File Detection in Network Traffic Using Approximate Matching

Vikas Gupta

Kongens Lyngby 2013
IMM-M.Sc.-2013-51

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-51

Summary (English)

Virtually every day data breach incidents are reported in the news. Scammers, fraudsters, hackers and malicious insiders are raking in millions with sensitive business and personal information. Not all incidents involve cunning and astute hackers. The involvement of insiders is ever increasing. Data information leakage is a critical issue for many companies, especially nowadays where every employee has an access to high speed internet. In the past, email was the only gateway to send out information but with the advent of technologies like SaaS (e.g. Dropbox) and other similar services, possible routes have become numerous and complicated to guard for an organisation.

Data is valuable, for legitimate purposes or criminal purposes alike. An intuitive approach to check data leakage is to scan the network traffic for presence of any confidential information transmitted. The existing systems use slew of techniques like keyword matching, regular expression pattern matching, cryptographic algorithms or rolling hashes to prevent data leakage. These techniques are either trivial to evade or suffer with high false alarm rate.

In this thesis, *known file content* detection in network traffic using approximate matching is presented. It performs content analysis on-the-fly. The approach is protocol agnostic and file type independent. Compared to existing techniques, proposed approach is straight forward and does not need comprehensive configuration. It is easy to deploy and maintain, as only file fingerprint is required, instead of verbose rules.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark, in cooperation with Center for Advanced Security Research Darmstadt (Germany) in fulfilment of the requirements for acquiring an M.Sc. in Security and Mobile Computing.

The thesis deals with the problem of data leakage over an organisation's network and proposes a solution for it by performing bitwise content analysis of network packets using approximate matching.

The thesis consists of 7 Chapters and Appendixes, in which a detailed description of the proposed data leakage prevention method, conducted experiments, analyses of results and directions for future works are presented.

Lyngby, June 2013-2013

Vikas Gupta

Vikas Gupta

Acknowledgements

I thank M.Sc. Frank Breitinger, from Center for Advanced Research Darmstadt (CASED), for his guidance and supervision in all phases of this thesis work. His knowledge and advice have been invaluable throughout the course of this work. I would also like to thank Prof. Christian D. Jensen, Denmark Technical University, Prof. Danilo Gligoroski, Norwegian University of Science and Technology, and Prof. Harald Baier, CASED, for co-supervising this thesis work.

A special thank to Sebastian Abt for introducing me to Frank and providing valuable comments and feedback during the thesis project.

Thanks to all my colleagues at CASED specially Jinghua, Elakkiya and Ivan for making work at CASED fun and keeping me motivated throughout the duration.

Last but not the least, I would like to thank my family for their constant support.

Lyngby, June 30, 2013
Vikas Gupta

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Industrial revolution marked the onset of industrial age, similarly, digital revolution of past 2 decades has marked the onset of *digital age.* Humans are producing and consuming digital information at much higher rate than ever before. In the past two decades, Internet has spread with a breath taking pace and has become an inseparable part of modern life. Internet offers world-wide broadcasting capability, a seamless platform for information dissemination, and a medium for collaboration and interaction between individuals irrespective of their geopolitical location. The ease of collaboration, outreach and cost effectiveness invited organisations to use Internet for performing business. The productivity of employees have increased manifold as a result of technology.

Email is one of the most important communication mechanism in any organisation and according to an estimate, around 101 billion business emails are exchanged per day in 2012 [20]. Enterprises are awash with ever-growing data of all types, easily amassing terabytes of information. As per Harvard Business Review, Walmart collects more than 2.5 petabytes of data every hour from its customer transactions [1].

Every technology has its own dark side. Defending connected networks have always been a challenge. Well-known examples of successful attacks against

[1] `http://www.hbr.org/2012/big-data-the-management-revolution` (last accessed 2013-June-25).

computer networks include Morris Worm in the 1980's [16], to Chinese hackers compromising US weapon systems' designs in 2013. Administrators are facing a challenge in keeping confidential information from leaving the networks. At present, there is a constant increase in data breach incidents since 2009. According to Open Security Foundation (OSF), 1605 incidents were reported last year, which is a steep increase of 45% compared to 2011. Some major data breach incidents reported within last couple of years[2] are [8]:

- Attackers were successful in compromising 77 million records of Sony Corporation in April 2011.
- LinkedIn was allegedly hacked of 6.5 million password hashes in June 2012.
- In April 2013, more than 50 million customer's names, emails, birth dates, and hashed and salted passwords of LivingSocial were accessed by hackers.

Data is valuable, whether for legitimate or for criminal purposes. Ensuring the security and privacy of the data is a major challenge. Information leakage, including company's *intellectual property* or user information, is becoming more frequent. Unauthorized use of information can incur enormous financial cost to an organisation. As per McAfee, malicious insiders had the largest average number of compromised records per breach of 72,325 [24].

In order to minimize this risk, an intuitive approach is to scan the network traffic for *known files'* fragments, where *known files* refers to the files which are confidential as per company's policy and not allowed to go outside company's network. Traditionally, organisations added keywords like 'confidential' or 'secret' to ensure that no such file having these keywords leaves the network. But such keyword based approach is easy to evade. A more foolproof technique is required, than just looking for a specific keyword.

1.1 Keywords

Approximate matching, similarity hashing, network sniffing, file identification, known content detection, data leakage prevention, packet content analysis.

[2] `http://datalossdb.org/{index/largest,statistics}` (last accessed 2013-June-25).

1.2 Problem Description

One of the main aim of an attack on an organisation's network is to obtain confidential information, ranging from the credit card numbers of its customers to future expansion plans of the company. Today's network is so voluminous that manual inspection for data leakage is impractical and an expensive alternative. In order to address the security issues, companies install intrusion detection and prevention systems (IDS), firewalls and virus scanners. However, only two-third of all the data breaches are the result of hacking attacks - according to OSF, 36% of all recorded incidents are involving insiders[3].

Data loss prevention systems are developed to check and block outgoing traffic for known confidential information. These systems perform deep packet analyses and use multiple methods to detect confidential content in network traffic, for instance, searching for keywords, patterns or regular expressions. These systems also use cryptographic hash functions for content identification, like in digital forensics. But such approach cannot be used in case of network traffic, as file content is split and spread over many packets.

In this work, a new approach using approximate matching to detect *known file content* in network traffic is presented. Approximate matching is a technique for identifying similarities between some digital objects, like files, storage media, network streams etc. It is based on the logic of identifying and picking up some features or attributes which are unique to each object and can be used to identify and compare them. This collection of features is the signature/fingerprint of the object under investigation.

The working of the proposed approach is straightforward. For each network packet sniffed, a fingerprint is generated using approximate matching algorithm. This packet's fingerprint is compared against a pre-computed list of fingerprints of protected/confidential files, if packet's content matches with content of a protected file, data breach incident is reported. Unlike existing techniques, the proposed technique is simple to use and does not need comprehensive configuration. It can be easily deployed and maintained as only fingerprints are required, unlike providing verbose rules.

The proposed approach can also be used for incoming traffic into a network. For instance, an adversary can send a malicious code, which is a modification of a previous known version, to evade IDS or virus scanners, which can be detected by approximate matching approach.

[3] `http://datalossdb.org/statistics` (last accessed 2013-June-25).

1.3 Motivation and Benefits

Data leakage or information leakage, can be defined as unauthorized transmission of data (or information) from within an organization to an external destination or recipient [19]. As highlighted in previous section, insiders, whether intentional or inadvertent, constitute a major source of data leakage. Instant messages, peer-to-peer file transfer, email, cloud storage etc. are some of the internet based vectors which are easily accessible in an organisation to cause a data breach. Further, use of SSL or other encryption technology has made detecting attempts of data leak practically more difficult.

The current state of practice regarding the technical ability to defend and monitor Internet-based attacks is not sufficient. Attackers use sophisticated techniques to evade from surveillance and it is becoming impossible to defend using current practices. As per Bendrath [1], "the anonymity enjoyed by today's cyber attackers poses a great threat to the global information society, the progress of information based international economy, and the advancement of global collaboration and cooperation in all areas of human endeavor".

There exist no detailed previous work addressing the issue of data leakage over network. The existing data leakage prevention systems are commercial and closed source and not much information is available about their working or performance. The prototype developed in this thesis can be used to check data leakage and catch the malicious insiders. Some of the scenarios where it can be used are enumerated below [23] [8]:

Scenarios

- Illegal software downloads in a network using peer-2-peer or ftp.
- A confidential bid is leaked by an insider to a competitor through email or cloud storage services.
- A financial services firm produces valuable research that is forwarded by an insider to unauthorized distribution channel.
- A spreadsheet containing personal medical data of patients is posted to a public website and the mistake goes unnoticed for a long time.
- Identifying and blocking webpages, text files or emails dealing with a specific content. For instance, since January 2011 Russia started a Internet surveillance plan to protect kids from Internet pedophiles.

- Detecting and deleting malware or spam before it reaches its victim and uncover self-distributing malware.

1.4 Research Goals

Research goals of this thesis are:

1. Is it feasible to identify *known file content* in network packets using approximate matching?
2. Which approximate matching algorithm can be used in such a tool?
3. Develop a prototype using approximate matching algorithm.
4. Keep the network latency introduced by performing such packet inspection as low as possible.
5. Does present approach have acceptable false alarm rate?

1.5 Methodology

Literature study - in order to find out what is the current state of the art for data leakage detection in network traffic. To the best of our knowledge this is the first work to identify *known file content* in network traffic. Though, data leakage prevention system is the most relevant technology to be considered.

Implementation - is carried out in two steps. First, the feasibility of the approach is established. Secondly, an effort is made to improve the performance and detection rate by identifying the core problems and addressing them.

1.6 Contribution

The main contribution of this work aims at establishing the feasibility whether approximate matching algorithms can be used in detecting *known file content* in network traffic. The focus is to make a working prototype which uses approximate matching and signals presence of known file by reporting the filename detected and the source and destination IP address involved in such a transmission.

Furthermore, tests showed that approximate matching algorithm, `mrsh-v2` is best suited to use for network traffic analyses compared to `sdhash`. `mrsh-v2` is more efficient in terms of per packet processing time than compared to `sdhash`. Also, using `mrsh-v2` at application layer, i.e., lower layer headers stripped off, gives a 100% file detection rate in single packet analysis mode, and more than 98% detection rate in case of stream based analyses mode.

1.7 Notation and Terms

This section explains all notations and terms which are used in this thesis.

- Approximate matching, a.k.a similarity hashing previously.
- `sniffer` refers to the prototype/tool developed during this thesis to test the approach.
- 'Known files' refers to the files/content which an organisations wants to protect, is confidential, or other way around, an organisation does not want to enter its network (e.g. a known malware).
- DLPS refers to data leakage and prevention systems.
- IDS refers to intrusion detection systems.
- SPA refers to single packet analysis mode while analysing network traffic.
- STA refers to stream analysis mode while analysing network traffic.

1.8 Structure of Thesis

The rest of the thesis is organized as follows. Chapter 2 introduces concepts of approximate matching, communication network's model and network packet inspection. These concepts are used in designing the prototype, developed for performing network packet analysis.

Chapter 3 summarises how the implementation and testing is performed. It discusses about the file set used for emulating *known files*, how the approximate matching algorithms are integrated into the prototype and eventually the overall proceedings to perform a test. The results thus obtained are elucidated in chapter 4. The results chapter talks about the average true positive, false positive and false negative scores and the processing time taken to perform a match for each packet. An improvement of initial single packet analysis approach is proposed by using stream based analysis and discussed in the later half of the chapter.

Chapter 5 gives an overview of the related research and development in the field of data leakage prevention. Chapter 6 is about the discussion and analyses of the work performed. While, the thesis is concluded by presenting a conclusion and future work that could be performed in chapter 7.

Chapter 2

Foundation

This chapter presents the foundation for the work performed in this thesis. Section 2.1 gives an overview of hashing, while section 2.2 talks about cryptographic hash functions, their characteristics and properties, and shortcomings of cryptographic hash functions. Section 2.3 gives an introduction to the concept of approximate matching. In the initial half of the section, evolution of approximate matching is presented, while in the later half working of `sdhash` and `mrsh-v2` is presented. Properties and characteristics of approximate matching is also presented in this section.

Section 2.4 summarises the communication network models: OSI and TCP/IP model. Network terminologies used in this work are also defined in this section. Whereas, section 2.5 and section 2.6 discusses concepts of packet inspection and encoding schemes respectively.

2.1 Hash Functions

A *hash function*, is a routine or an algorithm that maps arbitrary strings into binary strings of fixed length. In simple words, hash functions compresses the data, i.e, the output is shorter than the input. While a *hash* is a number that is generated from some data using an hash function, in such a way that the distribution of the numbers look random and there is a low probability that the same number is re-used. *Hash* can be interchangeably used with *hash value, hash code, hash sum* or *checksum.*

Hash functions enable quick table lookup or data comparison tasks. Some utility of hash functions are, finding items in database, detecting duplicate or similar records in a large file, to build caches when data is stored on slow media etc. Hash functions can be broadly classified into cryptographic and non-cryptographic hash functions based on some properties, which are discussed in Sec. 2.2.1. Some examples of non-cryptographic hash functions are Fowler-Noll-Vo (FNV) hash function and Java hashCode(), while cryptographic hash functions can be exemplified by MD5 (Message Digest family), SHA1, SHA2, SHA3 (Secure Hash Algorithm family).

2.2 Cryptographic Hash Functions

Cryptographic hash functions (crypto hash) are most used for message integrity check and digital signatures. Crypto hash are generally faster than encryption algorithms, and therefore it is typical to compute the digital signature or integrity check of some document by applying cryptographic processing to the document's hash value, which is small compared to the document itself. Also, making a digest public does not divulge any information about the content of the original document. For instance, certain websites provide MD5 or SHA-1 hashes of the binaries available for download along with the binaries. Such practice ensures that the binary is authentic and is not compromised by an attacker.

Next section enumerate the properties and requirements of a hash function to call it a crypto hash. Note, all the crypto hash functions are hash functions but not the other way around.

2.2.1 Requirements and Properties

For a hash function to be called as a cryptographic hash function, it should has the following properties [31]:

Compression: Hash function *h(x)* should produce a fixed-length output string s with bit-length n, for any given input string k of any arbitrary finite length.

Ease of Computation: Every hash-value of hash function *h(x)* is efficient to compute in software and hardware.

Preimage Resistance: It is computationally infeasible to find an input string *x'* (preimage) such that $h(x') = s$ for any given output string s for which corresponding input string x is unknown.

Second Preimage Resistance: It is computationally infeasible to find an input string x' (second preimage) for any given input string x such that $h(x') = h(x)$.

Collision Resistance: It is computationally infeasible to find two distinct input strings x and x' such that $h(x) = h(x')$.

Non-correlation: Input string x and output string s are not correlated in any way. Every bit of input string x affects every bit of output string s.

Near-collision Resistance: It is computationally infeasible to find two input strings x and x' such that $h(x)$ and $h(x')$ hardly differ.

Partial-preimage Resistance: It is computationally infeasible to find any substring of input string x for any given output string s even for any given distinct substring of input string x.

Computationally infeasible implies that solving the underlying problem is not possible in polynomial time or constrained memory.

2.2.2 Problems with Cryptographic Hashes

Cryptographic hash functions have found a wide spread use in the industry and a significant effort is being made to develop more secure hash functions. But crypto hashes suffer with a few shortcomings, especially to perform *known file filtering.* In [42], Roussev enumerated certain scenarios in which crypto hashes cannot be used as filters:

1. **Identification of embedded/trace evidence:** A file of one format is embedded in another file of a different format. For example, a jpg embedded in pdf file. As per conventional approach, hash of jpg will not be able to indicate the presence of it in pdf file.
2. **Identification of code versions:** Softwares are updated and patched very frequently and thus making it infeasible to maintain crypto hash inventory of all the files for every single version.

3. **Identification of related documents:** Documents undergo transformations as they are updated. Identifying the original source of document is not possible with crypto hashes.

4. **Correlation of memory and disk sources:** During a forensic investigation it is needed to be able to correlate memory captures and disk images. The run-time layout and content of an executable/document are different from the on-disk representation, but still have identifiable commonalities. Due to this difference, conventional hashes will fail.

5. **Correlation of network and disk sources:** Files transmitted over a network are fragmented and interleaved. Current approaches require time-consuming packet flow reconstruction and protocol parsing to extract transmitted files before any hash filtering can be applied.

Cryptographic hashes (ideally) depend on every bit of the input, making them inherently fragile and unsuited for similarity detection. Above stated shortcomings of conventional hash algorithms emphasise the need of an alternative family of algorithms, which is efficient in solving above problems.

2.3 Approximate Matching

Crypto hashes (by design) can only give simple yes/no answers, e.g. two files matched using SHA-1 will either match or not. There is a need of an algorithm which provides a probabilistic answer - a number between 0 and 100, when two similar files are compared. The confidence score should be low when a small amount of content in the two files is similar and a high score when ratio of similar content is high. *Approximate matching* addresses some of the issues raised in Sec. 2.2.2.

Before delving into details of approximate matching, ways in which matching can be performed is presented. Matching can be performed in 3 different ways and are elaborated below [43]:

Bitwise matching uses only the sequence of bits in a digital object. The matching is performed irrespective of any structures within the data stream. Approximate matching algorithms come under this category.

Syntactic matching uses internal structures present in digital objects. For example, pdf and jpg files have some standard structure and that can be used during matching.

Semantic matching uses contextual attributes of the digital object. This matching is more closer to the human perception. For instance, a facial recognition algorithms will look for facial patterns in images. The facial recognition algorithm works agnostic to the format of the image feed it uses.

Approximate matching or similarity hashing is a technique for identifying similarities between or some digital objects, like files, storage media, network streams etc. The underlying logic is to identify and pick some features or attributes from each object and compare them. This collection of features is the signature of the object under investigation. The confidence score thus generated should be based on the number of features shared by the object.

In past few years, an effort has been made and a few such algorithms have been proposed. `ssdeep` is known to be first of its kind and was accepted well by the industry. Services like VirusTotal[1] uses `ssdeep` to identify malwares. `sdhash` is successor to `ssdeep` but uses totally different approach for similarity detection. `mrsh-v2` is one of the latest approximate matching algorithm proposed and have a significant performance advantage over its peers. These algorithms and evolution of approximate matching is discussed in upcoming sections. But firstly, we discuss about a special data structure called Bloom filters.

2.3.1 Bloom Filters

Bloom filter is a space-efficient randomized probabilistic data-structure for representing a set in order to support membership queries. The concept of Bloom filters was proposed by Burton Bloom in 1970 [2]. Lets try to understand the need of Bloom filters with an example. During a forensic analysis, the investigator has a reference hash set of 50 million hashes (H_{set}). During the investigation, investigator hashes each file he encounters and compares the generated fingerprint against H_{set} to identify known content. For every query in H_{set}, approximately 26 main memory accesses are expected and each of which causing a delay of tens of CPU cycles. Such a memory constrained workload severely underutilizes the CPU, which directly slows down the investigation process. Bloom filters provide an alternative to increase the speed of lookup operations and reduce space requirements and in turn increasing the efficiency [40].

Bloom filters find extensive use in the field of computer networks, specifically in network routing and traffic filtering [10] and in this section, their use in

[1] virustotal.com is a free online service which analyses malwares and urls, and facilitates quick detection of malicious softwares

Table 2.1: Bloom filter's false positive rate variation with bits/element [40].

No. of Hashes	8	10	12	16
4	0.0240	0.0117	0.0064	0.0024
6	0.0216	0.0083	0.0036	0.0009
4	0.0255	0.0081	0.0031	0.0006

cryptographic hashing and approximate matching is discussed.

Working

The working of a bloom filter is straightforward. An *empty bloom filter* is a bit array of m bits, all set to 0, used to represent a set S of n elements. A bloom filter uses k independent hash functions $h_1, ..., h_k$ with range (1,...,m). In order to insert an element s into the filter, we compute $h_0(s), ..., h_{k-1}(s)$ where each h outputs a value between 0 and $m - 1$. Thus, each hash function sets the corresponding bit within the Bloom filter.

To check the presence of an element s', we compute $h_0(s'), ..., h_{k-1}(s')$ and check if the bits at the corresponding positions are set to one. If all bits are set to one, s' is assumed to be a member of S with a high probability. If atleast one of the bits is set to zero, it could be concluded that s' is not member of S. The filter will never return a false negative. However, filter can return a false positive, i.e, it may return a 'yes' for a element which was never inserted [6].

After the insertion of n elements, the probability of Bloom filter returning a false positive is a nonlinear function of the bit-per-element ratio m/n and the number of hash functions k. The variation in false positive rate with different parameter combination is depicted in Table 2.1 [40]. Fig. 2.1 shows an overview of how a Bloom filter works.

Continuing with the scenario described above, instead of computing k separate hashes for a given file, file's cryptographic hash can be split into several non-overlapping subhashes, and use them as if different hash functions have produced them. A 128-bit MD5 hash can be split into four 32-bit separate hashes. This will reduce the memory lookup from 26 to just four. Also, the false positive rate of less than 0.3 per million doesn't pose severe practical hindrance.

Bloom Filters also find extensive use in approximate matching algorithms to represent hash values. Bloom filters allow fast comparison of similarity hashes

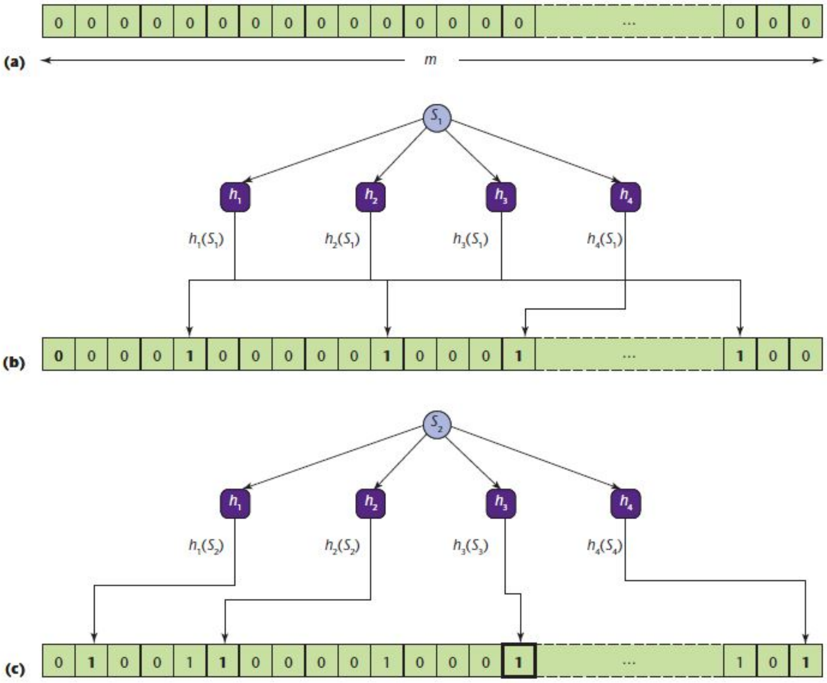

Figure 2.1: The insertion of two elements into a Bloom filter using four hash functions: (a)an empty Bloom filter; (b) a Bloom filter after the insertion of one element, S_1; and (c) a Bloom filter after the insertion of a second element, S_2. Each insertion sets four bits in the filter; some bits might be selected by different elements, $h_4(S_1) = h_3(S_3)$, which can lead to false positives [40].

using the Hamming distance. `sdhash` and `mrsh-v2` use Bloom filters for performing similarity matching. They are discussed in Sec. 2.3.4 and Sec. 2.3.5 respectively.

2.3.2 Evolution of Approximate Matching

Cryptographic hashes made searching for a object, which is exact copy of reference object, easy to perform, but searching for a similar object is still a challenging task. *Avalanche effect*[2] in cryptographic hash functions make them unfit for

[2] A slight change in input will alter the output significantly, e.g, half the output bits flipped. It is a desirable property of cryptographic algorithms.

similarity detection.

The idea of using characteristic features of one object to compare with others to establish similarity has been there for decades. This idea can be defined as *data fingerprinting*, use of more resilient features of an object to identify it. One of the first attempt for data fingerprinting was done by Michael Rabin in 1981 [37]. His idea was based on random polynomials, and with original purpose "to produce a very simple real-time string-matching algorithm and a procedure for securing files against unauthorized changes" [37]. Rabin fingerprint approach is like a checksum with low, quantifiable collision probabilities that can be used to efficiently detect identical objects.

Udi Manber's *sif* unix tool, developed in 1994, is capable of quantifying similarities among text files [30]. Sergey Brin in his pre-Google years used Rabin fingerprinting in a copy-detection scheme. Broder et al. applied Rabin fingerprinting to find syntactic similarities among web pages [11].

2.3.3 Rolling Hash

Rolling hash is a hash function where the input is hashed in a window that moves through the input. The rolling hash functions uses a small *context* of a few bytes to procduce a pseudo-random value h_r. The rolling hash maintains a state solely based on the last few bytes from the input. While processing, each byte is added to the state and removed from the state after a set number of other bytes have been processed. Let the input be of n characters, b_i be the ith character of the input. At any position p in the input, the state of the rolling hash will depend only on the last s bytes of the file. Thus, the value of the rolling hash, r, can be expressed as a function of the last few bytes as following: $r_p = F(b_{p+1}, b_p, b_{p-1}, ..., b_{p-s})$

Rolling hash is used in Rabin Karp string search algorithm [27].

2.3.4 `sdhash`

`sdhash` was proposed by Vassil Roussev in 2010 [41]. `sdhash` uses a totally different approach for similarity matching. It uses concept of similarity digest hashing and hence getting its name (`sdhash` = **s**imilarity **d**igest **hash**). `sdhash` extracts statistically improbable features using the Shanon entropy, where a feature is a byte sequence of 64 bytes. Each of the above selected feature is then hashed using a cryptographic hash function SHA-1 [18]. When a Bloom filter is full, a new filter is added to accommodate the remaining features. Thus, a *similarity digest* consists of a sequence of Bloom filters and its length is about 2-3% of input length.

Bloom filters have predictable probabilistic properties. Thus, for comparison of two `sdhash` signatures a Hamming distance-based measure D(•) is calculated. The match score gives an approximate estimate of the fraction of features that two filters have in common. To compare two digests, for each of the filters in the first digest, the maximum match among the filters of the second is found [42]. The resulting matches are then averaged.

The similarity distance $SD(F,G)$ for digests $F = f_1 f_2 ... f_n$ and $G = g_1 g_2 ... g_m, n \leq m$, is defined as:

$$SD(F,G) = \frac{1}{N}\sum_{i=1}^{n} max\ D(f_i, g_j) \quad \text{where } j = 1...m$$

`sdhash` computes a normalized Shannon entropy measure, as emprical probability of encountering a 64-byte feature can neither be directly estimated nor could such observation be practically stored and looked up. These normalized features are placed into 1000 classes of equivalence.

2.3.5 `mrsh-v2`

Multi-resolution similarity hashing (MRSH) [44], proposed by Roussev et al., is a variation of `ssdeep` [28]. `mrsh-v2` is updated version of MRSH, proposed by Breitinger & Baier [6] and is based on the concept of multi-resolution similarity hashing and context triggered piecewise hashing [28].

`mrsh-v2` identifies trigger points in the input byte sequence to divide it into chunks. This division into chunks uses a pseudo random function prf and a modulus called block size b. A window of fixed size 7 slides through the whole input, byte for byte, and prf generates a pseudo random number r at each step over the window. If $r \equiv -1\ mod\ b$, the byte sequence in the window is a trigger point and thus the end of the chunk. The implementation aims at having a fingerprint length of 0.5% and hence of $b = 160$ bytes.

Each chunk identified above is hashed using FNV [33]. The hash generated is of 64 bit. In order to insert a FNV hash into $m = 2048$ bit Bloom filter, 11 bit sub-hashes are constructed based on the least significant 55 bits of the FNV hash. Lastly, each sub-hash sets one bit within the Bloom filter. For example, if a sub-hash is $100101001_{binary} = 129_{hex} = 297_{decimal}$, then the bit 297 in the Bloom filter is set to one.

A maximum of 160 chunks per Bloom filter are allowed in mrsh-v2 by design. If this limit is reached a new Bloom filter is created. Hence, the final fingerprint obtained is a sequence of Bloom filters. Variable length fingerprints are generated by mrsh-v2, unlike traditional hash functions [31].

mrsh-v2 Block Size

Block size b is the amount of chunk per Bloom filter. If the block size is small, then it provides better coverage and higher sensitivity but requires more storage and processing time. On the other hand, a bigger block size covers less detail and thus is less processing intensive on comparison.

A trade-off is required to be maintained between the block size and the processing time. In Table 2.2 gives an overview of how the hash length varies with change in block size.

Table 2.2: Relation between block size and hash value length for mrsh-v2 [6].

Blocksize	128	160	256	320	512
Expected length in %	1.250	1.000	0.625	0.500	0.313

Comparison with sdhash

mrsh-v2 is a significant improvement over its predecessors. Computation time for mrsh-v2 is lower than sdhash, and closest to classical hash function SHA-1. A relative comparison of computation time is tabulated in Table 2.3.

mrsh-v2 offers better content coverage than sdhash, where coverage in case of approximate matching means that every byte of the input influence the output [7].

Additionaly, mrsh-v2 provides two modes for performing a comparison, fragment detection mode and file similarity mode. In case of known content detection in network traffic, each network packet is containing fragment of the original file. Thus, fragment mode is ideally suited to the thesis goal.

Table 2.3: Relative comparison of approximate matching algorithms with SHA-1 [6].

SHA-1	mrsh-v2	sdhash 2.0	ssdeep 2.8
1.000	2.054	11.236	2.798

In the present work, we will use sdhash and mrsh-v2 for performing matching on network traffic and establish which one of them is more suitable for real time deployment.

2.3.6 Other Approximate Matching Algorithms

ssdeep

ssdeep was proposed by Kornblum in 2006 [28]. The tool ssdeep produces context triggered piecewise hashes, commonly referred to as *fuzzy hashes.* Working of ssdeep is simple:

1. Break up the file into pieces using the result of a rolling hash functions.
2. Use another hash function to a produce a (small) hash for each piece.
3. Concatenate the results to produce the hash signature for the whole file.

In [42], a comparison of ssdeep and sdhash is presented. Some of the key comparison results are summarised in Table 2.4. In brief, ssdeep have inferior detection rate than sdhash and also it is slower than mrsh-v2 and sdhash (see Table 2.3). By design, ssdeep's performance is highly dependent on the presence of a large, continuous chunk of common data, and thus making it unfit to use for network traffic analysis. ssdeep is not considered as a candidate algorithm in this work.

bbhash & mvHash-B

Some other candidates which are also considered during the initial phases are: bbhash [5] and mvHash-B [4]. But these algorithms have performance issues. bbhash is to slow and not fit to be used in case of network packet analysis [6]. While, mvHash-B is file type dependent and thus not further considered.

Table 2.4: `ssdeep` vs `sdhash`: True, false, and total known positives [42].

Set	`ssdeep`		`sdhash`		Total
	TP	FP	TP	FP	
pdf	39	28	45	25	46
doc	40	31	51	7	53
All	653	310	1124	71	1189

2.3.7 Properties of Approximate Matching Algorithms

Inspired from cryptographic hashes, in [6] Breitinger et al. proposed properties for approximate matching. The properties are divided into two groups: general and security properties. These properties provide parameters for comparing two approximate matching algorithms. The properties are discussed below:

General Properties:

1. **Compression:** The output of approximate matching is much smaller than the input. The shorter the output the better it is. Unlike conventional hash algorithms, the output is not a fixed-length hash value. The compression is a desired quality because, firstly, a short hash value is space-saving and secondly, the comparison of small hash values is faster.

2. **Ease of Computation:** Generating a hash value is 'fast' for all kind of inputs. The processing time should be comparable to classical hash functions like SHA-1. This property ensures the usability in practice.

3. **Similarity Score:** Comparison of approximate matching hash values is more complex than compared to traditional hashes, which use Hamming distance. Input of a comparison function are two hashes to be compared, returning a score between 0 and X, where X being the maximum match score. A maximum match score is indicative that two files are identical or almost identical. Generally similarity score is between 0 and 100 and represents a percentage value.

Security Properties:

1. **Coverage:** Every byte of an input should influence the hash value. Statistically, given a certain byte of the input, the probability that this byte does not influence the input's digest is insignificant.

2. **Obfuscation resistance:** It should be difficult to achieve a false negative/non-match. For example, let f be a file under investigation. It should be difficult to manipulate f to f' so that a comparison yield a non-match but they are still very similar.

2.3.8 Use Cases

Approximate matching potentially have extensive use in forensic analysis by identifying similar objects, malware or junk mail detection. In [6], use of approximate matching is broadly classified into two categories: for file identification and fragment detection. Each of the use cases are discussed below:

File Identification: In computer forensics a database of fingerprints of known malware or files from previous investigation is maintained. During the investigation process, fingerprints of the new identified content is generated and matched against this database to quickly identify the new content. The segregation of hashes into known-to-be-good, known-to-be-bad and unknown input can further simplify the forensic investigation.

Blacklisting. The main challenge for an active adversary is to conceal suspect files from an automatic identification by investigators, anti-virus software or junk mail scanner etc. In case of cryptographic hash functions it can be trivially done by flipping a single bit, but it is not possible in case of approximate matching.

Whitelisting. In case of whitelisting, cryptographic hash functions are the preferred choice. For instance, an active adversary can manipulate the *ssh daemon* of an operating system and include a backdoor. Thus, the original file and the modified file are very similar although it is a malicious ssh daemon. The whitelisting is out of scope of consideration, as no adversary will like to manipulate a file to look like a suspect file.

Fragment Detection: An investigator is encounters a hard disk which is formatted using quick-mode. The only way to analyse the data is by analysing the low level hdd blocks. SPH can be used in analysing these file fragments.

2.4 Communication Networks

This section describes network terminology frequently used in this work and the framework for the specification of network's physical components and their functional organization and configuration. OSI and TCP/IP are two important reference model for network architecture discussed in Sec. 2.4.2 and Sec. 2.4.3. TCP/IP model will be used as a reference in this thesis.

2.4.1 Network Terminologies

Some terminologies frequently used with network traffic are described below [3]:

Segment A segment is the unit of end-to-end transmission in the TCP protocol. A segment consists of a TCP header followed by application data.

IP Datagram An IP datagram is the unit of end-to-end transmission in the IP protocol. An IP datagram consists of an IP header followed by transport layer data.

Packet A packet is the unit of data passed across the interface between the internet layer and the link layer. It includes an IP header and data. A packet may be a complete IP datagram or a fragment of an IP datagram.

Frame A frame is the unit of transmission in a link layer protocol, and consists of a link-layer header followed by a packet.

2.4.2 OSI Reference Model

Open System Interconnection (OSI) Model [46] is a conceptual model proposed in 1983. This model characterizes the internal functions of a communication system by partitioning it into abstraction layers. As per the model, communication network can be divided into 7 logical layers. Each layer is a collection of similar functions. A layer provides services to the layer above it and receives services from the layer below it. The proposed 7 layers and their respective functions are discussed below:

Physical Layer This layer defines the electrical and physical specifications for devices. This layer ensures that if one side sends 1 bit, then the other side receives 1 bit, not as a 0 bit. In a nutshell, this layer is concerned with transmitting raw bits over a communication channel.

Data Link Layer At this layer, data packets are encoded and decoded into bits. It accomplishes this task by having the sender break up the input data into *data frames* and transmit the frames sequentially. *Acknowledgement frame* is returned in confirmation on receiving a correct frame.

Network Layer This layer is responsible for controlling the operation of the subnet. *Switching* and *routing* technologies are implemented on this layer.

Transport Layer This layer provides transparency in transfer of data between end users. Transport layer ensures the reliability of a given link through flow control, segmentation/ de-segmentation, and error control. It is true end-to-end layer, all the way from the source to the destination.

Session Layer The session layer allows user on different machines to establish session between them. The operation of setting up a session between two processes is called *Binding*. In some protocols this layer is merged with the transport layer.

Presentation Layer This layer is concerned with the syntax and semantics of the transmitted information. It ensures the independence from data representation by translating between application and network formats.

Application Layer This layer is closest to the user. This layer directly interacts with software applications that have a communicating component. HTTP protocol is works on this layer of the network.

2.4.3 TCP/IP Model

This model was proposed by Cerf and Kahn in 1974 [13] and divides the network into four layers. Each of these layers are discussed below in detail:

Link Layer TCP/IP model does not discuss much about the link layer. Though, it is the lowest component layer and ensures that TCP/IP can work on any hardware. This layer is used to move packets between two hosts on the network.

The Internet Layer This layer is responsible for injecting the packet into any network and sending the packet to potentially multiple networks. Internet layer defines an official packet format and protocol called *Internet Protocol* (IP). IP performs two basic functions:

- Host addressing and identification by having hierarchical IP addressing system.
- Secondly, packet routing.

ICMP and IGMP are some protocols that are carried over IP.

Transport Layer This layer allows the peer entities on the source and destination hosts to perform a conversation. Major responsibility of this layer are: end-to-end message transfer independent of the underlying network, along with error control, segmentation, flow control, congestion control, and application addressing. Two end-to-end transport protocols have been defined for this layer: *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP).

Application Layer Unlike OSI model, TCP/IP model does not have a session or presentation layer. Application layer directly interacts with the transport layer to communicate over the network. All the higher level protocols like FTP, HTTP, SMTP etc. work on this network layer.

In this thesis, TCP/IP model will be used as a reference. All the experiments and results are reported in accordance to TCP/IP model.

2.4.4 Maximum Transmission Unit

Maximum Transmission Unit (MTU) is defined as the maximum size datagram that can be transmitted through a network [35]. The MTU depends on the link layer technology being used for the network. In case of IEEE 802.3 Ethernet, MTU is 1500 bytes [22].

2.4.5 TCP Segmentation Offload

Large Segment Offload (LSO) technique, it is used to increase the outbound throughput of high-bandwidth networks by offloading packet processing time from CPU to the Network Interface Card (NIC). When LSO is applied on TCP, it is called as TCP Segmentation Offload (TSO). The working of TSO can be explained with the help of an example. Let a unit of 65,536 bytes is to be transmitted by the host device. Assuming MTU of 1500 bytes, this data will be divided into 46 segments of 1448 bytes each before it is transmitted over to network through the NIC. Process of dividing the data into segments before sending it over the network is handed over to NIC instead of CPU. NIC will break down the data into smaller segments, and add corresponding TCP, IP and data link layer protocol headers. This significantly reduces the work done by the CPU. Large Receive Offload (LRO) is a similar technique to GSO, but applied for incoming traffic [17].

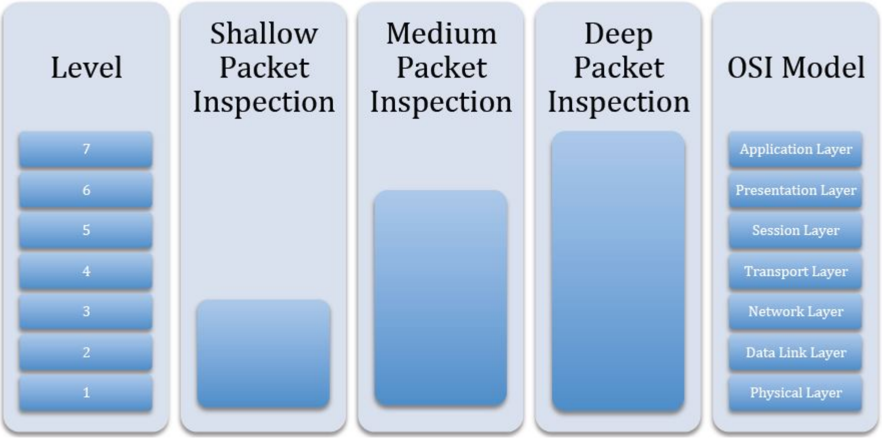

Figure 2.2: Packet inspection depths [34].

2.5 Packet Inspection

Packet analysis is the technique of analysing the content of the network packet to search for protocol non-compliance, viruses, spam, intrusions or, for the purpose of collecting statistical information. One of the first approach to analyse network packet was *static packet inspection.* In this approach, as the name suggests, each packet is considered one at a time and examines each packet based on the header information like: source IP, destination IP, source port and destination port. *Static packet inspection* is very easy to deploy but also very easy to evade.

The next approach in the packet inspection evolution is *stateful/dynamic packet inspection* (SPI). SPI is similar to *static packet inspection* but one main difference. In this, the packet filter is aware of the new and an established connection and maintains a state of the connection.

On the basis of network layer of operation, packet inspection can be broadly classified into three categories: Shallow, medium and deep packet inspection.

2.5.1 Shallow Packet Inspection

In shallow packet inspection, the headers of the network packet are parsed, and the results are compared to a rule set. In this approach, a tuple of 5 elements is maintained for each connection. The 5 elements are: Source IP address, Destination IP address, Source transport layer address, destination transport layer address and service type. The rules are defined based on these fields or a combination of them. The traffic is allowed or denied depending on whether it adhere to the rules or not [14].

2.5.2 Medium Packet Inspection

Medium packet inspection (MPI)is done by Application Proxies (AP) or gateway. These AP are placed inline with network routing equipment and thus ensuring that all the network traffic pases through these proxies. MPI allows to parse network traffic on basis of data format types. For example, MPI can be used to prevent client computers from receiving flash files from YouTube, or image files from social network files [14].

2.5.3 Deep Packet Inspection

Deep packet inspection (DPI) "is a computer network surveillance technique that uses device and technologies that inspect and take action based on the contents of the packet, i.e. it consider the complete paylaod of packet rather than just the packet header which includes data up to layer 7 of OSI model" [14].

Fig. 2.2 shows the packet inspection classification with respect to the OSI model.

2.6 Encoding Schemes

Digital systems can store information only in the form of *bits.* A bit can only have two values: 1 or 0. In order to convert these stored bits into something meaningful like English alphabet, numbers and images, one requires an *encoding scheme* or *encoding. Encoding* is a rule which gives some meaning to the data stored using it.

Encoding is the process of transforming a sequence of characters into a specialized format for (efficient) transmission or storage. While decoding is inverse of encoding and defined as the transformation of an encoded sequence back to the original one. ASCII, Base64 or Unicode are commonly used encoding schemes. For example, bit sequence 01100010 is letter *b* in ASCII encoding. A string of 1s and 0s is broken down into parts of eight bit each.

Encoding is used in plethora of ways in network traffic. For instance, SMTP is a text based protocol and thus sending binary data is prone to getting corrupt. To circumvent this problem, SMTP binary payload can be encoded with a binary-to-text encoding schemes like Base64, Base16 (hexadecimal) or Uuencoding [26].

It can be concluded that, in spite two inputs are completely identical, the underlying byte structure can be different depending on the encoding scheme used [8].

Chapter 3

Experimental Setup

The following chapter summarises the steps to setup the development environment and to perform the tests. The first section discusses about the tools and libraries used to create the prototype and to ensure its memory and runtime efficiency.

Section 3.2 talks about the file set used for emulating the 'known file' of an organisation. A corpus of random data files is used for establishing general feasibility of the approach and a corpus of commonly used file types is used to emulate real world data. The protocols and tools used for generating network traffic containing the 'known file' and steps taken to ensure that it mimics the traffic in an organisation's network are discussed in Sec. 3.3. Sec. 3.4 describes how the packet matching is performed for 4 TCP/IP layers.

Sec. 3.5 gives an insight on how `mrsh-v2` and `sdhash` are integrated to perform network traffic analyses smoothly. While, the last section summarises the overall procedure followed to perform various tests.

As discussed in Sec. 2.3, `mrsh-v2` and `sdhash` approximate matching algorithms are used in the developed prototype. In further text, the term `sniffer` is used to refer to the prototype/tool developed.

3.1 Infrastructure

The development and testing is done on a machine running Fedora 17 with Linux kernel 3.8 on Intel Core 2 Duo (3 Ghz) processor, with 4 GB RAM. The prototype is implemented in C/C++ in order to have a good runtime efficiency. Libpcap-1.2[1] is used for capturing network traffic. Libpcap provides a system agnostic portable framework for collecting network statistics, security monitoring and network debugging. Tcpdump, wireshark[2], snort[3] are some popular open source projects using Libpcap.

3.1.1 Development Environment

The development is carried out in VIM editor, while debugging is performed using cgdb[4]. *cgdb* is a lightweight terminal-based interface to the GNU Debugger (GDB). *cgdb* provides a split screen view displaying the source code being debugged, and thus used instead of GDB.

Memory Leakage

Improper handling of memory allocation in C/C++ may lead to memory leakage. Memory leakage can significantly effect program's performance by reducing the available memory to the program. In worst case, the program may be terminated by the operating system. Memory leakage might go unnoticed in case of program that run for short time, as extra memory allocated will be released soon after the program's termination. In the present case, the program runs for a long duration, thus addressing memory leakage is of utmost importance. To check and prevent memory leakage *Valgrind* (version 3.8.1) is used. *Valgrind* is an open source tool for dynamic analysis, used for memory debugging and memory leak detection. It is a framework comprising of various tools to perform dynamic testing. A typical use of valgrind during the prototype development is shown in Listing 3.1.

```
valgrind -v --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers
     =20 --track-fds=yes --track-origins=yes <executable>
```

Listing 3.1: Use of *valgrind* for detecting memory leakage.

[1] www.tcpdump.org (last accessed 2013-June-25).
[2] wireshark.org (last accessed 2013-June-25).
[3] snort.org (last accessed 2013-June-25).
[4] www.cgdb.github.io (last accessed 2013-June-25).

Table 3.1: File size statistics for RS and TS corpus.

Corpus	Total Files	Avg. Size	Min. Size	Max. Size
Random Files	1000	410.12 KB	4.00 KB	14 MB
TS Corpus	3282	487.06 KB	4.00 KB	17 MB

Execution Time Profiling

In order to analyse the execution time of the constituent parts of a program, *gprof* is used; available through operating system's package manager *yum*. The programs source code is compiled with `-pg` option of `gcc`. A profile file, *gmon.out* is created which is used to calculate the amount of time spent in each routine. *gprof* gives run-time figures based on a sampling process, so it is subject to statistical inaccuracy. Inspite of this limitation, it gives a good approximate idea about the program's execution times. Linux's `time` command provides the execution time of the overall script/program and cannot be used to determine the execution time of comprising routines of a program.

3.2 File Set

This section elaborates on the corpus used for emulating 'known file' of an organisation and how it is generated.

Two data sets are developed, one comprising of random data (RS) and other of real world data (TS). RS is used for establishing the feasibility of the approach, while TS is used to emulate real world scenario.

Random data is generated using Linux's `/dev/urandom/` together with `dd` command (Listing 3.2). RS consists of 1000 files with random data.

```
dd if=/dev/urandom of=./dev_random/file bs=1024 count=file_size
```

Listing 3.2: Linux's command to generate random files.

TS is developed by modifying *t5* corpus[5]. *t5* corpus is widely used within digital forensics and includes commonly used file types, like pdf, doc, jpg etc. *t5* does not contain executables, thus 400 executables from a machine running Windows

[5] `http://roussev.net/t5/t5-corpus.zip` (last accessed 2013-June-25).

are included. Also, html files in *t5* corpus are omitted, as html files are not used for transferring confidential information and they are similar to text files. Results for text files will hold true for html files as well. A general overview of the file sizes in the corpus is highlighted in Table 3.1.

In case of real world data, it is ensured that the corpus does not contain similar files. Similar files are detected using all-against-all-comparison of `mrsh-v2`. Two files having match score > 0 are called similar and either of the file is removed from the corpus. The corpus obtained after removing similar files is called test corpus (TS) and its constitution is shown in Table 3.2.

Table 3.2: Statistics of t5-corpus and test-set TS.

	jpg	gif	doc	xls	ppt	pdf	txt	exe	html
t5	362	69	533	250	368	1073	711	0	1073
TS	358	69	333	212	282	954	674	400	0

3.3 Network Traffic Generation

Network traffic is generated by transmitting RS and TS corpus using various application layer protocols. During the testing phase, three widely used application layer protocols are used: HTTP, FTP, and SMTP. However, the whole approach of using approximate matching for known file detection is protocol agnostic.

The network traffic is generated locally and is available for local use only. The traffic is accessed over the *loopback* network interface. The outside access to the generated traffic is blocked by making suitable changes to the respective configuration files and adding rules to the local firewall. The tools and commands used for generating network traffic for various protocols is discussed in detail below.

HTTP: *httpd* package provided with Fedora distribution is used for setting up a local HTTP server. Underlying *httpd* is Apache HTTP Server. *systemctl* is used for starting and controlling the state of *httpd* service. To generate network traffic, each file in the corpus is copied to the http server's directory (`/var/www/html` in Fedora) and accessed one at a time using Linux's `curl` command.

FTP: FTP traffic is generated by setting up a local FTP server using *vsftpd* package provided by *yum* package manager. The traffic is generated by making a request for each corpus file using Linux's `curl` command. Note, FTP can operate in two modes: binary mode and ASCII mode [21]. The tests are performed using the binary mode.

SMTP: SMTP traffic is generated using *mpack* package provided by *yum* package manager. *mpack* encodes the attachments in Base64 encoding. To generate traffic, an email is sent to a local email address (on the same machine) with a corpus file as an attachment to it. `sendmail` command can send emails only in ASCII (text) format, which is not considered to be a best practice to send binary files and thus not used for generating and testing SMTP network traffic.

Offline Network Packets

Performing each test by sending same corpus each time is a time intensive process. A typical test for generating match scores for `mrsh-v2` with TS takes more than 16 hours. For each test, the `sniffer` is started, followed by a sleep time of 3 seconds, to ensure the fingerprint list is read and ready to be used by the `sniffer`. Also, a sleep time of 5 seconds is maintained after generating network traffic to avoid mixing of traffic between analysis of two files.

In order to bring down the testing time for a single test, network packets are taken offline and written to the filesystem. `sdhash` and `mrsh-v2` are fast in performing file-against-file comparison. Running test against offline network packets reduces testing time for a single test by half. The testing times are still higher than expected, as `ext4` filesystem performance decreases when a directory contains a large number of files. In this case, a directory contains more than 290,000 network packets. Though, a slight improvement is achieved by arranging the packets in sub-directories, and it takes 7.5 hours to perform a test.

MTU & TSO

In this work, all the traffic for testing is generated on the loopback (*lo*) network interface of the device. By default, MTU (see Sec. 2.4.4) for *lo* is 65,535 bytes. The MTU is reseted to 1500 bytes using `ifconfig lo mtu 1500` command, to ensure that test environment emulates the real scenario.

Since, TSO (see Sec. 2.4.5) is used at end user machine only and approximate matching approach is applicable for the routers and other network devices as well, TSO is disabled for the experimentation. On a Linux host TSO can be disabled using `ethtool -K tso off` command.

3.4 Layer Specific Packet Matching

Packet matching can be performed for any of the 4 TCP/IP network layers (Sec. 2.4.3). The additional header included in the network packet for a given payload might effect the detection rate negatively. In theory, the detection rate for the packet stripped off all the layer specific header, which is effectively file fragment only, should give the highest possible match score. But detecting each header's length and hence stripping it off is a time consuming process. Therefore, a trade-off has to be made between the match score and processing time.

During the testing, the header for each layer is stripped off individually and then matching is performed. If the link layer and IP header are stripped to perform the match, it is called matching at TCP layer. For the remainder of this thesis, matching is said to be performed at a certain layer if all the headers are stripped off. For application layer matching, all lower layer headers are stripped off. Hence, only application layer header and the payload is considered.

For a given network packet, score and processing time for each layer is generated. Analysing these statistics, the network layer which shows the best behaviour is decided and is used.

3.5 Integrating `mrsh-v2` & `sdhash` into `sniffer`

The prototype uses approximate matching tools, `mrsh-v2` and `sdhash`. One of the method for integrating `mrsh-v2` and `sdhash` is by calling the respective executables from the `sniffer` directly. Any change to the internal source code in the future version of the tools won't effect the working of the prototype, until the user interface is changed. But accessing the executable from a program is a slow process and will introduce unwanted latency in the network traffic. An alternative to this is to import the source code to the prototype and make the respective method calls from the program itself. Since, the source code for both the tools is open source and easily available, approach to import the source code into the prototype is pursued. Integrating and calling respective algorithms from the `sniffer` is simple and described in the following sections.

3.5.1 sdhash

sdhash-2.3 is used in the present implementation of the sniffer. sdhash can be integrated by using the library *libsdbf.a* generated on compiling and installing sdhash on a system. In sdhash source code, *sdbf* class represents a hash of file, while *sdbf_set* is a vector comprising of list of *sdbf* objects. A *sdbf* object is created for all the 'known files' against which the comparison of a network packet is to be carried out, which all together constitutes a *sdbf_set* (*known_sdbf_set*).

A new *sdbf* object is created for each network packet by passing the pointer to the network packet to the *sdbf* constructor. This *sdbf* constructor generates a sdhash fingerprint for the network packet. sdhash implementation exposes a method for matching two sdhash sets against each other. Thus, each network packet's *sdbf* object is added to a singleton *sdbf_set* (*packet_sdbf_set*). *packet_sdbf_set* and *known_sdbf_set* are compared against each other to obtain the result. The sdhash returns match score as a string, which is parsed in the sniffer to perform further actions depending on the match score. A typical implementation of sdhash in the sniffer is shown in Listing 3.3.

```
/*creating new sdbf object for incoming network packet */
sdbf *sdbf_obj = new sdbf(packet_name, packet, SDHASH_BLOCK_SIZE, packet_length);
sdbf_set *sdbf_query = new sdbf_set();

/*singleton set for the network packet's sdbf object */
sdbf_query-> add(sdbf_obj);

/* comparing known files set with above created network packet's sdbf set.*/
result_string = set_known_files->comapare_to(sdbf_query, SCORE_THRESHOLD,
     SHDASH_SAMPLE_SIZE);
```

Listing 3.3: Code snippet for importing sdhash into sniffer.

3.5.2 mrsh-v2

For integrating mrsh-v2, source code is need to be imported and integrated for making respective function calls to perform packet matching. mrsh-v2 defines a *struct FINGERPRINT*, which contains mrsh-v2 fingerprint for a file. The approach with mrsh-v2 is similar to sdhash. *FINGERPRINT_LIST* is a list of *FINGERPRINT*. *FINGERPRINT* is created for each 'known file' and put together in a single *FINGERPRINT_LIST*. When a network packet is received, *FINGERPRINT* is created for that packet. mrsh-v2 exposes *fingerprint_against_all_comparison()* method to perform comparison of a single

FINGERPRINT against a *FINGERPRINT_ LIST*. A typical usage of `mrsh-v2` usage in `sniffer` is shown in Listing 3.4.

```
/*create FINGERPRINT for the incoming network traffic*/
FINGERPRINT *fp_pkt = init_empty_fingerprint();

/*hash the packet to generate the correspoding fingerprint */
hashPacketBuffer(fp_pkt,packet, lenght_packet);

/*compare known file's list with the network packet */
fingerprint_against_list_comparison(known_file_list, fp_packet);
```

Listing 3.4: Code snippet for importing `mrsh-v2` into `sniffer`.

`mrsh-v2` returns a *MATCH_ LIST struct* as a result of comparison performed, which is parsed by the `sniffer` to perform further operations depending on match score.

3.6 Per Packet Processing Time

Processing time per packet is the time taken to generate the fingerprint of the network packet and compare it against the 'known files' fingerprints. To perform match at layers other then link layer, the lower layer headers have to be stripped off, which also consumes some processing time and included in the final processing time for each packet. Time for stripping the header is also included in the processing time. In a nutshell, the time taken since the network packet is received by the `sniffer` and the corresponding result are reported for the packet, constitutes per packet processing time. From here on, per packet processing time will be referred as processing time, if otherwise specified.

Processing time is computed using *C++* `clock()` function in *time.h* header file. `clock()` returns the processor time consumed by the program, also called *CPU time*. The value returned is expressed in *clock ticks*, which are units of time of a constant but system-specific length. To calculate the actual processing time of a program, the value returned by *clock* is compared to the value returned by a previous call to the same function.

3.7 Proceeding

This section discusses the working of the prototype in detail. The steps to start a typical test are highlighted, including fingerprint list generation, network traffic generation, packet matching and its analysis. The proceeding is straight forward and the steps involved are discussed in upcoming sections.

3.7.1 Fingerprint Set Generation

A fingerprint set is a list/database containing fingerprints of ‘known files’ generated by approximate matching algorithm. Two separate fingerprint set are created for each algorithm. A Shell script is used to automate the processes of generating fingerprint sets. Each file from the corpus (RS or TS) is hashed using `mrsh-v2` and `sdhash` and the hash is added to the respective fingerprint set.

Listing 3.5 shows the commands used to generate fingerprint for a file using `mrsh-v2` and `sdhash`.

```
mrsh-v2: mrsh -p <directory> >> fingerprint.mrsh

sdhash: sdhash <filename> >> fingerprint.sdbf
```

Listing 3.5: `mrsh-v2` & `sdhash` commands to generate hash for a file.

All the fingerprints with `mrsh-v2` are generated with blocksize of 80 (see Sec. 2.3.5), as in a test scenario with 1000 random files, only 172,167 network packets are detected with blocksize of 160, while 180,509 packets are detected with blocksize of 80.

Encoded Traffic

In case for encoded traffic, a separate fingerprint set is to be created. During the testing, Base64 encoding is used. For creating the fingerprint set, each file is converted into Base64 format and this Base64 string is hashed using `mrsh-v2` and `sdhash`. It is observed that packets having Base64 encoded data have an additional newline character at each 72^{nd} position. In order to increase the detection rate of Base64 encoded files, newline character at every 72^{nd} is also added while creating fingerprint set for encoded traffic. The files are encoded to Base64 scheme using Python scripts.

3.7.2 Network Traffic Generation

As discussed in Sec. 3.3, network traffic is generated by starting protocol specific system daemon. The file is accessed by making protocol specific requests. In case of HTTP and FTP, requests are made using Linux's `curl` command. SMTP traffic is generated by using `mpack` command. The length of the content in the email body is 30 character for all tests. The commands used for generating the respective traffic are shown in Listing 3.6.

```
HTTP: curl ‘http://localhost/<file>’ > /tmp

FTP: curl -u <username:password> ‘ftp://localhost/<file_path>’ -o ‘/tmp’

SMTP: mpack -s <subject> -c <content type> <file> <email>

mpack -s "Matching" -c application/pdf 000010.pdf admin@localhost.localdomain
```

Listing 3.6: Commands to access HTTP, FTP and SMTP traffic.

To calculate true positive, false positive and false negative scores, the `sniffer` is executed for one file at a time and all the corresponding packets' statistics are written to a file. By doing so, it is ensured that network traffic of two files is not mixed.

3.7.3 Sniffing & Analysing Network Traffic

`sniffer` is responsible for sniffing network traffic, hashing the packets using either `mrsh-v2` or `sdhash` and comparing these network packets against the set of fingerprints of ‘known files’. Main settings and parameters required by `sniffer` are:

```
-a: Set algorithm mr=mrsh-v2 or sd=sdhash
-l: Input file containing list of hashed files
-o: Redirect the matching score output to a file
-O: read from offline packet dump
-w: write packets to file for offline use
-s: Get statistics of the session
-f: Filter in pcap format to filter sniffer packets.
-d: Network Interface to sniff packets from.
-t: threshold matching score
```

sniffer on startup, reads the input file containing the list of fingerprints of 'known files', created in Sec. 3.7.1, and loads the hashes into the RAM. When a network packet arrives, its fingerprint is created using the approximate matching algorithm specified during the startup. This generated fingerprint is compared against the hashes of 'known files'. If the match score is more than the specified threshold score, then the file detected is reported.

In order to keep the latency introduced by sniffer to minimum, the number of processed packets can be reduced by filtering the network traffic. sniffer supports *pcap filter format*[6] for filtering network traffic. While testing protocol specific packets like TCP's SYN, FIN and ACK packets, HTTP's GET requests are filtered out. Also, these small packets are unlikely to contain file payload.

All the match scores reported by the tool are redirected to a file. After completion of transmitting all the files in the corpus, the file containing the statistics is evaluated. Evaluation of such a file is discussed in the next section.

3.7.4 Result Compilation

The statistics file generated in the previous section is evaluated using Python scripts. A typical output of sniffer for a single packet looks like:

```
[*] Packet number:1
[*] Total packet:/corpus/t5/000109.jpg | whole_packet | 095
[*] IP Packet:/corpus/t5/000109.jpg | ip_packet | 097
[*] TCP Packet:/corpus/t5/000109.jpg | tcp_packet | 097
[*] Payload Packet:/corpus/t5/000109.jpg | payload_packet | 100
[*] Total Time: 0.030
```

The scripts reads output for each packet for all the files tested, and maintains score on per file basis as well as globally (all files together). The scripts calculates many parameters, some of the important one are: total number of packets encountered, match scores for all four network layers for each packet, processing time for each packet and average scores. The output of the analysis looks like:

```
THRESHOLD: 25
Total Files:3276
Total Files detected: 3276
```

[6] www.manpagez.com/man/7/pcap-filter (last accessed 2013-June-25).

```
Packets above threshold count 260230
total packet count: 290314
true positive packet count: 260111
false positive packet count: 41577
false true both count: 41458
false negative packet count: 30084
average true positive score: 95
average false positive score: 58
highest true positive score: 100
lowest true positive score: 26
highest false positive score: 100
lowest false positive score: 26
true positive percentage: 89.59
false positive percentage: 14.32
unique false positive percentage: 0.04
false negative percentage: 10.36
```

The parameters calculated above are discussed in detail in the next chapter.

Chapter 4

Experimental Results

This chapter discusses the results obtained after performing various tests to establish the feasibility and efficiency of the proposed approach. Initially, the terminology used in the rest of the chapter is described. Section 4.2 talks about the results obtained on using random data for emulating ‘known files’. In this section, feasibility of approximate matching based approach is established and also motivates which network layer and approximate matching algorithm to use.

Section 4.3 discusses about the results obtained while emulating real world scenario by using *TS* corpus. Section 4.4 and section 4.5 discusses the feasibility of detecting encoded traffic and embedded information using present approach respectively. While, in the last section stream based analysis approach is presented.

4.1 Terminology Used

Following terminology is used to describe the results obtained during the various tests [8].

True Positive (TP) is when a packet of a given file is detected correctly, i.e, approximate matching algorithm detects the file which is actually there in the packet.

False Positive (FP) is when approximate matching algorithm reports presence of another file in a packet of a given file. In simple words, false detection of a file in a packet.

False Negative (FN) is when approximate matching algorithm does not detect any file in a packet, though the packet is known to be of a file from the database.

Threshold (t) is the score to classify a match as true positive or false positive. As we only consider scores equal to or greater 25: $25 \leq t \leq 100$.

True positive rate (TPR) is the amount of network packets that yield a score $\geq t$ when comparing it against the original file digest.

False positive rate (FPR) is the amount of network packets yielding a score $\geq t$ but the packet match does not belong to the file. Note, that a false positive can have *1 or more* matches. For instance, a rate 4% means that 4% of the packets received *1 or more* matches with false files.

False positive only rate (FPoR) is the amount of network packets which have only false positives and no true positive.

False negative rate (FNR): is the amount of network packets yielding a score $< t$ but the packet actually belongs to a file.

4.2 Detection Rate with Random Data

In a first step, the packet analysis is performed using random data traffic, in order to establish the feasibility of the whole approach. Preparation and characteristics of random data corpus is discussed in Sec. 3.2. These random files are used to emulate the network traffic generated on transferring 'known files'. The network packet analysis is performed using both `mrsh-v2` and `sdhash` on all 4 TCP/IP network layers.

4.2.1 Detection Rate for Each Network Layer

A total of 189,509 packets are generated for 1000 random files transferred over the network while performing the test. Results are computed for all 4 network layers using both `mrsh-v2` and `sdhash` for each layer. The results obtained are summarised in Table 4.1 for mrsh-v2 and in Table 4.2 for sdhash. It can be clearly observed that, the average match scores increases on moving higher in the network protocol stack. The maximum average score is obtained when the matching is performed at the application layer. Secondly, the average scores are higher for mrsh-v2 than compared to sdhash.

A small modification is made to mrsh-v2 algorithm, the first and last bloom filter created for a packet are ignored while performing a match. By doing so, the application layer header does not influence the matching score. This slight modification increased the average scores.

The FNR is low for both the algorithms. The reason for packets going undetected, counted as false negatives, is the low entropy of the content in network packets. mrsh-v2 and sdhash detects unique features for performing match based on entropy in the content being analysed. If the entropy is low, the number of unique features detected will be less and while performing a match, a low score or no score is possible. An example of a file having low entropy content is documented in appendix C.

Table 4.1: Statistics for 4 network layers for mrsh-v2.

Layer	Avg. TP Score	Avg. FP Score	TPR	FPR	FNR
Ethernet	91	29	98.64	3.55	1.34
IP	91	29	98.60	3.59	1.35
TCP	92	29	98.60	3.79	1.39
Payload	95	29	98.48	4.28	1.51

Table 4.2: Statistics for 4 network layers for sdhash.

Layer	Avg. TP Score	Avg. FP Score	TPR	FPR	FNR
Ethernet	83	55	96.94	6.45	1.30
IP	87	57	97.15	6.60	1.08
TCP	88	57	97.18	6.72	1.02
Payload	89	60	97.09	5.85	2.12

True positive and false positive score distribution for mrsh-v2 is presented in Fig. 4.1 and for sdhash in Fig. 4.2. In case of mrsh-v2, around 80% of the packets having a true positive match give a score between 95-100 (78.83% have score=100), while for sdhash the true positive packets are distributed over the score >75 (without modification, mrsh-v2 have similar distribution like sdhash).

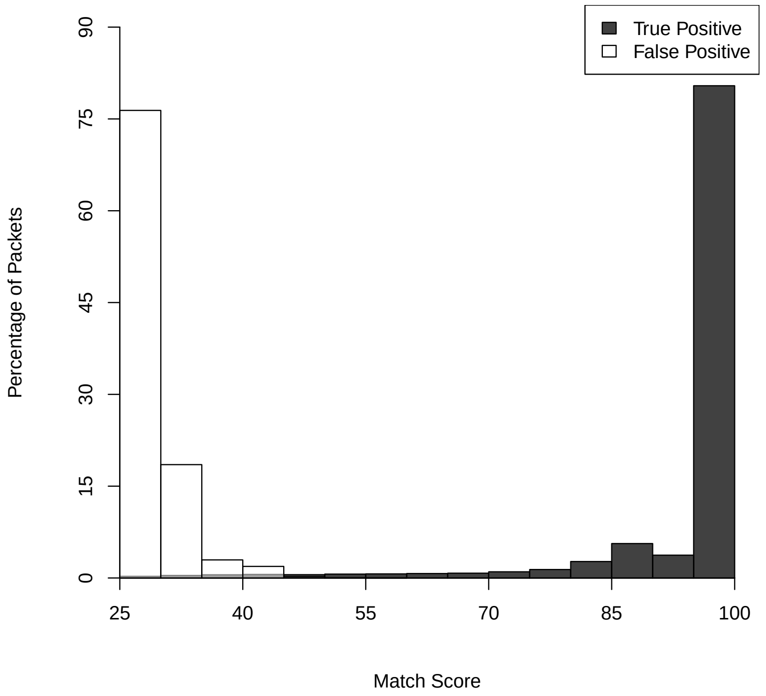

Figure 4.1: Score distribution for random traffic using `mrsh-v2`.

All the above discussion corresponds to detection of content in a single packet. It needs to be highlighted that all the files transferred during the experimentation, all of them are detected successfully. Where, a file is said to be detected, if at least one packet gives a match for that file.

4.2.2 Processing Time

The processing time (see Sec. 3.6) using `mrsh-v2` and `sdhash` is presented in Table 4.3. Processing time for link and application layers are only presented in the table, to keep it less verbose.

The data for both the algorithms is in agreement with the theory, that the processing time should increase as one moves up the network layers for performing

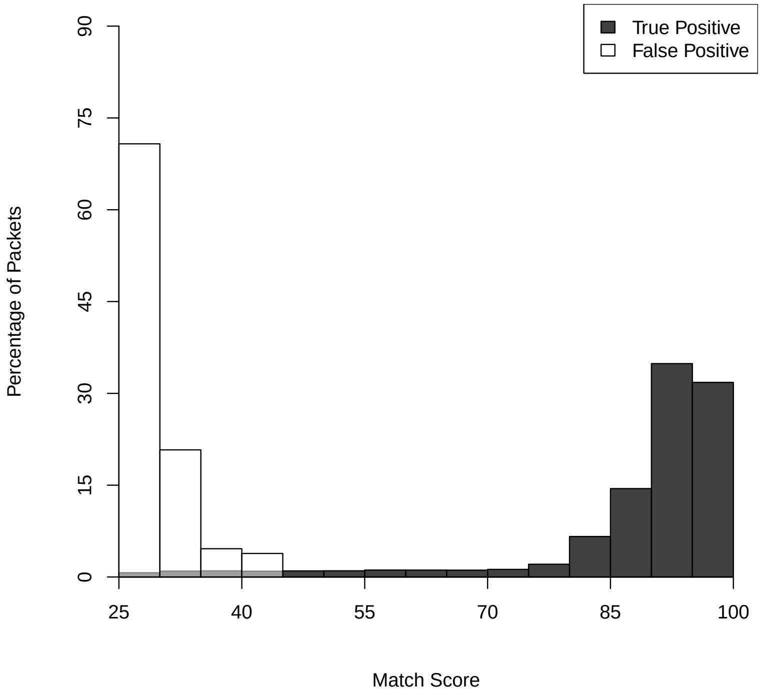

Figure 4.2: Score distribution for random traffic using `sdhash`.

a match. Though, the difference in processing time on two extreme layers (link and application) is very small. Also, it can be observed from the table that the processing time for `mrsh-v2` is about 5 times faster than `sdhash`, which is also the conclusion of [9].

The processing time is directly proportional to the number of fingerprints in the dataset to compare against. The larger the number of fingerprints, more processing time the matching will take. It is more elaborated in the next chapter.

4.2.3 Algorithm and Network Layer to Perform Matching

Latency refers to the delay incurred while processing the network data. In this case, latency is the difference between the time when the data has arrived, and

Table 4.3: Avg. processing time (in milliseconds) for `mrsh-v2` and `sdhash`.

Layer	Link	Payload
`mrsh-v2`	0.62	0.63
`sdhash`	3.33	3.36

to the moment it is available to the user. One of the main goals of the work is to keep the latency as minimum as possible.

In the previous section, it was concluded that processing time for `mrsh-v2` is faster than `sdhash`. Also, the average scores for `mrsh-v2` are higher than compared to `sdhash`. Considering the better performance of `mrsh-v2` over `sdhash`, `mrsh-v2` is used for performing further tests.

Also, the processing time for application layer is higher than the other lower layers, but the average scores are higher for the application layer. As a trade-off between the detection rate and processing time, it is decided to use application layer for packet matching. Also, at the application layer, the headers of other layers do not influence the match score and content anlaysis in true sense can be carried out.

Thus, the further results presented are with using `mrsh-v2` on the application layer. Also, in this section, research goals 1 and 2 are achieved.

4.3 Detection Rate with Real World Data

In the previous section, feasibility of the approach was established using random data. Also, it was decided to use `mrsh-v2` on application layer for performing packet matching. In this section, the testing is carried using the real world data (TS corpus) to emulate 'known file' network traffic.

A total of 290,314 packets are generated for 3282 files. A TPR of 86.69%, FPR of 14.32%, and FNR of 10.31% is obtained. A score distribution histogram is presented in Fig. 4.3. It is in agreement with the results obtained using random data. The frequency of true positives is dominated towards the right of the histogram. Also, the percentage of packets having true positive score in between 95-100 is also very high.

The FPR with real world data is higher than that with the random data. It is so, because of the existence of 'common subsequence'. For example, common headers of jpg files, or common headers of pdf files. Hexdump of a typical packet having header of a jpg is documented in appendix A. Such packets have more than one false positive match. Since, a jpg image can be present in a doc, a pdf or a ppt file, cross matching between file types is also observed.

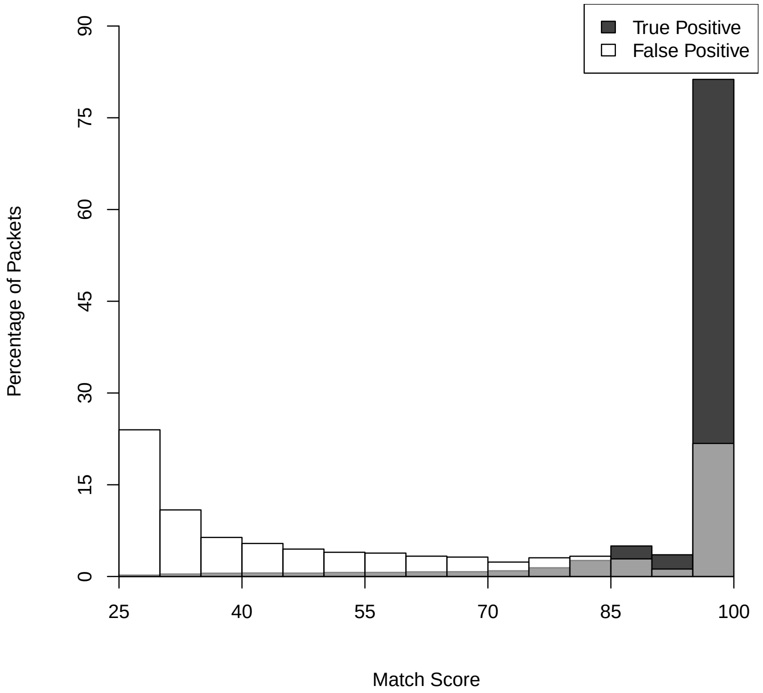

Figure 4.3: Score distribution for TS corpus traffic using `mrsh-v2` on application layer.

To identify the number of packets having false positive but no true positive, as 'common subsequence' give true positive as well, FPoR is used. FPoR gives an approximate idea of false positives caused without presence of 'common subsequence'. FPoR for present test scenario is 0.04%.

It is important to highlight that, as in case with random data, all files transmitted in this test scenarios are also successfully detected.

Table 4.4: Statistics of various file types in TS.

File Type	TPR	FPR	FPoR	FNR
jpg	96.92	11.85	0.02	3.05
gif	97.79	10.54	0.00	2.20
doc	74.11	14.96	0.09	25.80
xls	81.61	18.56	0.02	18.36
ppt	92.07	14.34	0.02	7.90
pdf	95.25	12.40	0.04	4.69
txt	87.63	14.80	0.08	12.35
exe	84.85	18.59	0.07	15.07
total	86.69	14.32	0.04	10.31

4.3.1 Results for Different File Types

As stated in previous section, all the files transmitted are detected successfully. In this section, detection rates for the various file types in TS are presented in order to identify which file types are best detected.

Though all the files are detected, but statistics for all the file types in the TS corpus are also calculated and compiled in Table 4.4, in order to identify which file types are easy to detect over the others. Pdf, jpg, gif have data in binary format and good TPR for them indicates that approximate matching approach is better than the existing ones. Existing approach are only good for detecting text data, this is discussed further in Chapter 5. Score distribution histograms for various file types are documented in appendix B.

4.3.2 Processing Time

In case of TS corpus, the number of fingerprints to compare against has increased to 3282. As per the expectation, the processing time is increased to 2.32 milliseconds.

4.3.3 Threshold Score

Till now the packets giving match score > 25 are only considered. From Fig. 4.3, it can be observed that for score > 85 the percentage of packets having false

Table 4.5: Statistics for Base64 encoded traffic.

Layer	TPR	FPR	FPoR	FNR
Payload	99.97	0.004	0.0002	0

positive is less than the percentage of packets having true positive. By increasing the match score's lower limit, the chances of encountering a true positive will be higher than a false positive. Also, the number of packets to process will reduce significantly and in turn helping to decrease the processing time. Section 4.6 talks about using stream based analysis in order to detect files and it uses the idea of higher threshold score just discussed.

4.4 Detection Rate for Encoded Traffic

Section 2.6 discusses the need for encoding network traffic. In an organisation there are plethora of applications used. Certain applications require to encode the data to work efficiently or are designed to use a specific encoding scheme. One such example is SMTP protocol. When the data is encoded, approximate matching approach is ineffective in detecting the 'known files'. In this section, a work around for such a scenario is proposed.

The naive approach is to identify the encoding used in the network traffic in real time and then decode the data and perform the match to identify known content. But this approach will be slow and processing intensive. To circumvent this problem, following approach is used and tested with SMTP protocol. A file is attached to the email and the attached file is encoded using Base64 scheme. The 'known files' are encoded using Base64 scheme and fingerprints are generated for each encoded file. The detection rate observed is shown in Table 4.5. The test comprised of only 500 files, randomly chosen from TS. All the 500 files are successfully detected using this approach.

In an organisation, it is trivial for an administrator to identify encoding scheme used by an application. The encoding specific fingerprint set can be created for the 'known files' and that can be used to detect data leakage by sniffing the application specific traffic.

4.5 Detection of Embedded Information

Approximate matching based approach can also be used for detecting embedded objects as well. An embedded file is a file that is hidden or store in another file. For example, a jpg file in a pdf or a video in a ppt. If the underlying byte structure of the embedded content is not significantly altered, i.e. the bit stream of the included content is unaltered, then it can be easily detected using the present approximate matching based approach.

To verify such a scenario, pdf and doc files are created by embedding a jpg file in each. These modified files are transmitted over the network and sniffed using the `sniffer`. The comparison is made against the fingerprint set of TS corpus. Notably, all the embedded files are successfully detected.

In a nutshell, if the underlying bit stream is not altered, then embedded files can be easily detected using approximate matching based approach. Detection of files in case of zipping them is out of the picture, as the raw byte structure is altered.

4.6 Stream Analysis

Till now, single packet analysis (SPA) approach is used for detecting files. In SPA, a file is declared to be detected if at least a single packet for a given file is matched. But there is a shortcoming to this approach. Due to ‘common subsequence’, many false positives are reported and thus causing a file to be falsely detected. To circumvent this problem, stream analysis (STA) is proposed.

STA determines the presence of a file in network traffic by considering match results from more than one packet of a connection stream. It is observed in SPA that, if a single packet gives a false positive, then that packet will give more than one false positive matches. The very low rate of FPoR supports this observation. This observation is used in defining two important parameters for STA. These parameters are defined below:

match_per_packet A threshold for the maximum number of matches a packet can have. If the number of matches are higher than this, then the packet is neglected. For instance, let $match\_per\_packet = 2$. Then, if a packet returns a true positive and a false positive, the match count is 2 for the packet and the packet is ignored.

packets_per_file A threshold for the minimum number of packets containing 'known file' content, required to declare a file to be detected. For instance, let *packets_per_file*=2. In network traffic, if 2 packets containing content of file A are detected, then this file A is declared to be detected.

The process for testing STA is similar to SPA. The files are sent over the network and the traffic is sniffed and analysed by the `sniffer`. In STA, a connection table is defined for each network connection stream. A connection table's row, is a tuple of *{Source IP, Destination IP, Source Port, Destination Port, Detection Status}*, where *Detection Status* is a table with each row is tuple of *{File Detected, packet count}*. Whenever a packet arrives, if the entry for the particular connection stream is not in the table, a new entry is added, else the corresponding entry in the table is updated. If the number of matches for a packet is more than *match_per_packet*, that packet is ignored. For a given file, if the number of matching packets = *packets_per_file*, the file is declared to be detected and all the counters are cleared for that connection stream.

In the test scenario, threshold match score = 100 is taken, as the number of matches with 100 are high (see Fig. 4.3). Also, a high confidence score reflects the certainty of the files presence. While *match_per_packet* = 5 and *packets_per_file* = 2 are taken. Motivation for taking *match_per_packet* = 5 is, in SPA it is observed that the minimum number of matches per packet is 1, followed by 5. Value 1 is true for all packets and hence 5 is taken.

Value of *packets_per_file* parameter depends on many factors. If the 'known file' database consists of many small sized files, then for a given file, the number of network packets generated are less. For instance, a file of 3 KB will generate only 2 packets of 1500 Bytes (ideally) size. If a database have similar files, then a packet will generate more than one false positive for one packet, though it is because of similar content rather than 'common subsequence'. Packets having high number of matches might be ignored and effectively reducing the number of packets available to detect a file. For instance, a file with size 4.5 KB will generate 3 packets of 1500 Bytes. If two packets match with more than one file, then detecting such file is difficult.

In the test scenario, $t = 100$ is taken. Traffic is generated for randomly selected 383 files from the TS corpus, with minimum file size of 4.8 KB. In this test scenario a true positive detection rate of 98.69% files is achieved.

STA has following advantages over SPA: Firstly, files can be detected with more confidence as false file detection, because of 'common subsequence', is avoided by using *match_per_packet* parameter. Secondly, as soon as a file is detected, the connection stream can be dropped, inhibiting the further file transfer. But this approach might not be able to detect small files, where the minimum file size depends on the parameter *packets_per_file* parameter.

All the testing for STA is done using offline packets, therefore exact processing time in live situation are is not available for this approach.

Chapter 5

Related Work

This chapter summarizes the related work for this thesis. Though not much previous work for data leakage prevention over an organisation's network is available. Most of the systems used are commercial and closed source, therefore information about their internal working, performance and detection rate are not available. Most of the information acquired is through the whitepapers published by McAfee and Securosis, which gives only an overview about the state of the art.

The organisation of the chapter is as following: Sec. 5.1 gives an overview of data leakage prevention systems and various content analysis technique used in them. These systems specialise in preventing data leakage and most relevant to the work proposed in this thesis.

Section 5.2 discusses how content matching is performed in intrusion detection systems. Though, these systems does not specialise in known content detection, but a simple string matching technique is used to detect malicious text/binary constructs in a network packet payload. A brief overview of signature matching is presented.

5.1 Data Loss Prevention Systems

The present work concentrates on the content analysis, thus in the following section a general overview of Data Loss Prevention Systems (DLPS) is given, while the various content analysis technique used by them is discussed in detail. For detailed information on DLPS, refer to [25]. Firstly, the three states of data are defined and this knowledge is further used to define DLPS.

5.1.1 States of Data

According to [29], the data of an organisation can be classified into 3 main states:

- **Data In Motion (DIM)** It comprises of data which is in the process of being transmitted over the network. It includes emails, instant messages, web traffic and so on.
- **Data In Use (DIU)** The data at a network endpoint, like desktop or USB device, comes under this category.
- **Data At Rest (DAR)** Information in the storage comprises DAR. For instance, data in FTP server or file systems.

The 3 states of data are also summarised in Fig. 5.1.

5.1.2 Definition

Data Loss Prevention Systems (DLPS) is a mechanism that identifies senstive information by content in DIM, DAR, or DIU, and prevent leakage to outside of an organisation [29].

DLPS helps organisations to comply with government regulations pertaining to privacy, the protection of sensitive data, and the maintenance of records. According to [25], key defining characteristics of DLPS are :

- Deep content analysis,
- Central policy management,

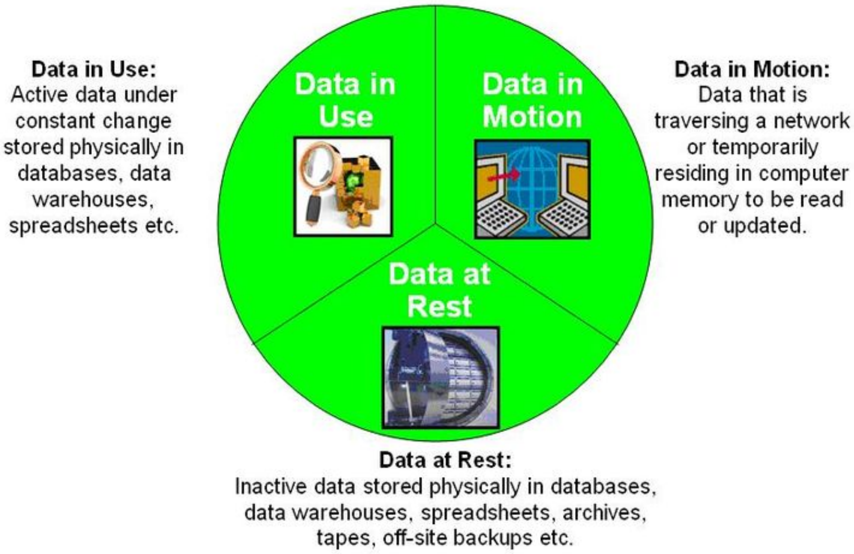
Data in Use:
Active data under constant change stored physically in databases, data warehouses, spreadsheets etc.
Data in Use
Data in Motion
Data in Motion:
Data that is traversing a network or temporarily residing in computer memory to be read or updated.
Data at Rest
Data at Rest:
Inactive data stored physically in databases, data warehouses, spreadsheets, archives, tapes, off-site backups etc.

Figure 5.1: The 3 states of data[a].

[a] `https://en.wikipedia.org/wiki/Data_at_Rest` (last accessed 2013-June-25).

- Broad content coverage across multiple platforms and locations.

By defining policies, a DLPS can automatically ensures that no sensitive data is stored, sent or accessed by an unauthorized person. Unlike white and blacklisting, DLPS blocks the activities involving leakage of sensitive data.

5.1.3 Content Analysis Techniques

In this thesis, various content analysis techniques used by DLPS are analysed and compared against the proposed approximate matching based approach. According to [25], there are following approaches used in DLPS for content analysis:

1. **Rule-Based/Regular Expressions:** This is the most common analysis technique offered by various content discovery tools. In this technique, the content is analysed for specific rules, e.g, 16 digits credit card numbers, social security numbers etc.

Best for: For detecting structured data like credit card numbers, healthcare codes/records. It is generally the first filter which is used while filtering data.

Strengths: Regular expression rules can be processed quickly and also easy to configure. It is a widely used technique, hence well understood and also easy to incorporate into diverse products.

Weakness: False positive rate is high. Also, inability to identify unstructured data.

2. **Database Fingerprinting:** In this technique, database dump or live data from database is used and only exact matches are reported. For example, a policy which looks only for credit card number in the organisation's customer base, thus ignoring employees of the organisation while buying online.

   Best for: This technique is best used for detecting structured data from databases.

   Strengths: A very low false positive rate is offered, as only exact matches are reported. This also enables protection of customer/sensitive data while ignoring other similar data used by the employees of the organisation.

   Weakness: Nightly dumps of the database won't contain transaction data since the last extract. On the other hand, live connection to the database will negatively effect the performance.

3. **Exact File Matching:** In this technique, cryptographic hash functions are used to generate fingerprint of the sensitive files. The content being monitored is also hashed using the same hash function, if the fingerprints match exactly, then the activity is reported.

   Best for: This technique is best used for media files or binaries where textual analysis is not feasible.

   Strengths: It is agnostic to file type, and have negligible false positive rate.

   Weakness: Evading such monitoring is trivial. Flipping a single byte in the file will change the fingerprint of the file, thus making it worthless for editable files like office documents and text files.

4. **Partial Document Matching (PDM):** This technique looks for a complete or a partial match of sensitive content. *Rolling hash* technique is often used and was discussed in Sec. 2.3.3. When outbound content is encountered, it is run through the same hash technique, and the hash values are compared for matches.

   Best for: This technique is useful for protecting unstructured content. For example, CAD files and source code files.

Strength: Effectively protect unstructured data, and have a low false positive rate. Doesn't rely on complete matching of large documents; can find policy violations on even a partial match.

Weakness: Common phrases in a protected document may trigger false positives. Trivial to avoid (ROT1) encryption is sufficient. It is not most efficient in detecting non-text files.

5. **Statistical Analysis:** Machine learning, Bayesian analysis and other statistical techniques are used to analyse the content and determine possible policy violations.

   Best for: Unstructured content where deterministic techniques, like partial document matching, would be ineffective. For example, a repository of engineering plans where partial document matching will be impractical because of high volatility or massive volume.

   Weakness: It is prone to false positives and false negatives. Also, requires a large corpus of source content.

6. **Conceptual/Lexicon:** This technique uses a combination of dictionaries, rules, and other analyses techniques to protect nebulous content that resembles an "idea". For example, traffic resembling to insider trading, sexual harassment, running a private business from a work account.

   Best for: Completely unstructured ideas that cannot be categorized into known documents, databases or other registered sources.

   Strength: Can be used to detect content with loosely defined policy violations.

   Weakness: In general, it is really hard to define such rule-set and requires considerable time and effort to develop a rule-set. It is very prone to high false positives.

7. **Categories:** Categories, also known as compliance templates, use combination of above described methods to detect certain types of content. For example, 1 credit card number + 1 partial date + expiry date keywords. If such a combination of information is find in the outbound data, the message is flagged.

   Best for: Any type of content which can be described as a category. Useful in case of content related to privacy regulations, or industry-specific guidelines.

   Strengths: It is extremely simple to configure and also saves considerable policy generation time.

   Weakness: It is only good for easily categorized rules and content.

The PDM technique is the most relevant of them all, and its comparison with approximate matching based approach is discussed in next chapter.

5.2 Intrusion Detection Systems

An intrusion detection system (IDS) inspects all the inbound and outbound network activity and identifies suspicious patterns that may indicate an attack on the network as a whole or on a system in the network. IDS help information systems prepare for, and deal with attacks. This is achieved by collecting information from various systems and network sources, and then analysing the information for possible security problems. In short, IDS acts like a 'burglar alarm'.

In this section, only the features which enable IDS to detect network packet content are discussed. For detailed information of IDS please refer to [15].

Network Intrusion Detection Systems (NIDS) are IDS which perform analysis for the passing traffic on the entire subnet. NIDS work in promiscuous mode, and matches the traffic that is passed on the subnets to the library of known attacks. *Snort* is one of the most popular NIDS. It is a cross-platform, lightweight IDS tool that can be deployed to monitor small TCP/IP networks and detect a wide variety of suspicious network traffic as well as outright attacks [39].

Whenever NIDS encounter a network packet, it performs a match against a predefined signature set in order to determine whether the packet is malicious or not. This step is called signature matching. Signature matching consists of two distinct operations: *packet classification*, which involves examining the values of a packet header fields, and *deep packet inspection*, in which the packet payload is matched against a set of predefined patterns.

Signatures for *snort* are specified using simple rule-based language. *Payload detection rule options* are the most relevant to content matching. Specifically, *content* keyword is the most interesting of them all.

By using *content* keyword, *snort* checks the payload content. "Whenever *content* options pattern match is performed, the Boyer-Moore pattern match function is called and the test is performed against the packet content" [38]. Such a test is successful only when the argument data string matches exactly with content in the packet's payload. *snort* can match both text and binary content.

Listing 5.1 shows a snort rule using *content* keyword. In this particular snort rule, *content* keyword argument is binary data and ensuring the detection of PNG file's 'magic' sequence.

```
alert tcp $EXTERNAL_NET $FILE_DATA_PORTS -> $HOME_NET any (msg:"FILE-IDENTIFY PNG
      file magic detection"; flow:to_client,established; file_data; content:"|89|
     PNG|0D 0A 1A 0A|"; within:8; fast_pattern; flowbits:set,http.png; flowbits:
     noalert; classtype:misc-activity; sid:20478; rev:1;)
```

Listing 5.1: Snort rule using content keyword.

Using *content* keyword for content detection is only good for matching small text or binary data, for example detecting a malicious string construct in a file transferred over the network. IDS does not specialize in preventing data leakage from a database of files. Detecting large file content will make the rules complicated and also severely slow down the processing speed.

5.3 Comparison

This section discusses how the approximate matching based approach compare against the other existing approaches. A comparison is made against the rolling hash approach and keyword based approach. The other methods, like using cryptographic hash functions and regular expressions are either easy to evade or not meant for detecting leakage of a document as a whole, therefore not discussed in this section.

5.3.1 Rolling Hash

Partial Document Matching (PDM) technique discussed in Sec. 5.1.3, internally uses rolling hash to compare documents and detect leakage. An introduction to rolling hash is presented in Sec. 2.3.3. As mentioned earlier, most of the DLPS are commercial and closed source and thus not considered for testing purposes. MyDLP[1] is an open source DLPS. To test detection rate of rolling hash, MyDLP Community Edition is used.

MyDLP is installed on a server and this server is used as a network proxy. All the traffic pases through this proxy and analysed by MyDLP for potential data leakage. For testing purposes, MyDLP is installed on a Ubuntu Server running in a Virtual Machine (VM). All the test traffic is diverted through this VM in

[1] `http://www.mydlp.org` (last accessed 2013-June-25).

order to be analysed by MyDLP. Though, post installation, the tool did not work smoothly and frequently broke down and required re-installation to make it work again.

MyDLP's PDM feature is only tested. Procedure to use PDM is described in [32]. During the test it is observed that MyDLP can only detect files which contain texts, like pdf, doc, txt, and not good for preventing leakage through other binary formats like, exe, jpg, gif etc. Rigorous tests to establish detection rate cannot be performed on MyDLP, as there were problems faced in running the tool reliably.

Drawbacks of Rolling Hash

Rolling hash based approach suffers with certain drawbacks. These drawbacks and how rolling hash compare against approximate matching based appraoch is disccused below.

For a given file there will be large number of rolling hashes generated. The number of hashes generated for a file depends on the size of the file and the window size used to generate rolling hashes. Due to large number of hashes generated for a given file, the number of lookups to perform a comparison is also large and thus slowing down the comparison process. Also, rolling hashes occupy large memory as one file have many hashes. Consequently, for a large database of confidential files, it is possible that all the rolling hashes cannot be loaded into RAM simultaneously and might effect the overall performance negatively.

In case of xls and doc files, it is observed that they contain large sequences of 0 and 1, called 0 run and 1 run respectively. With rolling hash, such runs could give large number of false positives. Hex dump of a xls having 0 and 1 run is documented in appendix C.

On the other hand, in approximate matching based approach, only one fingerprint is generated for one file or packet. Thus, requiring significantly less number of lookups to perform a comparison. Also, the length of a fingerprint of a file is small and can be easily loaded into the RAM and perform quick matches. Lastly, approximate matching algorithms detect unique features for a digital object by determining the entropy of the content. Thus, 0 and 1 run in doc and xls file would not cause false positive matches in this approach.

5.3.2 Keyword Matching

In this approach, important words in a document are determined by using some machine learning algorithms. These words from various files in the database are put together to make one big dictionary. While analysing network packets, if words from this dictionary occurs, then that connection is declared suspicious.

Drawbacks

Maintaining such a word dictionary is cumbersome. Adding and deleting a file is not straightforward, as the corresponding words need to be added or deleted from the dictionary. Also, a word in the dictionary could occur both in confidential and junk file simultaneously and thus can have high false positive rate. In case of approach proposed in this thesis, there is no dictionary required. Only a list of fingerprints for the files to protected is maintained. The false positive rate, as observed in the previous chapter is not so high.

To summarise, approximate matching approach is better than existing approaches as it is file type agnostic and easy to maintain. One file will have only one fingerprint and thus a file can be simply added or deleted by adding or deleting its corresponding fingerprint from the database. Also, calculating a fingerprint of a file is effortless.

Chapter 6

Discussion

This chapter discusses some general issues about the proposed approach. Initially, a discussion about the detection rate of the approach is presented. In the next section, classification of the approach based on level of packet analysis is discussed.

Section 6.3 looks into the statistics of processing time using `mrsh-v2` for packet analysis. A difference between content and context analysis is given in section 6.4 and elucidates that the present approach performs content analysis.

Section 6.5 discusses how to block the traffic once 'known file' content is detected in the network traffic. Whereas, Sec. 6.6 looks into the problem of how to successfully deal with encrypted traffic and ensure that protected data is not leaked.

Approximate matching based approach is an ingenuous way to deal with data leakage. As highlighted in Chapter 5, that the existing approaches are mostly commercial and closed source, and there exist not much prior work in detecting 'known file' in the network traffic robustly. The proposed approach is one of the first comprehensive work to deal with problem of data leakage.

Many DLPS uses cryptographic and rolling hash functions to check for data leakage. As mentioned previously, it is trivial to evade crypto hash based checking. Rolling hash holds some promise to acknowledge the problems with other approaches, but it suffers with certain drawbacks as well. Though, only MyDLP could be analysed for performance of rolling hash, which showed that not all file types can be detected using rolling hash. For instance, doc and pdf are detected but jpg and exe are not. Also, with rolling hash the number of look-ups performed in order to detect a match are high.

6.1 Detection Rate

The eventual goal of a data leakage prevention system is to stop the transfer of confidential information out of the network. In this work, transferring of file as a whole is considered. To ensure the detection rate is high, comparison is performed only for the application layer data and `mrsh-v2` is preferred over `sdhash`. Also, `mrsh-v2` is modified to ensure that application layer header does not influence the match score. During the testing with random data and real world data using SPA, all the files are detected successfully. While in STA, the detection rate of 98.69% is achieved. The low detection rate for STA is due to the presence of small files in the database and thus not generating enough packets to perform analyses. Since, none of the existing tools could be analysed thoroughly, the detection rate of them is not available. Consequently, a relative comparison of false detection rate cannot be made. Thus, the research goal 5 is considered to be partially achieved.

6.2 Level of Packet Inspection

The proposed approach analyses the content of the network packet, to determine whether ‘known file’ is being transmitted in the communication stream under investigation. As per the classification of packet inspection technology discussed in Sec. 2.5, present approach can be classified as *deep packet inspection* technique. The lower layer headers are used to get the source and destination IP address and ports and this information is used to maintain a table of ‘known files’ detected in the connection. While, lower layer headers - link, IP and TCP layer, are completely stripped to perform content analysis.

6.3 Processing Time

Major constituent of processing time in present approach is the time spent in iterating over the list of fingerprints to compare a given network packet. While analysing processing time for 61,700 network packets with `mrsh-v2` using *gprof*, it is found that 99.8% is spent comparing a network packet against the list of fingerprints (3280), while only 0.2% of the time is spent in hashing a network packet. If the number of fingerprints of 'known files' increases, then the processing time will also increase.

If it is possible to index `mrsh-v2` fingerprints, then the processing time can be reduced in comparing fingerprints. An `mrsh-v2` fingerprint is simply the Bloom filters' value, generated for the given file. At present there is no known way to index this information. This could be part of the future work.

6.4 Content vs. Context

In order to understand the intricacy of content and context, example of a letter and its envelope can be used. In such a case, the letter will be the content, whereas the envelope and environment around it will constitute the context. Context is inferred using the source, destination, size, recipients, sender, header information, metadata, time, format etc. Contextual analysis is a highly useful approach, like in detecting insider trading attempts.

While content analysis involves looking inside containers and analysing the content itself. Use of content analysis does not restrict the analysis to a certain specified context. If a data is declared to be confidential, it should be protected everywhere - not just in obviously sensitive container [25].

Content analysis is more difficult and time consuming than basic contextual analysis. In this thesis, the approach proposed performs a bitwise content analysis to detect the 'known file' and the results observed are promising.

6.5 Filtering/Blocking Traffic

The sole aim of using any data leakage tool is to block the traffic which is potentially leaking data. Most of the communication performed is synchronous and in real time. Thus actively monitoring the data and blocking is of utmost importance. In this section, some of the probable approaches are discussed which can be used to block traffic.

Bridge

A *bridge* is a device that connects two or more local area networks, or two or more segments of the same network [45]. A bridge can be used to filter network traffic. With a bridge, there is a system with two network cards which performs content analysis in the middle. If there is some malicious content (as per policy) being transferred, the bridge is capable to break the connection for that session. Bridging can be used in the present scenario for blocking traffic when known content is detected. But it suffers with a small drawback, i.e, it might not stop all the bad traffic before it leaks out the content. By the time content analyser gets enough traffic to make an intelligent decision, the good part of the content might have already been transferred. Considering the above shortcoming, it is not the best possible solution, but it holds the advantage that such an arrangement is protocol agnostic [25].

TCP Poisoning

Another possible approach to block or filter traffic is to use TCP poisoning. In TCP poisoning, TCP reset flag is used to terminate the connection. TCP reset flag is mostly set to 0 and has no effect, but when this bit is set to 1, it indicates the receiving endpoint should immediately halt using the TCP connection. The further packet received for the corresponding port number are also discarded by the end system. In a nutshell, TCP reset kills a TCP connection instantly [36].

The traffic is constantly monitored and as soon as data transfer against the policy is detected, the connection is terminated by sending TCP reset. This solution is simple to use, but have some disadvantages. Firstly, this works only for protocols based on TCP. Secondly, it is inefficient in case of protocols which keep trying to get the traffic through after a failed try. For instance, after TCP poisoning a single email message, the email server will keep trying to send it for 3 days, as often as every 15 minutes (as per settings). Lastly, the same issue as with bridging. By the time some nefarious activity is detected, some part of the traffic has already passed through.

Proxy

Each network packet can broadly be divided into address area and data area. Where, data area contains information written by the application program that created the packet and address area contains information to ensure that the packet is delivered to the right system and right application in the system [45]. Proxy servers (or application-level gateway) operate by examining incoming or outgoing packets not only for their source or destination addresses but also for information carried within the data area (as opposed to the address area) of each network packet. A proxy could be protocol/application specific and queues up network traffic before passing it and in turn allowing for deeper analysis of the traffic [25]. Since the traffic is being queued and analysed, it ensures that no part of the data is leaked in case some information is being leaked out.

Such gateway proxies can also be used as a reverse SSL proxy to sniff encrypted connections and discussed in detail in the next section.

6.6 Dealing with Encrypted Traffic

Many organisations in order to ensure a secure communication over the network use encryption. HTTPS, SSH, SFTP etc are some widely used protocols which support encryption. On an encrypted channel, the content is scrambled and the underlying bit stream is altered. In such a situation, detecting and preventing egress of ‘known file’ is difficult, unless the content is decrypted.

To circumvent this problem, man-in-the-middle (MITM) approach can be deployed. A proxy server is used to intercept the communication, which acts as a reverse proxy and launches a man-in-the-middle attack. Such an arrangement is well documented in case of HTTPS [12]. In an organisation environment, network administrator have enough authority to setup a reverse proxy server for intercepting encrypted traffic and checking network traffic content for known data.

In a desktop environment, MITM approach to intercept encrypted traffic can be tested using OWASP WebScarab[1]. WebScarab is a framework for analysing applications that communicate using HTTP and HTTPS protocol and supports MITM for HTTPS.

[1] `www.owasp.org/index.php/OWASP_WebScarab` (last accessed 2013-June-25).

6.7 Hardware Implementation

Implementing the traditional cryptographic hash functions into hardware is a common practice these days. Performing hashing in a dedicated hardware is more efficient than doing to on a CPU. `mrsh-v2` performs many low level operations, like bit shifting, to generate fingerprint of a content. Also, there exist hardwares which maintain network connection tables, e.g. routers. Thus, in theory it is possible to implement the proposed arrangement at the hardware level. This will significantly reduce the processing time. This could be an interesting future work to test for.

6.8 Limitation

During the design, implementation and testing phase no major limitations have been encountered for the proposed approach. Some minor limitations are enumerated below:

- Approximate matching based approach cannot detect a confidential file zipped and then transferred over the network. As discussed earlier, if the underlying bit stream is changed, approximate matching algorithms cannot detect the content.
- Malicious insider can take snapshots of the confidential data and transfer these snapshots. The present approach can not stop such data leakage attempt.
- If the minimum file size in the 'known file' database is small, then the file detection using stream based analysis might not always be reliable, as discussed in Sec. 4.6.
- The surveillance can be evaded by sending out files by encoding them, like using Base64 scheme. Since, the underlying original bitstream is altered by doing so. If encoding is being performed by some application, then such a situation can be dealt easily in an orgnisation's environment. Administrators can sniff the application specific traffic and check it against the 'known file' fingerprint set created for encoded traffic, discussed in Sec. 3.7.1.

Chapter 7

Conclusion & Future Work

The problem of data loss has become an important problem and a robust solution is need of the hour. Possible routes of data loss have become complicated and numerous, making countermeasures difficult to develop and deploy. The increased incidents of involvement of insiders in data leakage has raised a serious question on confidentiality of organisation's internal information, like intellectual property. In this work, the problem of identifying files in network traffic is considered. The problem with the existing technology is highlighted and need of open source tools and techniques needed to solve this problem is emphasised.

In order to solve this problem, bitwise content analysis of data in motion using approximate matching is proposed. Each packet is analysed for containing the 'known file'. It is successfully established that it is possible to detect files using this approach. To validate the technique and implementation, several scenarios are considered and tested. In a first step, random data is used to explore feasibility and establish a benchmark for what to expect from such a methodology. The tests with real world data showed promising results as well. Both binary and text based files can be easily detected using this approach. However, with real world data, problem of 'common substrings' persists. Wherefore, a easy extension is proposed of using stream based analysis.

In stream based analysis, a table is maintained for each connection stream. After analysing the results of single packet based approach, certain parameters are determined in order to prevent false positives caused by 'common substrings'. In this approach, a connection table of tuple: *{Source IP, Destination IP, Source Port, Destination Port, Detection Status}*, is maintained for each stream. After detecting a certain pre-defined number of packets, a file is declared to be detected in that stream. In such a case, the connection can be terminated immediately.

Compared to existing techniques, approach presented in this work is simple, straight forward and application layer protocol agnostic. The detection rate does not vary much for different file types and during testing all the files of various file types were detected successfully. In theory, the present approach is file type independent. Also, using such a tool is simple, as only the fingerprint of the file to be protected is needed. Thus, adding or deleting a file from the database is not cumbersome.

Future Work

In order to promote this approach further, there are several next steps. Some of the possible future work is enumerated below:

- A detail analysis of performance in high bandwidth network is required. Does in such situation, performance is comparable to existing methods? To enhance the performance, implementing this solution in hardware could be an alternative and was also discussed in previous chapter.
- What is the detection rate in case the database has similar files? How can the stream based analysis parameters be determined so that the detection of content is accurate.
- How accurate is this approach in detecting information leakage if partial content of a file is being transmitted.
- Analysis of commercial DLPS products and how the proposed approach fare against them.
- Can `mrsh-v2` fingerprints be indexed and in turn reduce the processing times?
- Finally, is it possible to tune approximate matching algorithms further, in order to receive better results?

Appendix A

Common Subsequence String

Hex dump of a packet containing ‘common subsequence’ and thus generates more than one false positive.

```
0000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 0000 5859 5a20 0000  ..........XYZ ..
0000020: 0000 0000 f351 0001 0000 0001 16cc 5859  .....Q........XY
0000030: 5a20 0000 0000 0000 0000 0000 0000 0000  Z ..............
0000040: 0000 5859 5a20 0000 0000 0000 6fa2 0000  ..XYZ ......o...
0000050: 38f5 0000 0390 5859 5a20 0000 0000 0000  8.....XYZ ......
0000060: 6299 0000 b785 0000 18da 5859 5a20 0000  b.........XYZ ..
0000070: 0000 0000 24a0 0000 0f84 0000 b6cf 6465  ....$.........de
0000080: 7363 0000 0000 0000 0016 4945 4320 6874  sc........IEC ht
0000090: 7470 3a2f 2f77 7777 2e69 6563 2e63 6800  tp://www.iec.ch.
00000a0: 0000 0000 0000 0000 0000 1649 4543 2068  ...........IEC h
00000b0: 7474 703a 2f2f 7777 772e 6965 632e 6368  ttp://www.iec.ch
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 6465  ..............de
00000f0: 7363 0000 0000 0000 002e 4945 4320 3631  sc........IEC 61
0000100: 3936 362d 322e 3120 4465 6661 756c 7420  966-2.1 Default
0000110: 5247 4220 636f 6c6f 7572 2073 7061 6365  RGB colour space
0000120: 202d 2073 5247 4200 0000 0000 0000 0000   - sRGB.........
0000130: 0000 2e49 4543 2036 3139 3636 2d32 2e31  ...IEC 61966-2.1
```

```
0000140: 2044 6566 6175 6c74 2052 4742 2063 6f6c   Default RGB col
0000150: 6f75 7220 7370 6163 6520 2d20 7352 4742  our space - sRGB
0000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000170: 0000 0000 0000 6465 7363 0000 0000 0000  ......desc......
0000180: 002c 5265 6665 7265 6e63 6520 5669 6577  .,Reference View
0000190: 696e 6720 436f 6e64 6974 696f 6e20 696e  ing Condition in
00001a0: 2049 4543 3631 3936 362d 322e 3100 0000   IEC61966-2.1...
00001b0: 0000 0000 0000 0000 2c52 6566 6572 656e  ........,Referen
00001c0: 6365 2056 6965 7769 6e67 2043 6f6e 6469  ce Viewing Condi
00001d0: 7469 6f6e 2069 6e20 4945 4336 3139 3636  tion in IEC61966
00001e0: 2d32 2e31 0000 0000 0000 0000 0000 0000  -2.1............
00001f0: 0000 0000 0000 0000 0000 0000 0000 7669  ..............vi
0000200: 6577 0000 0000 0013 a4fe 0014 5f2e 0010  ew.........._...
0000210: cf14 0003 edcc 0004 130b 0003 5c9e 0000  ............\...
0000220: 0001 5859 5a20 0000 0000 004c 0956 0050  ..XYZ .....L.V.P
0000230: 0000 0057 1fe7 6d65 6173 0000 0000 0000  ...W..meas......
0000240: 0001 0000 0000 0000 0000 0000 0000 0000  ................
0000250: 0000 0000 028f 0000 0002 7369 6720 0000  ..........sig ..
0000260: 0000 4352 5420 6375 7276 0000 0000 0000  ..CRT curv......
0000270: 0400 0000 0005 000a 000f 0014 0019 001e  ................
0000280: 0023 0028 002d 0032 0037 003b 0040 0045  .#.(.-.2.7.;.@.E
0000290: 004a 004f 0054 0059 005e 0063 0068 006d  .J.O.T.Y.^.c.h.m
00002a0: 0072 0077 007c 0081 0086 008b 0090 0095  .r.w.|..........
00002b0: 009a 009f 00a4 00a9 00ae 00b2 00b7 00bc  ................
00002c0: 00c1 00c6 00cb 00d0 00d5 00db 00e0 00e5  ................
00002d0: 00eb 00f0 00f6 00fb 0101 0107 010d 0113  ................
00002e0: 0119 011f 0125 012b 0132 0138 013e 0145  .....%.+.2.8.>.E
00002f0: 014c 0152 0159 0160 0167 016e 0175 017c  .L.R.Y.`.g.n.u.|
0000300: 0183 018b 0192 019a 01a1 01a9 01b1 01b9  ................
0000310: 01c1 01c9 01d1 01d9 01e1 01e9 01f2 01fa  ................
0000320: 0203 020c 0214 021d 0226 022f 0238 0241  .........&./.8.A
0000330: 024b 0254 025d 0267 0271 027a 0284 028e  .K.T.].g.q.z....
0000340: 0298 02a2 02ac 02b6 02c1 02cb 02d5 02e0  ................
0000350: 02eb 02f5 0300 030b 0316 0321 032d 0338  ...........!.-.8
0000360: 0343 034f 035a 0366 0372 037e 038a 0396  .C.O.Z.f.r.~....
0000370: 03a2 03ae 03ba 03c7 03d3 03e0 03ec 03f9  ................
0000380: 0406 0413 0420 042d 043b 0448 0455 0463  ..... .-.;.H.U.c
0000390: 0471 047e 048c 049a 04a8 04b6 04c4 04d3  .q.~............
00003a0: 04e1 04f0 04fe 050d 051c 052b 053a 0549  ...........+.:.I
00003b0: 0558 0567 0577 0586 0596 05a6 05b5 05c5  .X.g.w..........
00003c0: 05d5 05e5 05f6 0606 0616 0627 0637 0648  ...........'.7.H
00003d0: 0659 066a 067b 068c 069d 06af 06c0 06d1  .Y.j.{..........
00003e0: 06e3 06f5 0707 0719 072b 073d 074f 0761  .........+.=.O.a
00003f0: 0774 0786 0799 07ac 07bf 07d2 07e5 07f8  .t..............
0000400: 080b 081f 0832 0846 085a 086e 0882 0896  .....2.F.Z.n....
0000410: 08aa 08be 08d2 08e7 08fb 0910 0925 093a  .............%.:
0000420: 094f 0964 0979 098f 09a4 09ba 09cf 09e5  .O.d.y..........
0000430: 09fb 0a11 0a27 0a3d 0a54 0a6a 0a81 0a98  .....'.=.T.j....
0000440: 0aae 0ac5 0adc 0af3 0b0b 0b22 0b39 0b51  ..........".9.Q
0000450: 0b69 0b80 0b98 0bb0 0bc8 0be1 0bf9 0c12  .i..............
0000460: 0c2a 0c43 0c5c 0c75 0c8e 0ca7 0cc0 0cd9  .*.C.\.u........
0000470: 0cf3 0d0d 0d26 0d40 0d5a 0d74 0d8e 0da9  .....&.@.Z.t....
```

```
0000480: 0dc3 0dde 0df8 0e13 0e2e 0e49 0e64 0e7f  ...........I.d..
0000490: 0e9b 0eb6 0ed2 0eee 0f09 0f25 0f41 0f5e  ...........%.A.^
00004a0: 0f7a 0f96 0fb3 0fcf 0fec 1009 1026 1043  .z...........&.C
00004b0: 1061 107e 109b 10b9 10d7 10f5 1113 1131  .a.~...........1
00004c0: 114f 116d 118c 11aa 11c9 11e8 1207 1226  .O.m...........&
00004d0: 1245 1264 1284 12a3 12c3 12e3 1303 1323  .E.d...........#
00004e0: 1343 1363 1383 13a4 13c5 13e5 1406 1427  .C.c...........’
00004f0: 1449 146a 148b 14ad 14ce 14f0 1512 1534  .I.j...........4
0000500: 1556 1578 159b 15bd 15e0 1603 1626 1649  .V.x.........&.I
0000510: 166c 168f 16b2 16d6 16fa 171d 1741 1765  .l...........A.e
0000520: 1789 17ae 17d2 17f7 181b 1840 1865 188a  ...........@.e..
0000530: 18af 18d5 18fa 1920 1945 196b 1991 19b7  ....... .E.k....
0000540: 19dd 1a04 1a2a 1a51 1a77 1a9e 1ac5 1aec  .....*.Q.w......
0000550: 1b14 1b3b 1b63 1b8a 1bb2 1bda 1c02 1c2a  ...;.c.........*
0000560: 1c52 1c7b 1ca3 1ccc 1cf5 1d1e 1d47 1d70  .R.{.........G.p
0000570: 1d99 1dc3 1dec 1e16 1e40 1e6a 1e94 1ebe  .........@.j....
0000580: 1ee9 1f13 1f3e 1f69 1f94 1fbf 1fea 2015  .....>.i...... .
0000590: 2041 206c 2098 20c4 20f0 211c 2148 2175   A l . . .!.!H!u
00005a0: 21a1 21ce 21fb 2227                      !.!.!."’
```

Appendix B

Score Distribution for File Types Using `mrsh-v2`

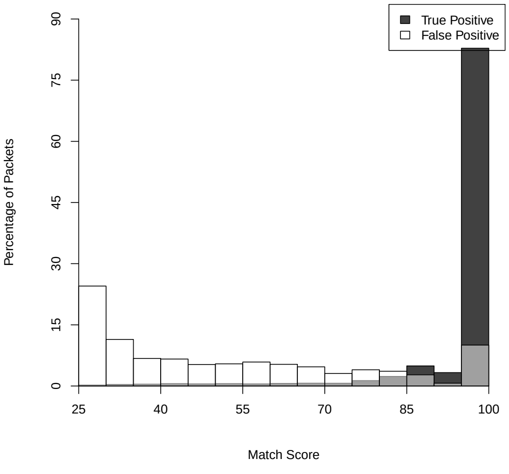

Figure B.1: Score distribution for doc file types using `mrsh-v2`.

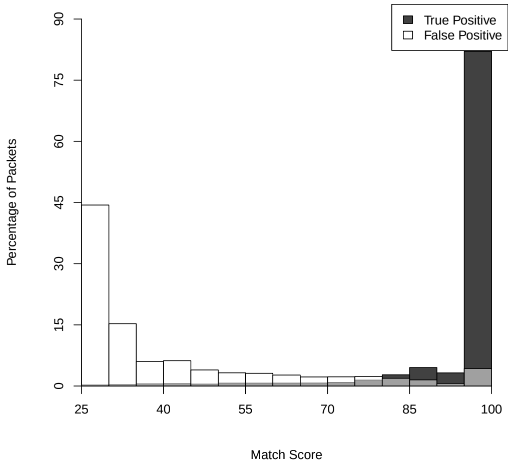

Figure B.2: Score distribution for exe file types using `mrsh-v2`.

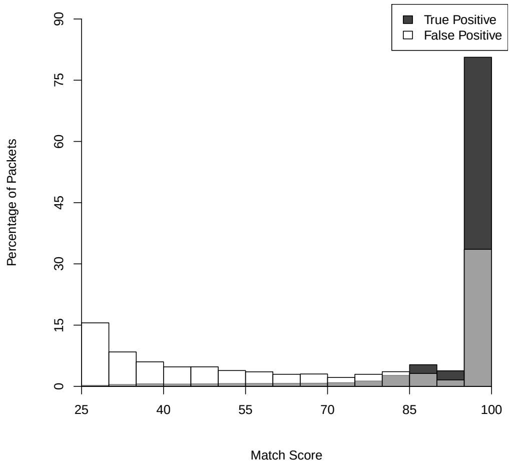

Figure B.3: Score distribution for pdf file types using `mrsh-v2`.

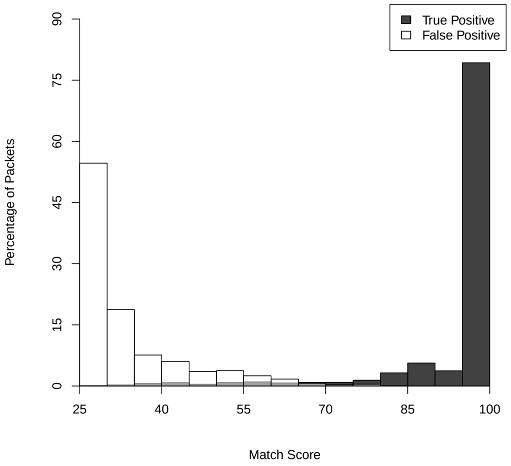

Figure B.4: Score distribution for gif file types using `mrsh-v2`.

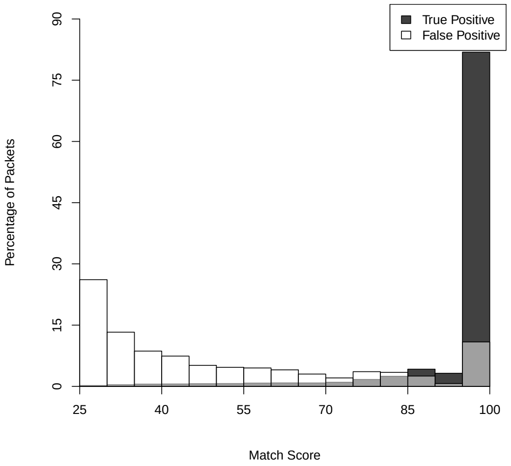

Figure B.5: Score distribution for xls file types using `mrsh-v2`.

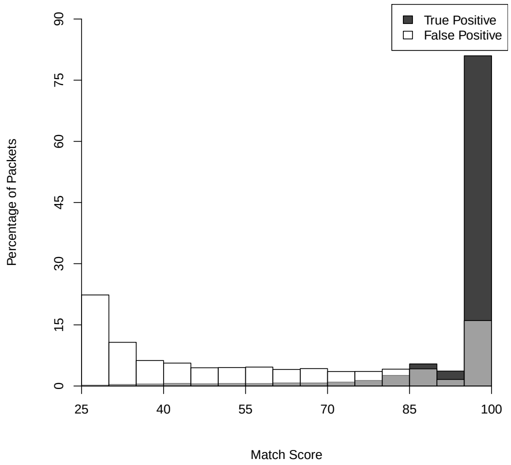

Figure B.6: Score distribution for ppt file types using `mrsh-v2`.

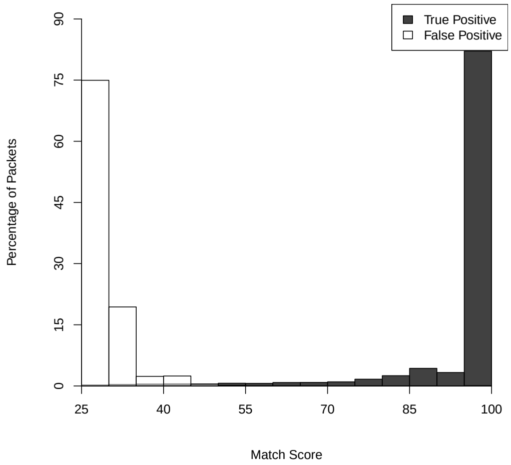

Figure B.7: Score distribution for text file types using `mrsh-v2`.

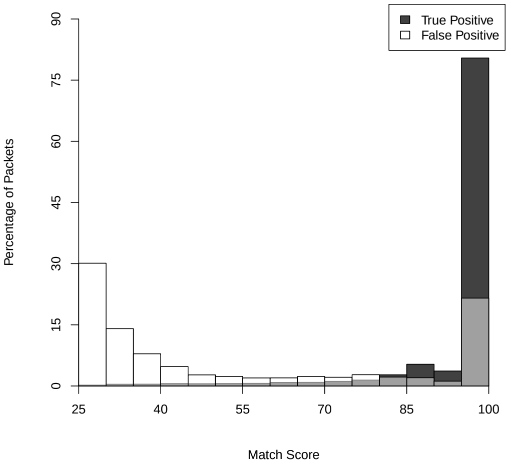

Figure B.8: Score distribution for jpg file types using `mrsh-v2`.

Appendix C

Hex Dump of a xls File

Hex dump of 003444.xls file from *TS* corpus showing 0 and 1 run.

```
0000050: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000060: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000070: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000080: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000090: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000a0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000b0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000c0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000d0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000100: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000110: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000120: ffff ffff ffff ffff ffff ffff ffff ffff  ................

0001650: 2200 2000 c0ff ffff ffff ffff ffff ffff  ". .............
0001660: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0001670: ffff ffff 0a00 0000 0000 0000 0000 0000  ................
0001680: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001690: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00016a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00016b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00016c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
00016d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00016e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00016f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001700: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001710: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001720: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001730: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001740: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001750: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001760: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001770: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001780: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001790: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00017f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001800: feff 0000 0400 0200 0000 0000 0000 0000  ................
0001810: 0000 0000 0000 0000 0100 0000 e085 9ff2  ................
0001820: f94f 6810 ab91 0800 2b27 b3d9 3000 0000  .Oh.....+’..0...
0001830: c400 0000 0800 0000 0100 0000 4800 0000  ............H...
0001840: 0400 0000 5000 0000 0800 0000 6800 0000  ....P.......h...
0001850: 1200 0000 8000 0000 0b00 0000 9800 0000  ................
0001860: 0c00 0000 a400 0000 0d00 0000 b000 0000  ................
0001870: 1300 0000 bc00 0000 0200 0000 e404 0000  ................
0001880: 1e00 0000 1000 0000 4c4f 434b 4845 4544  ........LOCKHEED
0001890: 204d 4152 5449 4e00 1e00 0000 1000 0000   MARTIN.........
00018a0: 4c6f 636b 6865 6564 204d 6172 7469 6e00  Lockheed Martin.
00018b0: 1e00 0000 1000 0000 4d69 6372 6f73 6f66  ........Microsof
00018c0: 7420 4578 6365 6c00 4000 0000 00eb ee3c  t Excel.@......<
00018d0: f113 bf01 4000 0000 808c 2019 3f6a be01  ....@..... .?j..
00018e0: 4000 0000 00d6 fd65 fd6e c101 0300 0000  @......e.n......
00018f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0001900: 0000 0000 0000 0000 0000 0000 0000 0000  ................

00037f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0003800: 0100 0000 0200 0000 0300 0000 0400 0000  ................
0003810: 0500 0000 0600 0000 0700 0000 0800 0000  ................
0003820: 0900 0000 0a00 0000 feff ffff 0c00 0000  ................
0003830: 0d00 0000 0e00 0000 0f00 0000 1000 0000  ................
0003840: 1100 0000 1200 0000 feff ffff 1400 0000  ................
0003850: 1500 0000 1600 0000 1700 0000 1800 0000  ................
0003860: 1900 0000 1a00 0000 feff ffff fdff ffff  ................
0003870: feff ffff ffff ffff ffff ffff ffff ffff  ................
0003880: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003890: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00038a0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00038b0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00038c0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00038d0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

```
00038e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00038f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003900: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003910: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003920: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003930: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003940: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003950: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003960: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003970: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003980: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003990: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039a0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039b0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039c0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039d0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00039f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0003a00: 5200 6f00 6f00 7400 2000 4500 6e00 7400  R.o.o.t. .E.n.t.
0003a10: 7200 7900 0000 0000 0000 0000 0000 0000  r.y.............
0003a20: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0003a30: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0003a40: 1600 0501 ffff ffff ffff ffff 0200 0000  ................
0003a50: 1008 0200 0000 0000 c000 0000 0000 0046  ...............F
0003a60: 0000 0000 0000 0000 0000 0000 8074 afae  .............t..
0003a70: fd6e c101 feff ffff 0000 0000 0000 0000  .n..............
0003a80: 4200 6f00 6f00 6b00 0000 0000 0000 0000  B.o.o.k.........
0003a90: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0003aa0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0003ab0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Bibliography

[1] Ralf Bendrath. Global technology trends and national regulation: Explaining variation in the governance of deep packet inspection. In *International Studies Annual Convention, February*, pages 15–18, 2009.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989. Status: STANDARD.

[4] Frank Breitinger, Knut Petter Astebøl, Harald Baier, and Christoph Busch. mvhash-B - a new approach for similarity preserving hashing. *7th International Conference on IT Security Incident Management & IT Forensics (IMF)*, 2013.

[5] Frank Breitinger and Harald Baier. A fuzzy hashing approach based on random sequences and hamming distance. *7th annual Conference on Digital Forensics, Security and Law (ADFSL)*, pages 89–101, 2012.

[6] Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. *4th International ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, 4, 2012.

[7] Frank Breitinger, Harald Baier, and Jesse Beckingham. Security and implementation analysis of the similarity digest sdhash. *1st International Baltic Conference on Network Security & Forensics (NeSeFo)*, August 2012.

[8] Frank Breitinger and Vikas Gupta. File detection in network traffic using similarity hashing. In *Annual Computer Security Applications Conference(ACSAC)*, August 2013. [Under review].

[9] Frank Breitinger and Kaloyan Petrov. Reducing time cost in hashing operations. *Ninth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, January 2013.

[10] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[11] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[12] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-middle attack to the https protocol. *Security & Privacy, IEEE*, 7(1):78–81, 2009.

[13] Vinton G Cerf and Robert E Icahn. A protocol for packet network intercommunication. *ACM SIGCOMM Computer Communication Review*, 35(2):71–82, 2005.

[14] Ajay Chaudhary and Anjali Sardana. Software based implementation methodologies for deep packet inspection. In *Information Science and Applications (ICISA), 2011 International Conference on*, pages 1–10. IEEE, 2011.

[15] Roberto Di Pietro and Luigi V. Mancini. *Intrusion Detection Systems*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[16] Ted Eisenberg, David Gries, Juris Hartmanis, Don Holcomb, M Stuart Lynn, and Thomas Santoro. The Cornell commission: on Morris and the worm. In *Computers under attack: intruders, worms, and viruses*, pages 253–259. ACM, 1991.

[17] Linux Foundation. `http://www.linuxfoundation.org/collaborate/workgroups.networking/gso`. [Online; accessed 25-June-2013].

[18] Patrick Gallagher and Acting Director. Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technologies, Federal Information Processing Standards Publication 180-1, 1995.

[19] Peter Gordon. Sans institute whitepaper: Data leakage - threats and mitigation. [Online; accessed 25-June-2013].

[20] The Radicati Group. Email Statistics Report: 2012-2016. `http://www.radicati.com/wp/wp-content/uploads/2012/04/Email-Statistics-Report-2012-2016-Executive-Summary.pdf`. [Online; accessed 25-June-2013].

[21] Per Gunningberg, Mats Bjorkman, Erik Nordmark, Stephen Pink, Peter Sjodin, and J-E Stromquist. Application protocols and performance benchmarks. *Communications Magazine, IEEE*, 27(6):30–36, 1989.

[22] C. Hornig. RFC 894: Standard for the transmission of IP datagrams over Ethernet networks, April 1984. Status: STANDARD.

[23] InDorse Technologies Inc. Emerging solutions for strengthening data loss prevention (dlp). http://www.indorse-tech.com/sites/default/files/InDorse%20DLP%20White%20Paper.pdf. [Online; accessed 25-June-2013].

[24] Mcafee Inc. Data loss by the numbers. http://www.mcafee.com/us/resources/white-papers/wp-data-loss-by-the-numbers.pdf. [Online; accessed 25-June-2013].

[25] SANS Institute. Understanding and selecting a data loss prevention solution, 2010.

[26] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[27] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[28] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *digital investigation*, 3:91–97, 2006.

[29] George Lawton. New technology prevents data leakage. *IEEE Computer*, 41(9):14–17, 2008.

[30] Udi Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference*, volume 1. San Fransisco, CA, USA, 1994.

[31] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, volume 5. CRC Press, August 2001.

[32] MyDLP.com. Mydlp administration guide. http://www.mydlp.com/wp-content/uploads/MyDLP-Administration-Guide.pdf, 2012. [Online; accessed 25-June-2013].

[33] Landon Curt Noll. Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/index.html, 1994–2012. [Online; accessed 25-June-2013].

[34] Christopher Parsons. *Deep Packet Inspection in Perspective: Tracing its lineage and surveillance potentials.* Queen's University, Surveillance Studies Centre, 2008.

[35] J. Postel. RFC 793: Transmission control protocol, September 1981. Status: STANDARD.

[36] J. Postel. RFC 793: Transmission control protocol, September 1981. Status: STANDARD.

[37] Michael O Rabin. *Fingerprinting by random polynomials.* Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[38] Martin Roesch. Snort users manual 2.9.4. `http://manual.snort.org`. [Online; accessed 25-June-2013].

[39] Martin Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[40] Vassil Roussev. Hashing and data fingerprinting in digital forensics. *Security & Privacy, IEEE*, 7(2):49–55, 2009.

[41] Vassil Roussev. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*, pages 207–226. Springer, 2010.

[42] Vassil Roussev. An evaluation of forensic similarity hashes. *digital investigation*, 8:S34–S41, 2011.

[43] Vassil Roussev, Simson Garfinkel, Frank Breitinger, John Delaroderie, Barbara Guttman, John Kelsey, Jesse Kornblum, Mary Laamanen, Michael McCarrin, Clay Shields, Douglas White, John Tebbutt, and Joel Young. The NIST Definition of Approximate Matching. Technical report, National Institute of Standards and Technologies, 2013 (to appear).

[44] Vassil Roussev, Golden G. Richard III, and Lodovico Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4:105–113, September 2007.

[45] Cisco Systems. Traffic regulators: Network interfaces, hubs, switches, bridges, routers, and firewalls. `https://learningnetwork.cisco.com/servlet/JiveServlet/previewBody/2810-102-1-7611/primch5.pdf`. [Online; accessed 25-June-2013].

[46] Hubert Zimmermann. OSI reference model–The ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.