A Comparison of Exact String Search Algorithms for Deep Packet Inspection

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Kieran Hunt

Grahamstown, South Africa
October 12, 2017

Abstract

Every day, computer networks throughout the world face a constant onslaught of attacks. To combat these, network administrators are forced to employ a multitude of mitigating measures. Devices such as firewalls and Intrusion Detection Systems are prevalent today and employ extensive Deep Packet Inspection to scrutinise each piece of network traffic. Systems such as these usually require specialised hardware to meet the demand imposed by high throughput networks. Hardware like this is extremely expensive and singular in its function.

It is with this in mind that the string search algorithms are introduced. These algorithms have been proven to perform well when searching through large volumes of text and may be able to perform equally well in the context of Deep Packet Inspection. String search algorithms are designed to match a single pattern to a substring of a given piece of text. This is not unlike the heuristics employed by traditional Deep Packet Inspection systems.

This research compares the performance of a large number of string search algorithms during packet processing. Deep Packet Inspection places stringent restrictions on the reliability and speed of the algorithms due to increased performance pressures.

A test system had to be designed in order to properly test the string search algorithms in the context of Deep Packet Inspection. The system allowed for precise and repeatable tests of each algorithm and then for their comparison.

Of the algorithms tested, the Horspool and Quick Search algorithms posted the best results for both speed and reliability. The Not So Naïve and Rabin-Karp algorithms were slowest overall.

Acknowledgements

Very many people have provided assistance and guidance throughout this research. I am indebted to my family for their support and encouragement throughout the years. Their influence has shaped me into the person I am today. For that, I am forever grateful.

To my supervisor, Barry Irwin, I am thankful for his mentorship, guidance and counsel throughout this research and, I'm sure, for many years to come. I could not have achieved this without him.

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs/CORIANT, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 90243). The author acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Contents

List of Figures

List of Tables

List of Listings

Part I

Introduction

Chapter 1

Introduction

As network usage grows, so too does the importance of network security. A study by Gantz, Florean, Lee, Lim, Sikdar, Lakshmi, Madhavan, and Nagappan (2014) showed that cybercrime in 2014 cost the world's businesses a combined $315 billion. On the frontline of traditional network security is the firewall. In the early years of computer networks, firewalls were simple devices used to control the flow of traffic into and out of a network (Ingham and Forrest, 2002). They worked by employing a set of elementary filters to discriminate against unwanted traffic. This simple approach to security proved fast but ultimately impractical as attacks grew more and more complex.

The next evolutionary step in the defence of computer networks came with the introduction of Intrusion Detection and Prevention Systems. Intrusion Detection Systems serve to monitor network traffic flow for signs of potentially malicious activity. An important component of IDSs and modern network firewalls is that of Deep Packet Inspection.

Deep Packet Inspection (DPI) is the process by which network packets have their payloads analysed for content that is of interest. Interesting content may take the form of a malicious attack, a data leak, the illegal transfer of copyrighted material, or communications unfavourable to the state amongst many other types (AbuHmed, Mohaisen, and Nyang, 2007). Deep Packet Inspection systems are required to very quickly and accurately assess the content of every packet. This is usually done by modelling interesting content and then using those models as fingerprints to match traffic in real time.

In order to meet the demands of modern networking systems, custom hardware can be employed to parallelise the process of Deep Packet Inspection (Dharmapurikar, Krishnamurthy, Sproull, and Lockwood, 2003; Yu, Chen, Diao, Lakshman, and Katz, 2006;

Parsons, 2014). This technique, although fast, has its drawbacks: The hardware required to perform at network speeds is usually expensive to purchase and maintain, is often proprietary, and makes scaling with the demand of the network very difficult. Network administrators must provision the maximum amount of hardware in order to meet any load.

This work extensively defines and compares an alternative technique for performing Deep Packet Inspection that, although slower than hardware-based methods, provides advantages that are of interest to the maintainer of a modern network. The alternative technique to be introduced is the use of traditional string search algorithms for Deep Packet Inspection. These algorithms have been proven to accomplish searches within text at very high speeds on general purpose processors. Such processors power just about every computing system available today, from servers in the largest datacentres to smart phons.

1.1 Problem Statement

Modern Deep Packet Inspection systems provide fast and reliable detection of network-based attacks and identification of other interesting traffic. In order to achieve the speed and reliability that these systems offer, they must forfeit cost, scalability and interoperability (Parsons, 2014).

Contemporary Deep Packet Inspection systems rely on custom hardware - such a field-programmable gate arrays or application-specific integrated circuits - to achieve the speeds required in modern networking scenarios (Parsons, 2014). This hardware is not readily available in existing data centres and, as such, must be purchased or developed in-house. Purchased systems, such as those from Palo Alto Networks[1], are financially expensive, often have mandatory yearly licensing fees and require special training and certification to manage (Palo Alto Networks, 2016).

A custom hardware-based approach to Deep Packet Inspection also introduces the issue of scalability (Dharmapurikar et al., 2003; Kumar, Turner, and Williams, 2006). Most networks see periodic traffic flow corresponding to the time of day, the day of the week, and even state holidays (Grondman, 2006; van Splunder, 2015). Often, systems that see periodic usage can take advantage of this by scaling to meet the needs of the current or near-future load based on heuristics learnt over time (van Splunder, 2015). In the case of

[1] https://www.paloaltonetworks.com/

custom hardware, this isn't possible as the hardware serves a singular use and cannot be repurposed at will.

Furthermore, modern Deep Packet Inspection, through these custom hardware solutions, often relies on proprietary software and protocols. Companies such as Cisco Systems[2], Hewlett-Packard[3], McAfee[4], and Juniper Networks[5] all sell proprietary Intrusion Detection Systems. This type of system has a high initial cost but also forms a closed ecosystem which prevents the owners of such systems from switching providers without substantial monetary investment (Eisenmann, Parker, and van Alstyne, 2009). Closed ecosystems are systems which do not interact with outside systems. These systems will accept network traffic and process it but will not provide a means of communicating with other devices. Such a closed ecosystem limits the network administrator to selecting additional systems, generally by the same company, which are able to interoperate with the current systems. Although the use of homogenous devices provides advantages, selecting such closed systems places all the power in the hands of that company (Eisenmann et al., 2009). A network administrator who has chosen to use this kind of closed ecosystem would have the following options when looking to upgrade their infrastructure: accept whatever cost the supplier decides to charge, not upgrade the system, or pay for a completely new system without the same constrained upgrade conditions.

An alternate means of achieving Deep Packet Inspection without hardware-based or hardware-assisted Deep Packet Inspection is done via software means (AbuHmed et al., 2007; Sourdis, 2007; Chaudhary and Sardana, 2011). Pure Software-based Deep Packet Inspection benefits from being mostly independent of the underlying hardware. Such methods are generally slower than hardware alternatives (AbuHmed et al., 2007) but do not succumb to the drawbacks listed above. These software-based approaches take advantage of the prevalence of general purpose processors available on commodity computing platforms in data centres today. Most research in the field of Deep Packet Inspection is done with custom hardware as the implementation.

There exist string search algorithms which have been proven to be very fast at traversing large amounts of text (Crochemore and Wojciech, 2002; Lecroq, 2007; Faro and Lecroq, 2013). These algorithms have not been benchmarked in the context of packet inspection which has different properties to large volumes of contiguous text. Certain algorithms

[2] http://www.cisco.com/
[3] http://www.hp.com/
[4] http://www.mcafee.com/
[5] http://www.juniper.net/us/en/

which may have a very well known performance in textual or bibliographic settings are not well understood in the constrained environment of Deep Packet Inspection.

This research aims to enable the use of commodity computational hardware for Deep Packet Inspection through string search algorithms. By the comparison of these string search algorithms, this research establishes a benchmark of algorithm performance and behaviour with packet data as the input.

1.2 Research Outline

The following research was conducted with these objectives in mind:

- Survey the current state of Deep Packet Inspection, with increasingly generalised summaries of the state of Intrusion Detection Systems, network firewalls and general network security.
- Compile and describe a sizable set of algorithms designed for exact string searching which have unknown performance in the context of Deep Packet Inspection.
- Construct a system for testing the performance of Deep Packet Inspection with support for different input types and that is easily extensible so that future algorithms may be added or the functionality of the system improved.
- Use the previously constructed system to extensively test the chosen algorithms with different kinds of both textual and packet data.
- Analyse the results of the various tests and present findings comparing the speed of each algorithm, the reliability, and their general performance in the context of Deep Packet Inspection.

1.3 Research Method

It is the goal of this research to complete the research objectives listed in Section 1.2 through initial literature review, the construction of a test system, subsequent implementation of the search algorithms, and then through thorough testing and exhaustive analysis of the resulting data.

The initial review of literature will be presented as an overview of the work done in the various fields related to network security, network firewalls, Intrusion Detection Systems, Deep Packet Inspection and then finally exact string matching algorithms.

The test system will be created using an initial design and then subsequent implementation based on the proposed design. The system itself will be implemented in Java 8[6] and provide support for both packet and textual data.

The test system and various string search algorithms will be tested over many iterations using specially designed datasets. The results of these tests will be analysed and compared among themselves.

1.4 Document Conventions

The definition of the term *packet* is overloaded even within the field of computer networking. In the OSI model (Aschenbrenner, 1986), packets refer to the data carried in layer three, the Network Layer. The term packet has also been used to mean transmitted data from layer two and up. For the rest of this research *packet* refers to the latter variation.

URLs for any website mentioned in this text have been added as footnotes. Within the electronic version of this text, references to parts, chapters, sections and subsections are all hyperlinks to the relevant places within the text. Citations are also added as references to the full citation within the bibliography. The names of the datasets as well as the algorithms are hyperlinks to their introduction in this text. Clicking one of these in the electronic document will move the reader to where the concept is discussed.

1.5 Document Structure

This document has been separated into three distinct parts, each part is then subdivided logically into chapters:

Part I provides an introduction and overview of the current state of the art for Deep Packet Inspection, an in-depth review of each of the algorithms selected for testing and information on the datasets used in the tests conducted in Part III.

[6] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

- Chapter 2 surveys the current state of the art of both software- and hardware-based Deep Packet Inspection, Intrusion Detection Systems, network firewalls and general network security.
- Chapter 3 presents each of the algorithms selected for testing during the course of this research. It also provides a comparison of their algorithmic complexity - a theoretical indication of their performance.
- Chapter 4 examines the datasets - both artificially constructed and real-world - used throughout the research.

Part II explains the software developed for the purpose of conducting and benchmarking Deep Packet Inspection using string search algorithms.

- Chapter 5 shows the design of the packet inspection framework.
- Chapter 6 discusses the implementation of the system and gives an example of its operation.

Part III analyses, compares and contrasts the string search algorithms. It juxtaposes a number of factors affecting the speed of the packet inspection and provides the results from the tests.

- Chapter 7 gives an initial comparison of the string search algorithms and discusses their advantages and disadvantages. It also selects four interesting algorithms and compares their speed against the length of the inputs.
- Chapter 8 drills down into a few select algorithms and analyses them in a variety of different ways.
- Chapter 9 rounds out and concludes the research presented below.

Chapter 2

Literature Review

As the number of people with internet access grows, so too does importance of keeping our digital information safe. With an ever increasing number of internet connected people and more of their lives lived through that interconnectivity, extra requirements to ensure the confidentiality, integrity and availability of their information are added. These requirements are not limited to the individual. Individuals, businesses, corporations, and governments all have much at stake.

This chapter looks at the work done previously in the fields of general network security (Section 2.1), network firewalls (Section 2.2), Intrusion Detection Systems (Section 2.3) and finally Packet Inspection (Section 2.4).

2.1 General Network Security

Network security is not a new concept. In the early days of geographically dispersed computer networks, ARPANET[1], the precursor to today's internet, was a collection of academic and military computer networks (Hauben and Hauben, 2006). The ARPANET was designed for openness and easy interoperability between computers on the network (Leiner, Cerf, Clark, Kah, Kleinrock, Lynch, Postel, Roberts, and Wolff, 2009). Within a few years of its inception, in 1986, the first major malicious security incident was identified.

The first computer program to automatically move between computers on a network was Creeper (Metcalf, 2014). The Creeper program, written in 1971 by Robert Thomas, would

[1] Advanced Research Projects Agency Network, `https://en.wikipedia.org/wiki/ARPANET`

run on TENEX systems and print the message: `I'M THE CREEPER : CATCH ME IF YOU CAN` (Metcalf, 2014). Shortly after, Ray Tomlinson (who famously invented the first email system (Ward, 2001)) wrote the Reaper program to move between computers and remove Creeper (Metcalf, 2014).

In 1986, a researcher by the name of Clifford Stoll[2], who at the time was working for the Lawrence Berkeley National Laboratory[3], was tasked with solving an accounting error in a system connected to the ARPANET (Stoll, 1989; Wuermeling, 1989). During his investigation, Stoll uncovered evidence of a spy known to be working for the Russian *Komitet Gosudarstvennoy Bezopasnosti* (known in the west as the KGB) on the system. This spy's intention was to use the ARPANET to gain access to military and government systems which were, at the time, also connected to the network (Stoll, 1989; Wuermeling, 1989). He was eventually caught using a honeypot set up by Clifford Stoll. Stoll, himself, initially struggled to gain the cooperation of the authorities due to this being the first record of such an incident and the overall infancy of computing in the public eye. Stoll (1989) has chronicled these events in his book *The Cuckoo's Egg*.

That first breach in network security was highly targeted and required active participation on the part of the attacker. Soon after, in 1988, the first widely-publicised automated instance of a network security attack was documented (Gardner, 1989). Coined the Morris worm after its author Robert T. Morris Jnr, this piece of software worked by exploiting known vulnerabilities within the systems connected to the ARPANET (Eisenberg, Gries, Hartmanis, Holcomb, and Lynn, 1989; Spafford, 1989a). The Morris worm was designed to gain access to a computer, make a copy of itself there and then move on to the next system. The processes is then repeated *ad infinitum* (Spafford, 1989b; Denning, 1989).

After these incidents and as a result of the increased understanding of the importance of network security, many different initiatives were set up to further the advancement of the field. An early addition was that of the network firewall (outlined in Section 2.2) which limited traffic flow into and out of a computer network based on predefined rules. These rules would specify criteria such as port number, source and destination IP addresses and even application-layer protocols.

[2]https://en.wikipedia.org/wiki/Clifford_Stoll
[3]https://www.lbl.gov/

2.1.1 Tenets of Network Security

As discussed earlier, the original design of networks using the OSI Model (Aschenbrenner, 1986) focused on flexibility, interoperability, and the standardisation of the protocols used. With its stack-based design and standard protocols, the OSI Model can be used to create networking environments perfectly suited to the situation. An example of this would be a web server; it would combine Ethernet, IP, TCP and then HTTP to serve web pages. Furthermore, the implementation of each layer of the stack can be swapped out without affecting the layers above or below it. If the administrator of this example web server wanted to serve its traffic over a wireless connection, they would just have to swap out the Ethernet portion for some kind of wireless protocol without affecting the higher level protocols on the stack. The process for implementing network security is not as well defined. This may be a consequence of the original design of the network work stack. When developing a secure network, the entire network needs to be considered rather than just the parts which are externally facing. A secure network needs to consider the following tenets:

- Confidentiality - measures by which sensitive and private information is prevented from being exposed (Perrin, 2008). This covers leaks to systems outside of a network as well as unauthorised users within the network.
- Integrity - ensuring that the data within your network can be trusted at any point either during storage or transport (Perrin, 2008). Data that is lost or corrupted, either by malevolently or accidentally can pose a risk to the reliability of the data.
- Availability - ensuring that data is available when it is needed (Perrin, 2008). If a network or system cannot supply data when required then there is little point in it storing that data.
- Authentication - the process by which an individual or system is verifiably who they present themselves as. A person may use a password, security token, biometric test, or some combination of those to verify their identity.
- Authorisation - strict policies surrounding the access to elements within a network, generally applied once authentication has taken place.
- Accountability - ensuring that every action within a system (and more specifically the network) is accounted for and a paper trail is left. It was the presence of a paper trail that led Stoll (1989) to discover the KGB spy on their network.

Effectively securing a network requires the consideration of those tenets at every level. Furthermore, knowledge of your attackers, your network's vulnerabilities and the level of security desired are all factors to consider when planning your network security (Dowd and McHenry, 1998).

2.1.2 Common Network Security Threats

The following list details common attacks on a computer network (Adeyinka, 2008):

- **Viruses** - infect files on a computer and usually propagate to many other files. They limit the integrity and availability of information as users are not able to access those files subsequent to them becoming infected.
- **System and Boot Record Infectors** - these attacks target a lower level than a virus. They infect areas of storage media that are part of the start-up process ensuring that they are run whenever the system starts. Other examples of this type of threat infect the system at a hardware level, rendering them very difficult to repair. This limits the integrity and availability of information.
- **Eavesdropping** - this kind of attack allows malicious parties to access information during transit. Confidentiality of important information is compromised.
- **Hacking** - hackers generally try to gain access to systems in order to create, steal or destroy information. Hacks to web servers are prevalent today and usually aim to steal users login information. Hackers will sometimes leave publicly visible messages[4] to brag about their accomplishments. They compromise confidentiality, integrity and availability.
- **Worms** - worms traverse systems on networks. They act autonomously, usually for the purpose of depositing a virus or trojan. The first instance of a computer worm is discussed earlier in this section.
- **Trojans** - these programs disguise themselves as benign applications such as something attached to an email or as a familiar program. They carry a payload similar to a worm which executes when the user or system opens the disguised file. The term trojan is a reference to the Trojan Horse written about by Homer in the *Iliad*[5].

[4]For a gallery of such messages: `https://www.google.co.za/search?q=hacked+website`
[5]`https://www.gutenberg.org/cache/epub/6130/pg6130.txt`

- **IP Spoofing** - attackers can change the source IP address of packets in order to gain access to a network. Rudimentary firewall systems with simple access policies may be vulnerable to these kinds of attacks. IP Spoofing can take advantage of a system with a poor approach to authentication which may by default trust packets from a certain IP address. An example of such a vulnerability is implicitly trusting traffic from IP addresses within a network (Ferguson and Senie, 2000).

- **Denial of Service** - attacks such as these leverage the poor configuration of a system or by a brute-force approach to limit the access to that system, thus compromising availability of information. A Denial of Service attack may send many requests to a system and by some means cause that system to allocate many resources to respond to that request. If enough requests are made the system's ability to respond to legitimate requests made be impaired.

- **Phishing** - phishing attacks attempt to take advantage of a layman's naïveté by posing as a legitimate request for information and leveraging the trust of others to steal confidential information. These kinds of attacks are often used to steal internet banking information, but can also be used to gain access to a computer network or system on that network.

2.1.3 Network Threat Mitigation Techniques

A variety of methods are deployed in order to mitigate threats caused by the attacks listed before. Such mitigating measures include (Adeyinka, 2008):

- **Cryptographic Systems** - a system for encoding information into a encrypted form and then decoding it again; only someone with access to a key is able to decode the encrypted information. These systems can be used to store sensitive data so that in the event of the data being leaked, whoever stole the data would need access to the key to read it.

- **Firewalls** - Firewalls act as a basic line of defence for traffic entering or egressing a network. Firewalls generally filter traffic based on IP address, UDP or TCP ports or subnets. Firewalls are traditionally designed to be closed to traffic unless an exception or rule has been made to allow it. Firewalls are discussed in-depth in Section 2.2.

- **Intrusion Detection Systems** - Intrusion Detection Systems employ heuristics to determine the threat level of traffic flowing into and out of a network as well as the behaviour of people and systems within the network itself. These systems have to monitor different kinds of traffic traversing a network looking for signs of malicious intent. Effective Intrusion Detection Systems must check for all of the attacks listed above and provide some kind of alert.

- **Anti-malware Software** - traditionally this software is deployed onto systems to detect signs of the presence of viruses, worms and trojans. Once found, anti-malware software generally removes or blocks the malicious software in order to protect the system. Modern versions of anti-malware software make use of quarantine techniques to silo the files in case the user deems them safe.

- **Internet Protocol Security (IPSec)** - this protocol provides a way to securely transmit information between two machines. It is a popular way of connecting two physically separated networks across an untrusted network (such as the internet). VPNs[6] are an example of software making use of IPSec; they bridge two networks in such a way that the machines on each of the networks appear to be on the same network.

- **Secure Sockets Layer (SSL)** - similarly to internet protocol security, secure socket layer provides encryption for data during transport. Unlike internet protocol security, secure socket layer encrypts data up to the protocol layer rather than just between machines. Secure socket layer is now the defacto encryption method for serving TCP traffic - most notably on the world wide web. The modern version of SSL is Transport Layer Security.

- **Content Filtering** - a more specific form of firewalling which looks at the content of the traffic itself. This mitigation technique is often employed by businesses to limit employee access to certain websites (Rouse, 2011).

2.1.4 Summary

This section has covered the general field of network security and the reasons for employing and encouraging its use throughout all computer networks. Many vulnerabilities exist and exploiters of such vulnerabilities mean to steal and compromise private information. It is

[6]Virtual Private Networks: `https://en.wikipedia.org/wiki/Virtual_private_network`

very important to have good knowledge of network security so that you may ensure that your networks are secure.

The coming sections will explore a number of measures used to counteract attacks on a computer network. These measures represent a small fraction of the methods used to secure a network.

2.2 Firewalls

The term *firewall* is described by the Oxford English Dictionary (Fowler, Fowler, and Allen, 1990) as

"A wall or partition designed to inhibit or prevent the spread of fire."

This term was subsequently used in computer networking to define a device or system used to separate one network from another (Zwicky, Cooper, and Chapman, 2000). Before the advent of network firewalls, simple routers separated one network from another. This separation could protect users and systems on one network from a misconfiguration (Ingham and Forrest, 2002) or noisy applications and protocols (Avolio, 1999) on another. The first firewalls appeared in 1987 (Ingham and Forrest, 2002) and and since have developed to include the following functions: network address translation, filtering, virtual private networks, and proxies.

For a device to be considered a firewall, it should satisfy the following requirements (Ingham and Forrest, 2002):

- Firewalls should separate two networks. They should form the boundary.
- All traffic from one network to another should flow through the firewall.
- The firewall must permit some kinds of traffic and block others.

Historically, the development of network firewalls has matched the levels of the OSI Model (Aschenbrenner, 1986) - initially only low-level inspection and filtering of traffic would occur but as time went on more and more levels of the protocol stack were understood, inspected and filtered by the firewall.

The key concept of a network firewall is that users and systems on one side of a firewall are trusted to a different extent to users and systems on the other side (Zwicky et al., 2000). The firewall separating the network of an academic institution, such as a university, from the rest of the internet is an example of such a system. Users on the university's network are trusted more than users on the internet. There are therefore usually fewer restrictions in place for traffic flowing within and out of the university's networked compared to traffic flowing from outside in. Users within the network may be permitted to serve web pages or share files with other users on the network but all of this traffic would be blocked from flowing out of the firewall.

Firewall administrators quantify trust by creating policies which describe how different kinds of traffic are to be treated. The different levels of trust placed on different kinds of traffic can be attributed to the following reasons (Ingham and Forrest, 2002):

- **Operating System and program security flaws** - Many operating systems or programs running within operating systems have known vulnerabilities. Often it is not possible to ensure that every machine connected to a network has been updated with the latest security patches - especially on networks where the administrators have no control over the connected computers - and so network administrators can use firewalls to limit the access of that machine to an outside network by blocking specific protocols or checking packets for exploits. An example of this would be to block telnet traffic flowing into a network.

- **Preventing access to information** - many businesses and governments implement firewalls which limit the access of users inside the network to information outside. For businesses this is often to block access to websites and services offering nonbusiness-related things. Governments can block information that does align with their political ideals (such as the Chinese government with their *Great Firewall of China*) or for the supposed protection of their inhabitants (like Britain's *Hadrian's Firewall*[7]).

- **Preventing information leaks** - Computer networks can contain machines with sensitive information on them. Due to the myriad of vulnerabilities in computers and the people using them, often internet firewalls are used to stop private information from leaving a network. To do this the firewalls need to have intimate knowledge of what constitutes confidential information and wrongly identifying such information could qualify as a degradation of availability.

[7] http://www.bbc.com/news/uk-23401076

- **Enforcing policy** - As some devices are not controlled by the network administrators, a firewall can be used to limit which applications and protocols are able to work within a network. Firewalls can also provide bandwidth monitoring and limiting services in bandwidth constrained networks.
- **Auditing** - Firewalls can record all traffic that flows through them. After a network attack has been recorded the audit logs stored by the firewall may prove important in preventing future attacks.

As discussed earlier, network firewalls often focus of specific layers of the network stack. The forthcoming subsections will discuss a few examples of network firewalls operating at different layers of the stack.

2.2.1 Layer 7 - Application Layer

As mentioned earlier, network firewalls follow a history that mimics the layers of the OSI model. The first description of a firewall that filters traffic was written by Mogul (1989) and the design was subsequently improved upon by Ranum (1992), creating the *Securing External Access Link (SEAL)* - one of the first commercially available firewalls.

The SEAL system provided an application-layer firewall through the use of various proxies. Proxies work by making connections to an external system on behalf of users both within and outside the network. The proxies provided users of the network with connections for: email and USENET, Telnet, FTP, WHOIS, and X Windows. The advantage of such a system of proxies was that the proxies could enforce the correct use of protocols as they were acutely aware of how the protocols were implemented. See Figure 2.1 for an example of such a proxy.

This kind of firewall is not without its drawbacks. Implementing application-layer proxies means that a separate proxy must be implemented for each application-layer protocol that the administrator wants to support. Furthermore, the client must often be aware of the proxy server and authenticate with it. For many applications this requires that the developer implements these changes and for applications which do not publish their protocols, this task may be impossible.

Although application layer proxies provide heightened security by making external connections on behalf of the users of a network, they are difficult to maintain and do not scale

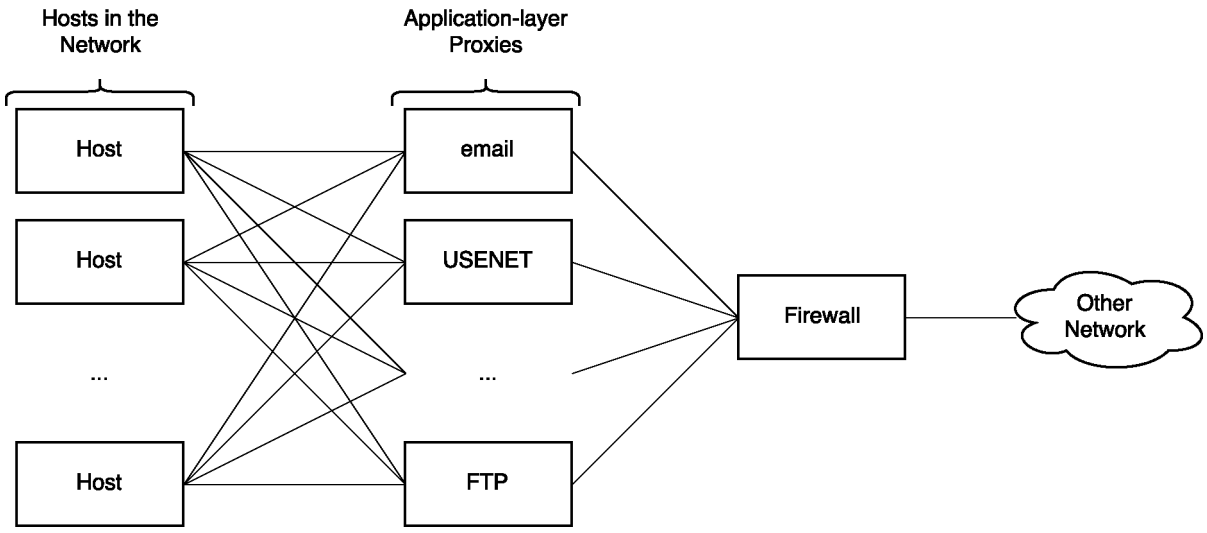

Figure 2.1: An example of application-layer proxies

to meet the number of different Application Layer protocols in use throughout modern networks today. The next subsection discusses the use of Transport Layer proxies to try and reduce the number of proxies that have to be supported.

2.2.2 Layer 4 - Transport Layer

The Transport Layer is mainly used by the TCP and UDP protocols. As there are fewer protocols, far fewer proxies need to be written to support more kinds of traffic. The advantage of working at the Transport Layers is that external traffic cannot spoof established TCP connections. This is becauses the firewall maintains the state of each connection (Ingham and Forrest, 2002). Unlike Application Layer proxies, Transport Layer proxies cannot enforce behaviour for protocols above the Transport Layer.

The first transport-layer gateway was written by Cheswick (1990) for AT&T[8] and a popular later implementation was the SOCKS (Koblas and Koblas, 1992) proxy. In SOCKS, rather than making a `socket()` call, the application would make a SOCKS call instead. All traffic would then be routed through a SOCKS server. Protocols such as Secure Shell are able to create local SOCKS servers and tunnel network traffic to external computers much in the same way as VPNs transparently create bridges between two networks. The popularity of the SOCKS protocol has since decreased in popularity.

[8] `https://www.att.com/`

2.2.3 Layer 3 - Network Layer

Like the Transport Layer, the Network Layer also has a very limited number of protocols that a firewall needs to be aware of. Internet Protocol - versions four (Postel, 1981) and six (Deering and Hinden, 1998) - are the most popular. Simple packet filtering at the network layer is a popular method of implementing network firewalls but they are limited in that they cannot keep state of ongoing connections (Ingham and Forrest, 2002; Chaudhary and Sardana, 2011).

Packet filtering is much faster than the other firewall implementations discussed previously as it does not require that the packets traverse the entire protocol stack (Corbridge, Henig, and Slater, 1991). Packet filtering is transparent to the users of a network in so far as they do not have to make alterations to their applications for it to work, unless those applications violate the policies of the Transport Layer Firewall.

Packet filtering firewalls are usually configured using a standard "5-Tuple" rule (Al-Shaer and Hamed, 2003). The following is a list of example criteria that a Network Layer firewall may use to distinguish wanted and unwanted traffic:

- Source address
- Destination address
- Transport layer protocol
- Flags set in the network layer header
- Various transport layer features - such as source and destination port

A drawback to pure packet filtering is that the identity of the originator of a packet cannot be confirmed. Packets only contain information about the originating IP address and not the user behind that address. On a shared system or behind some kind of NAT (see Subsection 2.2.4) it is almost impossible for a firewall operating purely at the Transport Layer to identify the originator of the traffic. Furthermore, IP addresses can be easily spoofed - rendering them insufficient methods of authentication, particularly in stateless firewalls (Ingham and Forrest, 2002).

The proxy methods mentioned in Subsections 2.2.1 and 2.2.2 provide safety that a packet filter cannot. As the proxies make connections on behalf of the users behind them, the

proxy administrators can ensure that the protocols used are up to date and correctly adhered to (Ingham and Forrest, 2002). Compared to the packet filters where the client makes the connection to the remote machine, this shrinks the attack surface to just the proxy system.

One way that a firewall can try and protect the clients from attacks based on bugs and flaws in their systems is by keeping state of the connections (Ingham and Forrest, 2002). Stateful firewalls track the state of connections by monitoring both layer three and four of the packets flowing through it.

2.2.4 Network Address Translation

Network Address Translation (NAT) was originally designed as a way to deal with the shortage of IPv4 addresses on the internet (Egevang and Francis, 1994) by sharing connections. A router implementing NAT could manage many clients - all behind a single or a few IP addresses. NAT routers work by using DPI to read and then rewrite the source address of outgoing packets to match their own external address. The NAT device then uses an unused port number from a layer four protocol to identify the connection which can then be looked up when a reply is received. NAT thus provides similar features to the proxies mentioned earlier in that a connection needs to already exist for packets to flow into the network. Devices wishing to receive unsolicited traffic (such as web servers) may use port forwarding to automatically send packets of a certain type directly to the device behind the NAT router.

NAT is not the happy medium that is desired. In order to work correctly, NAT must break the transport layer protocols and use the port numbers as identifiers for devices on its network. NAT was originally designed as a way for more devices to connect to the internet than there are IPv4 addresses available. The solution to that problem is IPv6 although it is still not widely adopted.

2.2.5 Other Firewalling Techniques

An alternative to the explicit proxying solutions discussed earlier is the transparent proxy (Chatel, 1996). A transparent proxy works by allowing the client to send packets to the remote device as if no proxy existed. All packets flowing into and out of a network are sent through the proxy and when the packets reach the proxy, the proxy creates the connection

with the remote device and replies to the client as if it is the remote device. Transparent proxies are required to understand all application-level protocol that they wish to proxy.

A further application of network firewalling is increased security through a concept know as *normalisation*. Attackers are often able to evade detection by using ambiguities in the stream of traffic that they send to a network (Handley, Paxson, and Kreibich, 2001). These attackers use techniques such as splitting traffic into smaller chunks, not adhering to a protocol specification, or exploiting a packet's time to live (TTL) so that is doesn't reach its intended recipient (Shankar and Paxon, 2003). For the latter of those examples, an attacker may be able to glean information about the topology of a network by whether it receives replies to its messages. A network normaliser works to correct some of these potential vulnerabilities by altering the network flow so that the traffic is normalised (Handley et al., 2001): Out of order packets are reordered and packets that cannot reach their destination are dropped.

2.2.6 Denial of Service

Network firewalls have also been employed to mitigate the effects of a denial of service (DOS) attack. DOS attacks aim to restrict the use of a system by overloading it in some way. These kind of attacks usually work by sending carefully constructed requests that are designed to take long to process, with enough of these kinds of requests, a system could become overloaded. Another method is to send so many valid requests that a system cannot process them all. This style of denial of service is usually referred to as distributed denial of service.

For these kinds of attacks, a network firewall is often a good tool to combat them. Input rate limiting will stop too many resource intensive requests from being accepted, and dropping packet originating from the DOS location is often sufficient to curb such a threat. This may limit the service's availability to legitimate users who are originating from those locations but the vast majority of benign requests are able to be received. Later, once such an attack has ended, the firewall could begin reaccepting traffic from those temporarily blocked locations.

No network security measure will ever be complete safe and sufficient without drastically compromising availability. The next subsection discusses a few shortfalls of firewalls.

2.2.7 Shortfalls

Firewalls are not the perfect security solution. Since the introduction of SSL in the 1990s[9], and with further enhancements to it and subsequent protocols, there is an ever increasing amount of encrypted traffic flowing through networks today. Encrypting traffic means that the information that it contains can only be read by someone who has the correct key. Unless actions be taken[10], firewalls are unable to directly read the data within packets forming part of true encrypted streams. This could provide a pipeline for sensitive information to leak out of a network or for malicious packets to enter under cover.

Cheswick (1990) describes networks as "a sort of crunchy shell around a soft, chewy center." That description becomes relevant when one considers the threat posed by users inside the network. Physical security is often undervalued or sometimes entirely ignored. In such cases malicious users may gain physical access to systems on the network. Through this vector, users could steal private data or plant malicious code.

2.2.8 Summary

As has been shown, firewalls are not entirely sufficient security devices on their own. Administrators need to try and find a balance between good, reliable access for their users as well as maintain a adequate level of security. In practise, administrators will employ many different security measures to try and approach the problem from different vantage points. Another such measure, which also makes use of Deep Packet Inspection, is that of Intrusion Detection Systems. These are discussed further in Section 2.3.

2.3 Intrusion Detection Systems

Another way of detecting intrusion attempts into a network is via the use of Intrusion Detection Systems. Intrusion Detection Systems are designed to gain insight into a network for the purpose of detecting malicious behaviour on or misuse of a network (Kemmerer and Vigna, 2002; Ashoor and Gore, 2011) by using pattern matching. What an Intrusion

[9]The protocol was originally designed by Netscape. An archived version of the original web page is available here: `https://web.archive.org/web/19970614020952/http://home.netscape.com/newsref/std/SSL.html`

[10]Some companies add themselves as certificate authorities to employee machines so that they may inspect encrypted traffic. They typically use systems such as Blue Coat: `https://www.bluecoat.com/`

Detection System finds is the evidence of an intrusion; this is often called its *manifestation* Kemmerer and Vigna (2002). The definition of the roles of an intrusion detection is quite fuzzy. Any system that detects intrusions in some way could be considered an Intrusion Detection System. The following lists some features which could classify a system as an IDS (Ashoor and Gore, 2011):

- Monitoring system and user behaviour
- Checking system configuration and identifying known vulnerabilities
- Assessing the integrity of a system
- Identifying attacks by known patterns or heuristics
- Identifying and recording policy violations

Initially, intrusion detection was done manually by network administrators. The administrators would monitor for intrusions by watching access to systems throughout their network (Kemmerer and Vigna, 2002). The network administrators would watch for logins or activity from users which seemed out of place. If an office secretary was logged as connecting the production database server that may indicate some form of compromise. The administrators relied on intuition which proved somewhat effective but did not scale to the size of modern computer networks. Following that (in the 1970s and 1980s) administrators would print off access logs and trawl through them looking for patterns which could signify suspicious behaviour. As the network administrators would do this periodically, it served more as a tool for detecting past intrusions rather than catching one in the act (Kemmerer and Vigna, 2002).

In the early 1980s, Anderson (1980) developed a system to monitor for intrusions automatically. Anderson's system would characterise typical use of computers by monitoring what time users were usually active, seeing which files they touch or program they used, and monitoring which devices they interfaced with. All of these properties where used to create a model of particular usage for a user with a specific role. Activities of users which did not fit into the model defined by their role could be a sign of an intrusion.

Traditionally, an IDS has been positioned behind a network firewall as described in Figure 2.2 (See Figure 2.3 for a better breakdown of the internals of an IDS). In this configuration, the firewall acts as a first line of defence whereafter the Intrusion Detection System would monitor the traffic approved by the firewall. The Intrusion Detection System, as the name

would imply, does not directly interact with the traffic. Usually, if it identifies a threat, the Intrusion Detection System would notify some other device. In the setup shown in Figure 2.2, the Intrusion Detection System might notify the firewall of the potential intrusion and the firewall would make appropriate changes to its rules to mitigate such a threat.

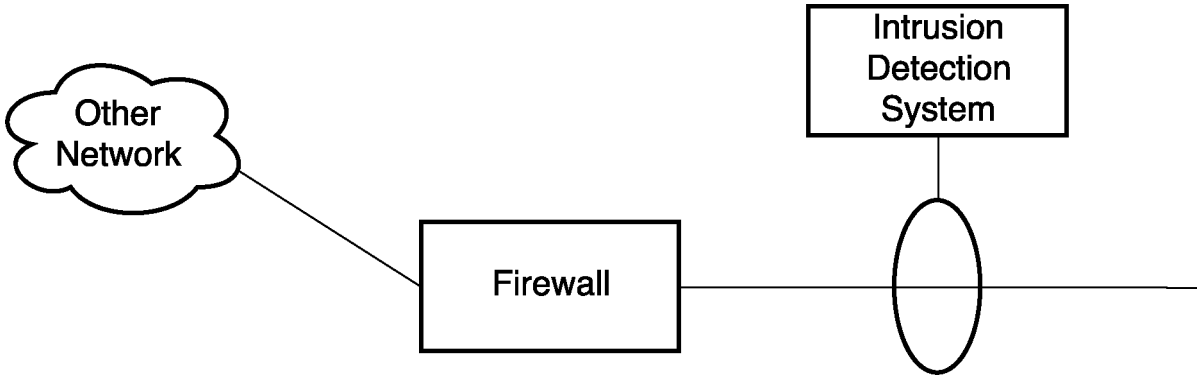

Figure 2.2: A traditional network setup with a firewall at the network edge and the Intrusion Detection System behind it.

Figure 2.3 shows a prototypical IDS. The first part of an IDS, often called the *preprocessor*, is responsible for collecting individual packets, grouping them by connection (most attacks span multiple packets), sorting, and finally decoding if necessary (Some attacks try to use different or obscure encodings in order to subvert IDSs) (Sourdis, 2007). These amalgamated connections are then passed over to the *detection engine* which tries to match the input against a known database of rules. The *detection engine* uses both packet classification and content inspection to try and ascertain the intent of the packet. Packet classification uses data in the header of the packet to establish identifying information such as the protocol and the packet's source. Packets originating from a IP address known to be a source for malicious traffic or of a protocol destined for a system which should not support that protocol should be a sign of malintent. Content inspection makes use of patterns and heuristics to identify the contents of the packet's payload. For more information of the rules used by Intrusion Detection Systems, see subsection 2.3.1.

An issue facing IDSs is that of data collection. Designers of such systems have to decide what level of data collection is enough to ensure the required security. Simple IDSs could simply log failed login attempts which may be used as part of detecting a compromised account or rogue employee. More complex systems could log every packet flowing through the network's firewall (Kemmerer and Vigna, 2002). Logging the right amount of data is important to ensure that patterns over a long period of time are detected and that those patterns are not hidden behind too much else.

Simply collecting the data is not enough. IDSs need to actually analyse the data to find

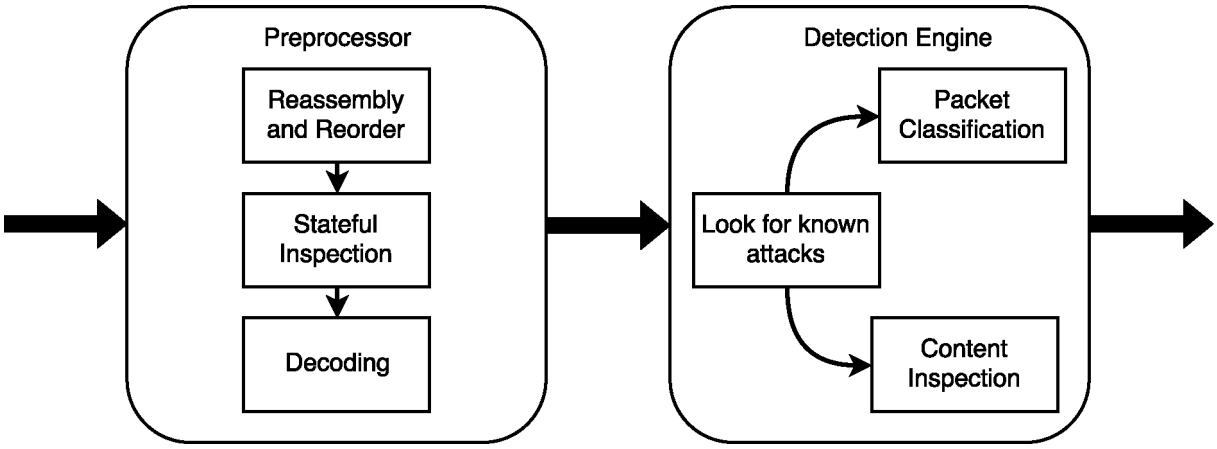

Figure 2.3: A typical network IDS (Sourdis, 2007)

evidence of intrusion. Broadly speaking there are two wide categories of techniques used for intrusion detection (Kemmerer and Vigna, 2002):

- **Anomaly detection** - This type of IDS models *normal* user and system behaviour and uses those models to identify irregularities which may indicate an intrusion (Denning, 1987; Ghosh, Wanken, and Charron, 1998; Chandola, Banerjee, and Kumar, 2009). Factors can include: when a user is usually active on their systems, what their usual tasks are, and whether they usually encounter access errors. Data indicating behaviour outside of the normal observed boundaries may be a manifestation of an intrusion. This is the same technique employed by Anderson (1980).

- **Misuse detection** - This approach to IDSs uses known threats to model potential attacks. Misuse detection works well to identify threats the same as or similar to known threats and so does not produce many false positives (Kemmerer and Vigna, 2002). This technique doesn't do well at detecting intrusions which do not match known signatures. Examples of systems using misuse detection include: Snort[11], OpenVAS[12], Fortinet[13], Checkpoint[14], Suricata[15], and Bro[16].

Once an Intrusion Detection System is confident that an attack is taking place, it must create some kind of response. Responses usually include all relevant information about the ongoing attack or intrusion. These responses are then sent to whichever system is

[11] https://www.snort.org/
[12] http://www.openvas.org/
[13] https://www.fortinet.com/products/fortigate/index.html
[14] https://www.checkpoint.com/products/ips-software-blade/
[15] http://suricata-ids.org/
[16] https://www.bro.org/

responsible for mitigating attacks. IDSs notify administrators through some kind of alert, siren or alarm (Kemmerer and Vigna, 2002).

There are two classes of issues which remain for developers implementing effective Intrusion Detection Systems. The first being effectiveness - how well IDSs detect intrusions - and the second being speed - how quickly IDSs detect intrusions. For an IDS to be effective it must be able to detect as many possible intrusions as possible. The intrusions are detected by matching traffic - or some other kind of data - with known rules or heuristics. Subsection 2.3.1 discusses the rules used by a contemporary IDS.

2.3.1 IDS Rules

IDSs use well-defined rules to specify what to look for in the packets they process. The rules themselves are generally written in some kind of domain specific language defined for the particular Intrusion Detection System. Snort is an example of an Intrusion Detection System which makes employs misuse detection to identify attacks on a network. The subsection that follows will be discussing rules specifically pertaining to the Snort IDS[11].

Snort's system for describing rules is both powerful and complex. All rules in Snort obey the structure presented in Listing 2.1. Table 2.1 gives a breakdown of the keywords used in Listing 2.1 and describes the features of the language.

```
1 action proto src_ip src_port direction dst_ip dst_port (options)
```

Listing 2.1: Snort rule structure

Variable	Description
`action`	The action to perform when the rule has been matched, examples include: `alert`, `log`, and `pass`.
`proto`	The protocol to match on, examples include: `tcp`, `udp`, and `icmp`.
`src_ip`	The source IP address.
`src_port`	The source TCP or UDP port.
`direction`	The direction of the traffic flow, examples include: `->`, `<-`, and `<>`.
`dst_ip`	The destination IP address.
`dst_port`	The destination TCP or UDP port.
`(options)`	Various other options.

Table 2.1: Snort's rule structure breakdown

An example of the simplest working rule can be found in Listing 2.2. This rule will create an alert for any packet containing a TCP header.

```
alert tcp any any -> any any (msg: Alert! ;)
```

Listing 2.2: A very simple working Snort rule

Rules in Snort and many other systems can be defined by either static patterns or regular expressions. Static patterns exactly describe the content of a packet which is of interest. Regular expressions (usually of the Perl-compatible[17] kind) describe a sequence or pattern to be found within the packet. An example of a Snort rule containing both a static pattern and a regular expression can be found in Listing 2.3.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (content:"ATTACK"; pcre:"/^
PASS\s*\n/smi"; within:10;)
```

Listing 2.3: Snort rule featuring a static pattern and a regular expression (Sourdis, 2007)

In Listing 2.3, `content:"ATTACK"` forms the static pattern part of the rule. Snort will use this rule to find the text `ATTACK` in each of the packets that it inspects. This also shows an example of a regular expression used for intrusion detection within Snort. In the example, `pcre:"/^PASS\s*\n/smi"` is an example of a regular expression-based rule. This particular regular expression can be broken down as follows[18]:

- **`^PASS`** - Matches the text `^PASS` literally.
- **`\s*`** - Matches any white space character as many times as possible.
- **`\n`** - Matches a newline character.

Finally, Listing 2.3 also contains one further constraint on the packet matching rule. The text `within:10` limits the second match to occur within 10 bytes after the first.

2.3.2 Effectiveness

IDSs need to achieve near perfect detection of intrusions into a network. To achieve this, modern Intrusion Detection Systems rely on misuse detection. Misuse detection relies on predefined rules based on known attacks and, as such, must be updated constantly

[17] `http://www.pcre.org/`

[18] Regex 101 provides a good explanation of the regular expression: `https://regex101.com/r/iL1qH4`

to keep up with the latest vulnerabilities (Kemmerer and Vigna, 2002). This approach is often only sufficient for attacks which are known to the system or are very similar to other attacks. New attacks or attacks specially designed to defeat this particular system may be enough to overcome an Intrusion Detection System with a fully up-to-date set of rules.

2.3.3 Performance

System performance is the crux of the problem and will be discussed through the rest of this research. There is always going to be a limit to the number of signatures that a system can search for in a finite amount of time. Figure 2.4 shows the results from a study by Schuff and Pai (2007) wherein the authors measure the amount of processing time spent on the Snort Intrusion Detection System during normal running. From that pie chart is is clear that a majority of Snort's processing time for each packet is spent on content inspection wherein it tries to match the content traffic to patterns defined in the rules.

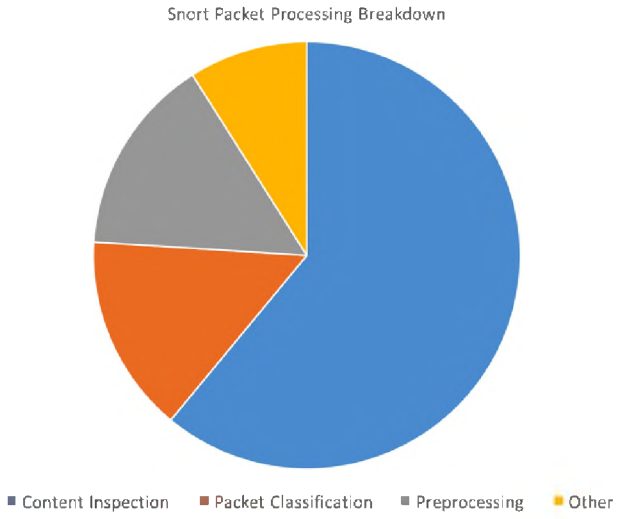

Figure 2.4: Snort processing time broken down by group (Schuff and Pai, 2007).

As mentioned earlier, although the speed of processors has been increasing in a similar way to network speed, the number of known attacks has also increased. An increase in number of attacks should directly result in a greater number of rules needed to perform effective intrusion detection. Sourdis (2007) gives a table showing the increasing number of Snort rules over a period of time from 2003 to 2007. In 2003 the number of rules stood at just over two thousand. By 2007 the number had increased to over eight thousand. In April 2016, for Snort version 2.9, that number had grown to almost nine thousand rules.

Sourdis (2007) showed that the number of regular expressions (Section 2.3.1) used in Snort rules increased dramatically between 2003 and 2007. Regular expressions allow for far more complex and flexible rules to be created. As a result of the increased flexibility gained by using a regular expression, it is likely that many rules which were previously implemented as distinct rules using the static-style expressions have been combined into a single rule using regular expressions. Regular expressions themselves have nondeterministic properties which, for poorly designed rules, could lead to massive slowdowns during processing.

With this in mind, the slow increase in the number of Snort rules can be attributed, in part, to the increased reliance on regular expressions. Variations on existing rules can be implemented within the rules themselves rather than as separate rules entirely. Furthermore, even though there are more networked devices than ever before[19], computer security is increasingly a concern for developers of such systems and often on the mind of even the general public which may encourage better diligence when it comes to ensuring that their systems are protected. Snort retires rules which they deem end-of-life (EOL). These rules are for vulnerabilities present in older systems or software not supported by Snort or for systems in which the vulnerability has been patched.

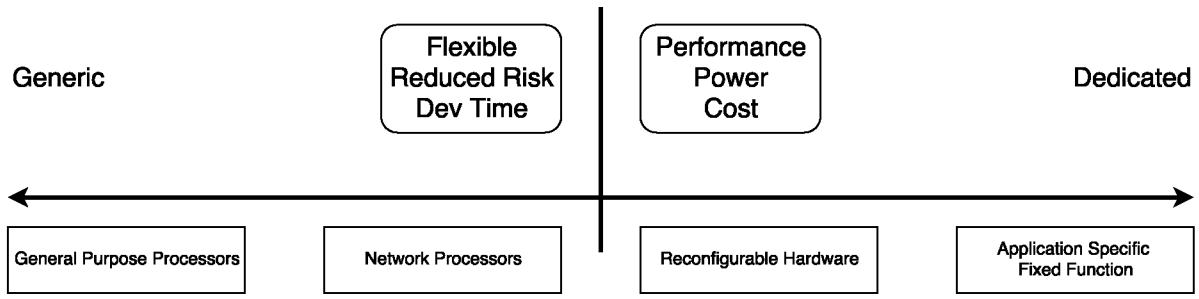

Figure 2.5: A comparison of the broad categories of IDSs available (Sourdis, 2007).

Implementation

In order to cope with the ever increasing performance demand put on IDSs, and in particular the speed required for packet processing, a number of different methods for implementing these systems are availabile. There are four broad categories (represented in Figure 2.5) under which an IDS implementation can fall (Shah, 2001; Sourdis, 2007; Becchi, Wiseman, and Crowley, 2009; Jiang and Prassana, 2009):

[19] See for instance the Internet of Things (IoT) movement: `https://en.wikipedia.org/wiki/Internet_of_Things`

- General Purpose Processors (GPPs)
- Network Processors (NPs)
- Reconfigurable Hardware
- Application Specific Fixed Function (ASIC)

Each of these implementation styles has its own advantages and disadvantages. This tradeoff usually manifests itself as a reciprocity between performance and flexibility.

General Purpose Processors are the standard processors found in data-centres and personal computers throughout the world. GPPs are easy to develop software for - owing to the fact that they are present on every software developer's personal machine - and cheap because of their ubiquity and flexibility. GPPs struggle, however, to match network line speeds because of their general design and linear processing pattern that they follow (Sourdis, 2007).

Network Processors try to take the advantages from GPPs - namely how easy it is to develop for the platform - but employ dedicated and specialised network hardware to further increase performance (Shah, 2001). Some of the networking work is offloaded to this special hardware whilst the inspection continues to run on the GPP. The obvious speed bottleneck is, again, the GPP.

Reconfigurable Hardware provides a middle-ground between the generic and slower software-based solutions and the dedicated hardware. Sourdis (2007) defines the difference between reconfigurable and reprogrammable as follows: "a reconfigurable device can support directly in hardware *arbitrary* functions on demand, while a reprogrammable device can choose only between its *predefined* (and committed at fabrication), *finite* number of functions." Generally speaking software based implementations are able to switch their behaviour on a per-packet basis whereas with reconfigurable hardware functionality changes may only be made when the entire ruleset is altered (Shah, 2001). Reconfigurable hardware typically refers to implementations on field programmable gate arrays (FPGAs).

Application Specific Fixed Function are typically ASIC devices used for very fast packet processing. ASICs sacrifice ease of development and flexibility for processing speed. ASICs tend to be very difficult to make changes to which means that rulesets are generally hardwired.

It is with Application Specific Fixed Function and Reconfigurable Hardware that the hardware-based solutions discussed earlier are implemented (Shah, 2001) and it is through general purpose processors (Yu et al., 2006) that this research's solution is implemented.

A solution to the limiting processing speed of an IDS is to split the processing up onto many different machines. Traffic will generally encounter a single, fast, centrally located machine which will send traffic to different machines based on load and other factors. The issue with splitting the traffic is that, depending on how the splitting is performed, intrusions may slip by as a single system performing the traffic analysis would not have the entire context of a connection to correctly identify an attack (Kemmerer and Vigna, 2002).

For IDSs built with custom hardware this solution could prove to be very expensive. For each additional machine the initial high cost of the hardware must be incurred again.

Another approach to meeting the speed demand of the network is to deploy the IDSs at or near the hosts on the network they are trying to protect. This approach means that network traffic has already been separated out due to the normal traffic routing algorithms employed in the network. Traffic reaching the IDS should just be intended for that host. This does mean that often many more pieces of hardware are needed to perform at the same speed as the previous approach. This approach tends forms part of a Host-based Intrusion Detection System (HIDS) through which the network traffic flowing into a host as well as the behaviour of programs and users of that system is monitored.

As has been shown, the problem with fast, reliable network security is still open. There are many facets which must be considered when implementing any solution and especially when dealing with network traffic in real time. The specific issues surrounding packet inspection and possible solutions are discussed in Section 2.4.

2.4 Packet Inspection

In modern computer networks, packet inspection is employed throughout these networks to provide a variety of services. Firewalls (Section 2.2) often make use of packet inspection to aid the filtering of unwanted traffic (Zwicky et al., 2000). Intrusion Detection Systems (Section 2.3) also make use of use packet inspection to detect anomalous or malicious behaviour (Kemmerer and Vigna, 2002).

In the field of packet inspection can be further separated into three distinct groups:

- Shallow Packet Inspection or SPI (Subsection 2.4.1) - 5-Tuple based inspection used in packet filtering firewalls (Ingham and Forrest, 2002).
- Medium Packet Inspection or MPI (Subsection 2.4.2) - Application proxies such as SOCKs proxies (Koblas and Koblas, 1992).
- Deep Packet Inspection or DPI (Subsection 2.4.3) - Allows implementors to exactly monitor and inspect the *content* of network traffic (Ingham and Forrest, 2002).

2.4.1 Shallow Packet Inspection

Shallow packet inspection (SPI) is the simplest form of packet inspection. SPI systems work by monitoring just the header portion of a packet - up to and including the network layer (Ingham and Forrest, 2002). In firewalls implementing SPI, the fields of the packet headers are used to decide whether a packet should be accepted or dropped. These decisions are based on blacklists or whitelists configured by a network administrator. In Figure 2.6, shallow packet inspection will, usually, have access to the Ethernet, IP, and TCP headers.

Typical SPI devices make use of 5-Tuple based configuration to define their behaviour. Listing 2.4 gives an example a 5-tuple entry (Al-Shaer and Hamed, 2003).

```
1 <order> <protocol> <src_ip> <src_port> <dst_ip> <dst_port> <action>
```

Listing 2.4: Example of 5-Tuple based configuration

A filtering blacklist contains a combination of 5-Tuple entries defining where traffic may not flow. A filtering whitelist contains 5-Tuple entries defining where traffic may *only* flow.

Shallow packet inspection provides only the most rudimentary form of packet analysis and is useful for simple firewalls but cannot be used to enforce complex policies reliant on the packet's payload (Ingham and Forrest, 2002).

2.4.2 Medium Packet Inspection

Medium Packet Inspection (MPI) is what is used by the application proxies described in Subsection 2.2.1. MPI is used to control the traffic flow through a network at the application layer (Mogul, 1989). With MPI it is possible to enforce the use of specific applications

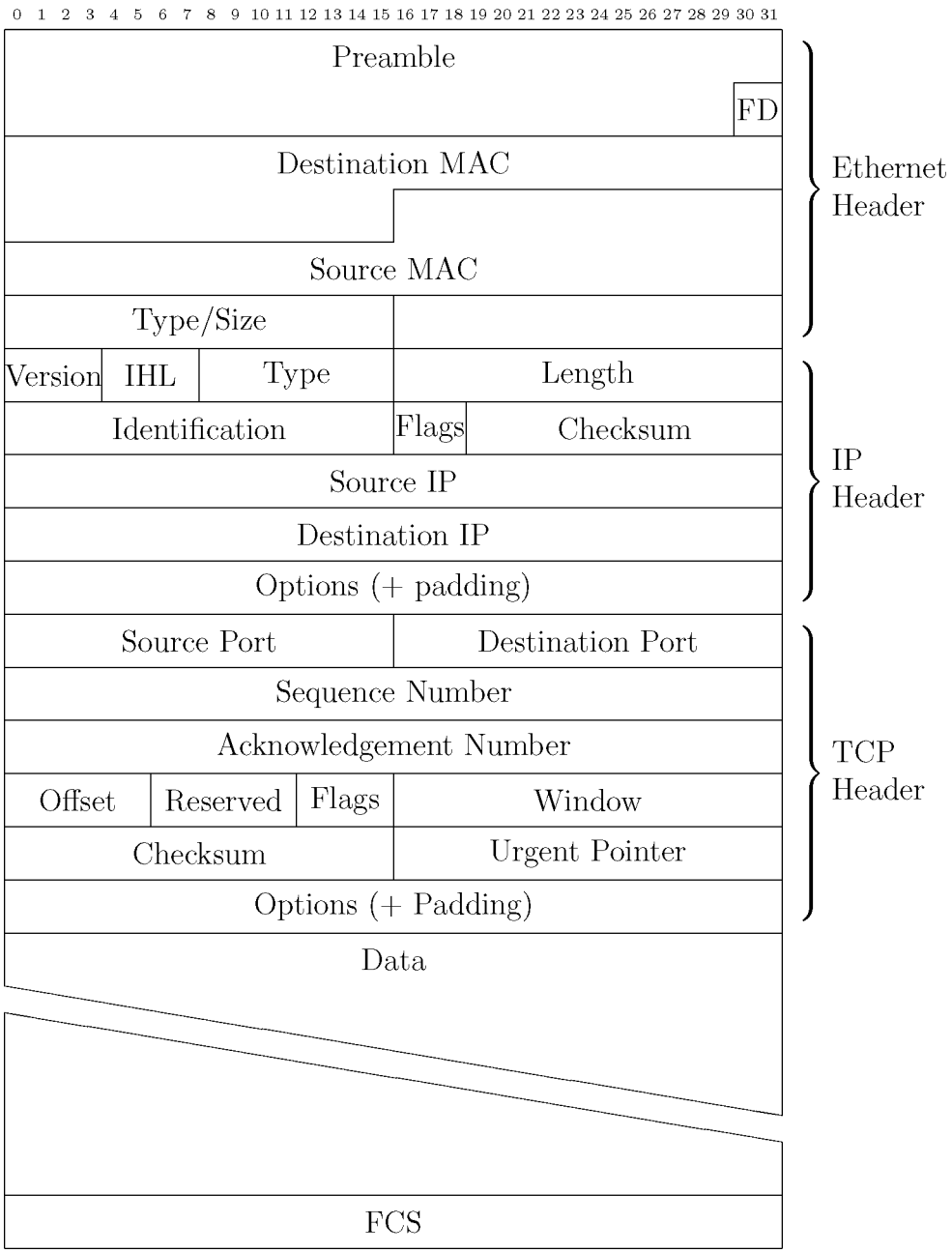

Figure 2.6: An example packet. Different levels of packet inspection have access to different protocols within a packet.

and protocols, thus limiting the potential attack surface of the network (Ranum, 1992). Systems implementing MPI are even able to restrict the kind of files being transmitted (usually by looking in the presentation layer) which can be used to curb file sharing to an extent (Parsons, 2009).

A further use of MPI is detecting anomalous behaviour in known protocols. A system implementing MPI is required to be aware of different application-layer protocols and check for packets which do not comply with the standards defined for those protocols (Handley et al., 2001). Figure 2.1 exemplifies proxies which perform MPI.

As mentioned in Subsection 2.2.1, these kinds of systems are required to have intimate knowledge of the protocols they wish to proxy. New protocols are designed and released every day. Some protocols are even obfuscated to the point where developing an application layer proxy is impossible. This type of approach to packet inspection cannot scale well if the administrators wish to continue supporting new protocols (Parsons, 2014).

2.4.3 Deep Packet Inspection

Deep Packet Inspection (DPI) is the process by which packets flowing through an Intrusion Detection System, firewall, or other system interested in network traffic are searched through for threats within their payload (Parsons, 2014). DPI can combine signature-matching and heuristics to assess the threat of the communication. In order to achieve the speed needed to match modern network bandwidth, custom application-specific integration circuits (ASICs) are often deployed to provide the speed required. For a Deep Packet Inspection system to perform correctly, firewalls need to maintain both the state of the underlying connection but also the state of any application using it. Sourdis (2007) describes DPI as "[analyse] packet contents and [provide] content-aware processing."

Sourdis (2007) describes the following requirements for DPI systems wishing to act in real time on network traffic:

- High processing throughput. Yu et al. (2006) emphasize the importance of this in their work - especially on general purpose processors.
- Low implementation cost. Custom hardware solutions present considerable costs.
- Ease of modifiability. New threats result in new rules and systems must be able to quickly implement these changes.

- Scalability. The traffic on networks is very periodic and DPI systems must be able to handle such scenarios.

It is not a trivial task for the technology behind DPI systems to keep up with modern networks. Although the speed of computer processors increases rapidly and predictably (Moore, 1965), so too does the speed of communication within and between networks (Neilsen, 1998). Furthermore, every day more and more network-based attacks are developed, these attacks are often used to create patterns and rules. The resulting patterns and rules are then used in DPI systems to detect those attacks.

DPI has many different applications. Internet Service Providers (ISPs) use Deep Packet Inspection to perform bandwidth limiting and cost analysis (Networks, 2011; Bendrath and Mueller, 2011; Mueller and Asghari, 2012). It is often useful for an ISP to know exactly what application-layer protocol is being used by customers on their network. ISPs may bill differently depending on the type of content being transferred (Mueller and Asghari, 2012). DPI also has many uses for securing a network - both at the network's edge with a firewall (Section 2.2) and within IDSs (Section 2.3).

One technique used by DPI systems is to identify the files being transferred by comparing a hash of the file to hashes of known files. First, complete streams spanning multiple TCP segments must be reassembled (Necker, Contis, and Schimmel, 2002), then by using the file or packet as an input to a hash function, a DPI system can quickly match that file to a file previously identified by the system or known to it via a signature (Callado, Kelner, Sadok, Kamienski, and Fernandes, 2010). Although fast, this approach is limited as even the smallest change to a file will cause it to be unidentifiable by its hash. A more refined, but programatically and computationally more intensive, approach is that of fingerprinting.

DPI systems can employ fingerprinting to identify the contents of a file within a packet (Parsons, 2014). To represent the fingerprint of a packet, the DPI system needs to have a deep understanding of the contents of a packet. If the packet's payload is a file from Microsoft Excel then the DPI system needs to be aware of the structure of an excel file in order to better understand its contents (Callado et al., 2010; Liao, 2015).

An example of fingerprinting used to identify traffic is given in Parsons (2014). In it Parsons describes the use of fingerprinting to identify traffic associated with the Skype[20] program. Skype uses encryption to mask the contents and even the true headers of

[20] https://www.skype.com/en/

packets that it transmits. To identify traffic from Skype, DPI systems have to use different measures.

When initiating a voice call in Skype, the Skype program will initially transmit a seemingly random burst of packets which, after further analysis by Bonfiglio, Mellia, Meo, Rossi, and Tofanelli (2007), can be shown to follow a pattern and in turn this pattern can be identified by heuristics. Thereafter, Skype traffic can be identified by Intrusion Detection Systems by the heuristics demonstrated in Bonfiglio et al. (2007).

2.4.4 Encrypted Traffic

While packet inspection, and more specifically Deep Packet Inspection, can be very effective at identifying patterns and rules in unencrypted traffic, it is far more difficult to perform the analysis on traffic that is encrypted(Sherry, Lan, Popa, and Ratnasamy, 2015; Lin, Lin, Prassana, Chao, and Lockwood, 2014). Traditionally application-layer protocols have mappings (assigned by the Internet Assigned Numbers Authority (IANA)[21]) to TCP or UDP port numbers and can be identified through those numbers. Often times, as is the case with Skype (Bonfiglio et al., 2007), even the protocol itself has been obfuscated by means of encryption and port numbers are assigned randomly (Alshammari and Zincir-Heywood, 2011). In 2005, Moore and Papagiannaki showed that the classification of traffic by port number alone is 70% accurate.

Through the analysis of the packet payload itself - and comparing it to known signatures - unencrypted traffic can be classified at near 100% accuracy (Moore and Papagiannaki, 2005). For encrypted traffic it is far more difficult. Researchers have employed techniques such as Hidden Markov models, Naïve Bayesian models, AdaBoost, RIPPER, Decision Trees, expert systems and Maximum Entropy methods (Alshammari and Zincir-Heywood, 2011). Further statistical models for classification are also employed.

Internet service providers are one of the major implementors of Deep Packet Inspection today (Hibberd, 2012). Subsubsection 2.4.5 will discuss the incentives for these ISPs to perform DPI. For many ISPs, Deep Packet Inspection only stretches as far as to identify the kinds of traffic flowing through their network and often, in the case of HTTP traffic, the specific website being accessed. ISPs and others wishing to identify traffic like this can resort to other measures. Examples of which include matching the destination or source IP address of a packet to a known domain name. These domain names can then be used

[21] https://www.iana.org/

to identify websites. Streams of small packets both towards and away from a client is a strong indication of VOIP traffic.

2.4.5 Why Perform DPI?

In discussing the technicalities of Deep Packet Inspection, a few examples of DPI in use have been briefly covered. DPI employed by ISPs for bandwidth monitoring and by network administrators for security reasons has been discussed. This subsection will formally define and categorise the reasons for ISPs, companies, and governments to perform any kind of packet inspection.

Parsons (2014) suggests that there are three main reasons for networks to perform DPI:

- **Technical** - DPI is used by network administrators for general security, access restriction and quality of service (QOS) monitoring (Parsons, 2014). As mentioned in Section 2.3, DPI was originally intended to improve the administrators' ability to detect intrusions into their network, and even prevent future intrusions (Kemmerer and Vigna, 2002). Furthermore, DPI provided vital logging which allows administrators to gain valuable insight into the kind of traffic traversing their network and, in the event of a breach, a historical view of how the attacker gained access to the network.

  A system designed to log HTTP traffic could keep track of which websites were being visited, separate traffic by upload and download or even the type of traffic being transmitted (images, movies, text, etc.). Such logs can be used to build usage patterns for users which in turn could be used by IDSs (Section 2.3) to identify an intrusion (Kemmerer and Vigna, 2002).

  DPI is often used to flag traffic as potentially interesting (for the reasons listed above) which could then be analysed *offline* (without having to match the network speed) (Yang, Liao, Luo, Wang, and Yeh, 2010).

  Deep Packet Inspection can also be used to identify the user that packet can originated from (usually via some kind of authentication) and provide services specific to that user. Within the context of an ISP, the user may only be allowed to transfer HTTP and SMTP traffic whereas other users may be allowed unrestricted traffic flow (Kumar et al., 2006; Parsons, 2014). Furthermore, ISPs may use DPI to intercept

HTTP traffic and add their own banners or advertisements to pages[22].

ISPs or network administrators may use DPI to identify time-sensitive (or realtime) protocols and give those packets priority during times of congestion.

- **Economic** - There are many economic reasons for ISPs to implement DPI. ISPs may offer different levels of service based on how much a customer is willing to pay and can then use Deep Packet Inspection to identify traffic and act according to the agreed level of service. Customers may be able to choose a basic internet connection package which limits the speed that they are able to connect to some services or specific websites with - DPI is instrumental in identifying these services or websites allowing the ISP to treat the traffic differently.

  ISPs can also use DPI to identify traffic from services that they themselves offer. That traffic may then be treated differently. Examples of special treatment to traffic include: zero rating it so that it does not count towards some kind of limited usage quota or giving the traffic preferential treatment during times of congestion. ISPs may even slow or block traffic to services offered by their competitors.

  As mentioned earlier ISPs could use DPI to distinguish traffic based on the service that a customer has chosen to pay for. Basic packages may only offer web browsing and email whereas - often at a cost - an advanced package may make the entire internet available. For a look at a rather dystopian idea of what an ISP could achieve through DPI see Figure A.1 (/u/quink, 2009) on page 143.

  The idea expressed by that graphic is that of the 'app-model' of the internet (Parsons, 2014). The concept of an 'app-model' is where connectivity is charged based on the application being used rather than the overall bandwidth consumed. An analogous example is that of an electricity supplier charging differently for the same amount of power used by a toaster and a kettle. Some ISPs try to frame this concept as a security feature that limits a client's attack surface by restricting the traffic to them. The debate surrounding these practices has been very heated. The term coined to describe the principal that these ideas subvert is Net Neutrality[23].

  A further economic incentive for DPI is that of detecting and subsequently stopping the illegal transfer of copyrighted material. ISPs using DPI could monitor traffic for signs of copyright infringement (often by using some kind of fingerprinting on media contained within the packets (Gupta, 2013)) and stop the transfers before

[22] Here is an example of this being performed by Telkom, South Africa's largest ISP: `https://www.reddit.com/r/southafrica/comments/3cnpit/telkom_is_using_a_maninthemiddle_attack_to_change/`

[23] `https://en.wikipedia.org/wiki/Net_neutrality`

they reach the recipient. ISPs are especially motivated to perform such monitoring when they themselves are holders to the rights of material[24].

ISPs are also able to generate revenue through the injection of "foreign code" (Parsons, 2014) into traffic. Such examples of this are the use of code injection for adding advertisements or tracking cookies into the packet's payload. Advertisements are paid for through some kind of ad network and the ISP would then be paid per view or click.

- **Political** - Governments have for long been concerned with the way their citizens communicate with each other. Before the advent of the internet, some governments would routinely monitor phone calls, telegrams or letters sent by people of interest to them. In the internet age, governments have been known to use DPI to monitor the communication of citizens; often this can be done in a way that is transparent to the person or group being watched.

  Governments can employ DPI to monitor network traffic for things such as: child pornography, communication that is unfavourable to the government, or even just encrypted traffic (which some governments look to ban[25]). Governments, such as the Chinese government[26], may block access to websites in an effort to limit free speech.

  The process of actively inspecting traffic for an entire country can be extremely demanding of resources and so governments often deploy a different strategy for ensure that DPI takes place. The strategy is known as *intermediary liability.*

  Intermediary liability is described as governments shifting the liability of what they deem to be illegal activity to the companies who transmit the data. It is thus those intermediary companies who are responsible for ensuring that the customers on their networks do not break the law by monitoring all traffic.

As has been discussed, there are many reasons to perform Deep Packet Inspection. Very few of these reasons seem to add positive value to the users of networks with the exception of a few security cases. The Net Neutrality argument continues, and the core technology behind it is Deep Packet Inspection.

[24] An example of this is Time Warner who operate in the US as an ISP (Time Warner Cable) and produce TV shows (through networks like HBO and Cartoon Network) and films (through production companies like Warner Bros.). See: `http://www.timewarner.com/`

[25] `http://www.itnews.com.au/news/uk-pm-wants-to-ban-encrypted-comms-399338`

[26] `https://en.wikipedia.org/wiki/Great_Firewall`

2.5 Summary

This chapter has introduced and discussed a number of different areas which have influence on Deep Packet Inspection. Section 2.1 discussed the overall field of network security and the various dangers posed to modern networks and the devices therein. Later, Section 2.2 took a deeper look into the history and status of network firewalling. It was shown how import packet inspection is in firewalls and just how important firewalls are to network security. Section 2.3 investigated the use of Intrusion Detection Systems in modern networks and saw how vital they were in keeping users of those networks protected. Intrusion Detection Systems were shown to rely heavily on Deep Packet Inspection in performing their duties. Finally, Section 2.4, took a in-depth look at the varying levels of packet inspection with an emphasis on Deep Packet Inspection.

It is with this that the prevalence and importance of Deep Packet Inspection in modern networks today has been established and the research presented hereafter justified.

Chapter 3 will investigate the field of string search algorithms and present a set of algorithms chosen to test and benchmark in the context of Deep Packet Inspection.

Chapter 3

Algorithms

String search algorithms prove useful in many different disciplines within the field of computer science (Crochemore and Wojciech, 2002). Traditionally, these algorithms have been used to search for key words or short sentences within large volumes of text (Stephen, 1994). String search algorithms are not, however, limited to searching through books. Many other kinds of information are stored in textual formats and benefit from the performance of string search algorithms. An example of this is genetic code[1] which is used to store entire genomes in a four-letter alphabet. Furthermore, the amount of binary data (of which textual data is a subset) that is stored and processed grows substantially every year.

Packet processing typically does not make much use of string search algorithms and in particular exact string search algorithms (Chaudhary and Sardana, 2011). The exact string search algorithms presented in this chapter are have been designed to run sequentially. Packet processing systems are usually implemented as custom hardware solutions and make use of highly parallelisable algorithms (Sourdis, 2007). The algorithms introduced and discussed in this chapter are best suited to implementations on the GPP-style hardware presented in Section 2.3 which, as shown there, are not particularly well suited to fast packet analysis (Chaudhary and Sardana, 2011).

The following chapter presents an introduction to Stringology (the study of string search algorithms), and then presents each of the string search algorithms selected for implementation and comparison.

[1] https://en.wikipedia.org/wiki/Genetic_code

3.1 Stringology Primer

The authoritative source of information on the exact string search algorithms - the algorithms which this research was limited to - is the *Handbook of Exact String-Matching Algorithms*, by Charras and Lecroq (2004). In it, the authors describe in detail a number of different exact string search algorithms, their seminal publications, and the special properties of each algorithm. In the text, a standard nomenclature was adopted for describing the algorithms in such a way that it was easy to compare one algorithm with one another. The same naming scheme is employed hereafter.

String-searching is defined as finding one or more **patterns** or **rules** in a piece of **text** or **input**. Patterns are represented as $x = x[0...m-1]$ where $\underline{m}$ is the length of the pattern. Text is represented as $y = y[0...n-1]$ where $\underline{n}$ is its length. The **alphabet** is the finite set of characters which the text or pattern may be comprised of. The alphabet is denoted as Σ with a size of σ. When a pattern is matched with some point in the text, the position of the match is noted by the index of the first matching character in the text.

The realm of string search algorithms has usually been within textual data (Crochemore and Wojciech, 2002). String search algorithms have, on occasion, been employed to search through biological sequences (Srikantha, Bopardikar, Kaipa, Venkataraman, Lee, Ahn, and Narayanan, 2010). Network packets, on the other hand, represent their data as a series of bytes. In some cases these bytes may just be encoded text but this is often not the case. For the purposes of this research, textual data has been reduced to its ASCII representation, i.e. `0x000` to `0xFFF`. When text is reduced to its byte representation, the same algorithms can be used for searching through packets and through plain text.

For string searching, the following terms and their definitions are relevant:

- **prefix** - a prefix p of a string a is defined as $a = p+q$ where q is possibly zero-length.
- **suffix** - a suffix q of a string a is defined as $a = p+q$ where p is possibly zero-length.
- **substring** - a substring s of a string a is defined as $a = r+s+t$ where r and t may be zero-length.

This research looks only at the string search algorithms that can be defined as exact string matching and single rule matching. As discussed earlier, a large selection of such string search algorithms has been amassed by Charras and Lecroq (2004). From this collection

of 34 exact string matching algorithms, a subset of nineteen algorithms was chosen to implement, benchmark and then compare. Table 3.1 provides a summary of the chosen string search algorithms. The summary includes: the name of the algorithm, the year it was published, its author, and the algorithmic complexity during searching. Each of the chosen algorithms share two important features: exact string matching and single rule matching. Those terms are defined as follows (Charras and Lecroq, 2004):

- **Exact String Matching** - all of the algorithms match exactly with substrings in the input text. Partial matches, no matter how similar to the pattern, are not considered matches.
- **Single Rule Matching** - all of the algorithms are designed to search for just a single rule at a time. In order to search for multiple rules simultaneously further parallelisation is needed. Two further categories of string search algorithms exists, namely algorithms which match a finite set of patterns and algorithms which match an infinite set of patterns.

Figure 3.1 shows all of the chosen algorithms plotted on a timeline based on their year or release and Big–θ classification. Every algorithm, except for the Naïve algorithm, was published in some kind of paper, journal article or technical report. The Naïve algorithm has no known year of invention. The year in which they were first published has been plotted along the x-axis.

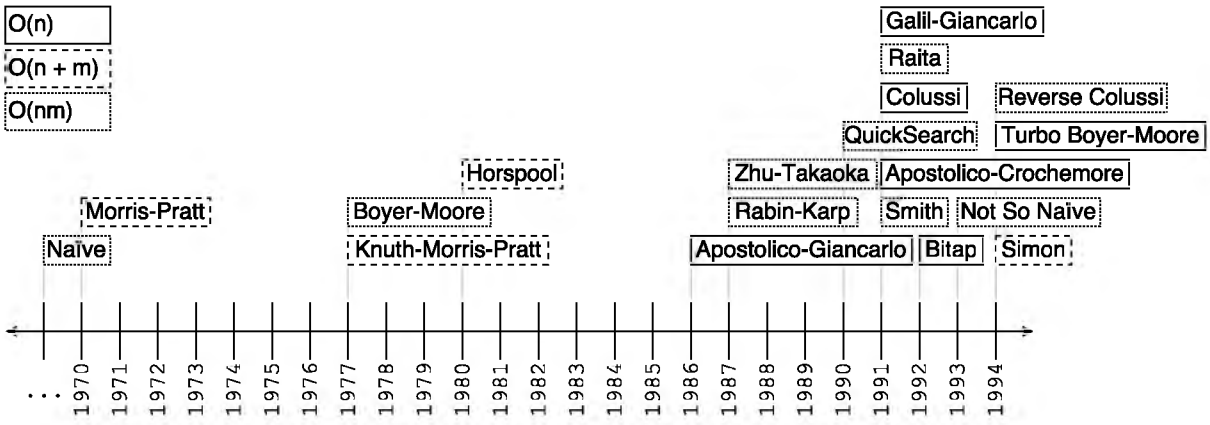

Figure 3.1: A timeline of the string search algorithms selected for this research

Each of the string search algorithms has a known theoretical performance. This performance is known as algorithmic complexity and usually written in Big-θ notation (presented in the *Time Complexity* column of Table 3.1 and as either a solid, dotted or dashed

Algorithm	Year	Author(s)	Time Complexity
Naïve			O(mn)
Morris-Pratt	1970	Morris and Pratt	O(n + m)
Knuth-Morris-Pratt	1977	Knuth et al.	O(n + m)
Boyer-Moore	1977	Boyer and Moore	O(nm)
Horspool	1980	Horspool	O(n + m)
Apostolico-Giancarlo	1986	Apostolico and Giancarlo	O(n)
Rabin-Karp	1987	Karp and Rabin	O(mn)
Zhu-Takaoka	1987	Feng and Takaoka	O(mn)
Quick Search	1990	Sunday	O(mn)
Smith	1991	Smith	O(mn)
Apostolico-Crochemore	1991	Apostolico and Crochemore	O(n)
Colussi	1991	Colussi	O(n)
Raita	1991	Raita	O(nm)
Galil-Giancarlo	1992	Galil and Giancarlo	O(n)
Bitap (Shift Or)	1992	Baeza-Yates and Gonnet	O(n)
Not So Naïve	1993	Hancart	O(nm)
Simon	1994	Simon	O(n + m)
Turbo Boyer-Moore	1994	Crochemore et al.	O(n)
Reverse Colussi	1994	Colussi	O(n)

Table 3.1: Implemented string search algorithms.

line bordering the algorithms on the timeline in Figure 3.1). This value is generally related in some way to both the length of the input and the length of the rule. Algorithmic complexity often only provides insight into processing speed where large variations in the length of the input (differing orders of magnitude) are present. In packet data, a limited range of input lengths is possible. The maximum length of a packet is defined by the maximum transmission unit (MTU) of the communications medium (Law, Diab, Healy, Carlson, Maguire, Anslow, and Hajduczenia, 2012). The performance of these algorithms may come down to minutiae within the algorithms themselves rather than their overall algorithmic complexity.

As these algorithms are designed to match just a single rule at a time, in order to match multiple rules searches needs to either be run sequentially or in parallel. In a sequentially designed system, only a single search for a rule may be run at a time. In parallel, many rules could be searched for at once. In modern processor architectures, CPUs feature multiple cores and each core is able to handle many threads at the the same time (Figure gives the basic idea of such a multi-core multi-threaded CPU). An upper bound for the number of concurrent searches exists. This upper bound is defined by the processor, the number of cores it has, and the number of hyperthreads each core supports.

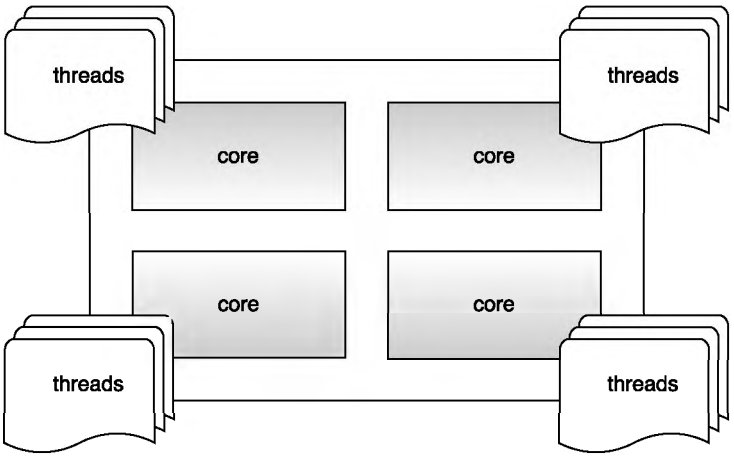

Figure 3.2: An example of a multi-core, multi-threaded CPU (Jenkov, 2014).

For a system with ten cores, running a search in a single thread (and therefore on a single core) is only using at most ten percent of the potential processing power. Splitting the search across ten cores should serve to give close to an order of magnitude speed increase. If those ten threads were saturating the processing speed of all ten cores (using one hundred percent of the available processing power), adding more threads might serve to reduce the benefits seen before. The overhead of switching between threads on a single core may start to adversely affect the processing time.

Many of the implemented algorithms make use of a preprocessing phase to aid the string search. The preprocessing phase will generally create some kind of data-structure based on the pattern to be searched for. As long as the pattern - or set of patterns - remains the same between successive searches, then the preprocessing phase does not need to be run again. In the context of packet inspection this means that the preprocessing phase will only need to run when a change is made to the ruleset.

The following sections will introduce and discuss each of the algorithms selected for testing.

3.2 Naïve

The Naïve algorithm is the simplest, and oldest, of the algorithms chosen for this research. It is often referred to as the Brute Force algorithm. The algorithm performs no preprocessing and always shifts the search window by exactly one position to the right (Charras and Lecroq, 2004). The algorithm has a search complexity of $\theta(nm)$ and an average of $2n$ text comparisons are made.

The algorithm works by checking all characters between 0 and $n - m$. After a mismatch, the window is shifted to the right by one character (Charras and Lecroq, 2004).

3.3 Morris-Pratt

The Morris-Pratt algorithm (1970) is an early refinement of the Naïve algorithm. Morris and Pratt noted that the Naïve algorithm 'wastes' information gathered during previous attempts when conducting the current attempt. The refinements made to the Naïve algorithm allow the Morris-Pratt algorithm to shift more than the single character done by the Naïve algorithm and simultaneously keep track of some already-compared pieces of text (Charras and Lecroq, 2004). As a result a number of character comparisons can be saved and the overall speed of the search improved (Aho, Hopcraft, and Ullman, 1974).

In order to keep track of the suffixes already compares, the Morris-Pratt algorithm makes uses of a 'next' table constructed using the search pattern. The Morris-Pratt algorithm has a search time-complexity of $\theta(n + m)$ and performs at most $2n - 1$ comparisons.

3.4 Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm (1977) is a refinement of the work previously done on the Morris-Pratt algorithm in Morris and Pratt (1970). The Knuth-Morris-Pratt algorithm hopes to improve the maximum length of shifts that the algorithm can perform during searching. The Knuth-Morris-Pratt algorithm has a search time-complexity of $\theta(n + m)$.

The maximum number of comparisons of a single character in the input is limited to $log_{\Phi}(m)$ where $\Phi = \frac{1+\sqrt{5}}{2}$ or the golden ratio (Knuth et al., 1977; Charras and Lecroq, 2004).

3.5 Boyer-Moore

The Boyer-Moore algorithm (1977) is often considered one of the fastest exact string search algorithms (Charras and Lecroq, 2004). The Boyer-Moore algorithm is used in the

very popular GNU Grep (Haertel, 2010) tool. It has a time complexity of $\theta(nm)$ and performs at most $3n$ character comparisons.

The Boyer-Moore algorithm performs its searches from the right-most character of the search window to the left most. In the case of a mismatch, two precomputed tables are consulted to determine how much to shift by (Charras and Lecroq, 2004). The algorithm is very fast for large alphabets(Lecroq, 1995). Large alphabets are defined as being much larger than the length of the pattern being searched for (Boyer and Moore, 1977).

3.6 Horspool

The Horspool algorithm (1980) is a refinement and simplification of the Boyer-Moore algorithm (Boyer and Moore, 1977). It simplifies the Boyer-Moore algorithm by only making use of the 'bad-character' shift table presented in Boyer and Moore (1977). Horspool noted that the 'bad-character' shift table was quite efficient for large alphabets; alphabets such as those provided by the ASCII (Bemer, 1960) or UTF-8 (Pike and Thompson, 1993) encodings.

The searching time-complexity of the Horspool algorithm is $\theta(nm)$ (Charras and Lecroq, 2004). The average number of comparisons with a single character can be shown to be between $\frac{1}{\sigma}$ and $\frac{2}{\sigma+1}$ (Baeza-Yates and Gonnet, 1992), where σ is the length of the alphabet.

3.7 Rabin-Karp

The Rabin-Karp algorithm (1987) (often referred to as Karp-Rabin) makes use of hashing to avoid constant recomparisons with the pattern. By computing a hash (or fingerprint) of the pattern (Aho, 1990) during the preprocessing phase, the algorithm is able to compare the hash of the current window into the text with that known hash of the pattern (Charras and Lecroq, 2004).

The speed of the algorithm can be negatively affected if the fingerprint is slow to compute or if many false positives are produced (Karp and Rabin, 1987). The Rabin-Karp also benefits from being able to search for many strings at the same time, with each pattern only adding an additional integer comparison during the search.

The Rabin-Karp algorithm has a searching time complexity of $\theta(nm)$.

3.8 Zhu-Takaoka

Yet another variant of the Boyer-Moore algorithm (Boyer and Moore, 1977) is that of the Zhu-Takaoka algorithm (Feng and Takaoka, 1987). The Zhu-Takaoka algorithm uses two characters for the Boyer-Moore algorithm's 'bad-character' shift table.

The Zhu-Takaoka algorithm exhibits a $\theta(nm)$ time-complexity for searches (Lecroq, 2007).

3.9 Quick Search

The Quick Search algorithm is another variant of the work done on the Boyer-Moore algorithm by Boyer and Moore (1977). In the Quick Search algorithm, when a mismatch has occurred, the window will necessarily shift by at least one character. Because of this, the first character after the window can be used in the Boyer-Moore algorithm's 'bad-character' shift table.

The Quick Search algorithm has been shown to have excellent performance for short patterns in long alphabets (Sunday, 1990; Stephen, 1994; Lecroq, 1995; Crochemore and Lecroq, 1996). The algorithm has a search time-complexity of $\theta(nm)$ (Charras and Lecroq, 2004).

3.10 Smith

The Smith algorithm (1991) is a amalgamation of both the Horspool (Horspool, 1980) and Quick Search (Sunday, 1990) algorithms. Smith noticed that computing the 'bad-character' shift with the rightmost character of the window (as done in the Horspool algorithm) could, sometime, give more of a shift than if it were calculated using the method described by the Quick Search algorithm (Smith, 1991). Smith proposed to take the maximum value from *both* of those method in order to maximise the shift each time (Smith, 1991).

The Smith algorithm completes its search with a time-complexity of $\theta(nm)$ (Charras and Lecroq, 2004).

3.11 Apostolico-Crochemore

The Apostolico-Crochemore (1991) algorithm builds on the work done in Knuth et al. (1977) by making use of the 'next' shift table. In the Apostolico-Crochemore algorithm, the maximum number of comparisons is bounded by $\frac{3}{2}n$ (Charras and Lecroq, 2004). This bounding is generally good for Deep Packet Inspection as it guarantees deterministic processing times for the large number of inputs usually seen in such an application. The Apostolico-Crochemore algorithm has a search time-complexity of $\theta(n)$.

3.12 Colussi

The Colussi algorithm (Colussi, 1991) is yet another refinement of the work done for the Knuth-Morris-Pratt algorithm by Knuth et al. (1977). The Colussi algorithm has a searching time-complexity of $\theta(n)$.

The algorithms itself works by splitting the pattern into two half, working from right to left on the first half and then left to right on the second (Breslauer, 1992; Galil and Giancarlo, 1992). As with the Apostolico-Crochemore algorithm, the number of comparisons is bounded by $\frac{3}{2}n$ (Charras and Lecroq, 2004).

3.13 Raita

The Raita algorithm (1991) has a slower search time-complexity than that of the Colussi, at $\theta(nm)$. This algorithm was designed to take advantage of what the author calls 'character dependencies' in English text.

In English, there is often a high dependency which governs the occurrence of characters in text. This dependency is strongest for characters positioned next to each other (Raita, 1991) and weakest at word boundaries. As an example, the character 'q' is almost always followed by the character 'u'. Raita argues that this dependency makes comparison of successive symbols from left to right (and from right to left) not profitable (Charras and Lecroq, 2004). Raita suggests that character comparisons should occur from the point of weakest inter-character dependency to strongest.

In practice, the algorithm works by first comparing the last characters, then the first, then the middle, and finally by comparing each of the other characters. The algorithm makes use of the Boyer-Moore algorithm's 'bad-character shift' function to compute the shift in the case of a mismatch.

In Smith (1994), the author argues that the improvement suggested in Raita (1991) was merely a result of the behaviour of the compiler, rather than the algorithm itself.

3.14 Galil-Gaincarlo

The Galil-Giancarlo algorithm (1992) builds on the work done on the Colussi algorithm (Colussi, 1991). It too has a search time-complexity of $\theta(n)$ (Charras and Lecroq, 2004). It improve the upper bound of the maximum number of text comparisons from $\frac{3}{2}n$, in the Colussi algorithm, to $\frac{4}{3}n$.

Galil and Giancarlo noticed that the Colussi algorithm had very poor performance for patterns which begin or end with repeated characters (Breslauer, 1992). When encountering such a set of repeated characters, the Colussi algorithm will shift by only a single character. Galil and Giancarlo devised a way to shift by more characters in such a case.

3.15 Bitap

The Bitap (or Shift Or) algorithm (Baeza-Yates and Gonnet, 1992) makes use of bitwise operations to perform its search and extends very easily to allow for non-exact matching (Baeza-Yates and Gonnet, 1992; Charras and Lecroq, 2004). The algorithm performs its search in $\theta(n)$ time. The algorithm works by representing the state of the search as some number, and then subsequent comparisons inflict some kind of arithmetic for logical operation on that number to represent the next state (Baeza-Yates and Gonnet, 1992; Wu and Manber, 1992; Crochemore and Lecroq, 1996).

3.16 Simon

The Simon algorithm (1994) is an example of a deterministic finite state automaton (DFA) used for string searching. It makes improvements on the basic DFA presented by Charras and Lecroq (2004) *et al.*

Simon notes that the number for edges used in the basic DFA is excessive. He posits that the number of edges can be bounded by $2m$ (Charras and Lecroq, 2004) and thus the maximum number of text comparisons is bounded by $2n - 1$. The algorithm has a time-complexity of $\theta(n + m)$.

3.17 Not So Naive

The Not So Naïve algorithm (Hancart, 1993) is an improvement on the base Naïve algorithm. It works works in much the same way as the Naïve algorithm except that the Not So Naïve algorithms tries to find two repeated characters and, when it does, shifts by two places instead of one (Charras and Lecroq, 2004).

The Not So Naïve algorithm has an average time-complexity of $\theta(nm)$.

3.18 Turbo Boyer-Moore

As the name would imply, the Turbo Boyer-Moore algorithm (Crochemore et al., 1994) is a variant of the Boyer-Moore algorithm (Boyer and Moore, 1977) or a simplification of the Apostolico-Giancarlo algorithm (Lecroq, 1995). The algorithm works by *remembering* the last suffix matched during the previous attempt, allowing the algorithm to jump by the length of that suffix or perform a 'turbo' shift.

The Turbo Boyer-Moore algorithm performs the search with a $\theta(n)$ time-complexity and will do, at most, $2n$ character comparisons (Charras and Lecroq, 2004).

3.19 Reverse Colussi

The Reverse Colussi algorithm (1994) is yet another variant of the Boyer-Moore algorithm by Boyer and Moore (1977). The algorithm has a search time-complexity of $\theta(n)$ and at worst will perform $2n$ comparisons (Charras and Lecroq, 2004) whereas the Boyer-Moore algorithm will perform $3n$ (Colussi, 1994). Like the Colussi algorithm (Colussi, 1991), the Reverse Colussi algorithm works by splitting the pattern into two halves.

3.20 Summary

In recent years many more string search algorithms have been invented (Faro and Lecroq, 2013). Most of these algorithms fall under the categories of automata- and bit-parallelism-based algorithms. These categories of algorithms differ from *most* of the algorithms listed above which are mostly classified as character-based comparison algorithms (Faro and Lecroq, 2013).

In Faro and Lecroq (2013), the authors compare many of the more modern string search algorithms experimentally. Through those experiments a few algorithms stood out as good performers. For patterns of longer length and alphabets of varying sizes, the SSEF algorithm (Külekci, 2009) was fastest overall. For smaller rule lengths the results varied a fair amount with with the EBOM (Fan, Yao, and Ma, 2009) and FSBNDM (Faro and Lecroq, 2009) algorithms showing good results.

In the preceding chapter, each of the chosen algorithms has been presented and discussed. Each of the nineteen algorithms help to form part of the backbone of the field of string search algorithms. Figure 3.3 gives an idea of the genealogy of the chosen string search algorithms.

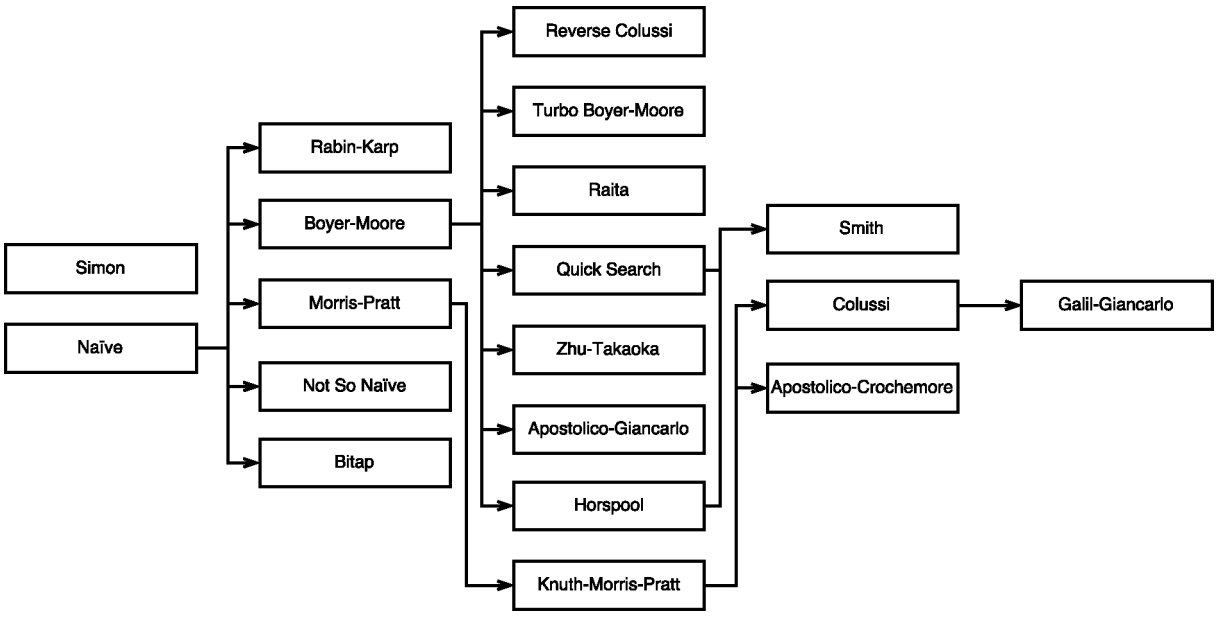

Figure 3.3: String search algorithms family tree

Figure 3.3 paints a rich picture of the constant collaboration and improvement that occurs within the string search algorithm field. The selected algorithms have two distinct categories: Naïve- and DFA-based algorithms. The Morris-Pratt, Boyer-Moore, Rabin-Karp,

Bitap, and Not So Naïve algorithms all descend from the Naïve algorithm. Refinements to the Boyer-Moore are made by the Horspool, Apostolico-Giancarlo, Zhu-Takaoka, Quick Search, Raita, Turbo Boyer-Moore, and Reverse Colussi algorithms. The Morris-Pratt algorithm is improved by the Knuth-Morris-Pratt algorithm. The Smith algorithm improves on both the Horspool and Quick Search algorithms. The Colussi and Apostolico-Crochemore algorithms make further improvements to the Knuth-Morris-Pratt algorithm and the Galil-Giancarlo algorithm makes refinements to the Colussi algorithm.

Each of the algorithms was designed to find a single pattern within a body of text. Some of these algorithms, like the Boyer-Moore algorithm, have found their way into mainstream use with implementations in popular programs (Haertel, 2010).

Chapters 5 and beyond present a harness for testing the algorithms listed above. They also perform various tests and present those results.

Chapter 4

Datasets

For the purpose of this research a number of datasets needed to be assembled, and constructed. Each of the datasets was designed to allow very specific questions to be posed about the algorithms. For the artificially constructed datasets, special care was taken to limit the possible variables affecting the processing speed of the algorithms. Table 4.1 presents the five datasets and other pertinent information.

Two types of data were used to construct the datasets. The first was PCAP data. PCAP files are logical collections of packets (Garcia, 2008). PCAP files are often created using the `tcpdump`[1] tool listening on a network interface. Packet data makes up the majority of the datasets, and is represented by: *Dataset A*, *Dataset C*, *Dataset D*, *Dataset E*, and *Dataset F*. The second type of data is textual and that makes up just *Dataset B*. Most of the datasets are PCAP files as they provide the best representation of network traffic. The textual data is a good representation of the kind of data traditionally parsed by these algorithms. Sections 4.1 to 4.6 discuss each dataset.

For the purpose of this research, textual data is treated as a single input whereas PCAP files are split into their constituent packets and each packet is treated as a separate input. This is discussed further in Chapter 5.

4.1 Dataset A

Dataset A is a PCAP file containing 10000 packets of real-world DNS data. The data represents requests from a network of clients to DNS servers and their responses.

[1] `http://www.tcpdump.org/`

Name	Description	$\bar{n}$	# Inputs
Dataset A	Real-world DNS traffic	109.61	10000
Dataset B	Full text of *Alice in Wonderland* by Lewis Carroll	163780	1
Dataset C	Randomly generated DNS traffic with a payload size between 0 and 1500 bytes	770.89	10000
Dataset D	Dataset C edited so that the payload just contains matches to the required rules	770.89	10000
Dataset E	Dataset C edited so that each packet is filled with a random number of matches	769.92	10000
Dataset F	Packets of fixed length - filled with a random number of matches	1500	10000

Table 4.1: Datasets used by the test system during the tests

This dataset is important as it represents actual network traffic - it is similar to the traffic that can be found passing through a network firewall or being examined by Intrusion Detection Systems and has a high proportion of textual content. This dataset does, however, have a flaw. The packets contained in *Dataset A* are on average about one hundred and ten bytes long. This is far shorter than a packet of average length which flows through networks today.

On a traditional ethernet network without jumbograms (Borman, Deering, and Hinden, 1999) the maximum payload size (referred to as Maximum Transmission Unit (MTU)) for packets traversing the network is 1500 bytes (Law et al., 2012). As seen in Table 4.1, the average length of the packets in *Dataset A* is approximately one hundred and ten bytes. Although these packets are a good representation of real-world DNS traffic, they are generally much shorter than a packet of average length.

Packet size variation is important as the length of the packet affects the time that each packet takes to process. As a results of the smaller processing time, the behaviour of the algorithms processing the packets may be concealed by overheads in the test system itself.

4.2 Dataset B

Dataset B is unique among the the datasets chosen for this research as it does not represent network traffic. *Dataset B* represents a large volume of text. Each of the string search algorithms introduced in Chapter 3 was originally designed to search through large textual datasets.

The text chosen for this dataset was *Alice in Wonderland* by Lewis Carol. The book was chosen because it is very large in comparison to the other datasets and the structure is a fair representation of prosaic English text. The book itself has for many years been freely available in the public domain and the copy used for this dataset was sourced from Project Gutenberg[2].

As discussed at the start of this chapter, text-based datasets are treated a bit differently to their packet-based counterparts. In a packet-based dataset, each packet is treated as an individual input the algorithm and are searched separately. In *Dataset B*, the entire body of text is considered as a single input. This approach allows for the use of *Dataset B* as a baseline for the expected performance of each algorithm.

4.3 Dataset C

Dataset C was created as a way of overcoming the shortfalls presented by *Dataset A* (Section 4.1). This dataset is a set of randomly generated DNS packets up to 1500 bytes in length.

To create such a dataset, the authors used Wireshark's `randpkt`[3] tool (Ramirez, 1999). The command used is given in Listing 4.1.

```
1 $ randpkt -b 1500 -c 10000 -t dns random_dns.pcap
```

Listing 4.1: Creating 10000 random DNS packets for *Dataset C*

Using the `randpkt` tool, ten thousand packets could be generated, with an overall mean length of seven hundred and seventy bytes. That's more than six hundred bytes longer than the mean packet length in *Dataset A*.

As *Dataset C* was randomly generated, the contents of each packet is just garbled bytes. This limits the effectiveness of algorithms which take advantages of partial or full matches in the test to skip comparisons altogether. This dataset will, however, allow us judge the speed of each algorithm whilst limiting the number of possible matches to the rules.

Figure 4.1 gives an example of a packet in *Dataset C*. The first few bytes of each packet is the standard header - filled with random contents. After the header, a random number

[2] https://www.gutenberg.org/ebooks/11
[3] https://www.wireshark.org/docs/man-pages/randpkt.html

of bytes are generated and placed into the rest of the packet. Each packet has a length, n, of up to 1500 bytes.

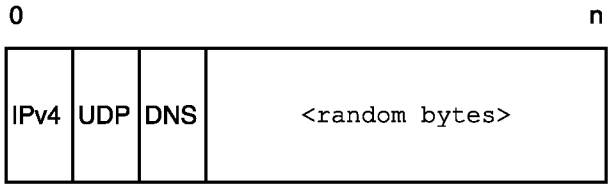

Figure 4.1: *Dataset C*

4.4 Dataset D

Dataset D was derived by editing *Dataset C* so that the contents of each packet would only contain matches. Each packet of the dataset remained the same length whilst its payload was replaced with the rules being searched for. The specific rules are listed in Section 7.1.

To create this dataset a Python[4] library, Scapy[5], was used to edit *Dataset C*. The set of rules was initially compiled into a Python list[6]. Following that the Scapy library was used to read each packet from *Dataset C*'s PCAP file. For each packet the order of the list of rules was randomised and the list turned into a single string. The string was then concatenated with itself repeatedly until the length of the string exceeded the length of the packet's payload. The string was then truncated to exactly the length of the packet's payload and finally the payload was replaced with the new string.

The resulting PCAP file was similar to the PCAP file for *Dataset C* in that each packet remained the same length as before. For an algorithm searching for the rules used to compile this dataset, matches will be found constantly.

Figure 4.2 shows what the structure of a packet in *Dataset D* would look like. Each packet from *Dataset D* is a modification of the corresponding packet in *Dataset C*. Instead of a random number of bytes being generated and used to fill the packet's payload, the rules are repeated and placed there instead. Each packet is the same length of its partner in *Dataset C*, with a maximum length of 1500 bytes.

[4] https://www.python.org/
[5] http://www.secdev.org/projects/scapy/
[6] https://docs.python.org/2/tutorial/introduction.html#lists

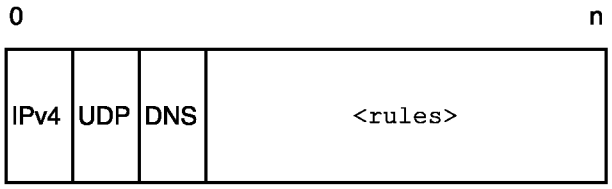

Figure 4.2: *Dataset D*

4.5 Dataset E

Dataset E is a further variation on *Dataset C* and *D*. In *Dataset D*, the number of matches in a packet is directly related to the length of each packet's payload. Larger packets necessarily have more matches than shorter packets. In *Dataset E*, the number of matches in a packet is arbitrarily defined by a randomly generated number.

The process for creating this dataset is very similar to the process described in Section 4.4. In *Dataset D*, the string of randomised rules is repeatedly concatenated with itself until the length of the new string exceeds the length of the packet's payload. For *Dataset E*, the same string is repeatedly concatenated but this process ends when the length of the new string exceeds some randomly generated number between zero and the length of the packet's payload. The bytes of the payload from zero to that randomly generated number are then replaced by the new string and the remaining bytes are left untouched.

The resulting PCAP file contains packets of random length with an arbitrary number of matches in each.

Figure 4.3 gives an example of a packet from *Dataset E*. In each packet, a random number of bytes are inserted up until the point marked n'. Between n' and n the rules are repeated the same way as *Dataset D*. Each packet in *Dataset E* is still the same length as the corresponding packets from *Dataset C* and *Dataset D* but with a random number of guaranteed matches in each.

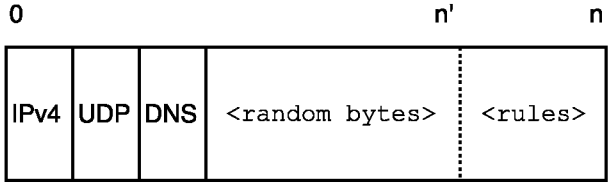

Figure 4.3: *Dataset E*

4.6 Dataset F

Dataset F was created in order to deal with a limiting property identified in *Dataset E*. In *Dataset E*, every packet has two independently varying values which govern the number of matches to rules in each. The independently varying values are the length of the packet and the randomly chosen number of bytes to fill part of the payload with. This property makes it very difficult to freeze one variable whilst allowing the other to vary. In *Dataset F*, the variable length packets have been eliminated while still allowing each packet to have a arbitrary number of matches.

`randpkt`, which was used earlier to create *Dataset C* will not work to create this new dataset. The packets generated by `randpkt` are always of random length; there is no way to fix the length of the packets. To overcome this limitation, Scapy was employed to create each packet.

Creating the packets needed for *Dataset F* proved to be a very similar process to that used in *Dataset D* and *Dataset E*. Initially, a default DNS packet is created using Scapy[7]. That snippet is given in Listing 4.2. The payload is generated in much the same way as what was done for *Dataset E*. A random number of rules are concatenated together and that is set as the packet's payload, since each of the packet need to have the same length the rest of the payload is set to random bytes.

```
1    packet = IP()/UDP()/DNS()
```

Listing 4.2: Creating a bare DNS packet with Python and Scapy

With *Dataset F*, the length of the packet has been kept constant while the number of matches is able to vary. Being able to edit the dataset itself means that later, when the results are analysed, the causes of processing time differences are better able to be separated.

Figure 4.4 shows the structure of a packet in *Dataset F*. Unlike *Dataset D* and *Dataset E*, each packet in *Dataset F* is 1500 bytes long. Packets are again created with the usual IP, UDP and DNS headers. After each header a random number (marked from the end of the header to n') of bytes is inserted. Finally the last part of each packet, from n' to 1500, is filled with repeated rules.

[7] `http://www.secdev.org/projects/scapy/`

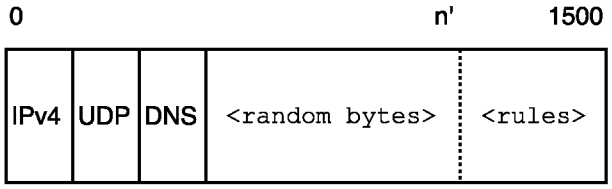

Figure 4.4: *Dataset F*

4.7 Summary

Each of the datasets described in the sections above have been designed in way that gives the researcher the most control over the data. These datasets allow for testing of properties of the algorithms (given in chapter 3) which are specifically important to Deep Packet Inspection. Part II presents the framework for testing these datasets against various algorithms.

Part II

Packet Inspection Framework

Chapter 5

Design

5.1 Introduction

For the purpose of this research a testing framework had to be constructed. The reason for this was to provide a platform to test string search algorithms in the context of packet inspection. Unlike textual data such as plain text files, packet processing must be tested by connecting to either a packet capture handle or by reading from a packet capture file.

The domain of string search algorithms has, traditionally, been for searching within the buffer of a text editor or in text files saved to disk. The packet data encountered by network firewalls or Intrusion Detection Systems is almost never so static. A packet capture handle provides a live interface between a program and a network interface; a PCAP file is a saved representation of a capture handle over a period of time.

As the purpose of this research is to provide a comparison between string search algorithms as they process packet data, the test system could be designed in such a way that it read PCAP files. Reading PCAP files, rather than reading packets directly off the wire, provides a way to reliably reperform tests using the exact same input data.

This chapter looks at the design of the test system developed to test the string search algorithms. It examines the broad design and focuses on a few goals for the finished design.

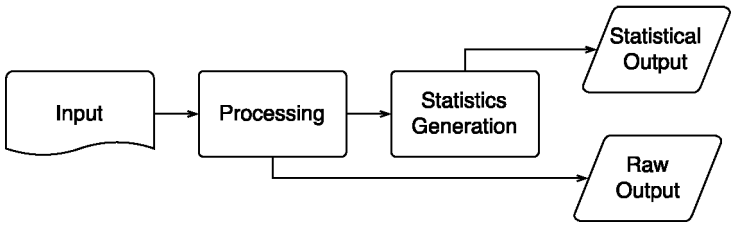

Figure 5.1: A diagram describing the broad design of the testing system.

5.2 Overall Design

The overall design of the test system is presented in Figure 5.1. The goal of this design was to provide a simple but configurable method of giving the test system input, allowing it to be configured by that input, run through the various tests, and then provide output in the form of both statistical information and raw data.

5.3 Input

As mentioned earlier, the test system needed to be easily configurable and the configuration had to persist in such a way that the same test could be run again at a later time. To meet those goals the design for the system's input was developed as shown in Figure 5.2.

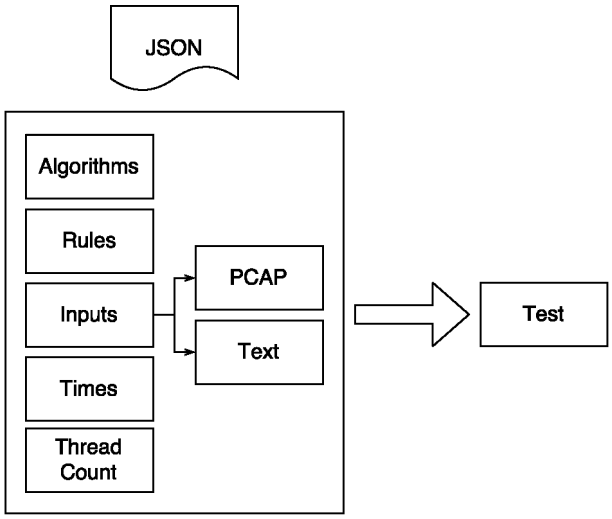

Figure 5.2: A representation of the input to the test system.

The test system's input consists of a single JSON (Crockford, 2006) file (an example of which can be found in Listing 6.1 in Chapter 6) with the following fields:

- **algorithms** - A list of the algorithms to test. Each algorithm needed to correspond with an algorithm listed in Table 3.1.
- **rules** - A list of rules or patterns to search for in the input.
- **inputs** - A list of inputs to perform the search on. Inputs could include both text and PCAP files.
  - For text files, the entire contents of that file is treated as a single input.
  - For PCAP files, each packet in the file is considered an individual input to the system.
- **times** - The number of times to perform the search. In order to limit the influence of external variation on the algorithm's performance and to establish a large number of results for each input, the system is designed to repeat tests as often as configured.
- **threads** - The number of active threads allowed during each search. Some algorithms perform differently depending on the number of threads available to it.

The JSON (Crockford, 2006) file is ingested by the system and converted into a test object. The test object represents an entire test and all details about that test such as the results of the test.

5.4 Processing

Once the test object has been constructed, the testing is able to begin. The design for the test hopes to ensure that each of the algorithms, rules and inputs are fairly evaluated and measured. A diagram representing the test design is presented in Figure 5.3. During a test the system performs the following sequence:

1. The system iterates through each of the test runs specified in the configuration (labelled in Figure 5.3 as "*For each run*"). At this point a unique identity is generated and assigned to the run, this is known as the run ID. Run IDs help to distinguish this run from another run performed using the exact same configuration.
2. For each run, the system will iterate through each of the algorithms specified in the configuration.

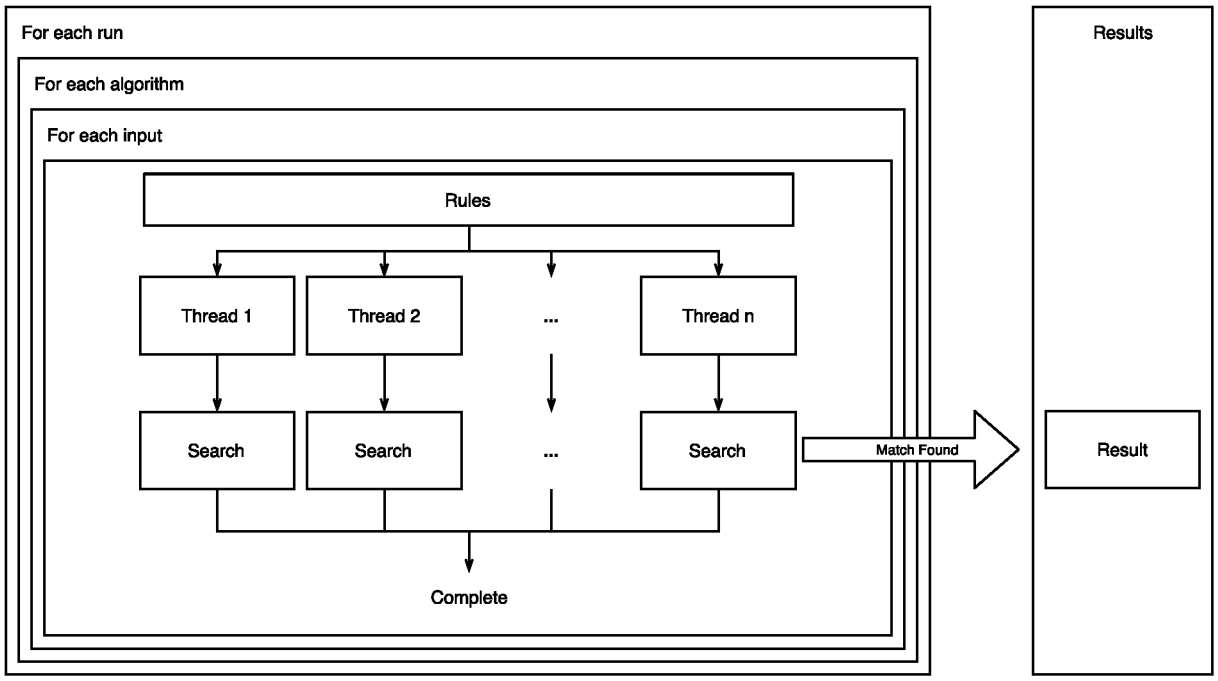

Figure 5.3: The functioning of the main processing logic of the test system.

3. For each of the algorithms, the system will iterate through each of the inputs.

4. At this point the test system should execute the search. The system will split each rule into its own thread and, when the total number of threads does not exceed the configured value, a search will commence.

5. Once each of the rules has been searched for the next iteration can continue.

6. During testing and if a match is found, the location of the match is logged to a result object. Each result object forms part of a larger results object which is managed by the test system.

After the test system has completed all its tests, just the results of the tests should be left. The results describe each iteration of the processor above as well as the locations at which matches were found.

5.5 Statistics Generation

It is from those results that the statistical generator is able to run. The statistics generator was designed to provide statistics for the different aspects of algorithm testing. In Figure

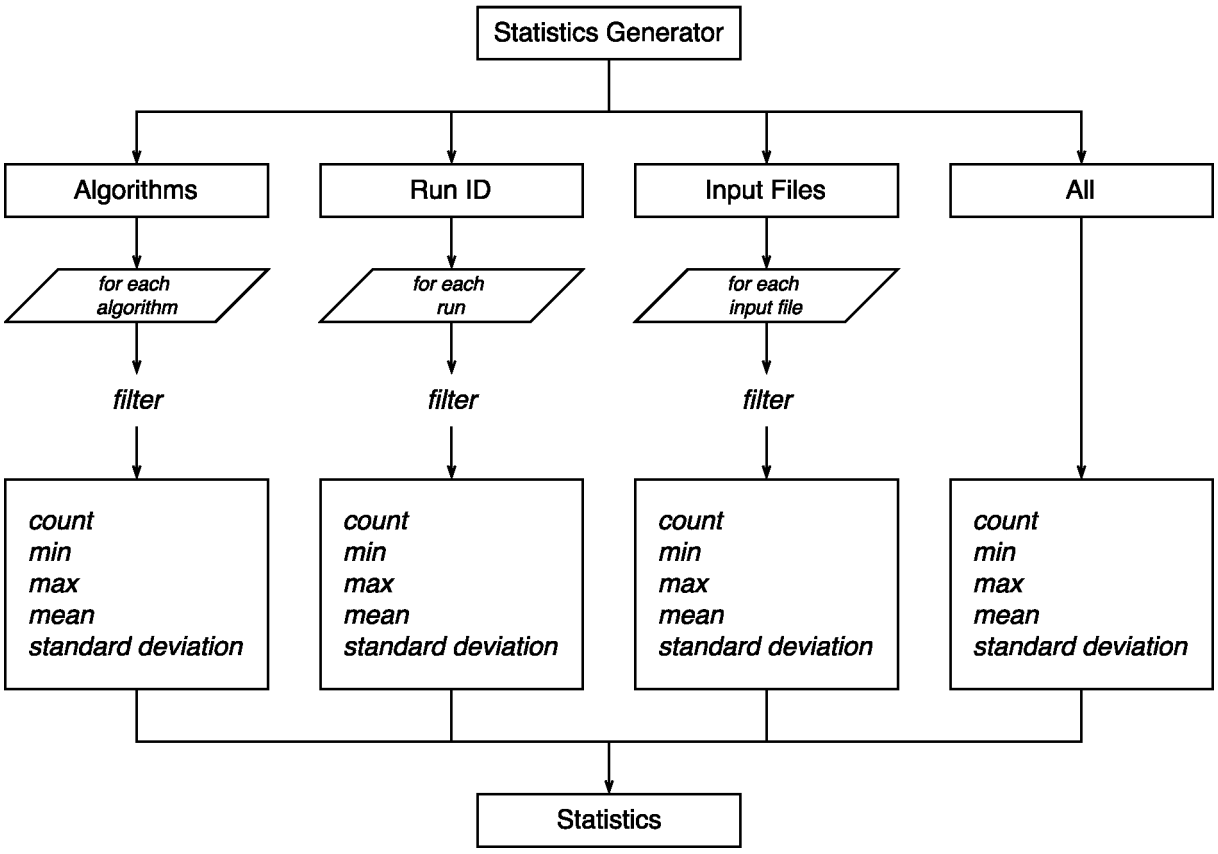

Figure 5.4: The flow of the statistics generation design.

5.4, the statistics generator is separated out into four components. Each of the components provides insights into the data by giving its minimum value, maximum value, mean value, the standard deviation and a count of the number of results. Each component is described below:

- **Algorithms** - For each of the algorithms specified in the input (Figure 5.2), the statistics generator creates statistics pertaining to a specific algorithm.
- **Run ID** - Because the test system support multiple test runs within a single test, the statistics generator creates statistics for each of those tests. This is especially useful in isolating tests which may present extreme data.
- **Input Files** - As described earlier, PCAP files are separated into the packets they contain - each packet is an input. For text files the file itself is an input. The statistics generator gives statistics per input file rather than for every input.
- **All** - This provides statistics for all of the results from the tests.

5.6 Statistical Output

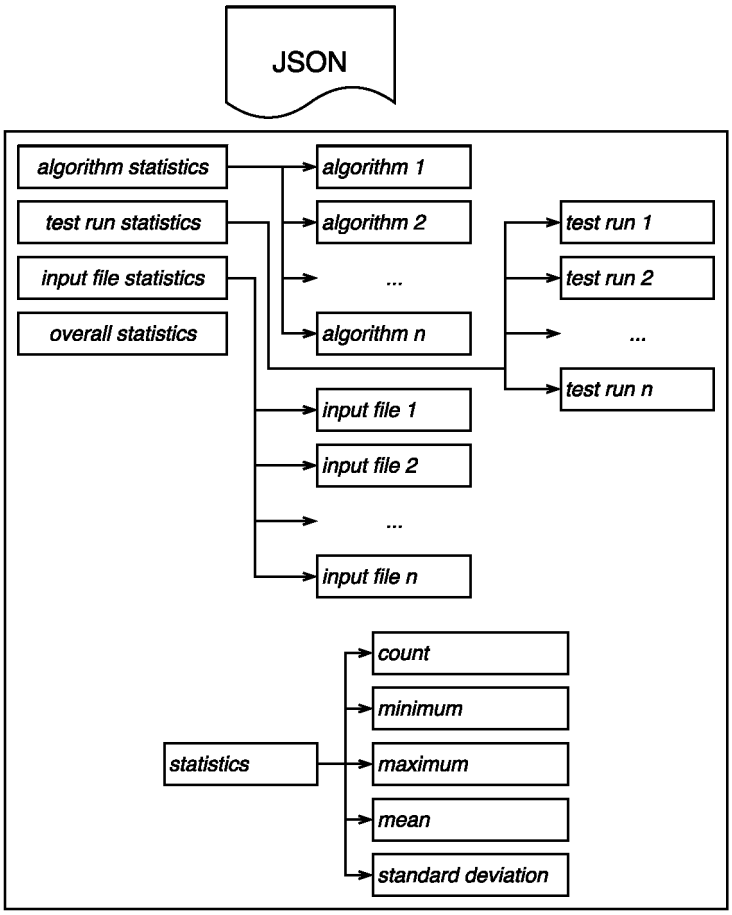

Figure 5.5: The structure of the output of the statistics generation

The test system is also responsible for writing the statistics to file. Figure 5.5 shows the design of a statistics file. For an example of such a file, see Listing 6.3 in Chapter 6. Each category of statistic is group in the output file. Within each category the statistics for the individual elements may be found. For the algorithm category, each element would correspond to an algorithm name. In Figure 5.5, this is labeled *algorithm 1* to *algorithm n*. The same goes for each other category of statistic. The statistics generated by the system are fairly simple but provide valuable insight into the behaviour of the algorithms. The following describes each one:

- *count* - the total number of results used to create these statistics.
- *minimum* - the smallest elapsed time for the results.
- *maximum* - the largest elapsed time for the results.

- *mean* - the average amount of time elapsed in the results.
- *standard deviation* - the standard deviation of the elapsed time.

Statistics such as the minimum and maximum values provides insight into the nature of outliers within the data. The standard deviation allows us to judge how variant the results of the algorithms are.

5.7 Raw Output

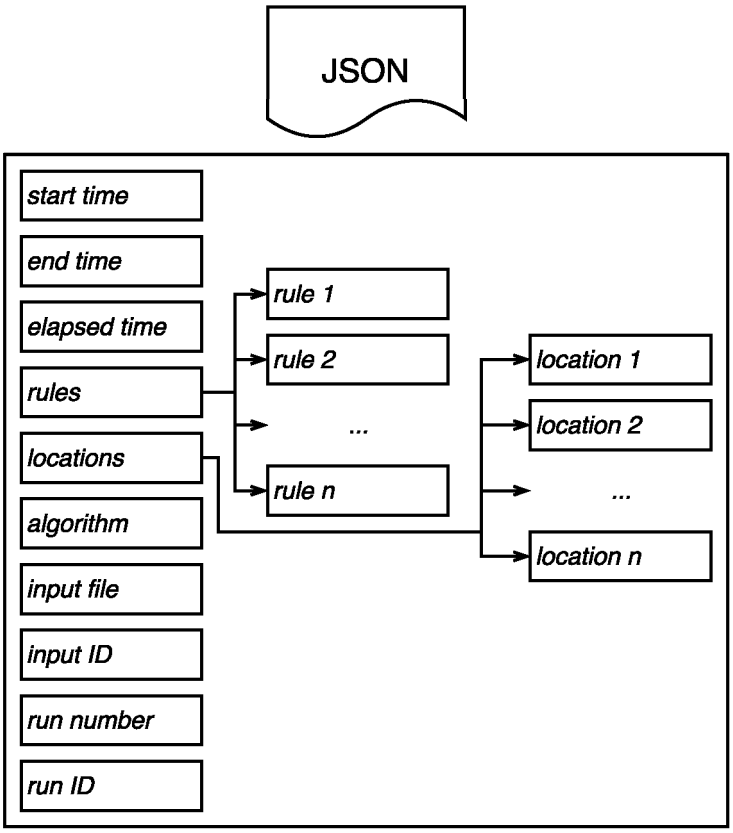

Figure 5.6: A representation of the output of the test system.

Following the statistics generation and output, the test system is then ready to write out the results. The results, like the input configuration, are in JSON (Crockford (2006)). As discussed earlier, for every search there is a corresponding result object. Each result object documents every aspect about the search that took place. The file that the test system writes out is a list of every result object generated during its run. Figure 5.6 gives an example of just one result of the many written to file once the testing is complete. For

an example of the result object, see Listing 6.4 in Chapter 6. In the list below, the details of that result object are dicussed:

- **start time** - the time that this particular search started.
- **end time** - the time that the search ended.
- **elapsed time** - the time the search took (effectively $endtime - starttime$).
- **rules** - the list of rules searched for. With its current design, the test system searches for the same rules in every different iteration. Future designs may require that the list of rules changes between search and so knowing which rules were searched for is very important.
- **locations** - a list of every location that the algorithm matched the rules to the input.
- **algorithm** - the algorithm used to perform the search.
- **input file** - the name of the file in which the input is contained.
- **input ID** - the ID of the input itself. Each of our packet-based datasets had ten thousand inputs. Each input was uniquely identified by this ID assigned to it.
- **run number** - the number of the run that this test corresponds to.
- **run ID** - the unique ID associated with the test run. This number distinguishes runs from different tests.

5.8 Summary

The design presented here represents the overall structure of the the test system. Through the use of easily replaceable modules it is hoped that the design of the test system promotes easy use and extensive future expansion.

The next chapter discusses the details surrounding the implementation of the test system.

Chapter 6

Implementation

Implementing the test system proved to be one of the more time consuming parts of this research. The original intention was for the test system to be implemented using the Rust[1] programming language.

The Rust programming language is touted as fast, memory safe, and highly concurrent. A full prototype system was developed, complete with a few algorithms, and proved to work as intended. At that point it was decided that a change of direction was needed. It was found difficult to create a test system that was as flexible and extensible as planned. The prototype system also suffered from being very difficult to extend.

As a result of this, the system was rewritten in Java[2]. Java was chosen for a few reasons. Firstly, it is the language that the author knows best and, over time, Java has proven to be an excellent language for developing robust, and highly extensible applications. The main drawback of using Java is that it is notably slower than a traditional language such as C to compile and run.

The relative speed lost when running software written in Java does not have a large effect on the research goal of this work. This work intends to compare string search algorithms with *each other* within the context of Deep Packet Inspection. Since the speed of the algorithms relative to each other is the important metric the absolute speed of program execution does not matter.

The system itself was developed using the IntelliJ IDEA[3] platform. The Pcap4J[4] library

[1] https://www.rust-lang.org/
[2] https://www.java.com/en/
[3] https://www.jetbrains.com/idea/
[4] https://github.com/kaitoy/pcap4j

was employed in order to interface with the packet capture files. Extensive unit tests for all of the search algorithms were implemented to ensure that the algorithms performed as intended. Each unit test was designed in such a way as to be tested against every single algorithm. With this it could be confirmed that the algorithms performed consistently.

Each of the algorithms was implemented by the author according to the design outlined in the respective originating papers. For references to those papers, see Chapter 3. The author decided on reimplementation rather than seeking a library. For the actual implementation see the author's git repository[5].

The test system itself was implemented as a command line application with all configuration being supplied by the configuration file. The system would output information to `stdout` as well as log to file. The statistics and results file were written to disk.

6.1 Example Test

In order to properly describe how the system was implemented and subsequently how a test would be run, the following section is intended to explore the system by running an example test.

The screenshots shown as Figures 6.1, 6.2, 6.3, and 6.4 show what is written to `stdout` during the example run. A number of lines have been omitted from between Figures 6.2 and 6.3 and from between Figures 6.3 and 6.4 for brevity. The omitted line repeated what had already been discussed.

6.1.1 Program Startup

Figure 6.1 shows output of the test system from the start of the program until the number of threads has been set. The following list describes what is happening within the program to produce the output seen in that figure:

- In the first 6 lines, the system prints out information about itself and about the author.

[5] `https://github.com/KieranHunt/dpi-algorithms-java`

Figure 6.1: Test run example screenshot 1. From the start of the system to setting the number of threads.

- By default the test system looks at the first command line argument for the location of the configuration file. If the configuration file isn't specified as a command line argument or if the specified file does not exist, the test system looks in the current directory for a file named `testConfiguration.json`. In this case the file was specified on the command line.

- The test system is able to find the file specified on the command line at the location identified. It is from this file that the test system will construct the forthcoming test.

- Once the file is found, it is parsed and an `Input` object is created. The *Input* object represents all of the data found in the configuration file.

- The first item that the system looks for in the configuration file is the list of rules. These are the the same rules that will be searched for later in the test. In this case the system found forty rules listed in the file.

- Next the system will try to identify the algorithms specified. A factory method is used to match the string representation of an algorithm to a list of known algorithms. If the algorithm can be matched, then it is created. If an algorithm is

incorrectly spelt or it has not been implemented an error is returned. At this point the preprocessing steps for each of the algorithms are run. The preprocessing steps require that the system be aware of each rule. The preprocessing phases complete fairly quickly and the system is ready to move on.

- In this example test, four algorithms have been specified in the configuration file.
- The system then checks for inputs. Inputs can either be text or PCAP files. Text files are used to create a single input and PCAP files are split so that a single input is created for each packet in the file.
- In this case a single file has been specified: `fixed_length_random_contents.pcap`.
- Inside the input file the system has found ten thousand individual packets. These packets are subsequently instantiated as individual inputs.
- The test system will then check to see how many times the tests should be run - again by checking a variable set in the configuration file. In our example the tests have been set to run twenty times. Twenty times was chosen to help promote statistical significance.
- Finally the system will check the maximum number of threads to use. This example test has been configured to use eighteen threads.

At this point the system is poised to start the tests. All of the preprocessing and configuration has been completed.

6.1.2 Testing

Figure 6.2 shows the start of a test. This is the same process represented in Figure 5.3 of the original design. The following describes the output listed in that figure:

- Like in the first six lines of Figure 6.1, the processing portion of the test system prints out a few lines to show that it has starting the tests.
- The system then prints out information relating to the tests. This information is designed to give a rough idea of how long the tests will take based on the number of variables to test. In this case the system was given ten thousand inputs, forty rules and four algorithms, it was set to repeat the test twenty times. This leads to the creation of *32 000 000* result objects.

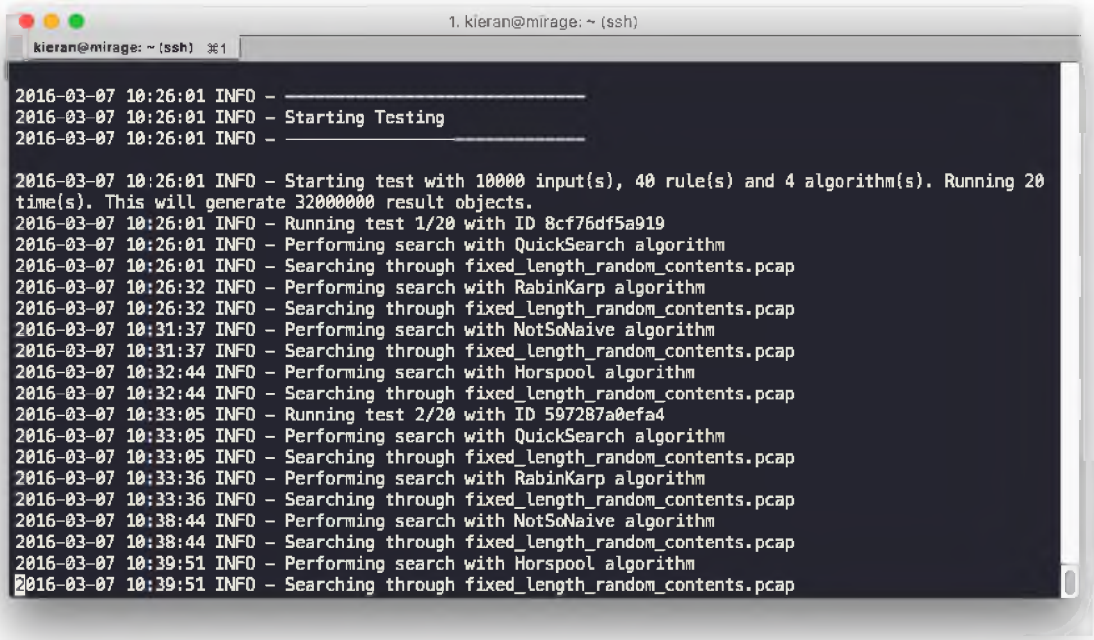

Figure 6.2: Test run example screenshot 2. From the start of the testing to somewhere into the tests.

- Following that, the system commences testing.
- As shown previously in Figure 5.3, the system starts by iterating through each of the test runs. The number of the test run is specified here as `1/20` and the test run's ID is also given.
- For each test run, each of the algorithms is iterated through. In this example the system has started with the Quick Search algorithm and follows that with Rabin-Karp.
- For each algorithm, the test system iterates through each of the inputs. Since the PCAP files can contain many thousands of separate inputs, not every input ID is listed during the search - merely the original input file is printed.
- The output is then repeated as the system iterates through each of the inputs, algorithms, and runs.

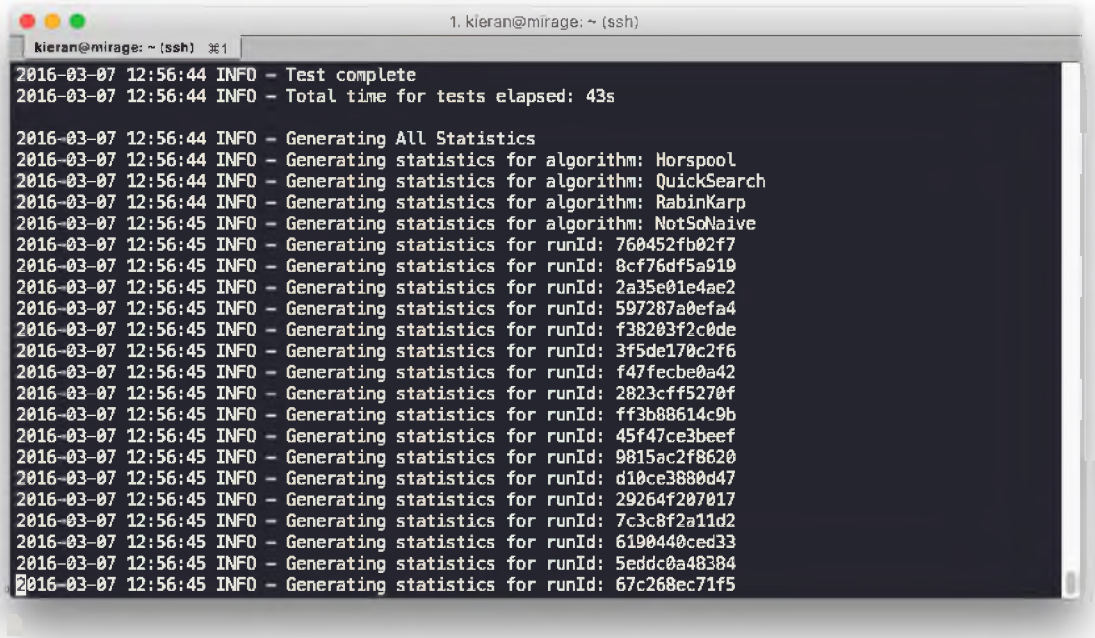

Figure 6.3: Test run example screenshot 3. End of the testing to statistics generation.

6.1.3 Statistics Generation

Following the completion of the search, the system generates statistics described by Figure 5.4. The following describes that process:

- The first two lines are the end of the testing process. Those lines state that the testing is finished and that this particular test took 43 seconds to complete.
- As described in Figure 5.4, the statistics generation is split into four separate parts.
- The test system uses a filter to isolate only the results belonging to the current category. In the screenshot, the algorithm and run ID categories are shown.
- The statistics generation begins with the algorithm category. Here the statistics are generated for all results relating to each of the algorithms. The algorithms in this particular test were: Horspool, Quick Search, Not So Naïve, and Rabin-Karp.
- Following that, statistics are generated for each of the run IDs. This allows us to isolate runs which may show extremes in processing times. These extremes are possibly related to other processes running on the test machine.

6.1.4 Output

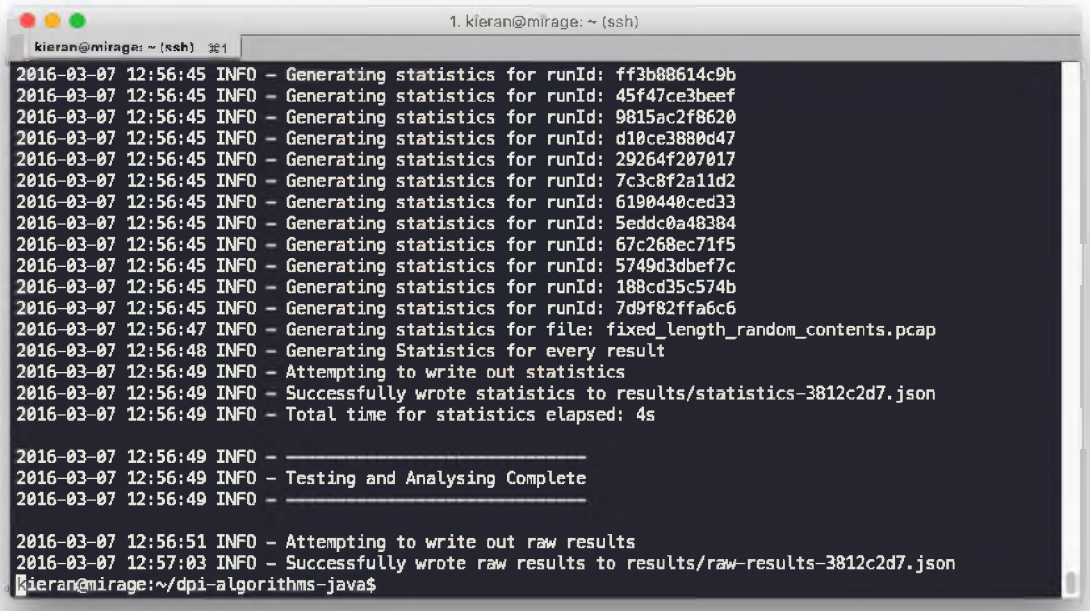

Figure 6.4: Test run example screenshot 4. Statistics generation to completion.

In the fourth and final screenshot (Figure 6.4), the system finishes generating the statistics, and writes out to file. That process is described as follows:

- The first part of this screenshot, lines one to fourteen, shows the last segment of the statistics generation. In this particular example the system was configured to run the tests twenty times. As such, statistics are generated for each run ID associated with a repeated test.
- Following the generation of statistics for each of the test runs, the system will then generate statistics about each of the input files. The input files generally group inputs of similar type (see the datasets listed in Table 4.1) and so statistics on a per-file basis are relevant. In this particular test there was just a single input file, `fixed_length_random_contents.pcap`, and so statistics are generated for just that file.
- Finally, as discussed in Chapter 5, statistics are generated for the entire set of results.
- Every single set of statistics contains the same information: minimum, maximum, mean, count, and standard deviation.

- Following the completion of the statistics generation, the test system will write the statistics to file. By default, the statistics are written to `results/statistics-<ID>.json`. Where `<ID>` is the unique identifier for this set of tests. The statistics are written out in JSON format (Crockford, 2006).
- The time elapsed whilst generating the statistics is then printed. In this case the statistics generation took four seconds to complete.
- Once the statistics generation is complete, the system prints out some text letter the user know.
- Finally, the test system will write the raw results to file. The raw results are written out as JSON (Crockford, 2006) and are written to `results/raw-results-<ID>.json` where the `<ID>` is the unique identifier for the run - the same as above - which separates results of one run from the results of another.
- For each run, the run ID is printed.

6.1.5 Test Configuration

As discussed before (In Sections 5.3, 5.6, and 5.7 and in Subsection 6.1.4), the test system uses JSON (Crockford, 2006) as a standard way of both ingesting configuration and as a way of outputting raw results and statistical information. Having a structured and documented way of transferring information to and from the application means that experiments are easily reproducible. This reproducibility stems from the ability to easily log and save the configuration used for a particular test; allowing for use in any subsequent run. Furthermore, data structured as JSON (Crockford, 2006) is widely used and understood by many systems and programming languages (Crockford, 2006).

Listing 6.1 gives an example of a test configuration used as input for the test system. This particular test configuration file was used to produce the test shown in Figures 6.1 to 6.4.

The structure of Listing 6.1 directly corresponds to design of the test configuration shown in Figure 5.2. As with Figure 5.2, there are five different fields which must be present for the system to run. Each of those five fields is discussed below:

- **`algorithms`** (lines 2 to 7) - the different string search algorithms to be tested are listed here. In this example, the Horspool, Rabin-Karp, Quick Search and Not So

```
1  {
2    "algorithms": [
3      "Horspool",
4      "RabinKarp",
5      "QuickSearch",
6      "NotSoNaive"
7    ],
8    "rules": [
9      "time",
10     "person",
11     ...,
12     "msn"
13   ],
14   "inputs": [
15     {
16       "type": "pcap",
17       "location": "fixed_length_random_contents.pcap"
18     }
19   ],
20   "times": 20,
21   "threadCount": 18
22 }
```

Listing 6.1: Example test configuration JSON file.

Naïve algorithms have been selected. The JSON specification (Crockford, 2006) does not include namespacing like in other notations such as XML (Bray, Paoli, and Sperberg-McQueen, 1998). Consequently, there is no way to limit which strings can be given as algorithm names. In the test system itself, checks are performed comparing the given algorithm name against a list of algorithms known to the system.

- **`rules`** (lines 8 to 13) - here the rules are given in a similar way to the algorithms. Each rule is specified individually and in the form of a string. In its current form, the test system can only handle rules in the form of text-based strings, although adding some kind of byte-based input method can be trivially achieved in the future.

- **`inputs`** (lines 14 to 19) - the inputs to be searched through are given here. As discussed previously, the inputs can either be in the form of text or PCAP files. Just a single type of input is used in Listing 6.1. The given input is of PCAP type and the location is specified. The configuration requires that both the type of the input as well as the location at which it can be found be specified for every input. In this example the input file with the name `fixed_length_random_contents.pcap` is given; this corresponds to *Dataset F*.

```
1 java -jar dpi-algorithms-java.jar testConfiguration.json
```

Listing 6.2: Running the test system.

- **times** (line 20)- here is where the number of time each test should be repeated is set. In this example the number of times that the tests have been set to run is twenty.
- **threadCount** (line 21)- the number of threads to use is given here. Any positive integer is valid but performance can heavily influenced by this number.

Typically the test system is run by passing the test configuration file in as the first parameter on the command line. The system could also be run through the development IDE or, in future, perhaps some kind of web interface. Listing 6.2 shows the command used to run the test system. Note that `testConfiguration.json` is the file name of the configuration file specified in Listing 6.1.

6.1.6 Statistics Output

Similarly to the configuration of the tests, the test system uses JSON (Crockford, 2006) to format its output, both for the statistical and raw results. The statistical output, as discussed earlier, is given a file name of the form `results/statistics-<ID>.json` where `<ID>` is the unique identity for that test. The unique identifier for each test is generated using Java's UUID Library. Listing 6.3 gives an example of the statistics output by the system.

The test system was designed to output the statistics like Figure 5.5. The actual output of the system given in Listing 6.3 closely matches the original design. The list below examines the finer details of that output:

- **algorithmStatistics** (lines 2 to 11) - Statistics pertaining to each of the algorithms tested. In the example four algorithms were tested. Only the statistical output has been shown for Rabin-Karp for brevity.
- **testRunStatistics** (lines 12 to 21) - Each test run has statistics associated with it. In Listing 6.3, the statistics for the test run with the ID `188cd35c574b` are given on line 13.

```
{
    "algorithmStatistics": {
        "RabinKarp": {
            "count": 200000,
            "min": 27001653,
            "max": 18532053519,
            "mean": 31470558,
            "standardDeviation": 46101820.53767535
        },
        ...
    },
    "testRunStatistics": {
        "188cd35c574b": {
            "count": 40000,
            "min": 1752744,
            "max": 12028122395,
            "mean": 11239891,
            "standardDeviation": 61286959.98985055
        },
        ...
    },
    "inputFileStatistics": {
        "fixed_length_random_contents.pcap": {
            "count": 800000,
            "min": 1680008,
            "max": 26757414816,
            "mean": 11294479,
            "standardDeviation": 72743902.6433408
        }
    },
    "overallStatistics": {
        "count": 800000,
        "min": 1680008,
        "max": 26757414816,
        "mean": 11294479,
        "standardDeviation": 72743902.6433408
    }
}
```

Listing 6.3: Example statistical output.

- **inputFileStatistics** (lines 22 to 30)- Statistics are given for each of the input files used to create the inputs. In this example the results for the `fixed_length_random_contents.pcap` file are given.
- **overallStatistics** (lines 31 to 37) - Statistics for every result are given too. These help to check the overall performance of the tests and provide a benchmark for comparing specific statistics to.

The system uses nanoseconds as its time unit of measurement. Every output using a unit of time is presented in nanoseconds. Each statistics object can be broken down as follows:

- **count** - A count of the number of results evaluated in this statistics object. For the Rabin-Karp-specific statistics, there were a total of two hundred thousand result objects used to generate these statistics.
- **min** - The minimum elapsed time featured in all of the result objects used to create these statistics. For the overall statistics - the statistics representing all results created during the test - the value was 1680008 nanoseconds.
- **max** - Similarly to the minimum elapsed time, the `max` field indicates the maximum elapsed time for the set of results used to create the statistics. For the `inputFileStatistics`, and specifically the file with the name `fixed_length_random_contents.pcap`, the maximum elapsed time was 26757414816 nanoseconds.
- **mean** - the mean amount of time for a packet to be searched through in this grouping of results. For the test run labeled with the ID `188cd35c574b`, the mean time is 11239891 nanoseconds or about eleven milliseconds.
- **standardDeviation** - Finally the standard deviation for the results is given on lines `8`, `18`, `28`, and `38`. For the Rabin-Karp algorithm in this example, the standard deviation is given as 46101820.53 nanoseconds.

6.1.7 Raw Results Output

The last file that the system writes to disk is the raw results. The raw results are every result created during the entire running period of the test system. They are identified by `raw-results-<ID>.json` where the `<ID>` is the unique identifier assigned to this test - the same identifier used for the statistical output. The structure of the raw results follows

```
1  [
2    {
3      "start": 206079193307938,
4      "end": 206079207132342,
5      "elapsed": 13824404,
6      "rules": [
7        "time",
8        "person",
9        ...,
10       "msn"
11     ],
12     "locations": [],
13     "algorithm": "RabinKarp",
14     "inputFile": "fixed_length_random_contents.pcap",
15     "inputID": "bf75faa5",
16     "runNumber": 1,
17     "runId": "188cd35c574b"
18   },
19   ...
20 }
```

Listing 6.4: Example raw results output

exactly with the design given in Figure 5.6. Listing 6.4 provides an example of the raw results written to file.

- **start** (line 3) - The time that this particular test started. This is the number of nanoseconds since the start of the Unix epoch (Thompson and Ritchie, 1975)[6].
- **end** (line 4) - This is the time that the test ended. Similarly to the `start` value, it is measured in nanoseconds since epoch.
- **elapsed** (line 5) - This is the `start` value subtracted from the `end` value to give the time elapsed during this particular search. 13824404 nanoseconds or about 138 milliseconds in our example.
- **rules** (lines 6 to 11) - A list of the rules searched for during the test. This list, at the time of writing, should be the same for each result in the test. For future additions to the test system the rules may vary per search. Our example test had forty rules but that list has been shortened to save space.
- **locations** (line 12)- The locations in the input at which a result was matched. This particular input did not have a single match to the rules.

[6]This value is usually defined as seconds since the start of the Unix epoch but for the purposes of these tests, wherein greater granularity was desired, nanoseconds since the start of the Unix epoch was used.

- **algorithm** (line 13) - The algorithm that performed the searched and produced this result.
- **inputFile** (line 14) - The file from which the input was sourced. Since different input files generally group inputs of similar type it is important to record which file this input emerged from.
- **inputID** (line 15) - The unique identifier for the input. This allows speeds to be compared for the same inputs in different tests.
- **runNumber** (line 16) - The number of the run in which the test was performed.
- **runId** (line 17) - The identifier for the run. since run numbers are reset when the test system starts and increase monotonically, this value uniquely identifies each run.

6.2 Summary

Chapter 6 presented the implementation of the packet inspection framework used throughout the body of this research. The outcome of this development work was a stable, robust and extensible system which allows for countless different test configurations and automates much of the manual processes which would ordinarily be associated with performing these kinds of tests. The test system itself produces detailed results which, in the case of the statistical results, can be used to compare the algorithms or, in the case of the raw results, are perfectly suited to ingestion by another system for further analysis.

Earlier, in Chapter 5, the design of this system was presented. The objective was to allow for the easy and repeatable testing of string search algorithms on network traffic and text files.

Part II has shown the structure and objectives of the test system used to test the string search algorithms. Next, in Part III, the testing of each of the algorithms using this test system is performed and their behaviour analysed.

Part III

Testing and Analysis

Chapter 7

Initial Algorithm Comparison

In Chapters 1 to 6, the the full context of this research has been established. The implications of Deep Packet Inspection (DPI) in modern networks was discussed. The need for approaches to DPI which can scale in the manner required by modern networks was identified. Chapter 3 discussed string search algorithms in detail.

The goal of this research is to establish a credible benchmark of string search algorithms in the context of Deep Packet Inspection. To achieve that, a collection of tests were run on each of the algorithms mentioned earlier. Each test sought to answer different questions pertaining to the algorithms' performance when processing packet data.

Part III separates the testing of the algorithms and their subsequent discussion into two distinct chapters. This chapter looks at an overall comparison of the algorithms and analyses their performance. From those results, four distinct algorithms were selected based on their processing speed. Those algorithms were then further analysed to better quantify their packet processing performance. In the next chapter (labeled as Chapter 8), the algorithms selected in this chapter are put through further tests in an effort to properly examine how they behave when processing packets of varying structure.

Each chapters contains a set of questions which are asked about the algorithms. Those questions are then answered using testing, statistical analysis and examination of plotted data.

Each of the graphs, and much of the statistics, have been created using the R programming language (Ihaka and Gentleman, 1996). The graphs themselves were produced using

the ggplot2[1] library (Wickham, 2011). This library proved invaluable in producing high quality plots, allowing the results to better speak for themselves.

7.1 Rules

In order to consistently test each of the string search algorithms, a standard set of rules was compiled. These rules were used as search patterns for the algorithms throughout the testing. Table 7.1 presents each of the selected rules.

Table 7.1: Rules used throughout the algorithm testing

Rules			
time	person	year	way
day	thing	man	world
life	hand	part	child
eye	woman	place	work
week	case	point	government
google	facebook	youtube	baidu
yahoo	amazon	wikipedia	qq
twitter	taobao	live	sina
linkedin	weibo	ebay	yandex
hao123	vk	bing	msn

The rules were specifically chosen as they are expected to appear often in both general English (of which *Dataset B* exemplifies) and in domain names (which can be found throughout *Dataset A*). The domain-name based rules may be similar to those used in a corporate firewall in order to restrict traffic to certain websites.

The first twenty rules were sourced from the Oxford English Corpus' *Facts about the language*[2] wherein the top twenty five most common nouns, verbs and adjectives in English are listed. The top twenty most common nouns were selected for testing as they appear often in both *Dataset A* and *Dataset B*.

The second twenty rules were sourced from Alexa's *The top 500 sites on the web*[3] wherein the publishers list the 500 most popular web sites on the internet. The top twenty were selected as they should appear often in various DNS requests.

[1] `http://ggplot2.org/`

[2] The list of words was sourced from `https://www.oxforddictionaries.com/words/the-oec-facts-about-the-language`

[3] `http://www.alexa.com/topsites`

It is using these rules that much of *Dataset D*, *Dataset E*, and *Dataset F* were constructed as discussed in Chapter 4.

7.2 Test Hardware

In order to ensure consistent results, the tests needed to be run on the same hardware each time. A server-style computer, made available through the researcher's research group, was chosen for this work. The device has the following important specifications:

- Two Intel® Xeon® E5-2620[4] Processors - a six core CPU with 2 hyperthreads per core for a total of 24 threads.
- 64 GB of RAM
- more than 5 TB of hard disk space.
- Debian 8.3
- Linux Kernel 3.16.0-4-amd64
- Java 8

It is with this system the the following results were generated. Each test was run using the command given in Listing 6.2 on a normal user account. The GNU Screen[5] terminal multiplexer was employed to detach shell sessions so that the test system could run autonomously and improve resilience to power cuts.

Each of the tests completed in this chapter were run with the rules discussed in Section 7.1. Eighteen threads were used consistently throughout and the tests were set to repeat 20 times each to both keep each test fair and to limit the influence of outside factors on the performance of each algorithm.

[4] http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI

[5] https://www.gnu.org/software/screen/

7.3 Algorithm Performance

The goal of this research is to establish the performance on these string search algorithms when processing packet data. The string search algorithms have shown themselves to be highly performant when searching through large volumes of text (Crochemore and Wojciech, 2002; Charras and Lecroq, 2004; Lecroq, 2007; Faro and Lecroq, 2013); the performance of these algorithms for very constrained inputs such as network packets is unknown.

For network devices implementing packet inspection, and in particular those who employ Deep Packet Inspection, the packet inspection should not adversely affect the overall speed of the network (Kumar et al., 2006). In network firewalls, this is extremely important as they present a single point through which all network traffic flows (Zwicky et al., 2000). Chapter 2 covers this is much more detail.

7.3.1 *Dataset A*

The goal of the first test was to establish a comparison of each of the string search algorithms using *Dataset A*. Since this dataset contains real-world data, comparing algorithms with it should give a good initial indication of their performance in practice. Figure 7.1 shows the results of that comparison. In that figure, the algorithms are ordered based on their mean packet processing times, where the algorithm with the smallest time is on the left and algorithms with increasingly longer packet processing times appear to the right.

By examining Figure 7.1, a few points are immediately clear. First, every algorithm, except for the Not So Naïve algorithm (Section 3.17), show extremely similar results. The faster algorithms on the left have a mean processing time of around 0.17 ms. The Not So Naïve algorithm posted a mean processing time 0.2 ms greater than that, at 0.37 ms.

The fastest string search algorithm observed during this this test was the Quick Search algorithm (Section 3.9) with the Horspool algorithm (Section 3.6) trailing closely behind.

The results of this test may, at first, seem fairly inconclusive. Although it is clear that the Not So Naïve algorithm performed very poorly, the other algorithms have such similar results that one may assume this test proves very little. On the contrary this test has shown us a very important piece of information. It would seem that, because *Dataset A* is

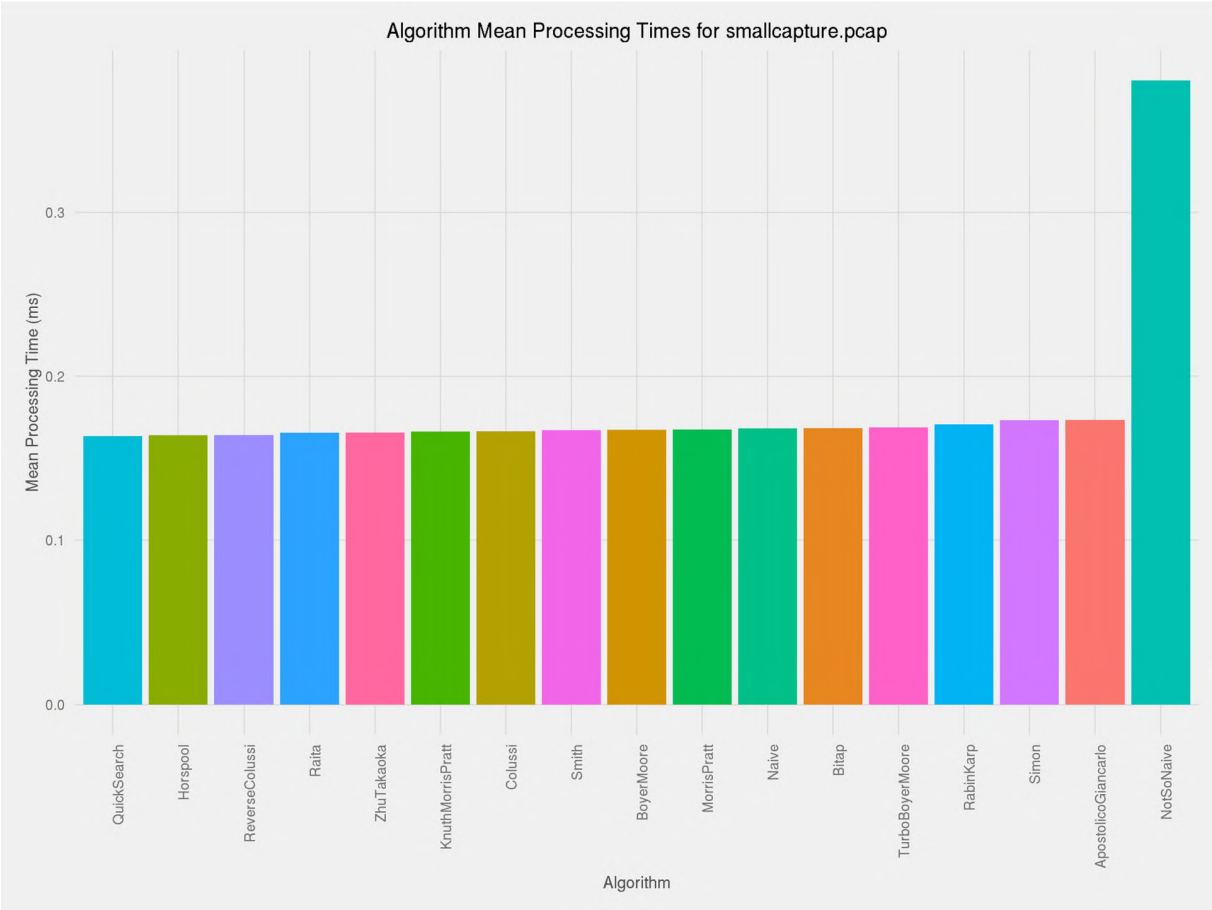

Figure 7.1: Algorithm mean input processing time for *Dataset A*, ranked by processing time.

comprised solely of short DNS packets (as described in Section 4.1), the relative differences in processing time that each algorithm's design would give are minimised by the incredibly small amount of data that each must process at a time. These algorithms were designed to process inputs of the order seen in *Dataset B* where each input is hundreds of thousands of bytes long. With inputs on average 110 bytes in length, *Dataset A* may conceal the true behaviour of these algorithms.

In most Deep Packet Inspection contexts - such as firewalls and Intrusion Detection Systems (Sections 2.2 & 2.3, respectively) - the average length of packets encountered is typically much longer than that given by *Dataset A*. As discussed before, the maximum size of an ethernet frame's payload is 1500 bytes[6] (Law et al., 2012) and further to this, security systems will often collect and search full connections of packets at a time (Handley et al., 2001) - thus multiplying the average search length.

[6]Jumbograms excluded (Borman et al., 1999)

7.3.2 *Dataset B*

The next test was devised in order to better compare these algorithms. In this test the algorithms are used to search through *Dataset B* (the full text of *Alice in Wonderland*). *Dataset B* provides a good representation of the typical input for one of these string search algorithms. The length of *Dataset B* is 163 780 bytes, four orders of magnitude larger than the average length of *Dataset A*. It is expected that the true speed of each algorithm should show better here than in the previous test. Figure 7.2 shows the results of that test. Again, as with Figure 7.1, the fastest algorithm is positioned on the left and the algorithms of increasing processing time follow to the right.

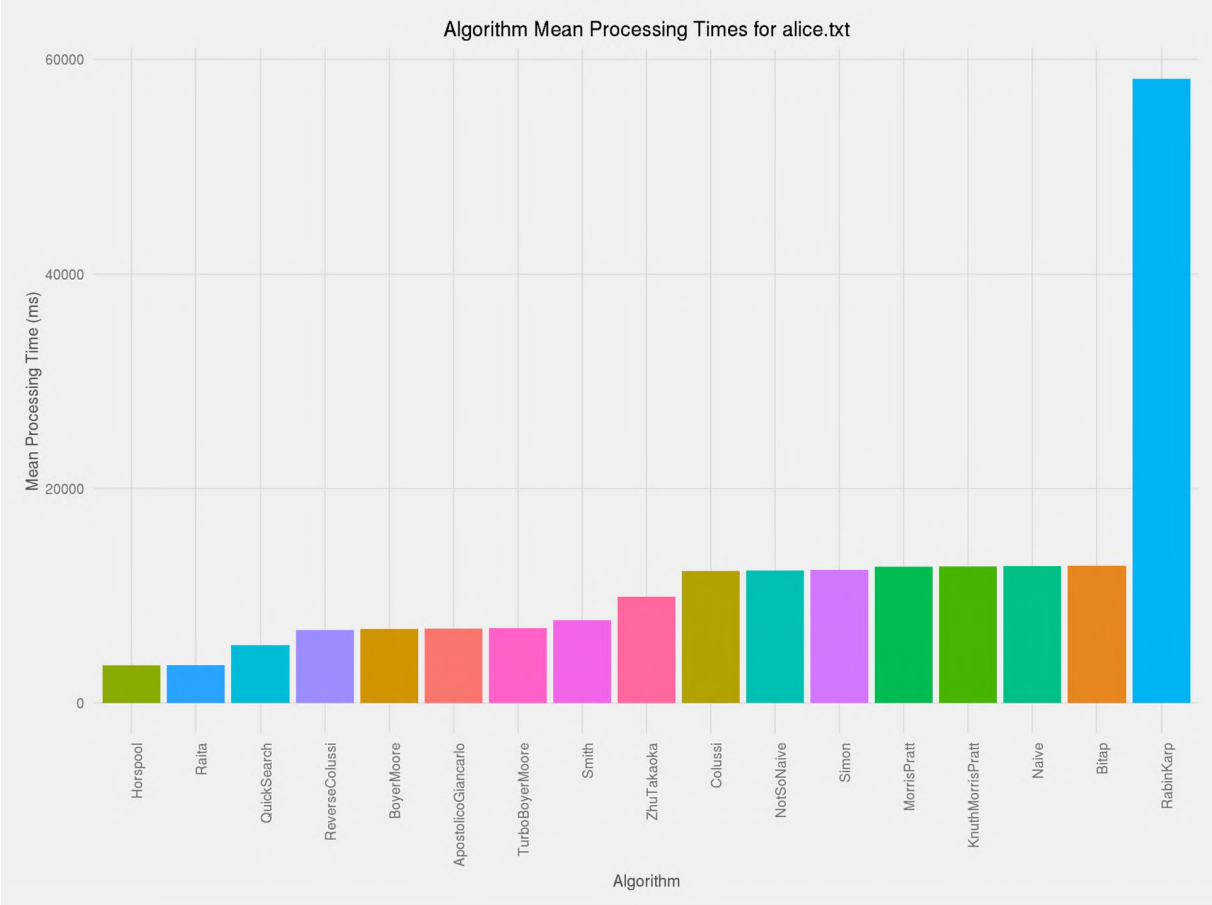

Figure 7.2: Algorithm mean input processing time for *Dataset B*.

Figure 7.2 clearly shows far more variation in the processing speed of the string search algorithms; the desired result when compared with the results presented in Figure 7.1. Again in these results a few salient points are immediately apparent. The Rabin-Karp algorithm (Section 3.7) stands out as the overall slowest algorithm. It is as much as six times as slow as the next slowest algorithm - the Bitap algorithm (Section 3.15) in this

case. The Horspool algorithm appears to be the fastest; a result similar to that in Figure 7.1 where the Horspool gave the second fastest processing speed.

Like the previous set of results, the results of this test on *Dataset B* has produced a few algorithms which appear to perform better than others. In Figure 7.1, the Quick Search and Horspool algorithms were the fastest overall and in the results on *Dataset B* those two algorithms make up two of the top three fastest algorithms. The Horspool and Quick Search algorithms would appear to be strong contenders in the context of Deep Packet Inspection.

In Aho (1990); Stephen (1994); Lecroq (1995); Crochemore and Lecroq (1996), the authors note that the Horspool algorithm - with its simplification of the Boyer-Moore algorithm using only the bad-character shift - shows discernibly favourable results for searches where the alphabet is very large when compared to the length of the pattern. These are exactly the kinds of searches being performed in our tests. In our case the alphabet is 256 characters whilst the longest rule ("government") is a mere ten characters long. This could be further improved by selecting a character encoding that allows for even larger alphabets such as UTF-8.

The length of each rule is bounded by the maximum length of a packet. That is to say that a rule necessarily must be shorter than the input for a match to be expected. With a large enough alphabet it is possible to 'force' the behaviour seen in these tests for even very long patterns.

Towards the right-hand side of the graph - the algorithms with slower processing speeds - the relative positions of the algorithms are not as consistent. In Figure 7.1, the Not So Naïve algorithm (Section 3.17) was clearly the slowest of them all; whereas in Figure 7.2 the Rabin-Karp algorithm is clearly the slowest.

This variable nature of the algorithms' relative positions may indicate some kind of behaviour not shown by their algorithmic complexity in Table 3.1.

What is particularly interesting about the Not So Naïve and Rabin-Karp algorithms is that, in their respective tests, they were by far the slowest overall and even to some degree eclipsed the results of the others. Further investigation of these algorithms should show why their performance is so poor.

7.4 Which algorithms vary the most?

An important feature of real-time systems such as firewalls and Intrusion Detection Systems is having a known upper bound on a given percentage of processing times (Zwicky et al., 2000). Having an upper bound on the times ensures that a system can be guaranteed to perform exactly as expected. If a packet inspection system takes too long to process a packet or the time to process a packet is not well known, this could have serious implications on the availability and quality of the service being offered. In Section 2.1, one of threats to network security that was discussed was denial of service.

Some systems quantify processing time boundaries using percentile metrics. A manufacturer might quote the latency of a DPI system in terms of a P50, P90, P99 etc. time. These latencies indicate the maximum time in which 50, 90 and 99 percent of operations complete respectively.

Denial of service has two distinct threat models. The first is known as distributed denial of service (DDoS) and is used to make a service unavailable by sending so many requests to it that it cannot keep up (Hoffman, 2013). Usually such attacks are performed by massive botnets[7] wherein computers controlled centrally are instructed to send many requests to the target system or service. Such attacks are common and are sometimes implemented as a means of censorship. Recently, the Chinese government has been linked (Anthony, 2015) to DDOS attacks on the code hosting platform GitHub[8]. It is believed that such attacks were designed to limit the availability of tools used to circumvent China's national firewall (Anthony, 2015).

The other mode of a DoS attack is more sophisticated than the first. It uses intimate knowledge of a system to send requests which are designed to use many of the system's resources. Minimal amounts of work done by the attacker can create substantial amounts of work for the system (Needham, 1993; Dougligeris and Mitrokotsa, 2004; Abliz, 2011). When a system accepts more work than it can handle this is sometimes called a *brown out*. There are many ways that a system could have such an attack performed on it and one of those is through packet inspection. A malicious entity could send packets which cause the DPI system to spend an inordinate amount of time checking that packet. If enough specifically crafted packets are sent to such a system, it can negatively impact the system's ability to provide service to legitimate traffic.

[7] https://en.wikipedia.org/wiki/Botnet
[8] https://github.com/

One way of mitigating these threats is to ensure that you can put a bound on packet processing times. A good indication of this is the variation of the processing time for each packet. This can be specifically quantified by the standard deviation of mean processing time for packets of equal length. Figure 7.3 gives a comparison of the standard deviation for the mean of each algorithm's processing time for *Dataset A*. The mean standard deviation is given in milliseconds.

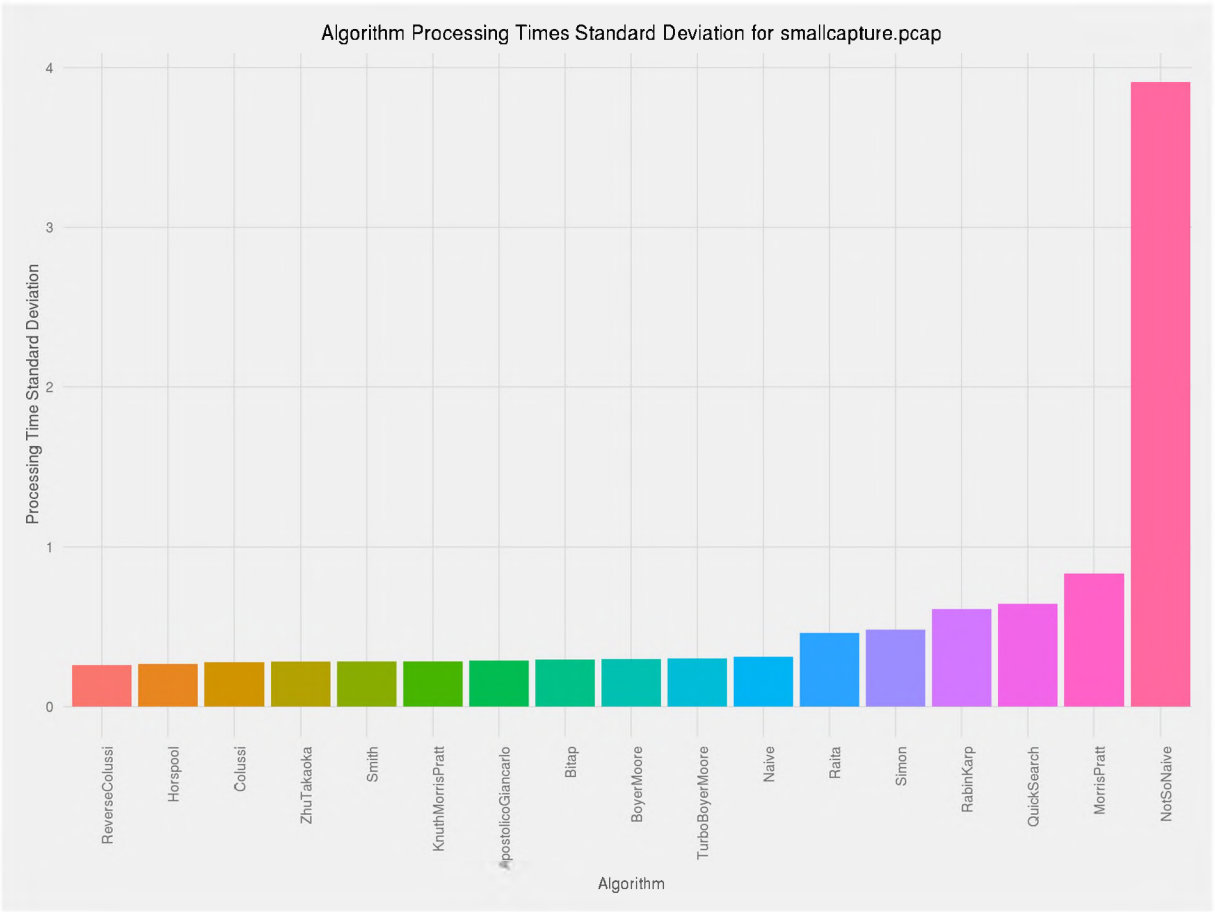

Figure 7.3: Mean packet processing time standard deviation for *Dataset A*

Figure 7.3 provides some interesting results. Obviously, the Not So Naïve algorithm shows massive variance in its processing speed. This kind of behaviour is unwanted because it would indicate some level on nondeterminism. Furthermore, the Quick Search algorithm appears on the higher end of the spectrum of mean standard deviations. For an algorithm with such fast performance as seen in Figures 7.1 & 7.2, this kind of result could mean that the Quick Search algorithm is not a suitable candidate for a good packet inspection algorithm.

The Horspool algorithm, on the other hand, shows very little variance in its processing times; well under 0.5 ms in this test. This indicates that the Horspool algorithm could

prove to be a very strong algorithm for Deep Packet Inspection. The standard deviation gives us an indication of how variant the processing speeds of each algorithm are. Some algorithms may have a few outliers which do not affect the standard deviation as much as highly variant data would. These outliers might indicate edge cases where the algorithm could encounter potential slow downs. That kind of behaviour in most undesirable.

One of the algorithms tested, the Smith algorithm (Section 3.10), was noted as being a combination of the Horspool and Quick Search algorithms. The Smith algorithm takes maximum value produced by the 'bad-character' shift functions of both the Horspool and Quick Search algorithms (which, in turn, have modified the 'bad-character' shift function by the Boyer-Moore algorithm (Section 3.5) (Boyer and Moore, 1977)). One would expect that an algorithm that pitted the shift functions of our two fastest algorithms against one another would be extremely quick. On the contrary the Smith algorithm shows mediocre results. In the tests with *Dataset A* and *Dataset B*, the Smith algorithm posted mean processing times which put it near the centre of the algorithms tested.

In order to better understand the behaviour of these algorithms, it is important that they are looked at individually. To better examine fully each algorithm, only the most interesting can be selected for further examination. With that, the following algorithms have been chosen:

- Horspool algorithm (Section 3.6) - Was the fastest algorithm to process *Dataset B* and the second fastest algorithm to process *Dataset A*. The Horspool algorithm also showed minimal variance in its processing time in Figure 7.3. These factors make it our strongest contender for an excellent Deep Packet Inspection algorithm and warrant its further examination.

- Quick Search algorithm (Section 3.9) - Was the only algorithm faster than the Horspool algorithm in the test using *Dataset A*. It performed poorly in the test of variation shown in Figure 7.3 but may prove to be a good algorithm to study because of that behaviour.

- Not So Naïve algorithm (Section 3.17) - This algorithm performed very poorly in the test of *Dataset A* but showed much better times when processing *Dataset B*. This algorithm is important to study because of its inconsistent behaviour; knowing why the Not So Naïve algorithm performs the way it does may give us better insight into what makes an efficient algorithm for Deep Packet Inspection.

- Rabin-Karp algorithm (Section 3.7) - Lastly, this algorithm was chosen because of its massively inefficient performance when processing *Dataset B*. A pertinent question for this algorithm is why its true behaviour wasn't apparent in the test with *Dataset A*.

These four algorithms give a good indication of the variety of behaviours exhibited by string search algorithms when performing Deep Packet Inspection. The next section looks at how each of these algorithms' processing times vary as a function of the length of the input.

7.5 Length Impact on Performance

As discussed in Section 7.3, algorithms that are too easily slowed by certain types of packets would not be viable in a real-world packet inspection scenario. In this section, the algorithms are examined based on their performance for inputs of varying length.

The speed of each of the implemented algorithms can be described by its algorithmic complexity. That complexity is usually classified using 'Big-θ' notation. The algorithmic complexity for all of the algorithms tested is well known and generally described in their originating papers. See Table 3.1 for each of the algorithm's Big-θ classification.

The algorithmic complexity for string search algorithms is usually expressed as some function of the length of the input (n) and the length of the rule (m). Big-θ notation describes the shape function that the algorithms approach as the factors tend to infinity (Bachman, 1894; Landau, 1909). The shape of the function determines how quickly the processing time increases as the other factors increase. Categorising the algorithms by the shape of the function means that any coefficients describing their behaviour are irrelevant as the factors tend to infinity.

Because the Big-θ notation categorises algorithms as they tend to infinity, the behaviour of these algorithms for very short inputs is not as well defined. As an example take the Big-θ notations: $\theta(n^2)$ and $\theta(n)$. For these two examples the algorithm with a complexity of $\theta(n^2)$ will be faster for very small values whereas the algorithm with a complexity $\theta(n)$ will fare better with longer inputs. If the second example was replaced with $\theta(1000n)$ - the 1000 usually being stripped off with Big-θ notation - then the second algorithm becomes far less viable for Deep Packet Inspection as it requires very long values to be more efficient than the first.

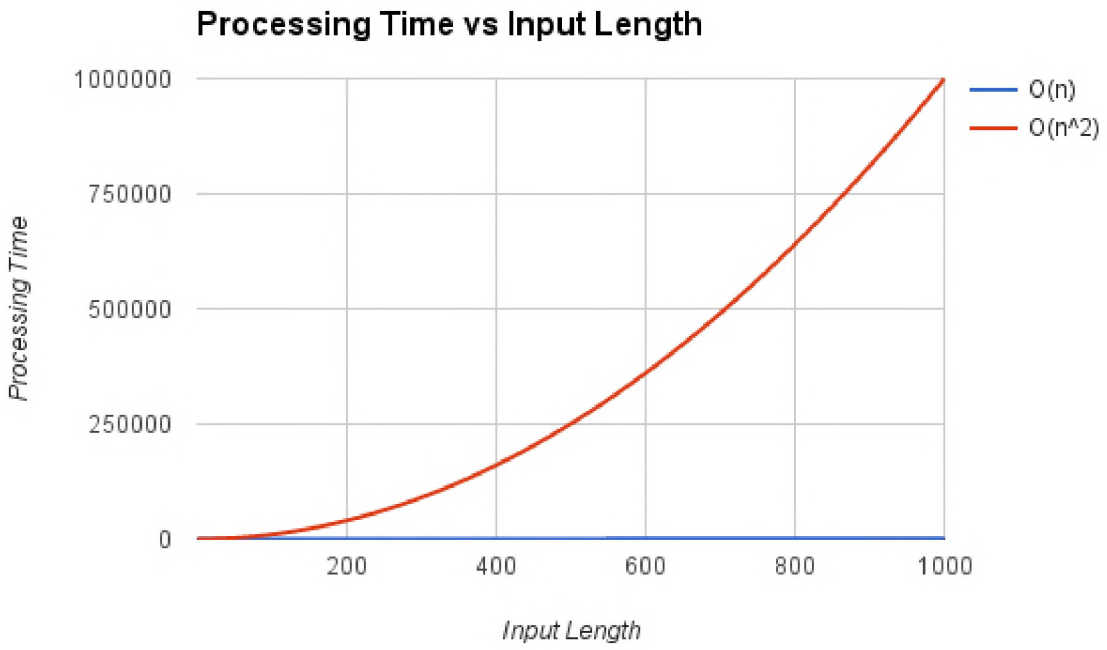

Figure 7.4

Table 7.2: The four chosen algorithms.

Algorithm	Complexity
Horspool	$\theta(n+m)$
Quick Search	$\theta(nm)$
Rabin-Karp	$\theta(nm)$
Not So Naïve	$\theta(nm)$

Figures 7.4 and 7.5 give a comparison of the possible behaviour of algorithms for smaller input lengths. In Figure 7.4, the function $f(n) = n$ is compared with $f(n) = n^2$. It is clear that, for even very small values of n, the $f(n) = n$ function is much more efficient than $f(n) = n^2$. But, as seen in Figure 7.5, if a large enough coefficient is applied to $f(n) = n$ the $f(n) = n^2$ function is more efficient for smaller input lengths.

The algorithmic complexities for the four algorithms chosen algorithms are given in Table 7.2.

From Table 7.2 it is evident why the Horspool algorithm has performed better than the others; it has a complexity related to the sum of the length of the rule and input whereas the other three algorithms all have complexities related to the multiplication of the length of the rule and input. In the larger table of algorithms (labelled as Table 3.1),

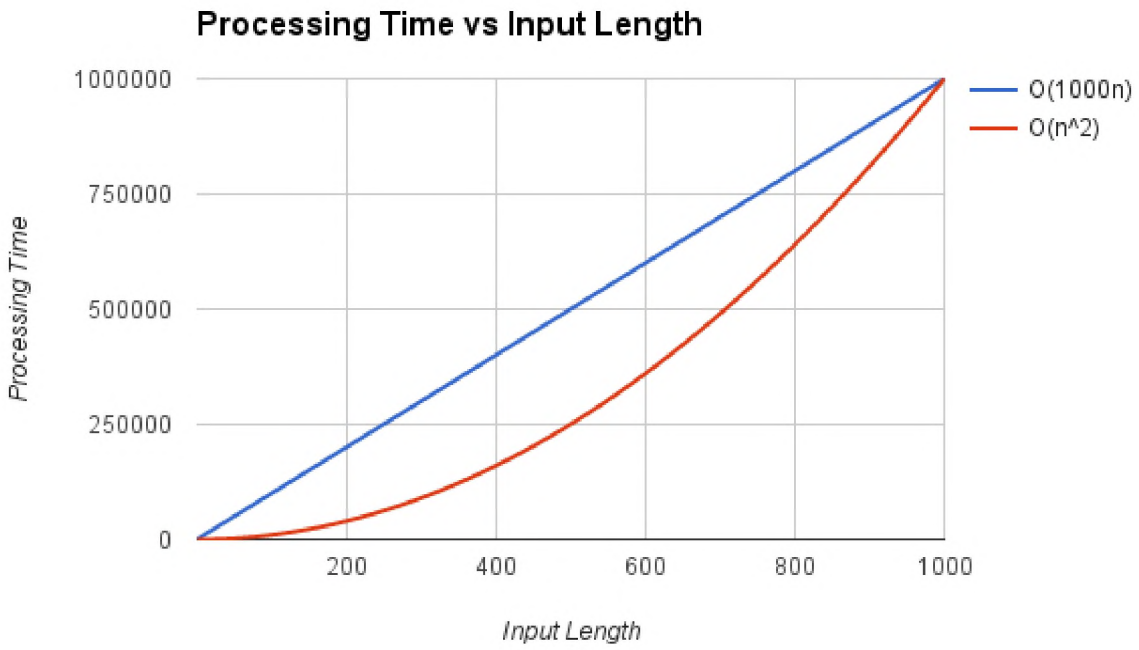

Figure 7.5

some algorithms even feature $\theta(n)$ time complexity but did not prove to be the fastest experimentally. The different speed of each of the last three algorithms seems to be hidden in the missing coefficients of the notation.

It is now time to to see how each of the algorithms fares when inputs of varying length are given. Figure 7.6 shows the relationship between input length and processing time for all of the algorithms combined using *Dataset A*. The plotted line indicates the smoothed conditional mean for every data point throughout the graph and the cone surrounding the line gives the confidence interval thereof.

From Figure 7.6, it is clear that the speed of the algorithms - the inverse of the time to process a packet - decreases as the length of the input increases. This is the expected result as none of the algorithms' processing times have an inverse relationship to the input lengths. The graph has been plotted with a logarithmic scale on the y-axis so as to not give too much emphasis on the large values. From this plot it is also clear that there are a large number of packets with a length of around 80 bytes and that the input times for those packets varies wildly - basically spanning the entire range of processing times.

The processing time versus input length for each algorithm must be looked at in order to better understand the four chosen algorithms. Figures 7.7 to 7.11 show these results.

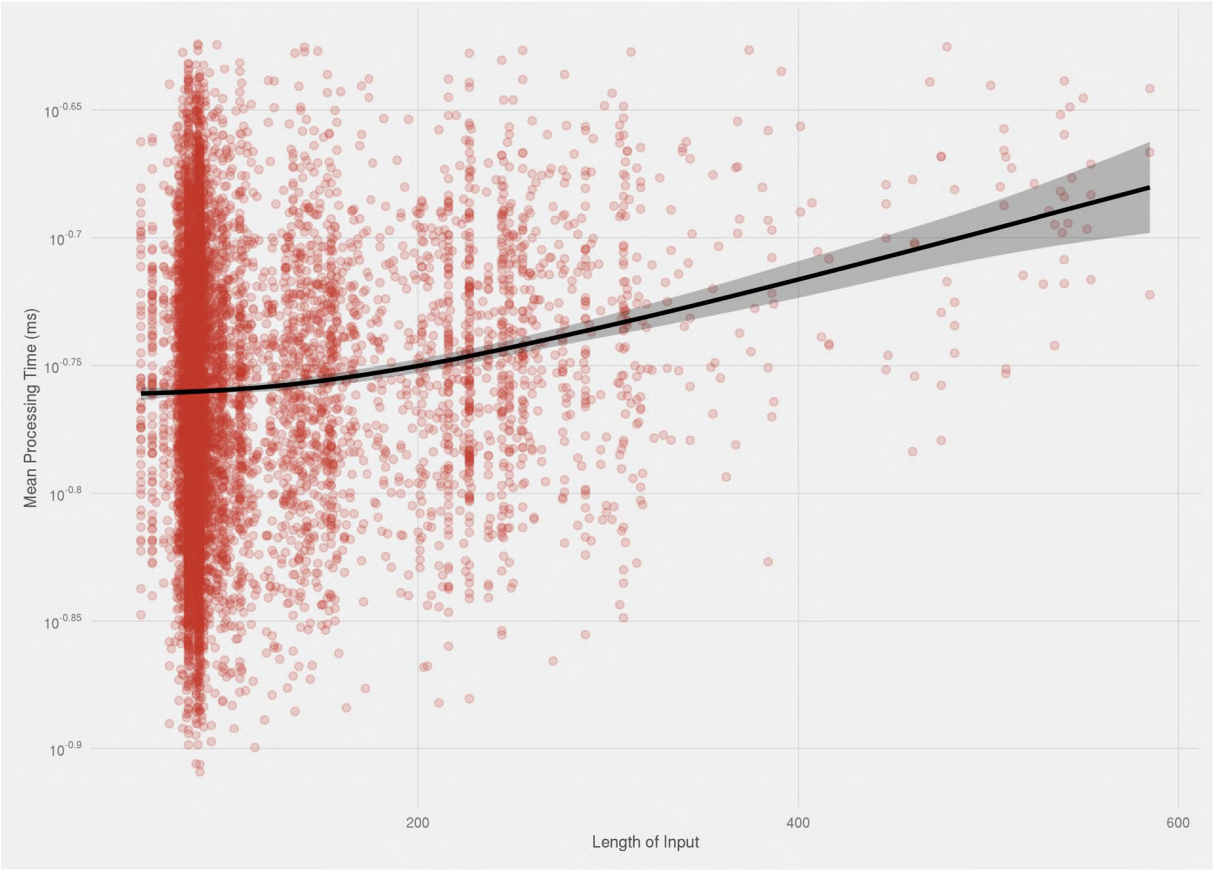

Figure 7.6: Overall mean processing time for combined algorithms versus input length for *Dataset A*.

7.5.1 Horspool

The Horspool algorithm (Section 3.6) is the first of the algorithms to be examined. As with the test shown in Figure 7.6, *Dataset A* was used to conduct this test. The results of the test are presented in Figure 7.7.

Figure 7.7 shows a good representation of the Horspool algorithm's behaviour at various input lengths. As you can see, by looking at the minimum processing time for each discrete input length, the processing time does increase with the input length. Again, a large cluster of data points around the 80 byte range can be seen (marked on the figure with a thick, vetical, black line); this would indicate that the dataset has a large number of packets of that length. Furthermore, the processing times for each packet would appear to remain fairly constant as the packet size increases when compared to the variation seen at the 80 bytes mark. This is evident by the distribution of points. The range of the y-axis is linear, unlike with Figure 7.6, and range from around 0.1 milliseconds to just over 0.6 milliseconds.

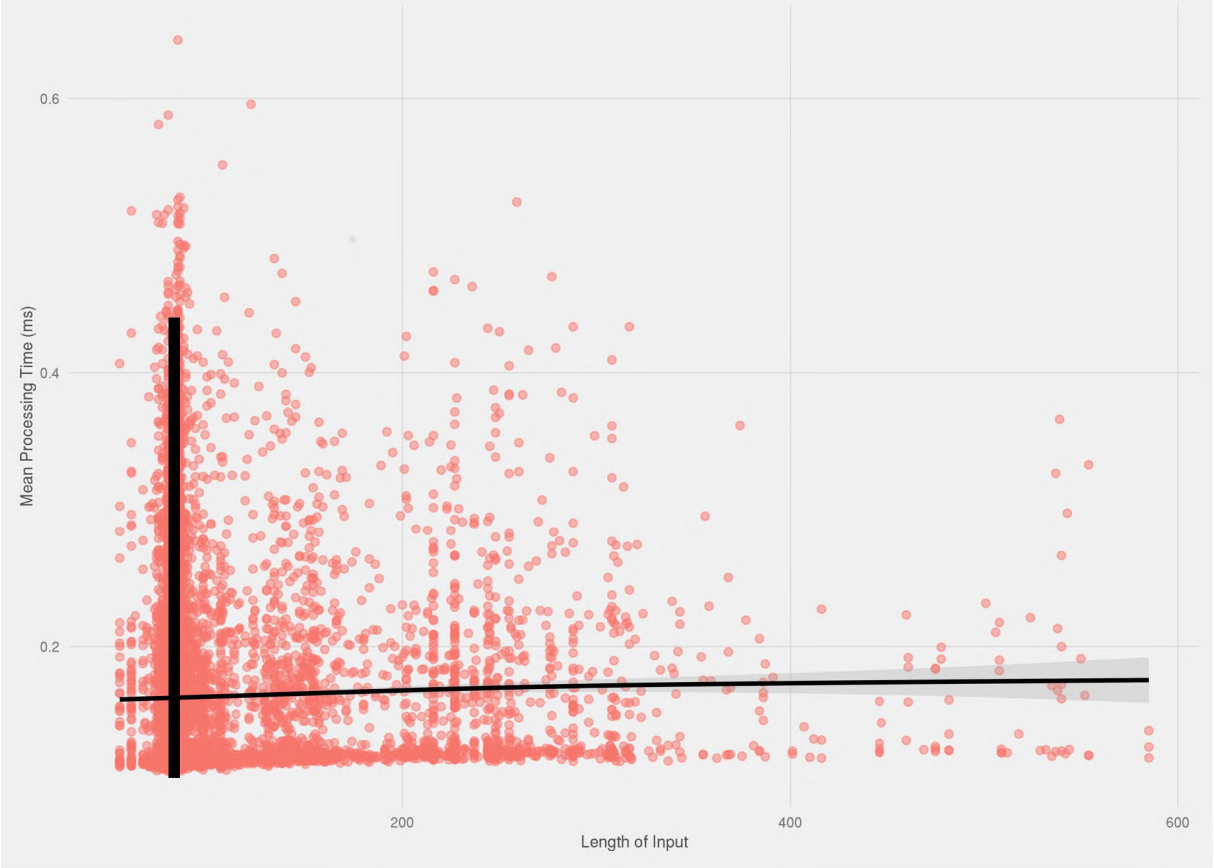

Figure 7.7: Horspool algorithm: Input processing time versus input length for *Dataset A*.

An issue with this dataset set may be that many of the packets of longer length are made up of packets which can be processed faster - an example of this is a packet without any matches and with very few partial matches which can take up precious processing time. An interesting experiment would be to compare the speed of packet processing versus the input length when there are no matches within the payload and so the algorithms can be compared more equally. This experiment is discussed in Section 8.1.

From Figure 7.7, it can be observed that the range of input lengths is between about 60 bytes to just under 600 bytes. As discussed in Section 4.3, the maximum payload size for ethernet frames is 1500 bytes (Law et al., 2012) and so *Dataset A* does not properly cover all of the input lengths that are currently achievable and commonplace in networks today.

7.5.2 Quick Search

The next test looks at the Quick Search algorithm (Section 3.9) and compare its processing speed with the length of the input. The Quick Search algorithm was also tested using *Dataset A* and the results are given in Figure 7.8.

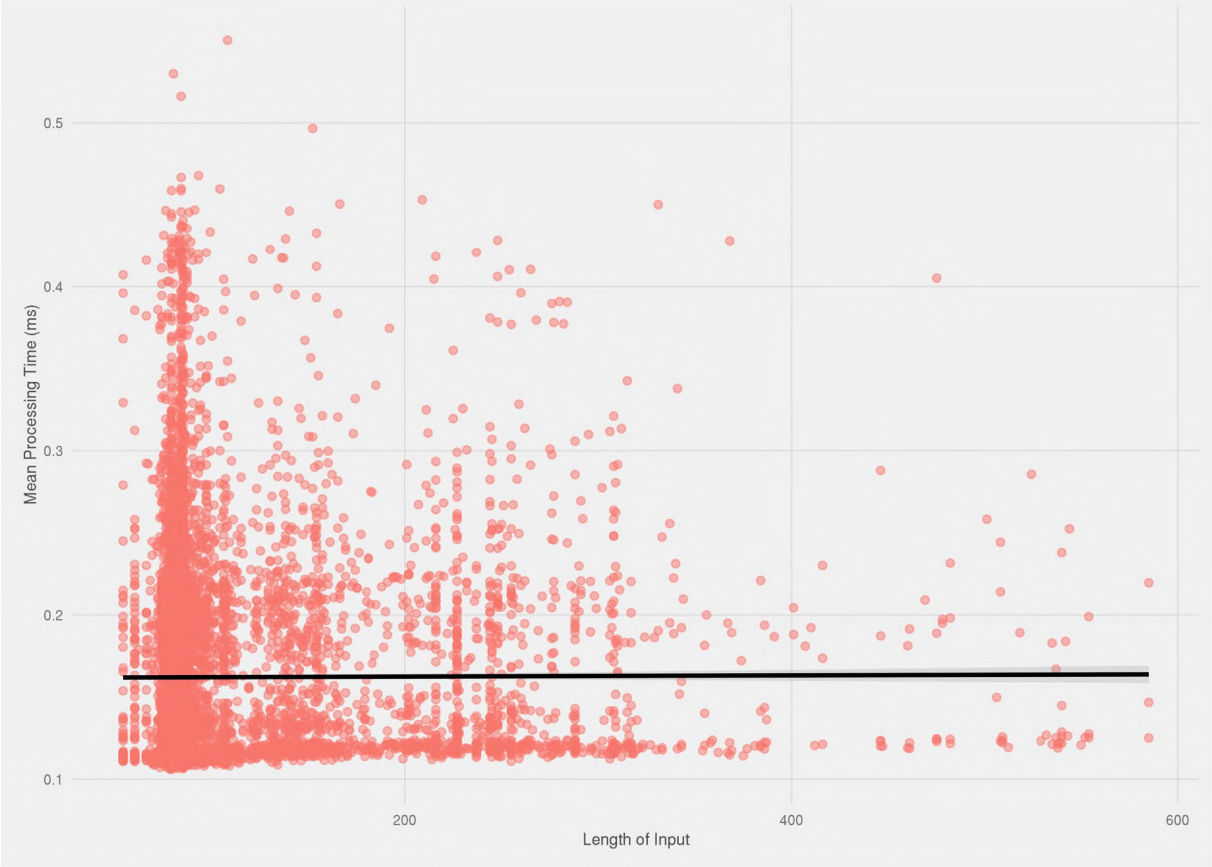

Figure 7.8: Quick Search algorithm: Input processing time versus input length for *Dataset A*

In the test discussed in Section 7.3 (where *Dataset A* was used as the input), the Quick Search algorithm performed better than the Horspool algorithm. This is evident from the results presented in Figure 7.8 when compared with those in Figure 7.7. The scale of the y-axis has been shifted down by 0.1 milliseconds indicating that the respective minimum and maximum values that the Quick Search algorithm produces is less than those produced by the Horspool algorithm. Furthering that point is that the gradient of the curve fitted to the data of the Quick Search algorithm is clearly less than that of the Horspool. This indicates that a smaller increase in processing time for longer packets is achievable with the Quick Search algorithm.

That last point - that the Quick Search algorithm appears to fare better at longer input

lengths than the Horspool algorithm - is not actually the case. This can be seen in Figure 7.2. In those results it is the Horspool algorithm that has shown a dramatically faster mean processing time than the Quick Search algorithm for a far longer input length.

In Figure 7.3, the Horspool algorithm is shown to have a lower deviation on its mean packet processing time than the Quick Search algorithm. This may not be immediately apparent from the data presented in Figures 7.7 and 7.8 but if one examines the results of both algorithms at the right-hand end of the graphs (results for longer input lengths) it is evident that the Quick Search algorithm shows more results of longer processing times than the Horspool algorithm does.

In Figure 7.8 (as seen again in Figure 8.1), there is a large variation of processing speeds for packets of around 80 bytes in length. This variation could be attributed to the varying contents of DNS packets. Each of these algorithms has differing procedures for dealing with partial or full matches. Packets without matches to the rules are treated differently to those with.

Figure 7.9 shows the results of a DNS lookup request for `ru.ac.za` (the researcher's university's domain). The results of the request indicate (see the last row of the response) that the size of the data received is 308 bytes in length. The same request for `news24.com` receives a response of 124 bytes and a request for `mybroadband.co.za` receives a response of 212 bytes. The larger size of the response for `ru.ac.za` can be attributed to the use of a large number of additional name servers and the support for IPv6 addresses in the form of `AAAA` records (Thomson, Huitema, Ksinant, and Souissi, 2003).

From this very small test it is clear that the length of DNS requests can vary quite a large amount but the majority of requests would appear to be small compared to the maximum size achievable on the medium.

7.5.3 Not So Naïve

The next algorithm tested was one of the slowest shown in earlier tests. It is the Not So Naïve algorithm (Section 3.17). The results of the test using *Dataset A* are presented in Figure 7.10.

The Not So Naïve algorithm was developed as an improvement on the traditional Naïve algorithm (Section 3.2). It aimed to keep the original design of the Naïve algorithm with a few added improvements. In the earlier tests - the results of which can be seen in Figure

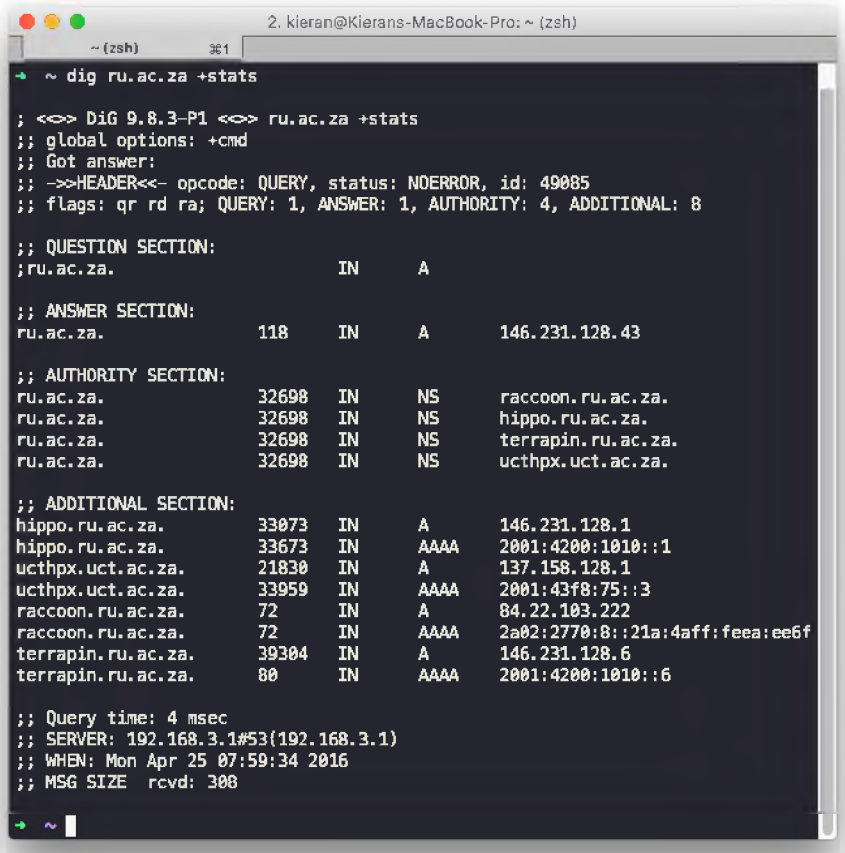

Figure 7.9: The results of a DNS lookup using the `dig` utility. The command used was `dig ru.ac.za +stats`.

7.1 - the Not So Naïve algorithm fared very poorly for *Dataset A*. In Figure 7.10, the range of values in the data from the test with the Not So Naïve algorithm is between about 0.2 milliseconds and 0.7 milliseconds. This presents a larger range of values than what was seen with the Horspool and Quick Search algorithms with the maximum value being far less than the other two.

In the previous two results, the following behaviour has been seen: for packets of shorter length (of which there are many) there is a large variation in the processing time but as the length of the packets increases the variation would appear to decrease. In Figure 7.10, the processing times at each of the input lengths appears to vary much more than what was seen in Figures 7.7 & 7.8. This behaviour is also exhibited in the comparison of standard deviations of each algorithm presented in Figure 7.3 where the Not So Naïve algorithm shows the largest standard deviation of them all.

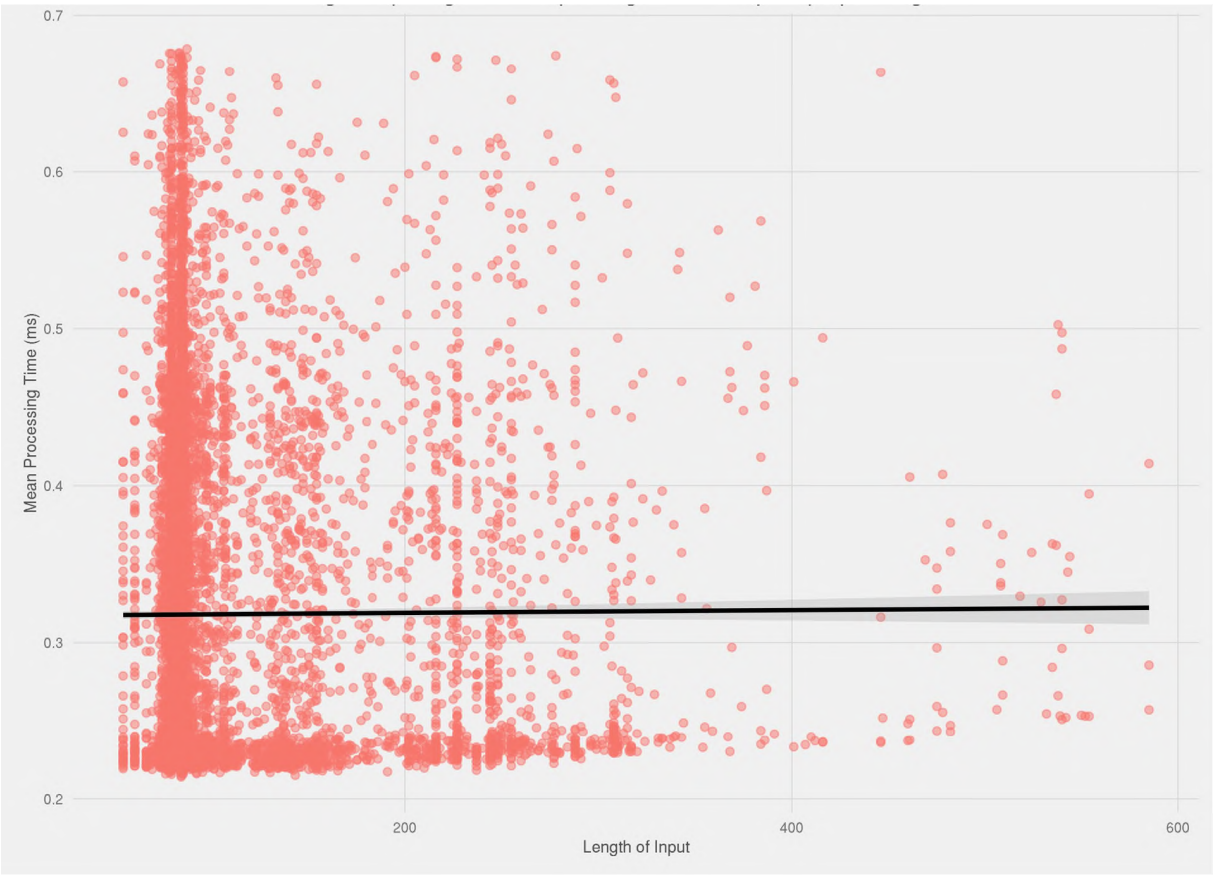

Figure 7.10: Not So Naïve algorithm: Input processing time versus input length for *Dataset A*.

If the behaviour of the Not So Naïve algorithm is similar to that of the Naïve algorithm, then the Not So Naïve algorithm could be expected to vary very little as the number of matches within a packet varies. This behaviour is a result of the Naïve algorithm not employing any special logic in the event of a partial or full match.

In Figure 7.10, the minimum processing time for packets of increasing length appears to increase as well. The line fitted to the data does not show much change as the length increases as the overall packet processing time for the longer length has remained about the same.

7.5.4 Rabin-Karp

The results of the Rabin-Karp algorithm (Section 3.7) processing *Dataset A* versus the input length are presented in Figure 7.11.

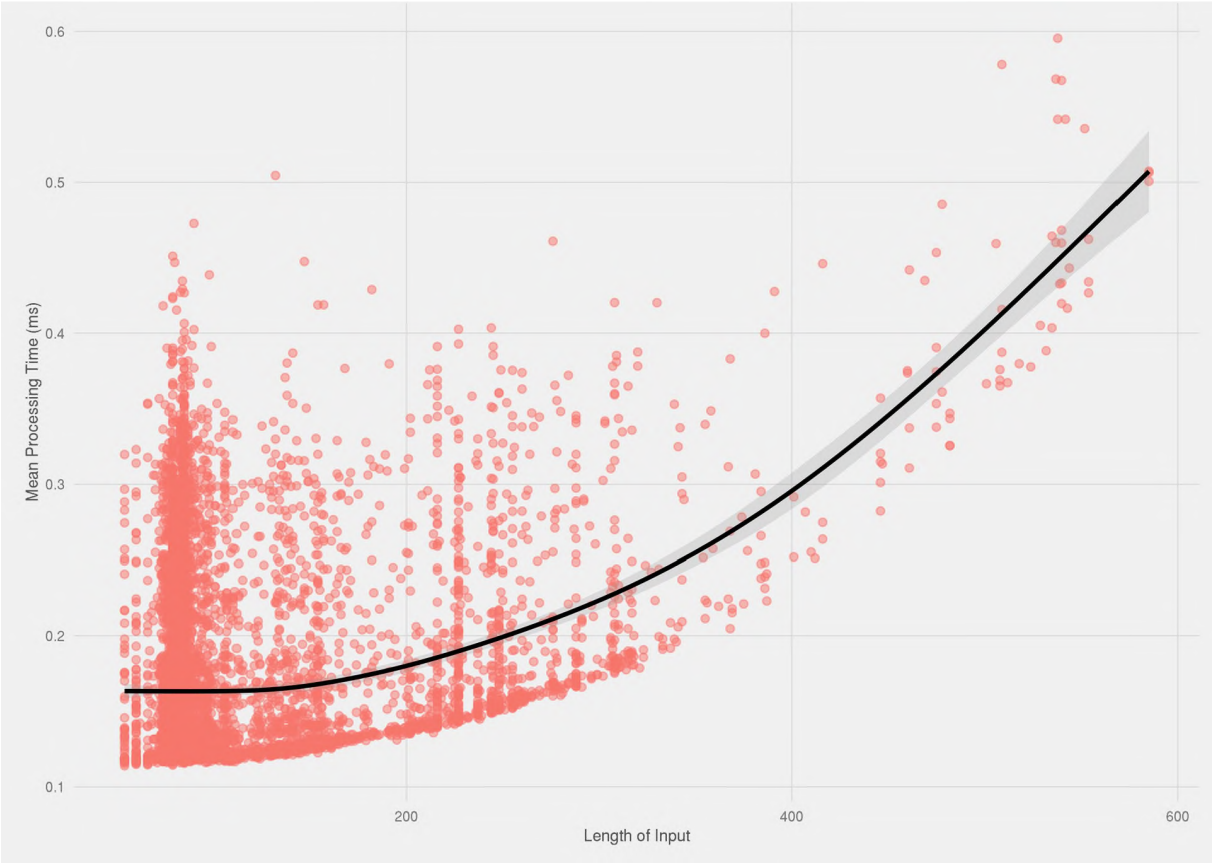

Figure 7.11: Rabin-Karp algorithm: Input processing time versus input length for *Dataset A*.

The Rabin-Karp algorithm, at least for *Dataset A*, is not the slowest. The slowest for *Dataset A* was the Not So Naïve algorithm. The Rabin-Karp algorithm was by far the slowest when processing *Dataset B* (a dataset with very long input). At the start of this section it was discussed that Big-θ notation strips the coefficients off of the functions that it describes. In this case it would appear that the Rabin-Karp algorithm was faster than the Not So Naïve algorithm for smaller input lengths. So it would appear that Not So Naïve has large enough coefficients that Rabin-Karp is faster for packets of DNS length. This is also evident from the maximum value shown in Figure 7.11, which is less than 0.6 milliseconds whereas the maximum value for Not So Naïve is just under 0.7 milliseconds.

The Rabin-Karp algorithm actually works in much the same way as the Naïve algorithm. There is a window into the input, the same size as the rule, which is shifted along, byte by byte, trying to find a match. The contents of the window are hashed and then compared with the known hash of the rule. If a match is found the window needs to be compared, byte by byte, with the rule to make sure that there was no false positive. False positives can occur because the hashing algorithm could transform two unique string into the same

value. It would appear that a combination of the continuous hashing and having to recheck after a potential match is made is enough to cause the behaviour seen in Figure 7.11.

It would seem that the Rabin-Karp algorithm is poised to take over as the slowest algorithm for packets just slightly longer than the ones in this dataset.

7.6 Summary

Table 7.3: Chapter 7 algorithm rankings

Rank	Speed *Dataset A* Subsection 7.3.1	Speed *Dataset B* Subsection 7.3.2	Variation *Dataset A* Section 7.4	Input Length *Dataset C* Sections 7.5
1	Quick Search	Horspool	Horspool	Horspool
2	Horspool	Quick Search	Rabin-Karp	Quick Search
3	Rabin-Karp	Not So Naïve	Quick Search	Not So Naïve
4	Not So Naïve	Rabin-Karp	Not So Naïve	Rabin-Karp

Table 7.3 gives the rankings for each of the selected algorithms tested in Chapter 7.

From the results presented in Table 7.3 and in the sections above it is clear that there is some very interesting behaviour associated with these string search algorithms. For inputs with lengths which are within their usual domain (inputs of book length) these algorithms have very well known behaviours. These behaviours can break down when the input lengths decrease to the size of packets. This nondeterminism can be detrimental to system performance in packet inspection contexts.

What is also evident from these results is that further examination is necessary. The data produced by these algorithms using *Dataset A* and *Dataset B* is indicative of real-world behaviour but does not provide the flexibility to fully examine the behavioural nuances exhibited by these algorithms.

This chapter has established a good comparison of each algorithm, their speed to process in the packet inspection domain, and compared that with their speed to process in their intended domain. This chapter has also taken a first cursory look at the algorithms' performance when compared to the length of the input - a property of the algorithms which could show good candidacy for an excellent packet inspection algorithm.

The following chapter further examines the four algorithms chosen and answer more questions which have arisen.

Chapter 8

Further Algorithm Comparison

The previous chapter started to look at the string search algorithms and how they perform in datasets mimicking network traffic in the real world. From those results, four algorithms were chosen, based on various behaviours, to further examine in order to establish a good body of knowledge representing their performance in the context of packet inspection.

The four algorithms, chosen for their unique behaviours, were:

- The Horspool algorithm (Section 3.6) - This algorithm was chosen as it showed the best processing speed when given *Dataset B* as an input and gave the second fastest mean time to process in the comparison for *Dataset B*.
- The Quick Search algorithm (Section 3.9) - The Quick Search algorithm was selected because it was the fastest to process *Dataset A* and was third fastest for *Dataset B*.
- The Not So Naïve algorithm (Section 3.17) - This algorithm was selected as it showed very poor processing times for *Dataset A* but recovered well when processing *Dataset B* to place within the middle of the algorithms performance-wise.
- The Rabin-Karp algorithm (Section 3.7) - The Rabin-Karp algorithm was by far the slowest algorithm when processing the textual dataset (*Dataset B*) but was not nearly as slow with *Dataset A*.

It is believed that these selected algorithms will show a good indication of how well string search algorithms perform Deep Packet Inspection and may give light to some of the edge cases which may also make them unsuitable. Generally, these algorithms are interesting as

the first two (Horspool and Quick Search) represent the faster end of the testing spectrum and the latter two (Not So Naïve and Rabin-Karp) represent the the slower end. These four edge cases were chosen in the hope that they may reveal what affects parts the their design affects a string search algorithm when it comes to packet inspection (be it positively or negatively).

This chapter will further scrutinise these algorithms and examine their behaviour under all manner of conditions. It will ask, again, how the speed of the algorithms is affected by input length (Section 8.1), how the number of matches affects the processing speed (Section 8.2, and how multithreading can change the behaviour of the algorithms (Section 8.3).

8.1 Performance versus input length with no matches

In the previous chapter, how the length of the input affects the speed at which these algorithms perform their packet inspection was discussed. When performing those tests, there are a number of issues with the real-world dataset (*Dataset A*) which came to light.

The average length of the inputs in *Dataset A* is 109.61 bytes (Table 4.1). This is far shorter than the maximum length of a ethernet frame, which is 1500 bytes (Law et al., 2012), and is still even less than half of that value.

The actual data in *Dataset A* is real-world data and the rules used in these tests are designed to match well with that data. As a result of this is that there are many matches to the rules within this dataset which may affect the comparison of the speed of the algorithm with the length of the input. A short input which produces many matches to the rules may affect the algorithm's time to process more than a long input with no matches.

In order to mitigate this a new dataset, *Dataset C* (Section 4.3) - which provides packets with lengths of up to 1500 bytes and contains no matches, was created. This dataset was created using Wireshark's `randpkt` utility (Ramirez, 1999) using the command in Listing 4.1. *Dataset C* has an average packet length of 770 bytes (See Table 4.1) which is far more than the 109.61 bytes of *Dataset A* and as such should force each of the algorithms to spend more time processing each packet than before.

8.1.1 Horspool

The following tests were designed to reexamine at how the length of the input affects the performance of the algorithm when there are no matches. Figure 8.1 shows a comparison of the Horspool algorithm versus the length of the input for *Dataset C*.

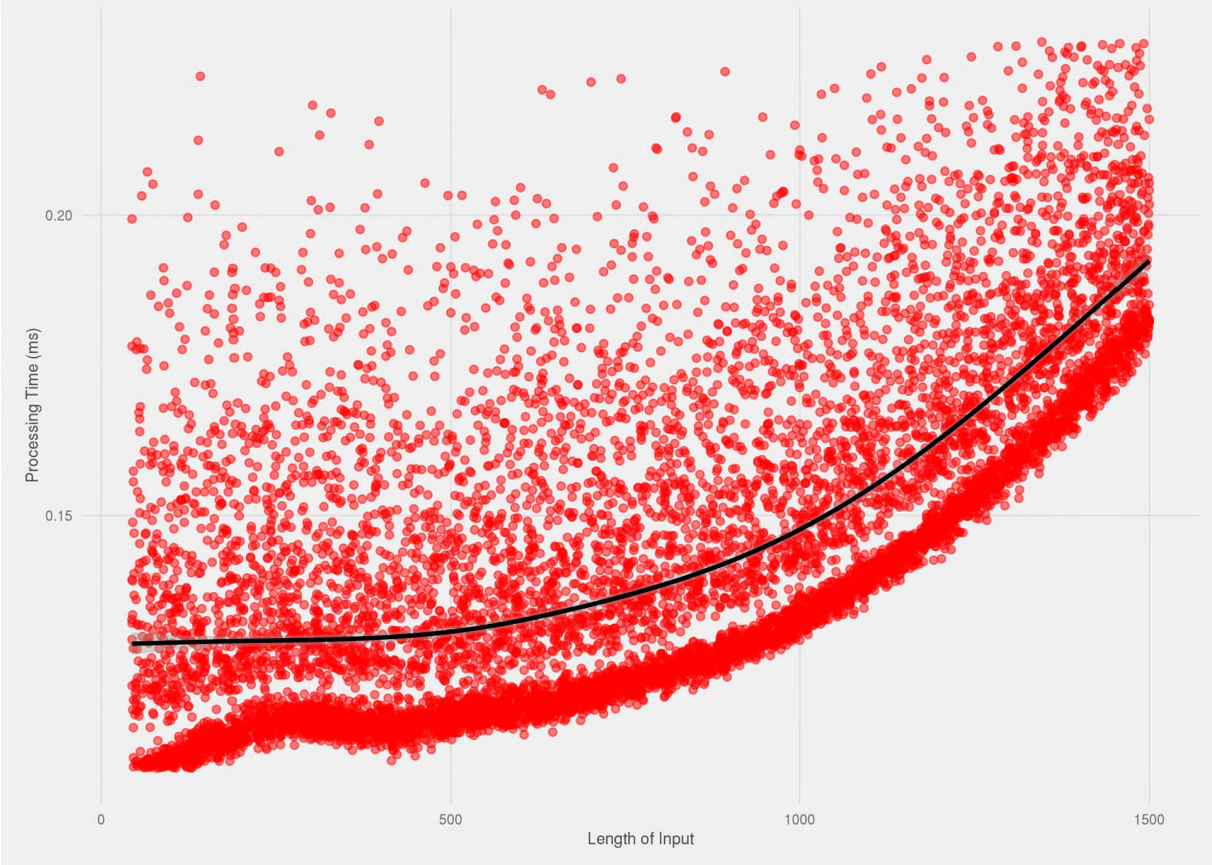

Figure 8.1: Horspool algorithm: Input processing time versus input length for *Dataset C*.

From Figure 8.1 quite a bit of interesting information is apparent. The graph itself shows a processing time of between 0.10 milliseconds to just under 0.25 milliseconds. The range of input lengths vary from about 50 bytes to 1500 bytes.

When comparing Figure 8.1 to Figure 7.7 one may conclude that these figures do not align well, but if you were to truncate the data in Figure 8.1 so that the maximum packet length is no more than 600 bytes, 8.1 would look very much like Figure 7.7. This shape of this algorithm can be attributed to the fact that it does not have to do much work for each input - as there are no matches it can fail early - and so it is just iterating through longer and longer arrays of characters. For each character there is some work that must be done and so this can account for the pattern seen here.

The hump in the data seen on the graph at about 250 bytes can be attributed to the initial overhead of the system and the behaviour of the algorithm thereafter (from about 500 bytes to 1500 bytes) is a true indication of the Horspool algorithm's actual performance.

8.1.2 Quick Search

Figure 8.2 shows the performance of the Quick Search algorithm when inspecting the data in *Dataset C*.

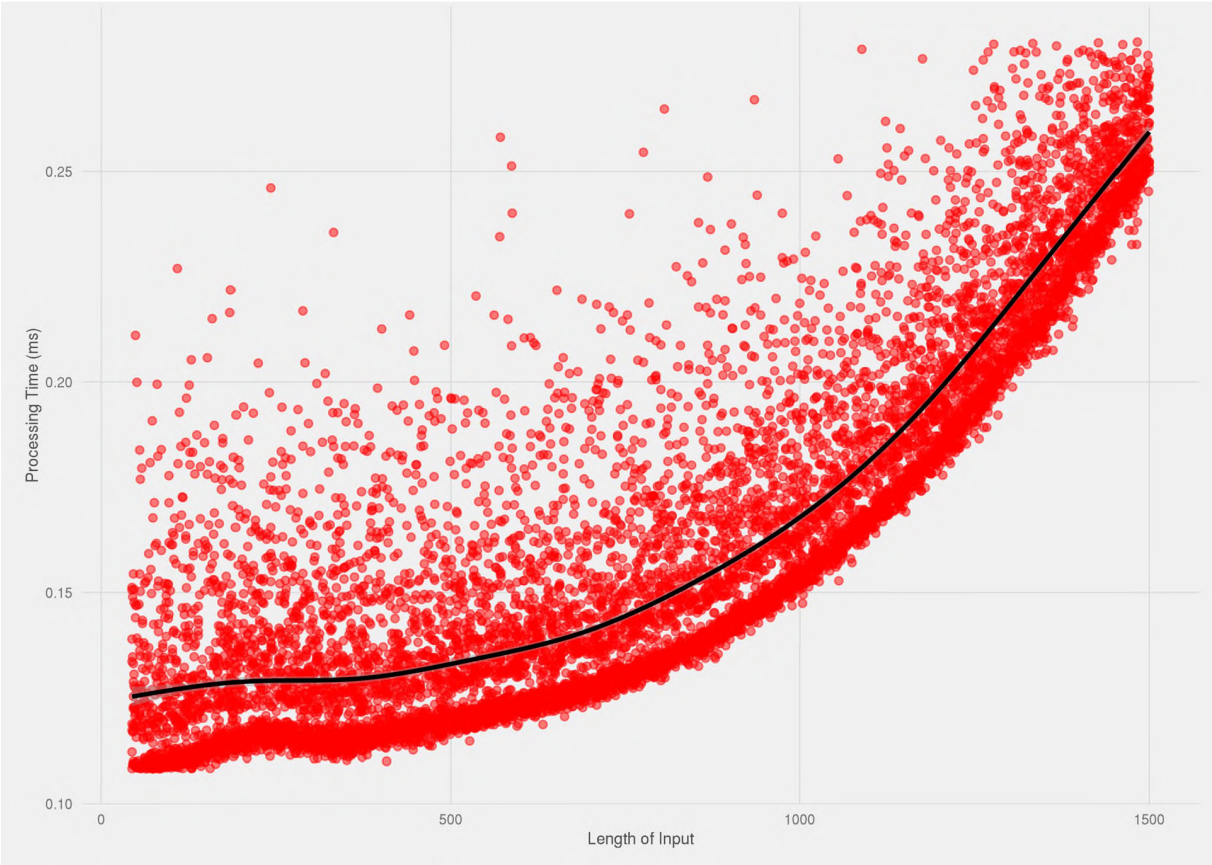

Figure 8.2: Quick Search algorithm: Input processing time versus input length for *Dataset C*

The Quick Search algorithm has a created results similar in look to the results produced by the Horspool algorithm. Unlike the Horspool algorithm, the Quick Search algorithm produced processing times ranging from about 0.12 milliseconds to almost 0.30 milliseconds. These processing times are higher than those produced by the Horspool algorithm.

In Section 7.3, it was concluded that, for shorter input lengths, the Quick Search algorithm was faster than the Horspool algorithm but, as the input length increased, the Horspool

algorithm would achieve better processing speeds. For packets of a length expected on a ethernet network, the Horspool algorithm has performed better than the Quick Search algorithm.

In Figure 8.2, there is a clear clustering of values near the minimum processing time for each input length as well as around the fitted curve. These clusterings merge as the input length increases indicating that the property affecting the packet processing that makes the gap seen in the clustering does not affect it as much when the length of the input and the time to processes increase. The disappearance of the clusters can be attributed to some small amount of work needed to be done to perform the search which is minimised over longer input lengths.

The variation of processing speeds for packets of similar length seen in Figure 7.8 is not as apparent in the latest set of results. This reduction in variation could be attributed to the removal of the variation introduced by matches within the inputs. An input with many matches may cause an algorithm to spend more time processing than when there are fewer matches. This reduction in processing time could be attributed to the extra load caused by reporting a match.

8.1.3 Not So Naïve

Figure 8.3 presents the packet processing speed versus the length of the input where there are not matches for the Not So Naïve algorithm. *Dataset C* was used for this test.

The Not So Naïve algorithm is the first of the two poorer performing algorithms to be examined in this chapter. In Figure 8.3 a plot of the processing time versus the length of the input is presented.

The maximum processing time of the Not So Naïve algorithm is clearly more than that shown for the Horspool algorithm in Figure 8.3 and the Quick Search algorithm in Figure 8.2. The minimum processing time For the Not So Naïve algorithm appears to be around 0.1 milliseconds whilst the maximum processing time is just under 0.7 milliseconds.

In this plot it would appear that the results are clustered around the fitted curve with few discernible outliers. Comparing this to what is seen in the results for the Quick Search algorithm (Figure 8.2) where there are two distinct clusters and quite a large number of outliers it can be concluded that the slower processing speeds, exhibited in the

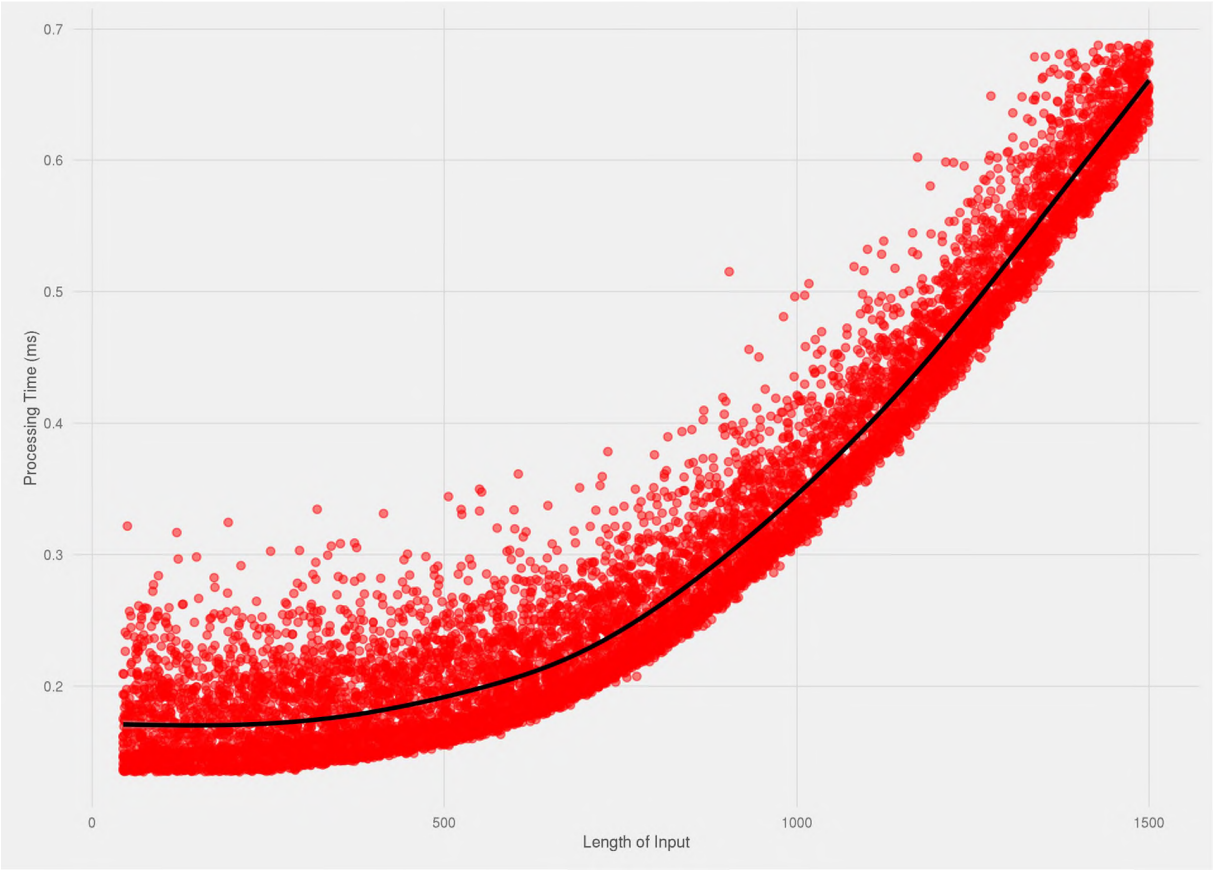

Figure 8.3: Not So Naïve algorithm: Input processing time versus input length for *Dataset C*.

Not So Naïve algorithm and towards the larger input of the Horspool and Quick Search algorithms, produce more tightly grouped results.

Figure 8.3 also shows an extreme rise in processing times following the 400 bytes mark. This corresponds to a point near the tail end of *Dataset A* which was not shown well in the results of the tests using that dataset. This extreme rise is probably attributable to its O(nm) complexity (see Section 3.1).

8.1.4 Rabin-Karp

Figure 8.4 presents the results of the same test for the Rabin-Karp algorithm, using *Dataset C*.

Even though the Rabin-Karp algorithm was not the slowest when processing *Dataset A*, it was by far the slowest when processing *Dataset B*. This is indicative of an algorithm

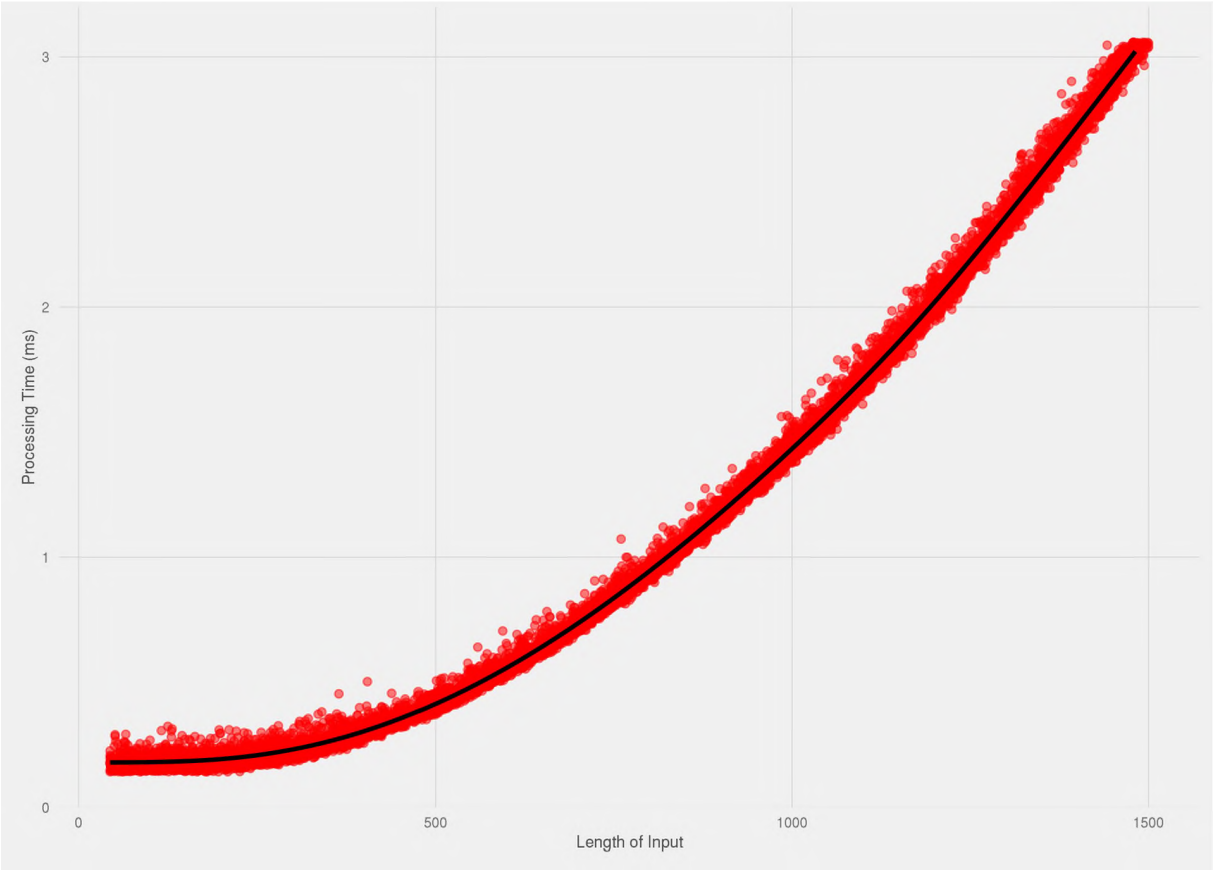

Figure 8.4: Rabin-Karp algorithm: Input processing time versus input length for *Dataset C*.

which is slower for longer inputs. This test aimed to see at which point this algorithm becomes slower than the Not So Naïve algorithm. If, during these tests, it was found that the Rabin-Karp algorithm performed better than the Not So Naïve algorithm then it could be established that the Rabin-Karp algorithm is better within the context of packet inspection than the Not So Naïve algorithm. This, however, was not the case as can be seen in Figure 8.4.

The processing times for the Rabin-Karp algorithm in *Dataset C* range between about 0.2 milliseconds to just over 3 milliseconds. The Rabin-Karp algorithm is by far the slowest of all four of the algorithms tested here.

There are a few points of interest on the graph. The first interesting point, again, is that for smaller input lengths (around 250 bytes), the Rabin-Karp algorithm performs about the same as the three previously examined algorithms. The second interesting point is how little variation is seen on the processing time through the range of input lengths.

The lack of variation seen in the Rabin-Karp algorithm's results, especially when compared

to the Horspool and Quick Search algorithms, can be attributed to its implementation. The algorithm does not care at all about partial matches - they do not affect its running - as partial matches will produce a hash completely different to the hash of the rule. As this dataset (*Dataset C*) was designed to produce no matches, this algorithm will always perform the exact same operations as it inspects the input. If the only change is the length of the input, it is then expected that there is very little variation on the processing times for the Rabin-Karp algorithm.

From the preceding tests the relationship of the length of the inputs to the algorithms and the time it takes for each algorithm to process its input has been succinctly determined. This kind of information is important when it comes to Deep Packet Inspection as Deep Packet Inspection usually has very strict requirements on the maximum time to process the packet's payload.

Through the use of the carefully constructed *Dataset C* (See section 4.3 for more information on that dataset) the influence of external variables on the performance of the algorithms has been eliminated by removing those variables. *Dataset C* does not perfectly represent traffic expected in real-world scenarios but does well represent a certain aspects thereof - namely the length of the packets.

8.2 Performance versus number of matches

In Section 8.1, the dataset itself was used to examine the behaviour of a certain aspect of the chosen algorithms. This section aims to see how the number of matches affects the processing time of the algorithms. In order to separate the number of matches from the length the input, a new dataset needed to be constructed.

This new dataset, *Dataset F* (Section 4.6) was constructed in such a way that the length of the input did not affect the number of matches. To construct this dataset, a PCAP file was created with 10000 packets - the same number of packets as every other packet-based dataset - and then each packet was filled with a random number of guaranteed matches to the rules and the rest of the packet filled with random bytes. Each packet then ended up being exactly 1500 bytes in length (See Table 4.1).

8.2.1 Horspool

It is with *Dataset F* that Figure was created to test the speed of the Horspool algorithm against the number of matches found in the dataset.

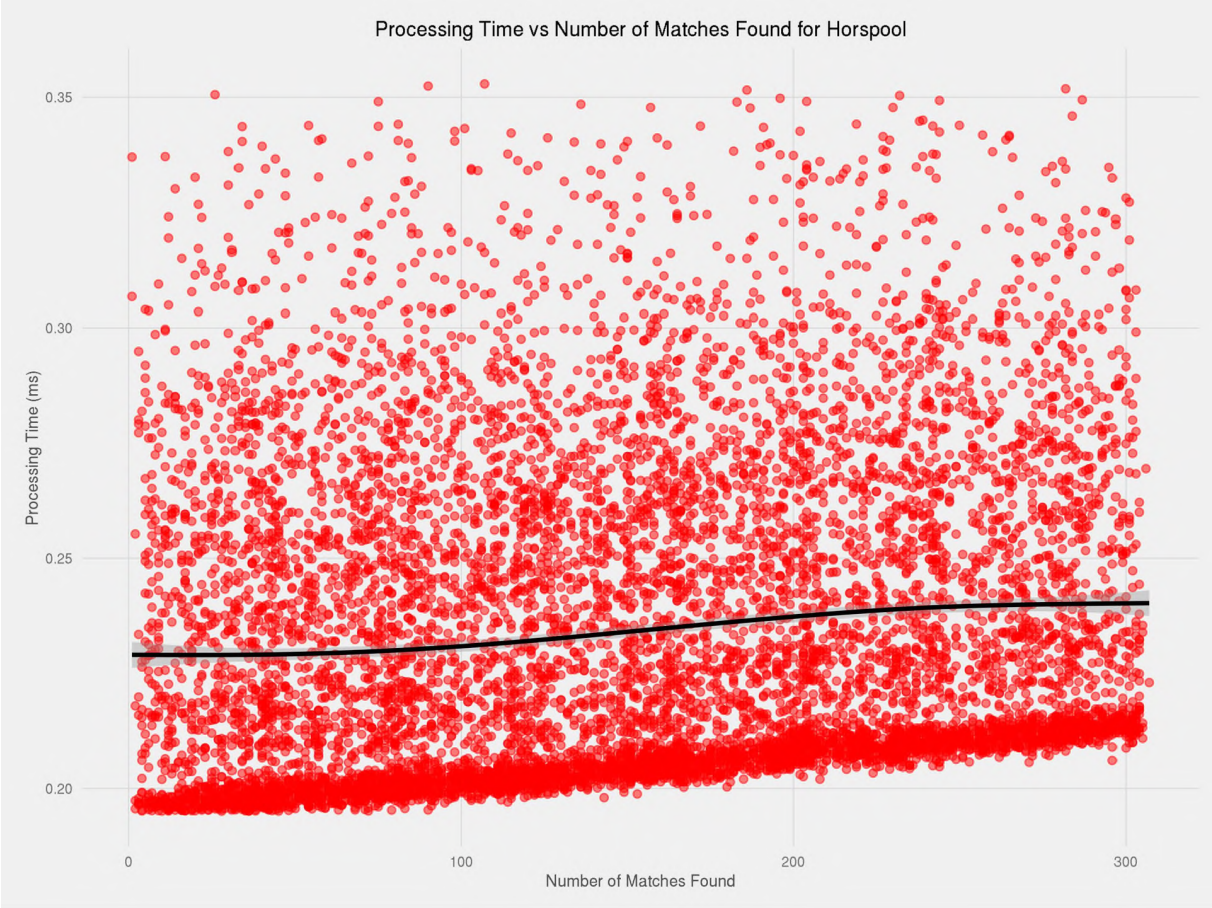

Figure 8.5: Horspool algorithm: Input processing time versus number of matches for *Dataset F*.

At first look, Figure appears to be a mess of highly variant data. Upon closer inspection there are a few elements which indicate how this algorithm actually performs.

The graph itself has processing time ranging from about 0.18 milliseconds all the way to just over 0.35 milliseconds. The curve fitted to the data indicates, by its slight upward trend, that the processing time of the algorithm does, in fact, increase as the number of matches increases.

An interesting point to note the the dramatic variation of the processing times for the same number of matches found and, because the number of matches is independent of the length of the input, this discrepancy must come from somewhere else. One explanation of

this behaviour is that, because the matches within each packet are randomised, there may be an overwhelmingly number of long matches in the data which take longer to process.

8.2.2 Quick Search

Figure 8.6 shows the results of the Quick Search algorithm in *Dataset F*.

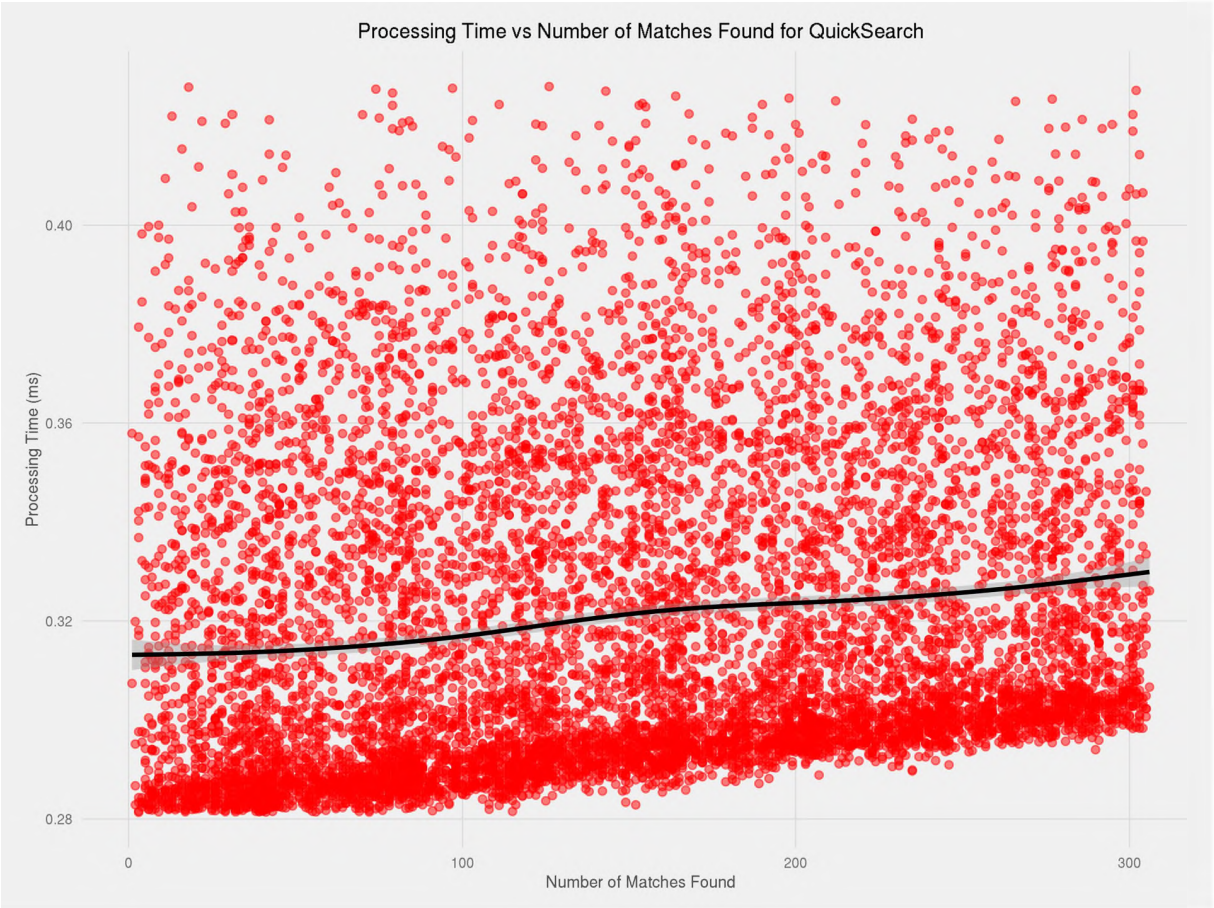

Figure 8.6: Quick Search algorithm: Input processing time versus number of matches for *Dataset F*.

The results of the Quick Search algorithm quite closely resemble those of the Horspool algorithm. These algorithms have shown similar results through the course of this research. The fitted curve to these results in Figure 8.6 shows un upward trend, steeper than that of the Horspool, which would also indicates a relationship between the number of matches and the length of the input.

8.2.3 Not So Naïve

Figure 8.7 presents the results of our next algorithm, the Not So Naïve algorithm, as it processes the inputs in *Dataset F*.

Figure 8.7: Not So Naïve algorithm: Input processing time versus number of matches for *Dataset F*.

The Not So Naïve algorithm's results show a very slight increase in processing time as the length of the input increases. The results here closely resemble the results presented earlier for the Horspool and Quick Search algorithms. In these results, a far larger variation on processing speed is seen. This variation would indicate that the relationship between number of matches and processing speed is not very strong.

The algorithm's processing speed may not exhibit a strong coupling with the number of matches because it spends so much time just processing the packet that the added processing overhead of an actual match is negligible.

8.2.4 Rabin-Karp

The forth and final comparison is with the Rabin-Karp algorithm and its processing of *Dataset F*. The results are presented in Figure 8.8.

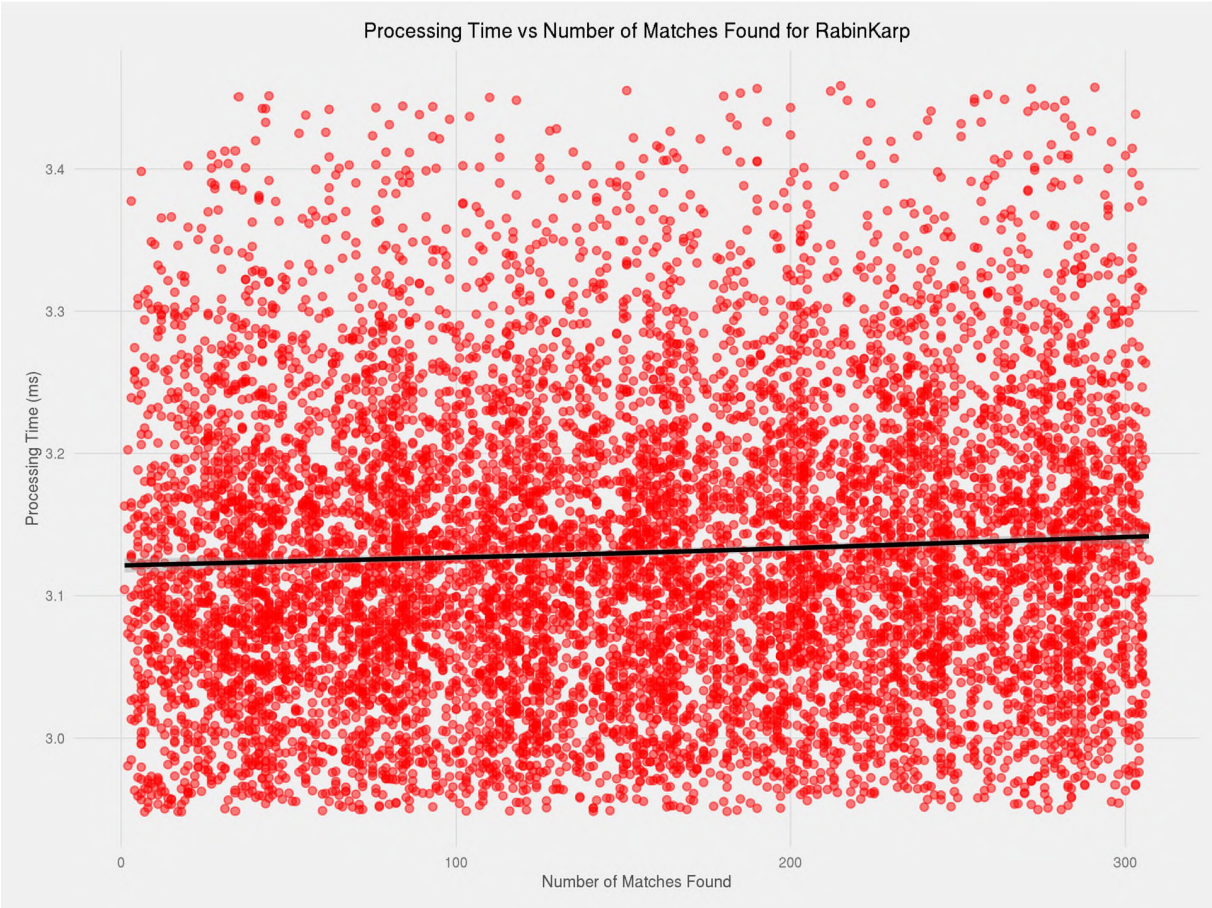

Figure 8.8: Rabin-Karp algorithm: Input processing time versus number of matches for *Dataset F*.

The Rabin-Karp algorithm's processing speed only shows a very small correlation to the number of matches found. This slight increase in processing time as the number of matches increase can likely be attributed to the overhead of the system as new matches are logged and not the algorithm itself.

The processing times themselves range from between about 2.9 milliseconds to just under 3.5 milliseconds. This processing time range would would appear to be just the natural variation in processing times for the Rabin-Karp algorithm with an input of 1500 bytes in length. This is, notedly, quite a large variance - especially when deterministic performance is needed for Deep Packet Inspection.

This section has examined the performance of each of the chosen algorithms when when the length of the input was fixed and the number of matches was varied. It is through this that the relationship between the number of matches and the processing time can be properly examined. The Horspool and Quick Search algorithms showed a much larger dependency on the the number of matches when compared to the Not So Naïve algorithm. The results of the Rabin-Karp test were devoid of a correlation between number of matches and processing speed. It may be that the faster the algorithm - the more it is affected by finding matches. There is certainly some overhead associated with reporting the matches.

8.3 How does multithreading affect processing speed?

This section examines how the use of multithreading can affect the performance of the chosen algorithms and tries to find a proper thread-count which guarantees best performance on the test system. As discussed in Section 7.2, the test system has a total of twenty four threads made available through the test hardware.

Each of the implemented algorithms were originally designed to search for just a single rule at a time. This is a perfectly justifiable decision especially since, at the time, most systems were running on single core, single thread CPUs (Lilly, 2009). Modern CPUs have been designed to allow for many physical CPU cores and even more hyper threads (Lilly, 2009). The test hardware (Section 7.2) is an example of such a design. With the advent of these multithreaded systems it would be wasteful to ignore all of the extra processing power available. The test system was designed with this in mind and runs each of the rules in parallel.

Multithreading in the test system works as follows:

- The number of threads to use is specified in the test configuration.
- A thread pool is created with a size equal to the number of threads specified.
- During run time, the test system assigns one thread from the threadpool to each rule.
- Once the rule has been searched for the thread is returned to the pool.

- If there are more rules than threads in the thread pool then the system must wait for a rule to finish before another one may be assigned a thread and start.

Essentially this design is a parallelism over multiple rules. A potential alternative implementation would be parallelism over multiple packets; it was not investigated here but could prove to be an interesting area of further research in the future. Further parallel implementations could also take a hybrid approach where both rules and packets are mixed and searched simultaneously.

With such a system, it is now possible to ask how the number of threads affects the processing speed of the algorithms. Table 8.1 shows each of the selected thread-counts for these tests.

Table 8.1: Thread counts used in the multithreading tests

Base 10	Base 2	Note
1	2^0	The same as the linear method of testing.
2	2^1	
4	2^2	
8	2^3	
16	2^4	The last value for which there are fewer program threads than processor threads.
32	2^5	The first value for which there are more program threads than processor threads but fewer program threads than rules.
64	2^6	The first value for which there are more threads than rules.

In the following tests, each of the selected algorithms was tested using *Dataset D* (Section 4.4, the dataset with a packets of random length but wherein each is filled entirely with matches to the rules, using the same rules as before but where the number of threads was varied each time.

8.3.1 Horspool

Figure 8.9 shows a comparison of processing times for each run of the Horspool algorithm at different numbers of threads. Each thread is assigned a different colour and the results are plotted as a function of the length of the input.

What is to be expected from these results is that, for smaller input lengths, fewer threads would be faster and as the input length increased, the number of threads for the most

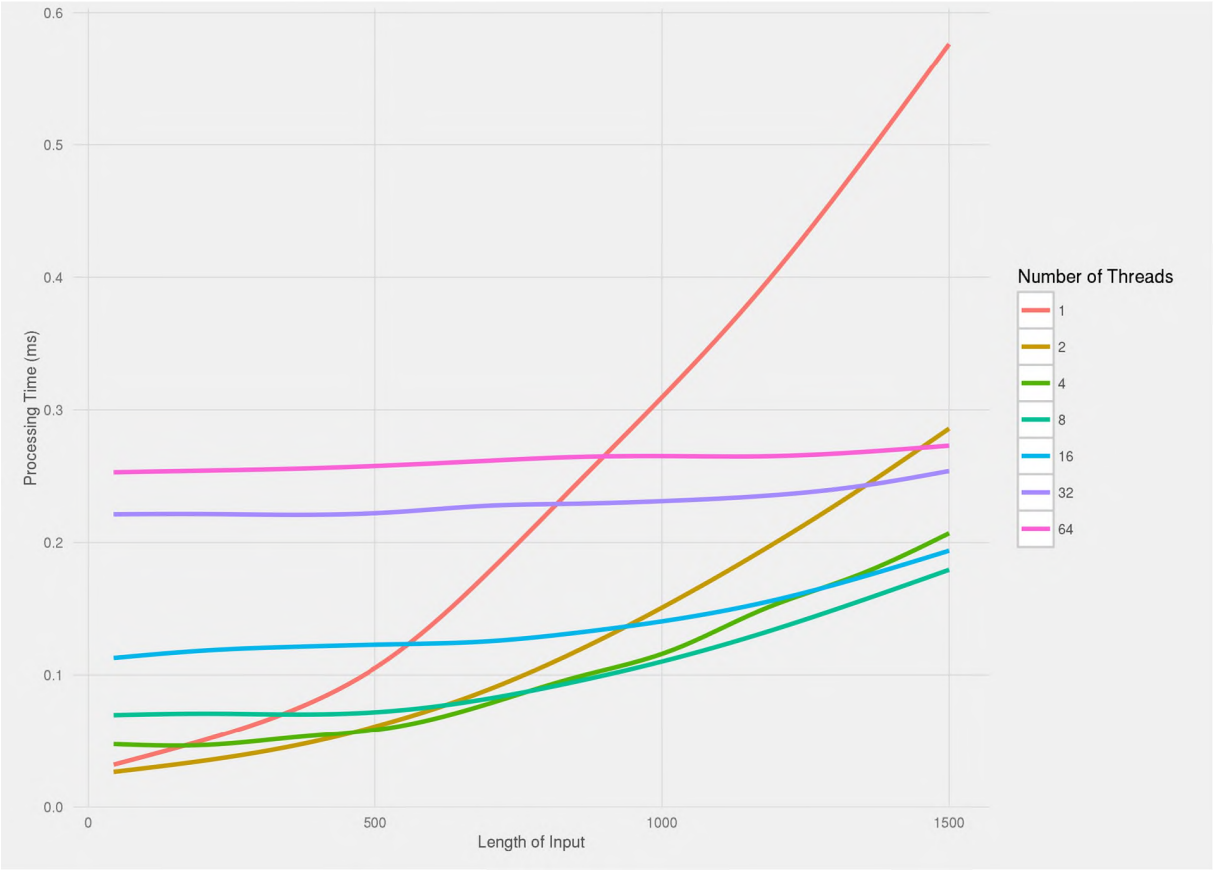

Figure 8.9: Horspool algorithm: Input processing time versus number of inputs for *Dataset D*.

efficient configuration would increase. For smaller inputs, the overhead associated with switching threads constantly would make the system slower and as the input grows, so too does the time to process, which makes the overhead for switching threads less impactful on the overall processing time.

In Figure 8.9, the behaviour that was just discussed is present. At the smaller input lengths, thread counts of one and two are far more efficient than higher thread counts. As the length of the input rises, the number of threads jostle for the fastest position until, at 1500 bytes, eight threads appears to be optimal.

The most efficient number of threads at the midpoint, 770 bytes, is either four or eight threads. It is at this thread-count, and for this particular machine and algorithm that the optimal number of threads is between four and eight.

8.3.2 Quick Search

Figure 8.10 gives a comparison of the processing speed for *Dataset D* of the Quick Search algorithm, for varying numbers of threads.

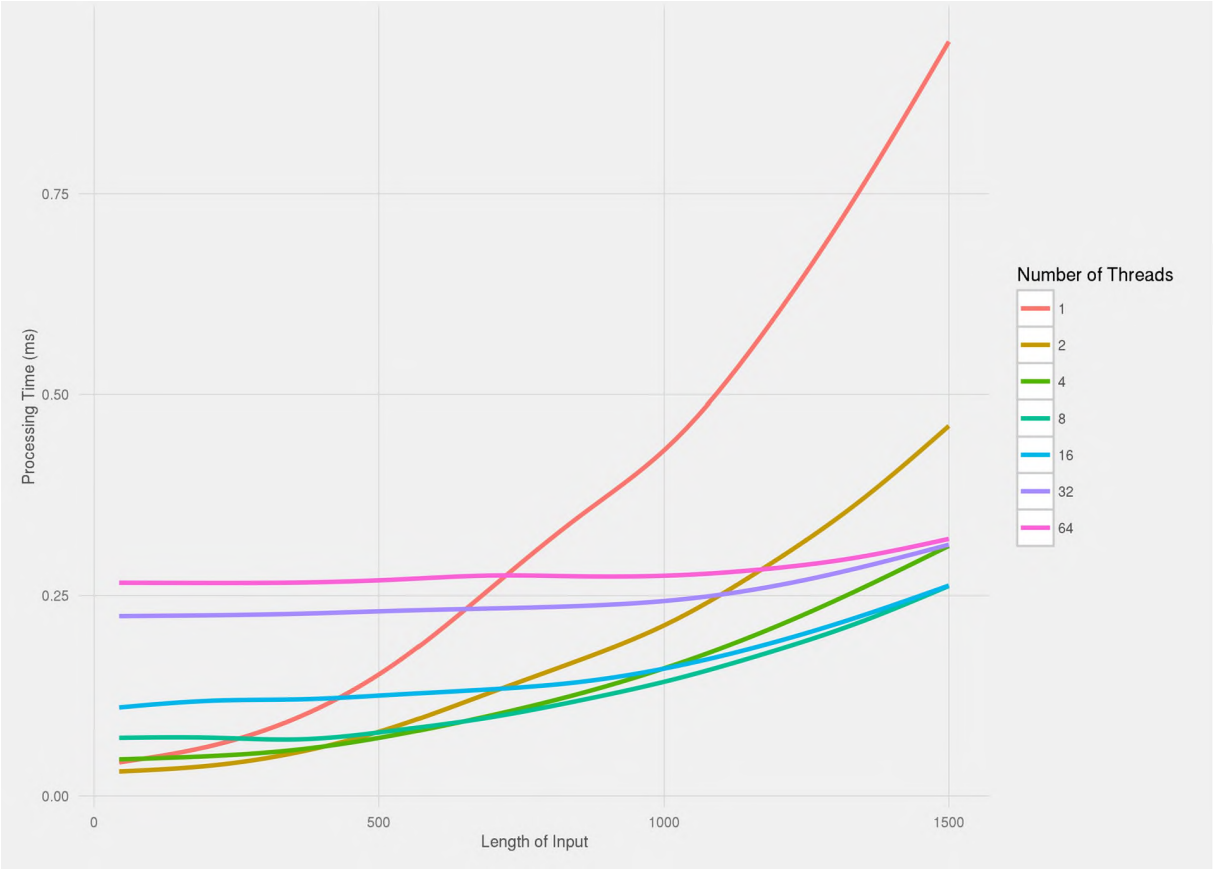

Figure 8.10: Quick Search algorithm: Input processing time versus number of inputs for *Dataset D*.

Earlier it was found that the Quick Search algorithm was more efficient than the Horspool algorithm for smaller input lengths. This relationship is evident in a comparison between Figures 8.9 and 8.10. In the Quick Search algorithm's graph, the four thread line becomes more efficient than the two thread line at inputs of about 400 bytes in length, in the Horspool algorithm's graph this crossover takes place at inputs of about 500 bytes in length.

The four thread line becomes less efficient than eight thread line at inputs of about 600 bytes in length and eventually, at inputs greater than 1500 bytes in length, the sixteen threaded line becomes the most efficient. The lines representing the results from the single thread test in both of the preceding graphs become totally inefficient very quickly, proving how important a multithreaded approach is to this problem.

8.3.3 Not So Naïve

Figure 8.11 shows the processing time versus the input length of the Not So Naïve algorithm with *Dataset D* as its input and for varying numbers of threads.

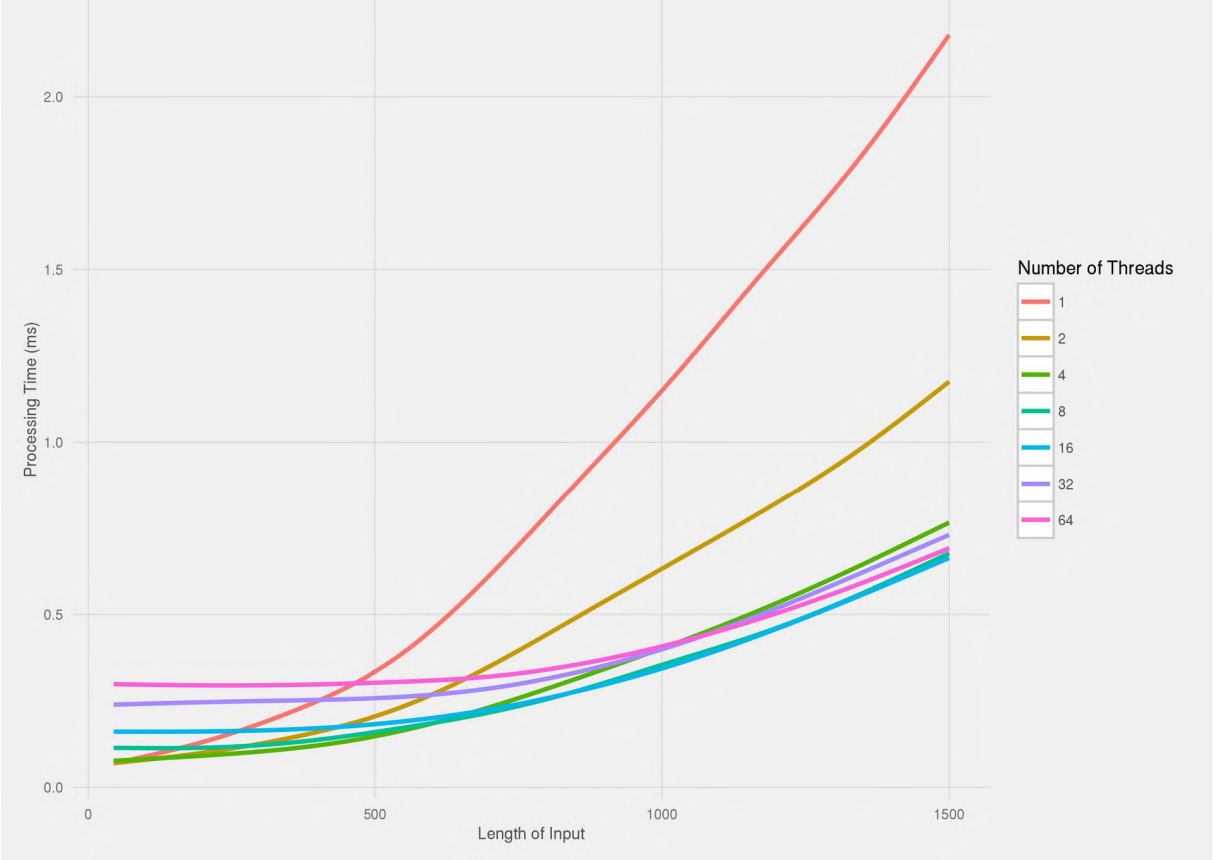

Figure 8.11: Not So Naïve algorithm: Input processing time versus number of inputs for *Dataset D*.

The Not So Naïve algorithm is one of the slowest algorithms in the test. Because of its slowness, more threads become more efficient faster than with the more efficient algorithms. For input lengths as low as about 770 bytes a sixteen thread configuration proved to be the most efficient.

Given the trend of the lines as they go from shorter input lengths to longer input lengths, one may think that, in the case of Figure 8.11, even more threads should be more efficient than sixteen threads at 1500 bytes. This would be the case for a system with a CPU capable of concurrently processing more threads but, for our test hardware with a maximum number of simultaneous threads of twenty four, thirty two threads would mean that there are more OS threads than there are available in the hardware. At this point

further overhead is introduced as thread vie for processing time on the CPU and there is further context switching as the operating system must try to fairly schedule the work.

This phenomenon - the hard limit set by the hardware itself - can be seen in the next test.

8.3.4 Rabin-Karp

The final test was performed using the Rabin-Karp algorithm to search through *Dataset D* using varying numbers of threads each time. Figure 8.12 compares the processing time of that test to the length of the input.

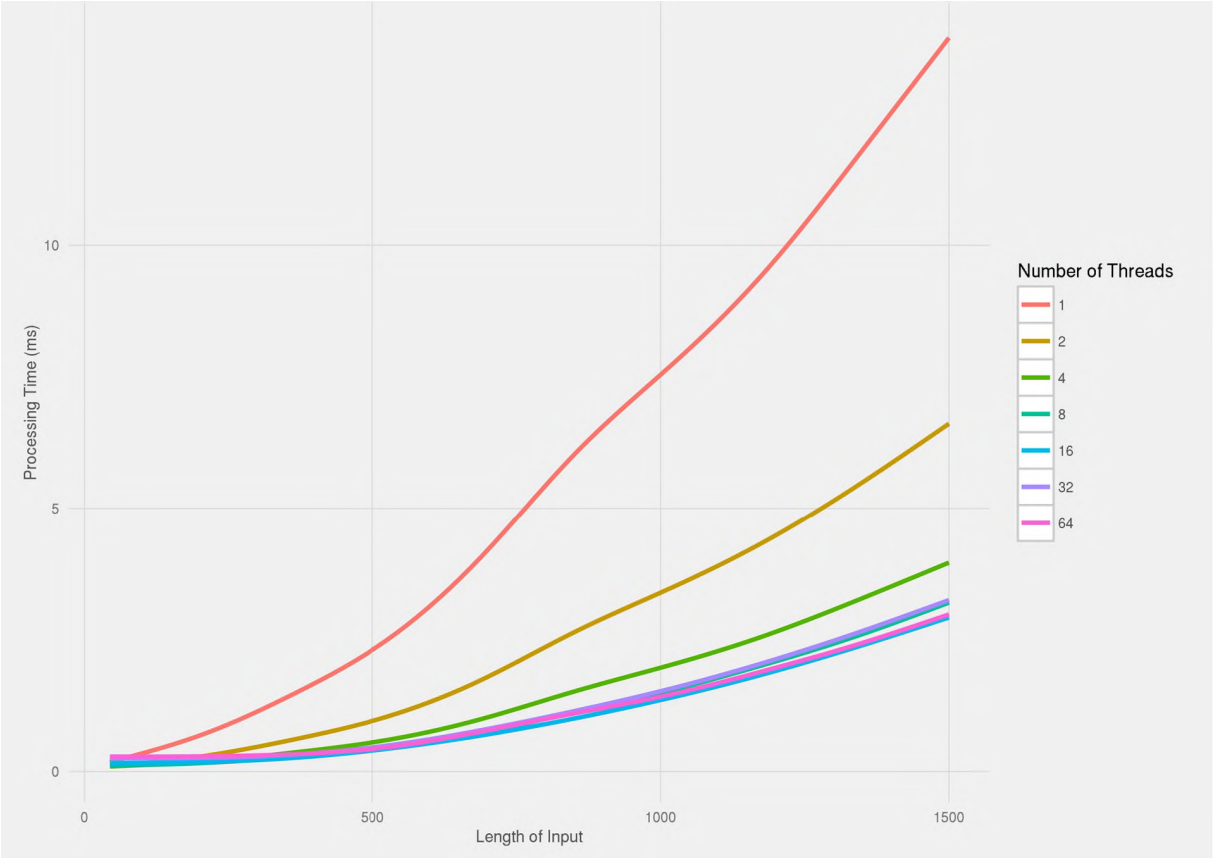

Figure 8.12: Rabin-Karp algorithm: Input processing time versus number of inputs for *Dataset D*.

This test, using the Rabin-Karp algorithm, is the most extreme of the cases that has been tested here. The Rabin-Karp algorithm proved to be very slow in previous tests (see Section 7.3). The slower algorithms have shown reach an efficient number of threads at lower inputs lengths than their faster counterparts. For the Rabin-Karp algorithm, at

the very smallest of input lengths, the eight threaded solution was most efficient but at inputs of about 200 bytes long the sixteen thread line takes over as most efficient and stays there for every other length of input.

As discussed earlier, for inputs of infinite length, and of the thread counts used in these tests, it is believed that sixteen threads would always be the most efficient. For packets of average length - about 770 bytes long - between four and sixteen threads has proven to be the most efficient number to use across the various algorithms used.

This section shows how important multithreading can be, especially with modern, highly parallelizable, CPUs. More than that, the points at which varying levels of multithreadedness are more efficient were investigated.

8.4 Summary

Table 8.2: Algorithm rankings for each test.

Rank	Speed *Dataset A* Subsection 7.3.1	Speed *Dataset B* Subsection 7.3.2	Variation *Dataset A* Section 7.4	Input Length *Dataset C* Sections 7.5 & 8.1	Matches *Dataset F* Section 8.2	Multithreading *Dataset D* Section 8.3
1	Quick Search	Horspool	Horspool	Horspool	Horspool	Horspool
2	Horspool	Quick Search	Rabin-Karp	Quick Search	Quick Search	Quick Search
3	Rabin-Karp	Not So Naïve	Quick Search	Not So Naïve	Not So Naïve	Not So Naïve
4	Not So Naïve	Rabin-Karp	Not So Naïve	Rabin-Karp	Rabin-Karp	Rabin-Karp

Throughout this chapter the four chosen algorithms have been put through further testing to really understand how they perform at the scale needed for Deep Packet Inspection. A number of questions were asked and answered surrounding the performance of these algorithms through the use of a multitude of different tests and inputs.

Table 8.2 gives a summary of the results presented in this and the previous chapter. The algorithms are ranked based on their performance in each of the tests conducted. For the 'Input Length', and 'Multithreading' tests, the algorithms have been ranked based on their speeds at 770 bytes. The 'Matches' tests have been ranked based on the performance of the algorithms at 150 matches.

From the results above it was shown how well the Horspool algorithm (Section 3.6) performed - both in terms of overall speed and determinism of processing time - much better than its peers. In this chapter it was also shown that faster algorithms seem to be more affected by the small overheads produced by the system itself. In Table 8.2, the Horspool

algorithm ranked first for five of six tests. The Horspool algorithm posted extremely good results in each of the tests and proved itself to be a very viable algorithm for packet inspection.

In the previous chapter, Chapter 7, the Quick Search algorithm (Section 3.9) showed the potential for becoming a good algorithm for packet inspection. It was later shown that this algorithm, although relatively fast, produced results that were quite variant. A large variation of processing times in a real-time packet inspection system could be open to unpredictable slowdowns and perhaps even cause denial of service. In Table 8.2, the Quick Search algorithm fared well in each of the tests expect for the test of variation. Although seemingly minor, this behaviour is most unwarranted. An algorithm such as the Quick Search algorithm may have, at first glance, shown strong signs of being a good packet inspection algorithm. After further investigation it was revealed that the Quick Search algorithm posses unwanted properties.

The slower algorithms were also examined in an attempt to see what hampered their performance. In the first test in Chapter 7, the Not So Naïve algorithm (Section 3.17) showed very poor processing times for *Dataset A*. Upon closer inspection it was found that the Not So Naïve algorithm was actually not the slowest overall. That title belonged to the Rabin-Karp algorithm. Table 8.2 shows how the Not So Naïve algorithm moved from being the poorest performing algorithm in the first test to being only the second worst performer in each of the subsequent tests.

The Rabin-Karp algorithm (Section 3.7) showed initially mediocre results and later proved to be extremely inefficient at longer length inputs. By examining such a slow algorithm the behaviour of these algorithms for longer processing times is now better understood. Table 8.2 shows the results of the Rabin-Karp algorithm in each of the performed tests. The behaviour of the Rabin-Karp algorithm is indicative of an algorithm which performs well for shorter inputs but has poor performance for inputs of about 1500 bytes.

Chapter 9

Conclusion

The research presented in this document has implemented, assessed and compared a large number of exact string search algorithms with a variety of different textual and packet-based inputs[1]. Each different test pitted the algorithms against each other in order to understand how they performed in the context of Deep Packet Inspection. Although the raw performance of these algorithms does not match the processing speed offered by other implementations (Chaudhary and Sardana, 2011), it is believed that this novel research may, one day, prove useful.

In order to conduct the tests and generate the statistics used throughout this body of work, a test harness needed to be constructed. Although systems for comparing string search algorithms (Faro and Lecroq, 2011) and others for performing packet inspection exist, no such system for comparing string search algorithms in the context of Deep Packet Inspection had been developed. The system developed proved extremely useful for generating precise, accurate data for later examination.

9.1 Document Recap

This section summarises the document in its entirety.

[1]In Chapter 7, ten thousand and one inputs were used on nineteen different algorithms over twenty separate test runs generating almost four million data points. In Chapter 8, ten thousand inputs were used on four algorithms in thirty six different tests, each test with twenty runs, for a total of just under thirty million data points.

Chapter 1 - Introduced the research itself, defined its scope and gave a layout of the document to come.

Chapter 2 - A comprehensive overview of the current state of general network security (Section 2.1), Intrusion Detection Systems (Section 2.3), firewalls (Section 2.2), and Deep Packet Inspection (Section 2.4).

Chapter 3 - An introduction to searching for strings in text and details about each of the implemented algorithms (Sections 3.2 to 3.19).

Chapter 4 - Discussed each of the six datasets used later in the tests. The datasets were labeled *Dataset A* to *Dataset F*.

Chapter 5 - The discussion of the test system began by giving an analysis of the design of the system itself. This chapter makes use of diagrams to succinctly describe the functioning of the system.

Chapter 6 - Continues the discussion of the test system, this time by discussing the implementation details. Example uses of the system of given - complete with screenshots of it during use.

Chapter 7 - The implemented algorithms were initially compared for speed using *Dataset A* and *Dataset B*, packet and textual datasets, respectively. The algorithms were then compared for variation on processing times. Four algorithms, Horspool, Quick Search, Not So Naïve, and Rabin-Karp were selected for further examination. The four selected algorithms had their speed compared to the length of the input to gain insight into their algorithmic complexity.

Chapter 8 - This chapter continues on the work done in Chapter 7 buy further examining the four selected algorithms. Another test comparing algorithm speed and input length was conducted, followed by a comparison of algorithm speed and number of matches. Finally the four algorithms were tested using varying numbers of threads to see what the optimal number of threads was for each.

9.2 Research Objectives

In Chapter 1, Section 1.2, a number of research objectives were listed and the following reports on the progress towards those objectives:

- The first goal of this research was to establish the current state of the art for software-based Deep Packet Inspection. Chapter 2, and specifically in Sections 2.1 to 2.4, works through the background behind this research from a network security point of view, concluding with a discussion on packet inspection and specifically Deep Packet Inspection in Subsection 2.4.3. The importance of network security (Section 2.1), and the role of IDSs and firewalls (Sections 2.3 & 2.2) was also emphasised.

- A set of string search algorithms which may perform well at Deep Packet Inspection was compiled and discussed in Chapter 3. Their specific algorithmic complexity as well as finer points about their history and behaviour were discussed.

- A test system was developed for the purpose of testing the algorithms. The system itself was designed in such a way that it can easily be extended to add more algorithms or alter its functionality. That system is discussed fully in Part II.

- The test system was then used to run each of the algorithms through a bevy of tests in order to compare them for use in Deep Packet Inspection. Various inputs were used which are documented during the testing in Chapters 7 and 8 and in Chapter 4.

- Lastly the results were heavily analysed and comments were made about some of the algorithms and their potential performance at Deep Packet Inspection. It was found that the Horspool algorithm performed very well in most of the tests, the Quick Search algorithm showed promise but had unwanted behaviour in the form of processing time variation, the Not So Naïve algorithm very slow in the tests with *Dataset A* and finally the Rabin-Karp was fairly quick for shorter input but slows heavily as the input length increases.

9.3 Future Work

The result of the work done during this research has been presented in Chapters 7 and 8. That work focused on the benchmarking, comparison and later analysis of a set of exact string matching algorithms which had originally been designed to find short strings in larger bodies of text. Those results allowed for the proper comparison of the algorithms in a number of scenarios and investigate the properties which are important to Deep Packet Inspection. In order to achieve the results that have been presented, a test system was designed to accurately measure the performance of the algorithms. This test system

proved itself to be a very robust platform for the kind of testing that was important for this research. The system itself was designed to be as extensible as possible and allow for very complex configurations and operation thereafter. With these details in mind the following is a list of future work in the field and on the test system:

- Only a fraction of the existing exact string matching algorithms were implemented for this work as time would not allow for implementation and testing of every available algorithm. Charras and Lecroq (2004), in their work, provide a list and analysis of thirty four algorithms and even more are available online through the work presented in Faro and Lecroq (2011). It is a fairly time consuming job to implement each to the string search algorithms but, once implemented, the test system accepts them easily.

- This work has focused solely on exact string matching algorithms (these are defined at the beginning of Chapter 3), there are very many other string searching algorithms which do not qualify as exact string matching algorithms. Some algorithms are able to match multiple rules at the same time and others have no limit to the number of rules they may match. It would be interesting to see a comparison of these other types of algorithms and contrast their results with those expressed in this research. Regular expression (Thompson, 1968) based matching is very popular today and a comparison with some algorithms implementing regular expressions would be enlightening.

- The test system itself can be relatively easily extended to add additional functionality. Example of such functionality might include additional configuration based on which byte a rule may start and end the matching. Alternative input types might be useful too and adding the ability for the system to directly read from a packet capture handle would be a useful addition.

- In addition to the previous point, a real-world study might prove to be very enlightening. Once the system has been extended to where a packet capture handle is understood, the system could act as a rudimentary firewall.

- Software-based approaches to Deep Packet Inspection - such as the ones discussed in this research - have been shown to not be as fast as other, hardware-based, implementations (Chaudhary and Sardana, 2011). This research does not make any direct comparisons between the speed of the string search algorithms and hardware-based techniques.

Although string search algorithms are not widely used in systems that implement Deep Packet Inspection, it is important to establish a comparison of these algorithms. Future advancements to the general purpose processors used in this research could mean that string search algorithms become more useful for Deep Packet Inspection. As more and more information is entrusted to computers, likewise does the requirement for good, practical security measures grow.

References

M. Abliz. Internet Denial of Service Attacks and Defense Mechanisms. Technical report, Department of Computer Science, University of Pittsburgh, 2011.

T. AbuHmed, A. Mohaisen, and D. Nyang. A Survey on Deep Packet Inspection for Intrusion Detection Systems. *Journal of Korean Communications (Information and Communication)*, 24(11):25–36, 2007.

O. Adeyinka. Internet Attack Methods and Internet Security Technology. In *Second Asia International Conference on Modeling & Simulation, 2008. AICMS 08*, pages 77–82, Kuala Lumpur, Malaysia, 2008. IEEE.

A. Aho. *Algorithms for Finding Patterns in Strings*, chapter 5, pages 255–300. Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity. Elsevier, 1990.

A. Aho, J. Hopcraft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. 978-0-201-00029-0. Addison-Wesley, 1974.

E. Al-Shaer and H. Hamed. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In G. Goldszmidt and J. Schönwälder, editors, *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003*, pages 17–30. IEEE, 2003.

R. Alshammari and A. Zincir-Heywood. Can Encrypted Traffic Be Identified Without Port Numbers, IP Addresses and Payload Inspection? *Computer Networks*, 55(6): 1326–1350, 2011.

J. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980.

S. Anthony. GitHub Battles "largest DDOS" in Site's History, Targeted at Anti-censorship Tools. Online `http://arstechnica.com/security/2015/03/github-`

`battles-largest-ddos-in-sites-history-targeted-at-anti-censorship-tools/` Date Accessed: 29 April 2016, 2015.

A. Apostolico and M. Crochemore. Optimal Canonization of all Substrings of a String. *Information and Computation*, 95(1):76–95, 1991.

A. Apostolico and R. Giancarlo. The Boyer Moore Galil String Searching Strategies Revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.

J. Aschenbrenner. Open Systems Interconnection. *IBM Systems Journal*, 25(3.4):369–379, 1986.

A. Ashoor and S. Gore. Importance of Intrusion Detection System (IDS). *International Journal of Scientific and Engineering Research*, 2(1):1–4, 2011.

F. Avolio. Firewalls and Internet Security, the Second Hundred (Internet) Years. *The Internet Protocol Journal*, 2(4):24–32, 1999.

P. Bachman. *Die Analytische Zahlentheorie*, volume 2. Teubner, 1894.

R. Baeza-Yates and G. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, 35(10):74–82, 1992.

M. Becchi, C. Wiseman, and P. Crowley. Evaluating Regular Expression Matching Engines on Network and General Purpose Processors. In P. Onufryk, K. Ramakrishnan, P. Crowley, and J. Wroclawski, editors, *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

R. Bemer. A Proposal for Character Code Compatibility. *Communications of the ACM*, 3(2):71–72, 1960.

R. Bendrath and M. Mueller. The End of the Net as We Know It? Deep Packet Inspection and Internet Governance. *New Media & Society*, 13(7):1142–1160, 2011.

D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing Skype Traffic: When Randomness Plays with You. *ACM SIGCOMM Computer Communication Review*, 37(4):37–48, 2007.

D. Borman, S. Deering, and R. Hinden. Request for Comments: 2675 - IPv6 Jumbograms. Technical report, IETF, 1999.

R. Boyer and J. Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 1998.

D. Breslauer. *Efficient String Algorithmics*. PhD thesis, Computer Science Department, Columbia University, New York, New York, 1992.

A. Callado, J. Kelner, D. Sadok, C. Kamienski, and S. Fernandes. Better Network Traffic Identification through the Independent Combination of Techniques. *Journal of Network and Computer Applications*, 33(4):433–446, 2010.

V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 2009.

C. Charras and T. Lecroq. *Handbook of Exact String-Matching Algorithms*. Institut d'électronique et d'informatique Gaspard-Monge, 2004.

M. Chatel. Request for Comments: 1919 - Classical versus Transparent IP Proxies. Technical report, IETF, 1996.

A. Chaudhary and A. Sardana. Software Based Implementation Methodologies for Deep Packet Inspection. In *2011 International Conference on Information Science and Applications*, pages 1–10, Jeju Island, Republic of Korea, 2011. IEEE.

B. Cheswick. The Design of a Secure Internet Gateway. In *USENIX Summer Conference Proceedings*, 1990.

L. Colussi. Correctness and Efficiency of Pattern Matching Algorithms. *Information and Computation*, 95(2):225–251, 1991.

L. Colussi. Fastest Pattern Matching in Strings. *Journal of Algorithms*, 16(2):163–189, 1994.

B. Corbridge, R. Henig, and C. Slater. Packet Filtering in an IP Router. In *Proceedings of the Fifth USENIX Large Installation and System Administration Conference*, pages 227–232, 1991.

M. Crochemore and T. Lecroq. *Pattern Matching and Text Compression Algorithms*, chapter 8, pages 162–202. CRC Press Inc., Boca Raton, Florida, 1996.

M. Crochemore and R. Wojciech. *Jewels of Stringology: Text Algorithms*. World Scientific, 2002.

M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up Two String-Matching Algorithms. *Algorithmica*, 12(4-5):247–267, 1994.

D. Crockford. Request for Comments: 4627 - The application/json Media Type for JavaScript Object Notation (JSON). Technical report, IETF, 2006.

S. Deering and R. Hinden. Request for Comments: 2460 - Internet Protocol, Version 6 (IPv6) Specification. Technical report, IETF, 1998.

D. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, (2):222–232, 1987.

P. Denning. The Science of Computing: The Internet Worm. *American Scientist*, 77(2): 126–128, 1989.

S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In D. Azada, editor, *2003 IEEE 11th Annual Symposium on High-Performance Interconnects*, pages 44–51, Stanford University, Stanford, California, 2003. IEEE.

C. Dougligeris and A. Mitrokotsa. DDoS Attacks and Defense Mechanisms: Classification and State-of-the-Art. *Computer Networks*, 44(5):643–666, 2004.

P. W. Dowd and J. T. McHenry. Network Security: It's Time to Take It Seriously. *Computer*, 31(9):24–28, 1998.

K. Egevang and P. Francis. Request for Comments: 1631 - The IP Network Address Translator (NAT). Technical report, IETF, 1994.

T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, and M. S. Lynn. The Cornell Commission: on Morris and the Worm. *Communications of the ACM*, 32(6):706–709, 1989.

T. Eisenmann, G. Parker, and M. van Alstyne. Opening Platforms: How, When and Why? In A. Gawer, editor, *Platforms, Markets and Innovation*. Edward Elgar Publishing, 2009.

H. Fan, N. Yao, and H. Ma. Fast Variants of the Backward-Oracle-Marching Algorithm. In *Fourth International Conference on Internet Computing for Science and Engineering*, pages 56–59, 2009.

S. Faro and T. Lecroq. Efficient Variants of the Backward-Oracle-Matching Algorithm. In 20, editor, *International Journal of Foundations of Computer Science*, volume 6, pages 967–984. World Scientific, 2009.

S. Faro and T. Lecroq. SMART: String Matching Research Tool. Online `http://www.dmi.unict.it/~faro/smart/` Date Accessed: 10 May 2016, 2011.

S. Faro and T. Lecroq. The Exact Online String Matching Problem: A Review of the Most Recent Results. *ACM Computing Surveys*, 45(2):13, 2013.

Z. R. Feng and T. Takaoka. On Improving the Average Case of the Boyer-Moore String Matching Algorithm. *Journal of Information Processing*, 10(3):173–177, 1987.

P. Ferguson and D. Senie. Request for Comments: 2827 - Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical report, IETF, 2000.

H. Fowler, F. Fowler, and R. Allen. *The Concise Oxford Dictionary: firewall, n.* ISBN: 0-19-861200-1. Clarendon Press - Oxford, 1990.

Z. Galil and R. Giancarlo. On the Exact Complexity of String Matching: Upper Bounds. *SIAM Journal on Computing*, 21(3):407–437, 1992.

J. Gantz, A. Florean, R. Lee, V. Lim, B. Sikdar, S. Lakshmi, L. Madhavan, and M. Nagappan. The Link between Pirated Software and Cybersecurity Breaches. Technical report, National University of Singapore and IDC, 2014.

L. Garcia. Programming with Libpcap - Sniffing the Network From Our Own Applications. In *Hakin9*, volume February 2008, pages 38–46. HAKIN9 MEDIA SP, 2008.

P. Gardner. The Internet Worm: What Was Said and When. *Computers and Security*, 8 (4):305–316, 1989.

A. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings. 14th Annual Computer Security Applications Conference, 1998*, pages 259–267. IEEE, 1998.

I. Grondman. Identifying Short-term Periodicities in Internet Traffic. Master's thesis, University of Twente, 2006.

V. Gupta. File Detection in Network Traffic Using Approximate Matching. Master's thesis, Norwegian University of Science and Technology, 2013.

M. Haertel. Why GNU grep is Fast. Online `https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html` Date Accessed: 9 May 2016, 2010.

C. Hancart. *Analyse Exacte et en Moyenne D'algorithmes de Recherche D'un Motif dans un Texte.* PhD thesis, Université Paris Diderot, 1993.

M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Security Symposium*, pages 115–131, 2001.

M. Hauben and R. Hauben. Behind the Net: The Untold History of the ARPANET and Computer Science. *Netizens: On the History and Impact of Usenet and the Internet*, 2006.

M. Hibberd. Encryption: Will It Be the Death of DPI? Online `http://telecoms.com/39718/encryption-will-it-be-the-death-of-dpi/` Date Accessed: 03 May 2016, 2012.

S. Hoffman. DDoS: A Brief History. Online `https://blog.fortinet.com/post/ddos-a-brief-history` Date Accessed: 9 May 2016, 2013.

R. N Horspool. Practical Fast Searching Strings. *Software: Practice and Experience*, 10 (6):501–506, 1980.

R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

K. Ingham and S. Forrest. A history and survey of network firewalls. Technical report, University of New Mexico, 2002.

J. Jenkov. Java Concurrency / Multithreading Tutorial. Online `http://tutorials.jenkov.com/java-concurrency/index.html` Date Accessed: 9 May 2016, 2014.

W. Jiang and V. Prassana. Large-Scale Wire-Speed Packet Classification on FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 219–228. ACM, 2009.

R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

R. Kemmerer and G. Vigna. Intrusion Detection: A Brief History and Overview. *Computer*, (4):27–30, 2002.

D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):232–350, 1977.

D. Koblas and M. Koblas. SOCKS. In *In UNIX Security Sympsium III Proceedings*, 1992.

M. Külekci. Filter Based Fast Matching of Long Patterns by Using SIMD Instructions . In J. Holub and Žďárek, editors, *In Proceedings of the Prague Stringology Conference*, pages 118–128, 2009.

S. Kumar, J. Turner, and J. Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 81–92. ACM, 2006.

E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, volume 1. Teubner, 1909.

D. Law, W. Diab, A. Healy, S. Carlson, V. Maguire, O. Anslow, and M. Hajduczenia. IEEE Standard for Ethernet. Technical report, IEEE Standards Association, 2012.

T. Lecroq. Experimental Results on String Matching Algorithms. *Software: Practice and Experience*, 25(7):727–765, 1995.

T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102 (6):229–235, 2007.

B. Leiner, V. Cerf, D. Clark, R. Kah, L Kleinrock, D. Lynch, J. Postel, L. Roberts, and S. Wolff. A Brief History of the Internet. *SIGCOMM Computer Communications Review*, 39(5):22–31, 2009.

Y. Liao. A survey of software-based string matching algorithms for forensic analysis. In *Proceedings of the Conference on Digital Forensics, Security and Law*, page 77. Association of Digital Forensics, Security and Law, 2015.

P. Lilly. A Brief History of CPUs: 31 Awesome Years of X86. Online `http://www.pcgamer.com/a-brief-history-of-cpus-31-awesome-years-of-x86/` Date Accessed: 10 May 2016, 2009.

Y. Lin, P. Lin, V. Prassana, H. Chao, and J. Lockwood. Guest editorial deep packet inspection: Algorithms, hardware, and applications. *IEEE Journal on Selected Areas in Communications*, 32(10):1781–1783, 2014.

J. Metcalf. Creeper & Reaper. Online `http://corewar.co.uk/creeper.htm` Date Accessed: 4 May 2016, 2014.

J. Mogul. Simple and Flexible Datagram Access Controls for Unix-based Gateways. Technical report, Western Research Laboratory, 1989.

A. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In C. Dovrolis, editor, *Passive and Active Network Measurement*. Springer, 2005.

G. Moore. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 3(20):33–35, 1965.

J. H. Morris and V. R. Pratt. A Linear Pattern-Matching Algorithm. Technical report, University of California, Berkeley, 1970.

M. Mueller and H. Asghari. Deep Packet Inspection and Bandwidth Management: Battles over BitTorrent in Canada and the United States. *Telecommunications Policy*, 36(6): 462–475, 2012.

M. Necker, D. Contis, and D. Schimmel. TCP-Stream Reassembly and State Tracking in Hardware. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002*, pages 286–287. IEEE, 2002.

R. Needham. Denial of Service. In *CCS '93 Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 151–153. ACM, 1993.

J. Neilsen. Nielsen's Law of Internet Bandwidth. Online `https://www.nngroup.com/articles/law-of-bandwidth/` Date Accessed: 07 April 2016, 1998.

Arbor Networks. Arbor E100. Online `http://www.lextel.com/wp-content/uploads/E100Datasheet.pdf` Date Accessed: 14 February 2016, 2011.

Palo Alto Networks. Education. Online `https://www.paloaltonetworks.com/services/education` Date Accessed: 1 May 2016, 2016.

C. Parsons. Deep Packet Inspection and Law Enforcement. Online `https://www.christopher-parsons.com/deep-packet-inspection-and-law-enforcement/` Date Accessed: 3 May 2016, 2009.

C. Parsons. *The Politics of Deep Packet Inspection: What Drives Contemporary Western Internet Service Provider Surveillance Practices*. PhD thesis, Department of Political Science, University of Victoria, 2014.

C. Perrin. The CIA Triad. Online `http://www.techrepublic.com/blog/it-security/the-cia-triad/` Date Accessed: 2 May 2016, 2008.

R. Pike and K. Thompson. Hello World. In *Proceedings of the Winter 1993 USENIX Conference, Berkeley, CA, USENIX Association*, pages 43–50. USENIX, 1993.

J. Postel. Request for Comments: 791 - Internet Protocol. Technical report, IETF, 1981.

T. Raita. Tuning the Boyer-Moore-Horspool String Searching Algorithm. *Software: Practice and Experience*, 22(10):879–884, 1991.

G. Ramirez. `randpkt`. Online `https://github.com/wireshark/wireshark/blob/master/randpkt.c` Date Accessed: 10 May 2016, 1999.

M. Ranum. A Network Firewall. In *Proceedings of the World Conference on System Administration and Security*, 1992.

M. Rouse. Content Filtering (Information Filtering). Online `http://searchsecurity.techtarget.com/definition/content-filtering` Date Accessed: 11 June 2016, 2011.

D. Schuff and V. Pai. Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface. In *Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

N. Shah. Understanding Network Processors. Master's thesis, University of California, Berkeley, 2001.

U. Shankar and V. Paxon. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *2003 Symposium on Security and Privacy, 2003.*, pages 44–61. IEEE, 2003.

J. Sherry, C. Lan, R. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. Technical report, ACM SIGCOMM Computer Communication Review, 2015.

I. Simon. String Matching Algorithms and Automata. In *Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science.* Springer, 1994.

D. Smith, P. Experiments with a Very Fast Substring Search Algorithm. *Software: Practice and Experience*, 21(10):1065–1074, 1991.

D. Smith, P. On Tuning the Boyer-Moore-Horspool String Search Algorithms. *Software: Practice and Experience*, 1994.

I. Sourdis. *Designs and Algorithms for Packet and Content Inspection.* PhD thesis, Delft University of Technology, 2007.

E. Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, 1989a.

E. Spafford. The Internet Worm Program: An Analysis. *Communications of the ACM*, 19(1):17–57, 1989b.

A. Srikantha, A. Bopardikar, K. Kaipa, P. Venkataraman, K. Lee, T. Ahn, and R. Narayanan. A Fast Algorithm for Exact Sequence Search in Biological Sequences Using Polyphase Decomposition . *Bioinformatics*, 26(18):i414–i419, 2010.

G. Stephen. *String Search Algorithms*, volume 3 of *Lecture Notes Series on Computing*. World Scientific, 1994.

C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, 1989.

D. M. Sunday. A Very Fast Substring Search Algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

K. Thompson and D. Ritchie. UNIX Proframmer's Manual. Technical report, Bell Telephone Laboratories, 1975.

S. Thomson, C. Huitema, V. Ksinant, and M. Souissi. Request for Comments: 3596 - DNS Extensions to Support IP Version 6. Technical report, IETF, 2003.

/u/quink. Here's a new scenario I just created illustrating what happens if net neutrality disappears. [PIC]. Online `https://www.reddit.com/comments/9yj1f/heres_a_new_scenario_i_just_created_illustrating`, 2009.

J. van Splunder. Periodicity Detection in Network Traffic. Master's thesis, Mathematisch Instituut, Universiteit Leiden, 2015.

M. Ward. H@ppy Birthday to You. Online `http://news.bbc.co.uk/2/hi/in_depth/sci_tech/2000/dot_life/1586229.stm` Date Accessed: 6 May 2016, 2001.

H. Wickham. ggplot2. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(2): 180–185, 2011.

S. Wu and U. Manber. Fast Text Searching: Allowing Errors. *Communications of the ACM*, 35(10):83–91, 1992.

U. Wuermeling. New Dimensions of Computer-Crime - Hacking for the KGB - A Report. *Computer Law & Security Review*, 5(4):20–21, 1989.

C. Yang, M. Liao, M. Luo, S. Wang, and C. Yeh. A Network Management System Based on DPI. In *2010 13th International Conference on Network-Based Information Systems (NBiS)*, pages 385–388. IEEE, 2010.

F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 93–102. ACM, 2006.

E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls*. O'Reilly Media, Inc., 2000.

Appendix A

Figure A.1, by /u/quink (2009), shows the possible outcome if net neutrality is not enforced. This particular example shows the potential offerings of an Internet Service Provider who discriminates heavily based on the kind of traffic being transmitted.

Figure A.1: If ISPs did not respect Net Neutrality (/u/quink, 2009)

Appendix B

Listing B.1 shows the Python code used to create *Dataset A* to *Dataset F* in Chapter 4. The code is available online at `https://github.com/KieranHunt/pcapcreator`.

```
from scapy.all import *
import sys
from random import shuffle
import string

max_payload_size = 1500 - 39

url_params = sys.argv

a = rdpcap(url_params[1])

rules = ["time", "person", "year", "way", "day", "thing", "man",
    "world", "life", "hand", "part", "child", "eye",
         "woman", "place", "work", "week", "case", "point", "
   government", "google", "facebook", "youtube", "baidu",
         "yahoo", "amazon", "wikipedia", "qq", "twitter", "
   taobao", "live", "sina", "linkedin", "weibo", "ebay",
         "yandex", "hao123", "vk", "bing", "msn"]

packets = []

for i, packet in enumerate(a):

    # Dataset D

    shuffle(rules)
    random_rule_string = ''.join(rules)
    length_of_payload = len(packet.payload.payload.payload.
```

```
    payload.payload)
26    length_of_rule_string = len(random_rule_string)
27
28    divisor = max_payload_size / length_of_rule_string
29
30    random_rule_string = random_rule_string*(divisor + 1)
31
32    packet.payload.payload.payload.payload.payload =
    random_rule_string[:length_of_payload]
33
34    # Dataset E
35
36    shuffle(rules)
37    random_rule_string = ''.join(rules)
38    length_of_payload = len(packet.payload.payload.payload.
    payload.payload)
39    length_of_rule_string = len(random_rule_string)
40
41    divisor = max_payload_size / length_of_rule_string
42
43    random_rule_string = random_rule_string*(divisor + 1)
44
45    if (length_of_payload != 0):
46        random_value_less_than_length = random.randint(1,
    length_of_payload)
47
48        packet.payload.payload.payload.payload.payload = ''.join
    (random.choice(string.lowercase) for x in range(
    random_value_less_than_length - 1)) + random_rule_string[
    random_value_less_than_length:length_of_payload]
49
50    # Dataset F
51
52    shuffle(rules)
53
54    start_non_random = random.randint(0, max_payload_size)
55    random_rule_string = ''.join(rules)
56    length_of_rule_string = len(random_rule_string)
57
58    divisor = max_payload_size / length_of_rule_string
59
60    random_rule_string = random_rule_string*(divisor + 1)
61
62    payload = ''.join(random.choice(string.lowercase) for x in
    range(start_non_random - 1)) + random_rule_string[
```

```
    start_non_random:max_payload_size]
63
64      packet = a = IP()/UDP()/DNS()
65      packet.payload.payload.payload = payload
66
67      packets.append(packet)
68
69  wrpcap("dataset.pcap", packets)
```

Listing B.1: Example code for editing and creating PCAP files with Python and Scapy