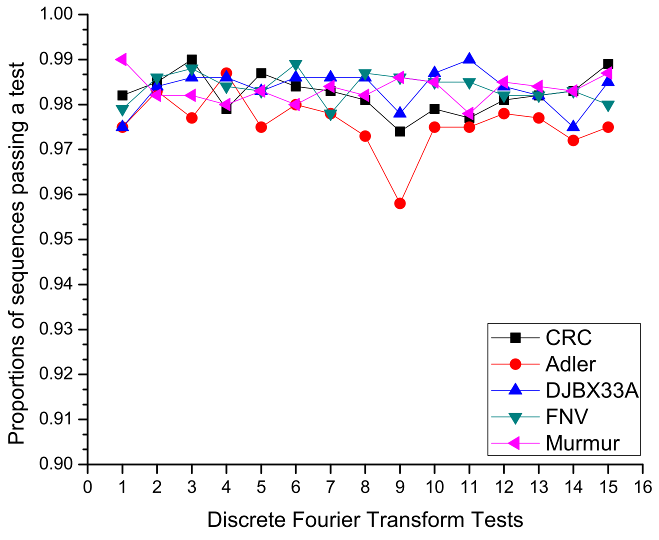


Figure 6.7: Proportion of sequences passing Discrete Fourier Transform Test

.

form Test of all the hash functions. It is evident from the histogram that distribution between zero and one is non-uniform. In sub-interval 5, 7 and 9, we have no frequency count for all hash functions. The standard deviation of all the hash functions for frequency Test is on the higher side (i.e. ranging between 76.928 and 79.133). Adler hash function has least standard deviation i.e. St. Dev = 76.928 and Murmur hash function the most standard deviation i.e. St. Dev = 79.133.

6.1.5 The Cumulative Sums (Cusums) Test

The focus of this test is to look for the large number of ones and zeros occurring at the start and the end of an input binary sequence. Additionally, it identifies the intermixing of zeros and ones across the entire bit sequence

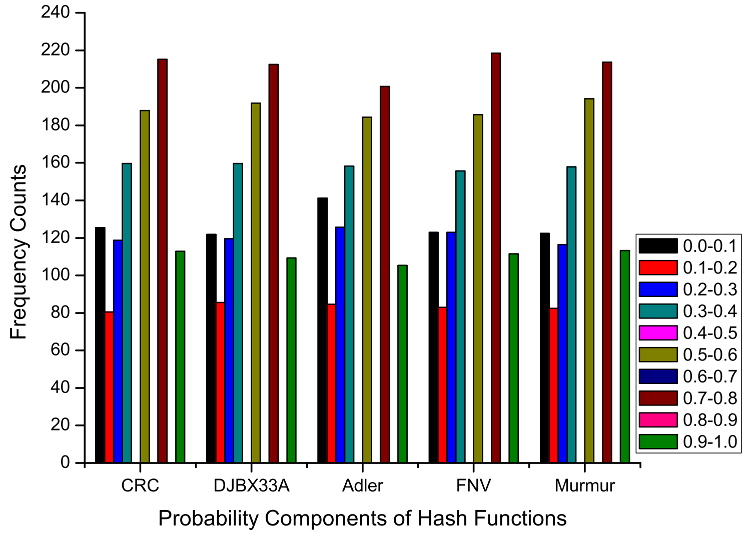


Figure 6.8: Histogram of P-values for Discrete Fourier Transform Test

.

[24].

The reference distribution for this test is normal distribution. This test is performed in forward (Mode 0) and backward (Mode 1) directions for the input bit sequence, in order to determine the number of ones and zeros at the start and the end of the input bit sequence, respectively. The smaller values of ones and zeros are desirable, which is an indication that the ones and zeros are distributed evenly.

It has been observed in Figure 6.9 that the proportion of sequences generated by Adler hash function have not been able to achieve the required threshold of passing this test as compared to other hash functions, figure 5. Additionally, the rest of the hash functions (CRC, DJBX33A, FNV and Murmur) have a similar test statistics.

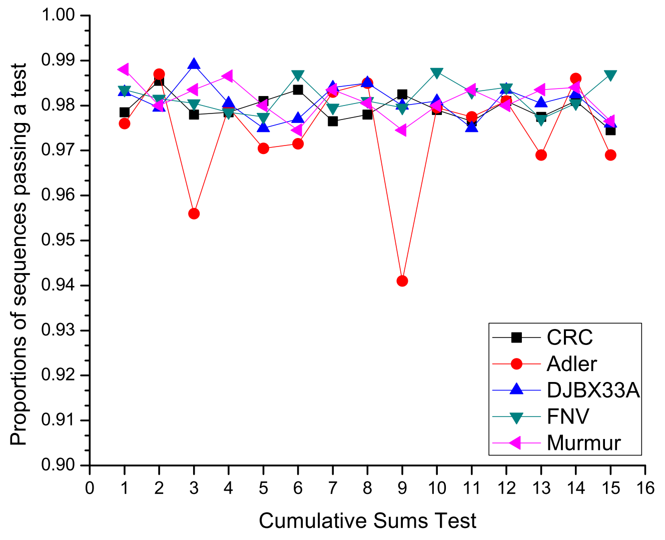


Figure 6.9: Proportions of sequences passing Cumulative Sums Test

.

Figure 6.10 shows the histogram of P-values for Cumulative Sums Test of all the hash functions. It is evident from the histogram that distribution between zero and one is not quite uniform. The standard deviation of all the hash functions for Cumulative Sums Test ranges between 30.069 and 35.751. DJBX33A hash function has least standard deviation i.e. St. Dev = 30.069 and Adler hash function the most standard deviation i.e. St. Dev = 35.751.

6.2 Analysis of Empirical Results

The NIST test suite contains fifteen tests; however, only five tests have been employed in our case in order to determine the randomness of non-cryptographic hash functions. Each test calculates a P-value, and if the

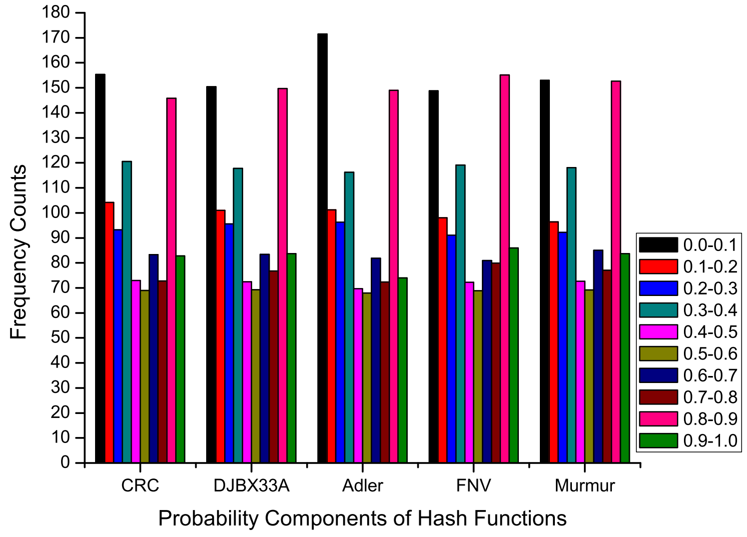


Figure 6.10: Histogram of P-values for Cumulative Sums Test

.

calculated value is greater than or equal to 0.01, the test is considered successful. This means that the input bit sequence is random.

Due to their lack of statistical properties, the rest of the tests in NIST test suite are unable to give us the required evaluation.

6.2.1 Proportion of passing a test based on P-value

Success rate of the tests are based upon the empirical results. For each hash function the percentage of passing a specific test is calculated as:

$$\text{Success Rate} = \frac{\text{Number of successful tests}}{\text{Total number of tests}} \times 100 \tag{6.2}$$

Table 6.2 shows the success rate of the tests performed for all hash functions. Adler hash function is found to be a non-uniformly distributed hash function, whereas for the rest of hash functions (CRC, DJBX33A, FNV and Murmur) the outcomes are quite close and inconclusive. Further in depth analysis is required to decide a hash function with better randomizing features.

Table 6.2: Success rate of hash functions.

Hash Functions	**Frequency Test**	**Cusums Test**	**Runs Test**	**Longest Run Test**	**Spectral Test**	**Mean Success Rate**
CRC	0.00%	40.00%	100.00%	73.33%	73.33%	57.33%
Adler	0.00%	33.33%	73.33%	0.06%	13.33%	24.01%
DJBX33A	0.00%	60.00%	100.00%	53.33%	80.00%	58.67%
FNV	0.06%	66.66%	100.00%	33.33%	80.00%	56.01%
Murmur	0.06%	53.33%	100.00%	53.33%	80.00%	57.34%

6.2.2 Uniform distribution of P-values

The standard deviation of frequency counts can be used as a metric to analyze the distribution of P-values between zero and one. The lesser the standard deviation, the more uniformly distributed is the output of a hash function.

Figure 6.11 shows the standard deviation of frequency counts for all the hash functions. It is evident that outputs of Adler hash function are less uniformly distributed and the distribution of P-values between zero and one for rest of hash functions (CRC, DJBX33A, FNV and Murmur) are quite similar.

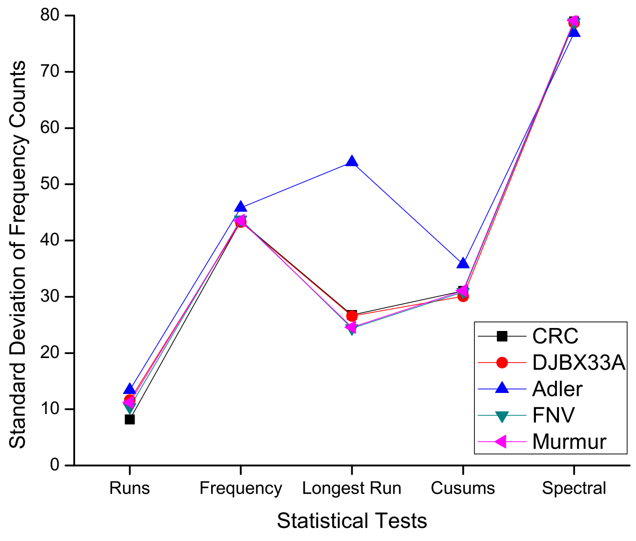


Figure 6.11: Standard Deviations

.

6.3 Verification of Statistical Results

The results of statistical tests are verified by the implementation of hash table using respective non-cryptographic hash functions. The hash table is populated with URL entries. The number of probes for each entry in hash table is computed. Various tests are performed for computing the average number of probes by varying the hash table size and load factor. The load factor is computed as:

$$\text{Load factor} = \frac{\text{Number of URL entries in Hash Table}}{\text{Total size of Hash Table}} \tag{6.3}$$

To avoid wastage of valuable storage, the hash table is initialized with 8 slots. When the load factor reaches 0.666, it resizes. To keep the resize operation memory efficient, it follows the following conditions:

If URL entries in hash table $<$ 50K; Then hash table size $\times$ 4

If URL entries in hash table $>$ 50K; Then hash table size $\times$ 2

Table 6.3: Implemented Hash Table Specifications.

HT Size	**TruncatedBits**	**LoadFactor(0.2)**	**LoadFactor(0.4)**	**LoadFactor(0.6)**	**LoadFactor(0.666)**
8	3	2	3	5	5
32	5	6	13	19	21
128	7	26	51	77	85
512	9	102	205	307	341
2048	11	410	819	1229	1365
8192	13	1638	3277	4915	5461
32768	15	6554	13107	19661	21843
131072	17	26214	52429	78643	87373
524288	19	104858	209715	314573	349490
2097152	21	419430	838861	1258291	1397962
8388608	23	1677722	3355443	5033165	5591846
33554432	25	6710886	13421773	20132659	22367384
134217728	27	26843546	53687091	80530637	89469537
536870912	29	107374182	214748365	322122547	357878150
2147483648	31	429496730	858993459	1288490189	1431512600

URL storage in our hash table is performed by initially computing the hash of a URL entry, the computed hash is then truncated to get the index of a slot depending upon the current size of hash table. If the slot is vacant the entry is stored in it, in case it is occupied, a backup recursive algorithm is executed by utilizing the unused upper bits to find a secondary storage location. The process repeats until a vacant slot is found. If the first slot calculated is vacant the number of probes required to retrieve it will be 1, whereas, the number of probes will vary depending upon the number of times the recursive function is called.

The first column in Table 6.3 shows the size of hash table which is a multiple of 2. The truncated bits of a 32-bits hash entry are used for indexing purpose. The rest of the table shows the number of URL entries in hash table depending upon the load factor.

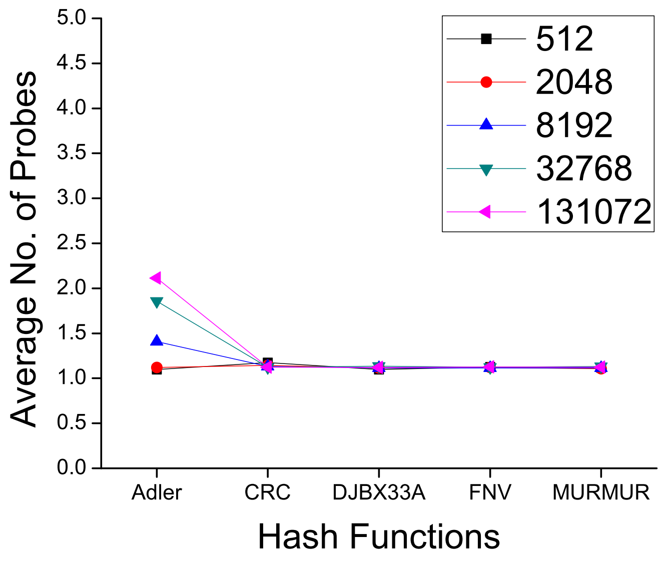


Figure 6.12: Average Number of Probes for 0.20 Load Factor

The performance of hash Table is dependent upon the load factor. The greater the load factor the greater will be number of probes for each URL entry. In Figure 6.12, where the load factor is 0.20, the average number of probes is ~1.15 for hash table size up to 2048 entries. As the size of hash table increase, the performance of Adler Hash function falls and its average number of probes reaches 2.112. The performance of the rest of hash functions (CRC, FNV, Murmur and DJBX33A) remains unchanged.

Figure 6.13 shows the average number of probes taken by each hash func-

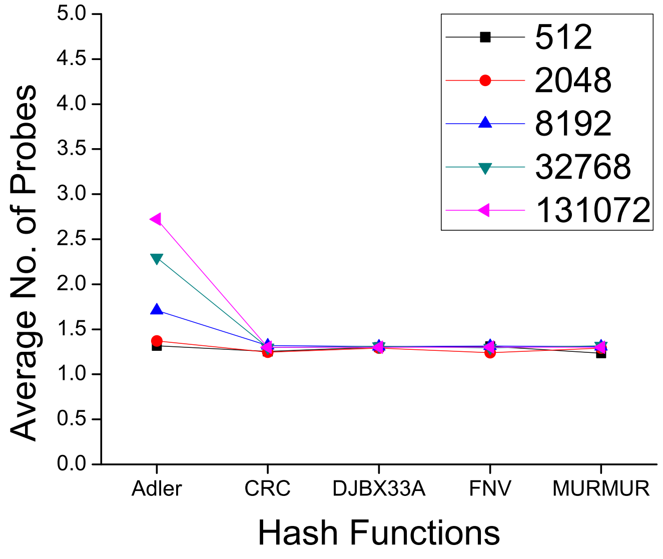


Figure 6.13: Average Number of Probes for 0.40 Load Factor

tion at load factor of 0.40. It can be seen that average number of probes for all the hash functions are around 1.25 up to hash table size of 2048 entries. The performance for all the hash functions remains the same except Adler Hash Function as the average number of probes rises to 2.75 with increasing the hash table size.

Figure 6.14 shows the average number of probes taken by each hash function at the load factor of 0.60. It is observed that the variations in average number of probes for Adler hash function are quite large. The average number of probes for the rest of hash functions (CRC, FNV, Murmur and DJBX33A) are quite similar. The average number of probes for hash table size 32768 is 1.5, but it gets better as the table resizes itself as the number of truncated bits are raised from 15 to 17 which adds more randomness and the average number of probes falls to 1.0.

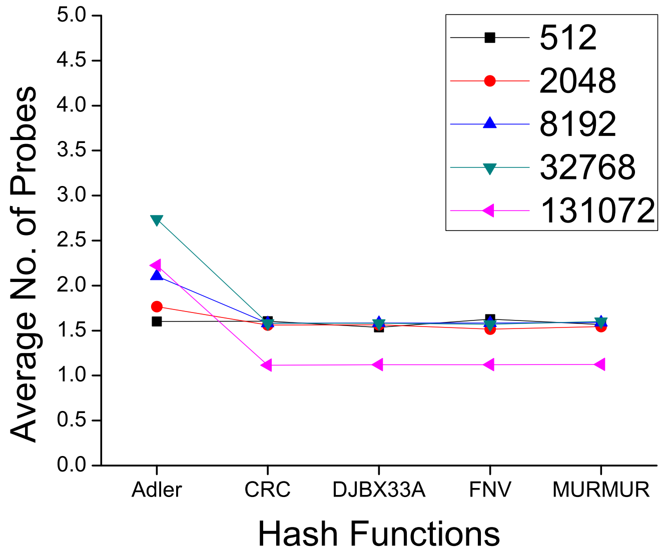


Figure 6.14: Average Number of Probes for 0.60 Load Factor

Chapter 7

Results and Discussion

The proposed real-time URL based filtering system in this thesis is capable of maintaining a localize blacklist database and provides finer level of filtering using higher layer information. Following are the four core components of this system,

- Initial packet processor (IPP)
- Hash based URL storage (HUS)
- PIntelligent Classifier (IC)
- Application level firewall

The performance of these core components is evaluated using Dataset 2. Each captured file is replayed using TCPREPLAY application, which transmits the already captured packets at the desired data rate on specified Ethernet card. The real-time URL based filtering system application is run on the same system and the application captures the replayed traffic from the Ethernet card. Dataset 2 comprises of 24 hours of data captured at School of

Electronic Engineering and Computer Science (SEECS), NUST, Islamabad Data Center. The results are shown in four groups, each of six hours. Following are the results and discussion of three of the core components, which are explored in this work,

7.1 Initial Packet Processor (IPP)

The purpose of IPP is to capture real-time network traffic and perform deep packet inspection to extract higher layer information i.e. URL from each packet. The performance of IPP is evaluated in terms of packet drop ratio.

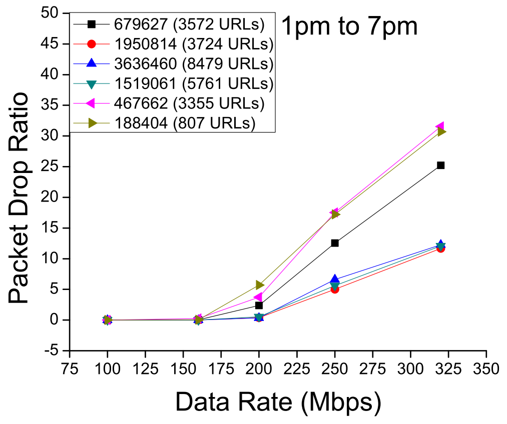


Figure 7.1: Packet Drop Ratio (1pm to 7pm)

.

Figure 7.1-7.4 shows the packets drop ratio in terms of variation in data rate. It can be seen that packets drop ratio is zero for data rate of 160 Mbps and less than 5% when data rate reaches 200 Mbps. The packet drop ratio increases with further increase in data rate and is less than 30% for data rate of 325 Mbps. The variation in packet drop ratio for data rates

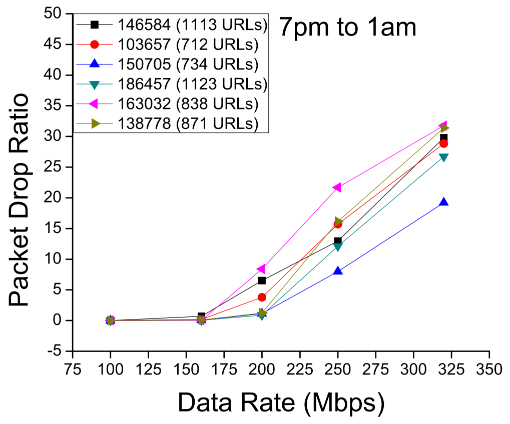


Figure 7.2: Packet Drop Ratio (7pm to 1am)

.

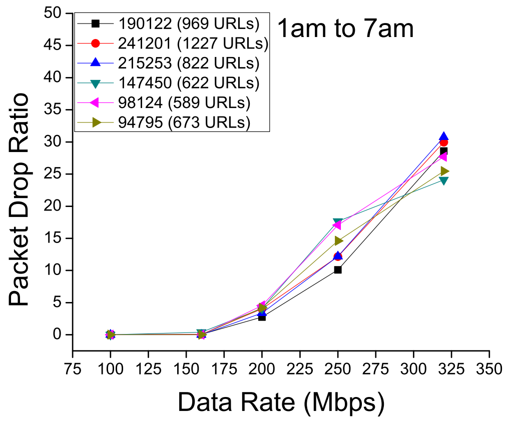


Figure 7.3: Packet Drop Ratio (1am to 7am)

.

$160 Mbps is due to variable sizes and types of captured packets.$

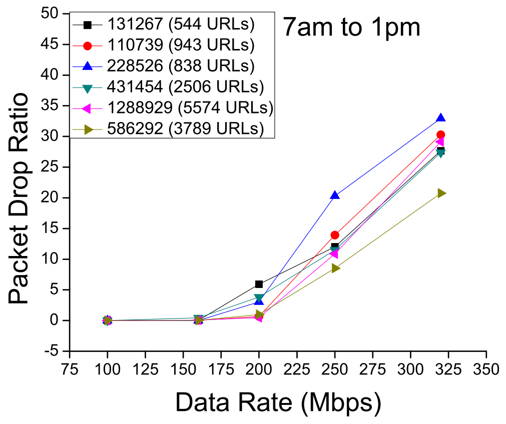


Figure 7.4: Packet Drop Ratio (7am to 1pm)

.

7.2 Hash based URL storage (HUS)

The purpose of HUS is to store URLs along with their respective IP address. It is based upon full URL matching (FUM) algorithm as discussed in Data Structure section. It consists of associated database of requested URL address and respective IP address. The database is stored on secondary storage as (URL, IP) pairs, such that to avoid duplication.

The performance of HUS is evaluated in terms of URL storage rate. The URL, IP pairs are stored without duplication on secondary storage. Figure 7.5-7.8 shows the URL storage rate of our real-time URL based filtering system application. It can be seen that there is an initial loss of 15 to 20% in URL storage rate due to I/O operation with secondary storage device. There is further drop of 10% for data rate up to 160 Mbps. URL storage rate reduces with the increase of data rate. A drop of around 50 to 60% in URL storage rate can be seen in Figure 7.5-7.8.

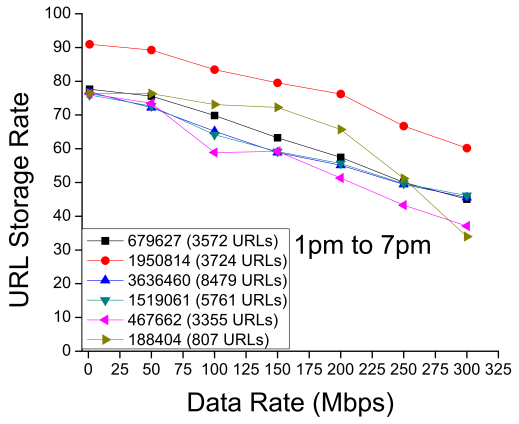


Figure 7.5: URL Storage Rate (1pm to 7pm)

.

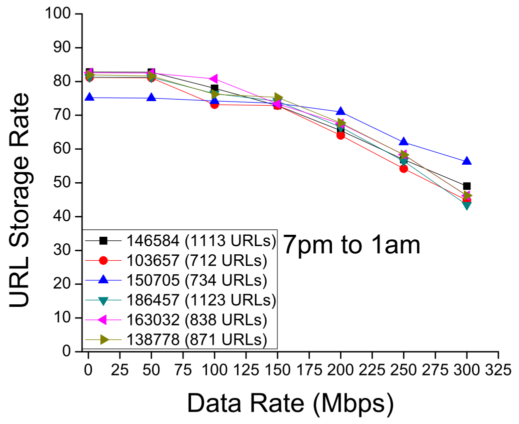


Figure 7.6: URL Storage Rate (7pm to 1am)

.

7.3 Application level firewall

The purpose of application level firewall is to filter URLs based upon a black-list database, which is maintained by intelligent classifier (IC). The updating

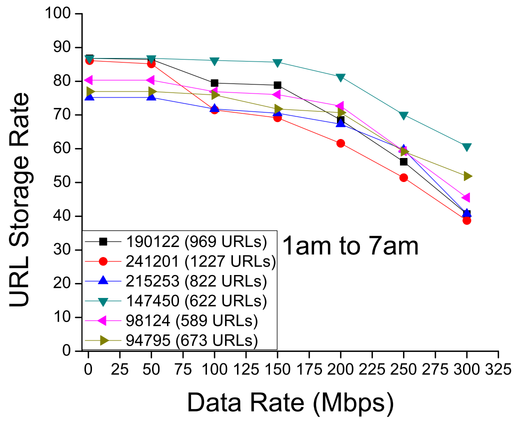


Figure 7.7: URL Storage Rate (1am to 7am)

.

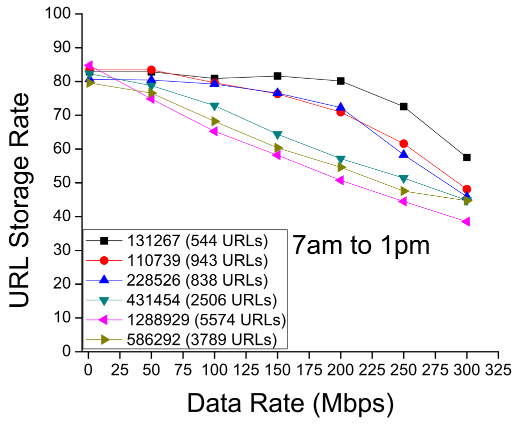


Figure 7.8: URL Storage Rate (7am to 1pm)

.

of blacklist database is an offline activity, whereas the requested URLs are compared to blacklist database in active mode. The performance of application level firewall is evaluated in terms of URL lookup rate and false positive rate.

7.3.1 URL Lookup Rate

The URL lookup operation of our real-time URL based filtering system is based upon partial URL matching (PUM) data structure. The suspicious URL entries are decomposed in URL components, which are stored in blacklist database of Application level firewall. In a URL lookup operation, a request URL entry is compared to blacklist database. If entry exists in blacklist database, the URL request is dropped otherwise forwarded to respective web server.

As seen in Figure 7.9-7.12, the URL lookup rate is 100% for data rate up to 160 Mbps. The URL lookup rate drops to 95% at data rate of 200 Mbps. The 40 Mbps increase in data rate caused 5 % loss in URL lookup rate. There is reduction of 20 to 30% in URL lookup rate as data rates reach 325 Mbps.

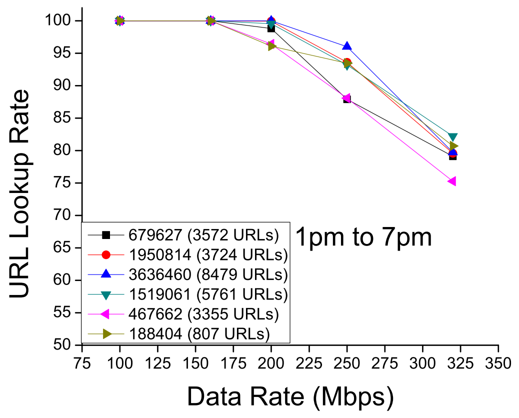


Figure 7.9: URL Lookup Rate (1pm to 7pm)

.

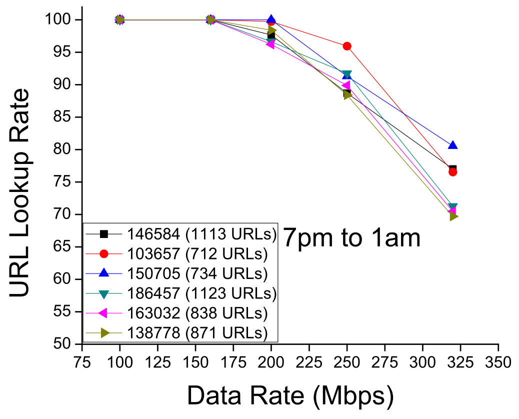


Figure 7.10: URL Lookup Rate (7pm to 1am)

.

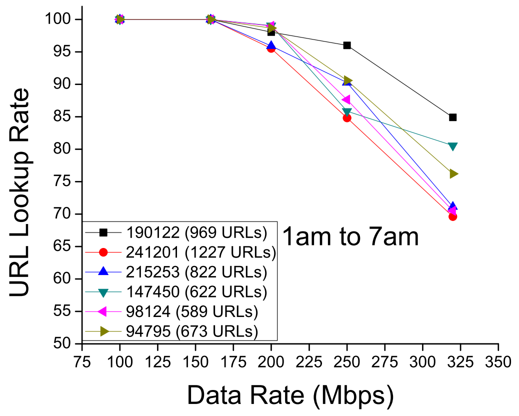


Figure 7.11: URL Lookup Rate (1am to 7am)

.

7.3.2 False Positive Rate

A false positive occurs, when a suspicious URL entry is not identified correctly and processed as a benign URL entry. Normally hash based data structure suffers from false positives due to hash collisions. This is catered in

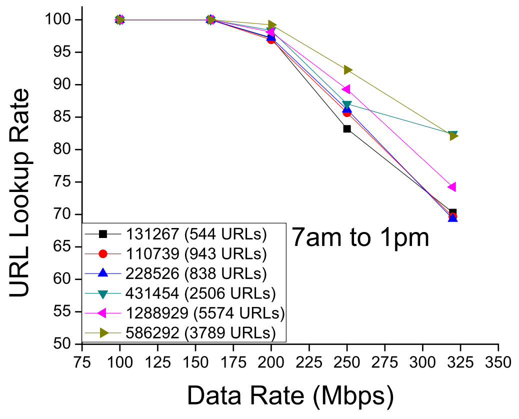


Figure 7.12: URL Lookup Rate (7am to 1pm)

.

our implementation by the use of two distinct non-cryptographic hash functions, one is used for slot calculation and other is used for compressing the URL entry to fix length of 32-bits.

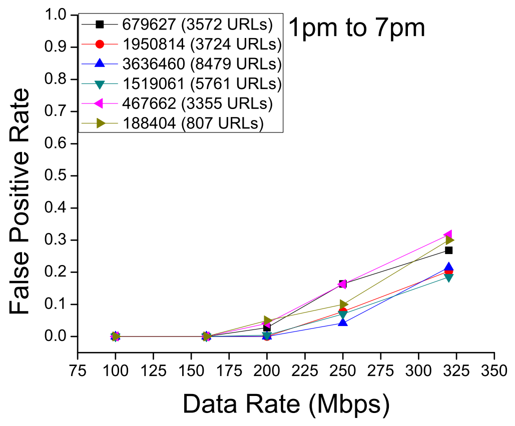


Figure 7.13: False Positive Rate (1pm to 7pm)

.

Figure 7.13-7.16 shows zero false positive at data rate up to 160 Mbps.

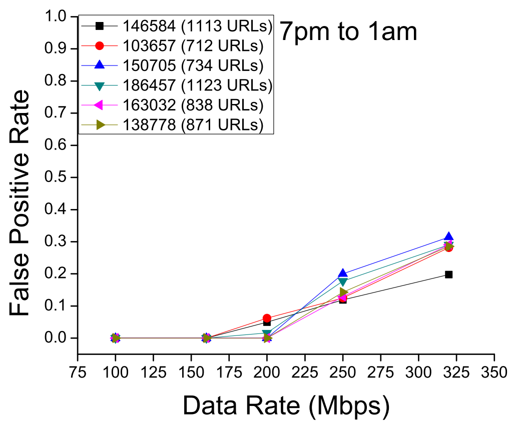


Figure 7.14: False Positive Rate (7pm to 1am)

.

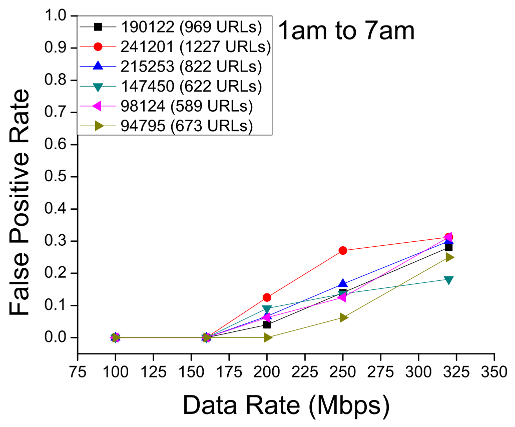


Figure 7.15: False Positive Rate (1am to 7am)

.

There is increase of 0.05 in false positive rate when data rate further exceed and reaches 200 Mbps. The false positive rate ranges from 0.18 to 0.30 for data rate up to 325 Mbps. These false positives are not due to hash collisions but due to the packet drop ratio, which start increasing when data

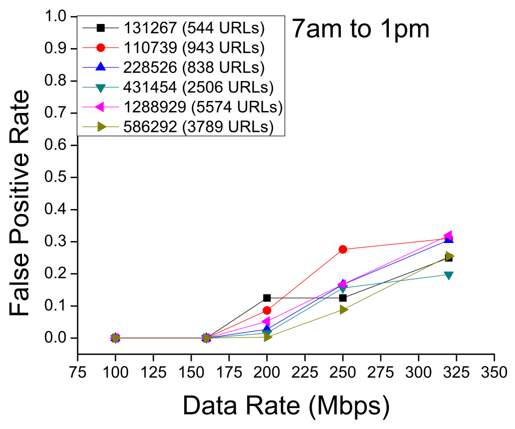


Figure 7.16: False Positive Rate (7am to 1pm)

.

rate exceeded 160 Mbps and kept on rising with increasing data rate.

Chapter 8

Conclusion and Future work

8.1 Conclusion

Randomness testing of non-cryptographic hash function for real-time URL based internet filtering applications is presented. The thesis focuses on the use of hash table for real-time internet filtering application. Hash tables suffer from false positives because of hash collisions which are associated with hash functions. Since the performance of hash functions is data dependent so there is need to explore behavior of hash functions for URL based string inputs. To the best of our knowledge there is no study to explore non-cryptographic hash functions in terms of randomness for URL based string inputs. Statistical Analysis is performed on the sequences generated using five widely used non-cryptographic hash functions for hash table implementations: 1) CRC, 2) Adler, 3) FNV, 4) DJBX33A, and 5) Murmur. The comparative analysis of tested non-cryptographic hash functions shows that the Adler hash function is not suitable for hash table implementation, whereas, the rest of non-cryptographic hash functions exhibit similar and better randomizing features which make them an attractive choice for hash table implementation. The results of these statistical studies have been verified by the implementation of hash table using these non-cryptographic hash functions. Based on our working with non-cryptographic hash functions for hash table implementation, we proposed a real-time URL based filtering system. Trace based experiments are performed using 24 hours data captured at SEECS Data Center for evaluation of this system. The results show that the system is capable of performing URL lookup operation with zero false positive for data rate up to 160 Mbps. False positives occurs for increase in data rate. These false positives are not due to hash collisions but due to increase in packet drop ratio.

8.2 Future Work

The future aim will be to explore the lexical and host-based feature of Uniform Resource Locators (URLs) for developing an anti-malware system using machine learning techniques. The system will be capable of performing the task of malware detection and protection in real-time.

Bibliography

[1] A. Appleby, "Murmurhash 2.0," 2013. [Online]. Available: http://code.google.com/p/smhasher/

[2] W. Chang, B. Fang, X. Yun, S. Wang, and X. Yu, "Randomness testing of compressed data," *arXiv preprint arXiv:1001.3485*, 2010.

[3] R. S. De Oliveira, C. Montez, and R. Lange, "On the use of hash tables in real-time applications," in *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on.* IEEE, 2009, pp. 1–8.

[4] D. Eastlake, G. Fowler, K.-P. Vo, and L. Noll, "The fnv non-cryptographic hash algorithm," 2012. [Online]. Available: http://tools.ietf.org/html/draft-eastlake-fnv-03

[5] C. Estbanez, J. C. Hernndez-Castro, A. Ribagorda, and P. Isasi, *Finding state-of-the-art non-cryptographic hashes with genetic programming.* Springer, 2006, pp. 818–827.

[6] C. Estbanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Journal of Software: Practice and Experience (2013)*, 2013.

[7] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, “Performance of the most common non-cryptographic hash functions,” *Software: Practice and Experience*, 2013.

[8] Y.-H. Feng, N.-F. Huang, and C.-H. Chen, “An efficient caching mechanism for network-based url filtering by multi-level counting bloom filters,” in *Communications (ICC), 2011 IEEE International Conference on.* IEEE, 2011, pp. 1–6.

[9] G. Fuxiang, W. Yanyan, S. Wenjun, and Y. Lan, “Design and implementation of the web access monitoring system based on url analysis,” in *Information Technology and Applications (IFITA), 2010 International Forum on*, vol. 1. IEEE, 2010, pp. 425–428.

[10] J.-l. Gailly and M. Adler, “Zlib compression library,” 2013. [Online]. Available: www.zlib.net

[11] J. 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,” in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on.* IEEE, 2012, pp. 1–6.

[12] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundation, Analysis and Internet Examples.* John Wiley & Sons, 2006.

[13] C. Henke, C. Schmoll, and T. Zseby, “Empirical evaluation of hash functions for multipoint measurements,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 39–50, 2008.

[14] N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto hardware hash functions for high performance networking asics," in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems.* IEEE Computer Society, pp. 156–166.

[15] K. Huang, G. Xie, R. Li, and S. Xiong, "Fast and deterministic hash table lookup using discriminative bloom filters," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 657 – 666, 2013.

[16] N.-F. Huang, R.-T. Liu, C.-H. Chen, Y.-T. Chen, and L.-W. Huang, "A fast url lookup engine for content-aware multi-gigabit switches," in *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, vol. 1. IEEE, 2005, pp. 641–646.

[17] B. Jenkins, "A hash function for hash table lookup," *Dr. Dobbs Journal (1997)*, 1997. [Online]. Available: http://burtleburtle.net/bob/hash/doobs.html

[18] J. Karasek, R. Burget, and O. Morsky, "Towards an automatic design of non-cryptographic hash function," in *Telecommunications and Signal Processing (TSP), 2011 34th International Conference on.* IEEE, pp. 19–23.

[19] D. E. Knuth, "Sorting and searching (the art of computer programming volume 3)," 1973.

[20] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, pp. 459–468.

[21] W. D. Maurer and T. G. Lewis, "Hash table methods," *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 5–19, 1975.

[22] T. Maxino, "Revisiting fletcher and adler checksums," 2006. [Online]. Available: http://repository.cmu.edu/isr/690/

[23] B. J. McKenzie, R. Harries, and T. Bell, "Selecting a hashing algorithm," *Journal of Software: Practice and Experience*, vol. 20, no. 2, pp. 209–224, 1990.

[24] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," DTIC Document, Tech. Rep., 2010.

[25] T.-F. Sheu, N.-F. Huang, H.-S. Wu, M.-C. Shih, and Y.-F. Huang, "On the design of network-processor-based gigabit multiple-service switch," in *Information Technology: Research and Education, 2005. ITRE 2005. 3rd International Conference on.* IEEE, 2005, pp. 240–244.

[26] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time url spam filtering service," in *Security and Privacy (SP), 2011 IEEE Symposium on.* IEEE, 2011, pp. 447–462.

[27] H. Yuan, B. Wun, and P. Crowley, "Software-based implementations of updateable data structures for high-speed url matching," in *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on.* IEEE, 2010, pp. 1–2.

[28] Z. Yuan, B. Yang, X. Ren, and Y. Xue, "Tfd: A multi-pattern matching algorithm for large-scale url filtering," in *Computing, Networking and*

Communications (ICNC), 2013 International Conference on. IEEE, 2013, pp. 359–363.

[29] J. K. M. S. U. Zaman and R. Ghosh, "A review study of nist statistical test suite: Development of an indigenous computer package," *CoRR*, vol. abs/1208.5740, 2012. [Online]. Available: http://arxiv.org/abs/1208.5740

[30] Z. Zhou, T. Song, and Y. Jia, "A high-performance url lookup engine for url filtering systems," in *Communications (ICC), 2010 IEEE International Conference on.* IEEE, 2010, pp. 1–5.

[31] M. R. J. Zobel, "Performance in practice of string hashing functions," in *Database Systems for Advanced Applications' 97: Proceedings of the 5th International Conference on Database Systems for Advanced Applications Melbourne, Australia April 1-4, 1997*, vol. 6. World Scientific Publishing Company Incorporated, p. 215.