A Comparison of String Search Algorithms for Deep Packet Inspection

Kieran Hunt

November 12, 2015

Abstract

Contemporary deep packet inspection systems often rely on custom hardware or entrenched ideas about the string search algorithms used. These algorithms have mathematically provable time and space complexities however not much is empirically known about their performance on real-world packet datasets. We felt that some string search algorithms could produce results that differed from their theoretical performance within the context of packet inspection. Furthermore, we sought to show that even algorithms with similar theoretical performances could produce differing practical results. Our approach was to reimplement a variety of the established string search algorithms and run them through a diverse set of tests with both real-world and constructed datasets. Our tests found that the Boyer-Moore and Horspool algorithms were the fastest overall with the Bloom filter being by far the slowest. These findings help to show which algorithms perform best in practice and can help future algorithm designers to improve on the current approaches. In practice we show which algorithms a designer of a deep packet inspection system should consider implementing based on our findings.

Contents

1 Introduction

Deep packet inspection (DPI) forms part of the packet filtering that takes place on many computer networks. Initially, packet inspection was nothing more than classification based on port number. DPI emerged later. For the general routing and very shallow inspection of a packet it is only necessary to look as far as the packet's headers. DPI takes the inspection further by closely examining the packet's payload data; its purpose is to detect data in a packet that is interesting to the network administrator. Generally, data that is interesting to the administrator is viruses, spam, incomplete or corrupted protocols, and other obvious intrusions. Today, newer techniques using statistical analysis are able to better inspect tunnelled or encrypted data (Cascarano et al., 2011).

Deep packet inspection is the most used form of packet inspection and classification today (Cascarano et al., 2011). From Internet Service Providers (ISPs) to Corporate Networks. The Deep Packet Inspection usually forms part of a greater Intrusion Detection System (IDS) or firewall. These IDSs have strict requirements surrounding the accuracy of DPI; misclassification of a malicious packet could help to compromise the entire network (Cascarano et al., 2011).

Today, many approaches exist to perform deep packet inspection. These approaches can be broadly split into two categories: hardware and software. A hardware implementation generally requires the use of expensive and custom systems usually either Ternary Content Addressable Memory (TCAM), Field Programmable Gate Arrays (FPGAs) or Multi-core processors (AbuHmed et al., 2007). Hardware is generally faster than software (citation here) but has slower compile times and does not scale as well. Software, like Bro[1] and Snort[2], is generally run on commercial off-the-shelf (COTS) machines or on virtualised infrastructure. Owing to the fact that it is not tied to the underlying architecture, software implementations are able to easily scale both vertically (speed of the machines) and horizontally (number of machines) to match the needs of the network.

Deep packet inspection faces multiple challenges which affect how well it can perform. A few of these challenges include:

- **Algorithm time and space complexity**. Different packet inspection algorithms each perform their inspection at different speeds and use different amounts of memory. These are classified by using the "Big-O" notation and more can be found in Subsection 2.9.
- **The number of intrusion signatures**. In AbuHmed et al. (2007), the authors point out that the number of signatures affects the inspection time and space usage. For most algorithms, more signatures generally means that the inspection will take longer and consume more memory. Furthermore, as more and more malicious software is released, the number of signatures needed to detect everything increases. Snort currently has 3331 community rules.
- **Detecting malicious content in encrypted data**. By very definition encrypted content cannot be viewed and as such deep packet inspection is ineffectual (AbuHmed et al., 2007). Systems in which encryption is terminated before the deep packet inspection takes place are available however they do require a level of trust between the end user and the party conducting the DPI that is only seen in a corporate or government environment.

Deep packet inspection is traditionally done within an IDS and as such needs to be fast enough that it can at least match the speed of the network. Failing to at least match the network speed will result in a slowdown of traffic flowing into the network, the queuing of yet to be inspected packets and, if the situation does not improve in time, the eventual dropping of packets as the queue fills. This result is certainly undesired as it negatively impacts the quality of service (QOS) experienced by users on the network.

Some deep packet inspection systems join packets together to recreate a session. Sessions can often span multiple packets because of the way the protocol was designed or if the data being transferred exceeds the maximum transmission unit (MTU) of the protocol and so the the data must be split. The DPI systems

[1] `https://www.bro.org/`
[2] `https://www.snort.org/`

identify sessions by looking for the fingerprint of the application-layer protocol, once the session has been identified it is often stored in the form: Source and destination IP address, the identified protocol, and the source and destination ports (Cascarano et al., 2011). This process is known as TCP/IP normalisation. Data encryption makes session identification very difficult as it is almost impossible to identify what the application layer protocol being used is.

Deep packet searching in general is very similar to searching through text. Packets contain a payload which is analogous to a body of text. Packet inspection differs from text searching in that searches through text are generally for a single or only a few terms at a time whilst in packet inspection scenarios the search terms can number in the thousands. The data in the packet itself can often be binary in nature too.

2 Algorithms

Deep packet inspection algorithms need to have the following traits in order to be considered for any commercial IDS (AbuHmed et al., 2007):

- **Deterministic performance**. The algorithm needs to have a known performance for a given set of signatures and inputs. Without a known performance the algorithm has the chance to slow down the entire packet inspection operation.
- **Memory efficiency**. An algorithm needs to be memory efficient. Like with time, the algorithm must occupy a reasonable memory footprint. Deep packet inspection machines have a limited amount of memory and algorithms should not exceed that.
- **Dynamic update**. Algorithms should be updatable on-the-fly. If a new vulnerability is discovered, the algorithms should be updatable without having to completely stop the entire inspection operation.
- **Scalability**. Algorithms should not only be able to scale vertically (by increasing the speed and memory of the machine) but also horizontally by adding additional cores and even entirely separate inspection machines.

A technique often suggested as a way to speed up the packet inspection process is that of imposing a hard limit on how far into a payload the inspection takes place. This imposes limitations on the use of Kleene closures in the inspection.

The following subsections give some information on the different algorithms used.

2.1 Naïve

In order to objectively compare algorithms we implemented a control algorithm and so the first (AbuHmed et al., 2007) search algorithm was chosen. Often called the Brute Force matching algorithm we've named it Naïve, this[3] algorithm has a $O(n^2)$ time efficiency. The algorithm simply tests every possible substring of the text against the search term. See Figure 1 for an example.

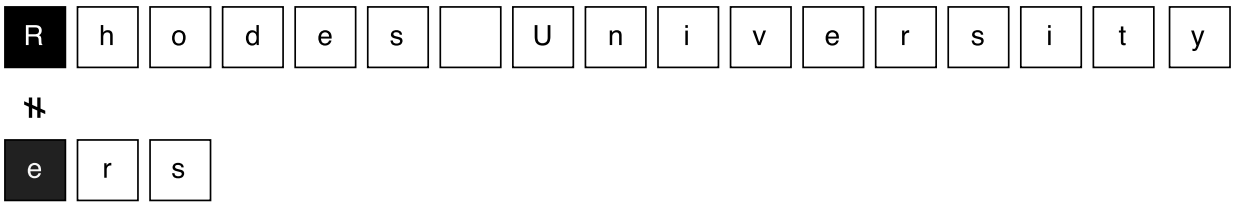
2.2 Aho-Corasick

This string search algorithm, originally proposed by Aho and Corasick (1975), matches all search terms simultaneously.

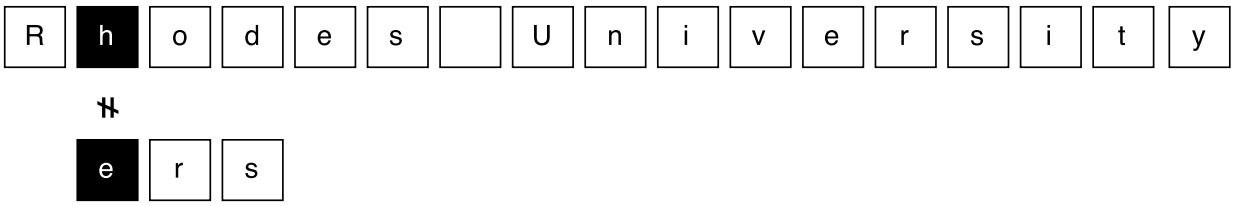
The Aho-Corasick algorithm was designed as an improvement on the trie (or keyword tree). A trie works by constructing a tree where each edge is labeled by a character and each node is the concatenation of edges leading up to the node. Nodes are then labeled with the index of the corresponding search term. For the search terms: he, she, his, hers ($P = \{he, she, his, hers\}$); a trie can be constructed (See **??**).

A lookup through a trie done by starting at the root, follow a path by matching the packet data against the edge labels for as long as possible. If the path leads to a node that is labeled: the string is a search term. If the path terminates at a node that does not have a label then no match is made. Lookups have $O(nm)$ time complexity.

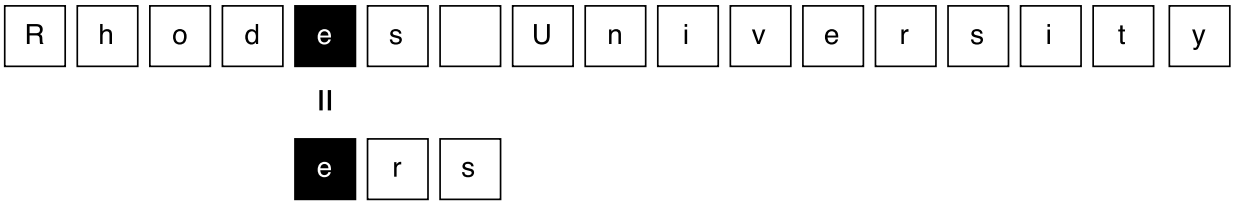
[3] `https://github.com/KieranHunt/dpi_algorithms/blob/master/src/dpi_interface/naive.rs`



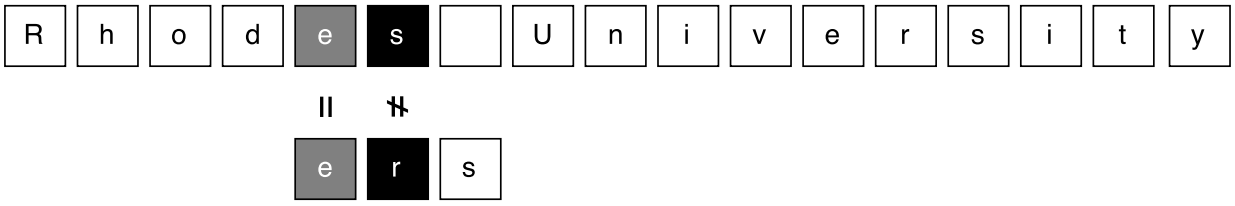
(a) The Naïve algorithm compares the first letter of the search term to the first letter of the text.



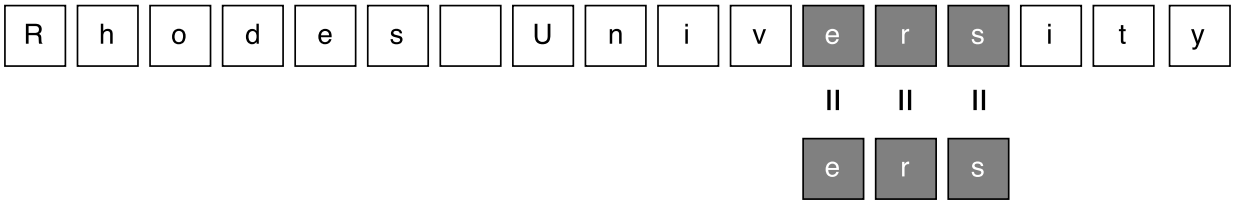
(b) Because the first letter of the search term did not match the first letter of the text, the Naïve algorithm now compares the second letter of the text to the first letter of the search term.



(c) After some iteration, the first letter of the search term is matched against the fifth letter of the text.



(d) The next letter of the text does not match the second letter of the search term. A match is not declared.



(e) As before letters from the search term match letters in the text. In this case the entirety of the search term was matched in the text and so a match is declared.

Figure 1: The process by which the Naïve algorithm conducts its search

As stated, the Aho-Corasick algorithm is an extension of the trie. The algorithm extends the trie into an automaton. The following functions are defined to determine the subsequent action given the current node and character being inspected: goto, failure and output.

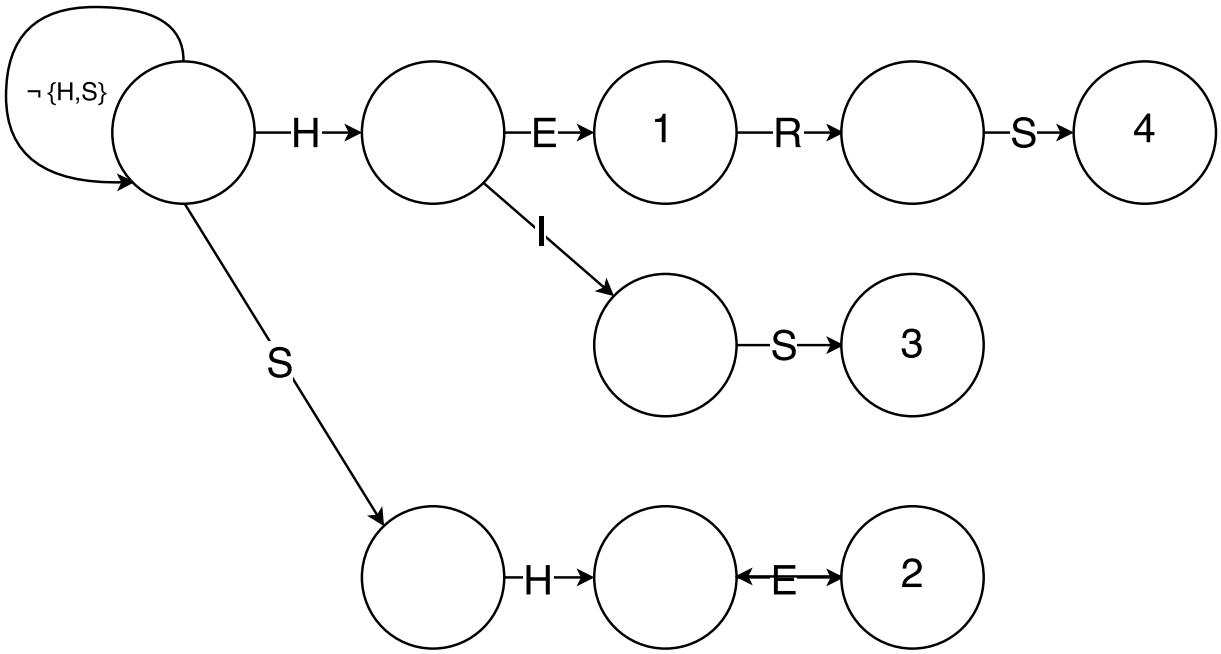


Figure 2: Trie (keyword tree) for the search terms $P = \{he, she, his, hers\}$

2.2.1 The goto function

Defined as $g(q, a)$, the goto function gives the next state by taking the current state (or node) q and a matching character a as parameters. If an edge (q, v) has the label a then the goto function is defined as $g(q, a) = v$. $g(0, a) = 0$ for any a that does not have an edge labeled out of the root node. For anything else $g(q, a) = \emptyset$. The goto function is represented as solid arrows in Figure 3.

2.2.2 The failure function

The failure function, or $f(q)$, defines the next state if no suitable state from the goto function has been found. In other words, when a mismatch occurs. The failure function finds the longest suffix of the label at q which is a prefix of a search term. The failure function is represented as dotted arrows in Figure 3.

2.2.3 The output function

The output function, $out(q)$ defines all search terms that have been recognised when entering q. The output function is represented as the text within the curly braces in Figure 3.

The algorithm has a proven linear time complexity proportional to the sum of the length of the packet, the search terms and the number of times a term is matched (Aho and Corasick, 1975). Formally it has $O(n + m + z)$ where n is the length of the packet payload, m is the length of the search terms and z is the number of times the pattern is matched.

With the small initial overhead encountered when constructing the automaton (computing the goto [see 2.2.1] , failure [see 2.2.2] and output [see 2.2.3] functions), the Aho-Corasick algorithm can shave off precious time by eliminating unnecessary work using the failure functions.

2.3 Bitap

Best known for its use in agrep (Wu and Udi, 1992)[4], the Bitap algorithm published by Baeza-Yates and Gonnet (1992) is an approximate string matching algorithm. That is to say that for a given search term

[4] `https://github.com/Wikinaut/agrep/blob/master/bitap.c`

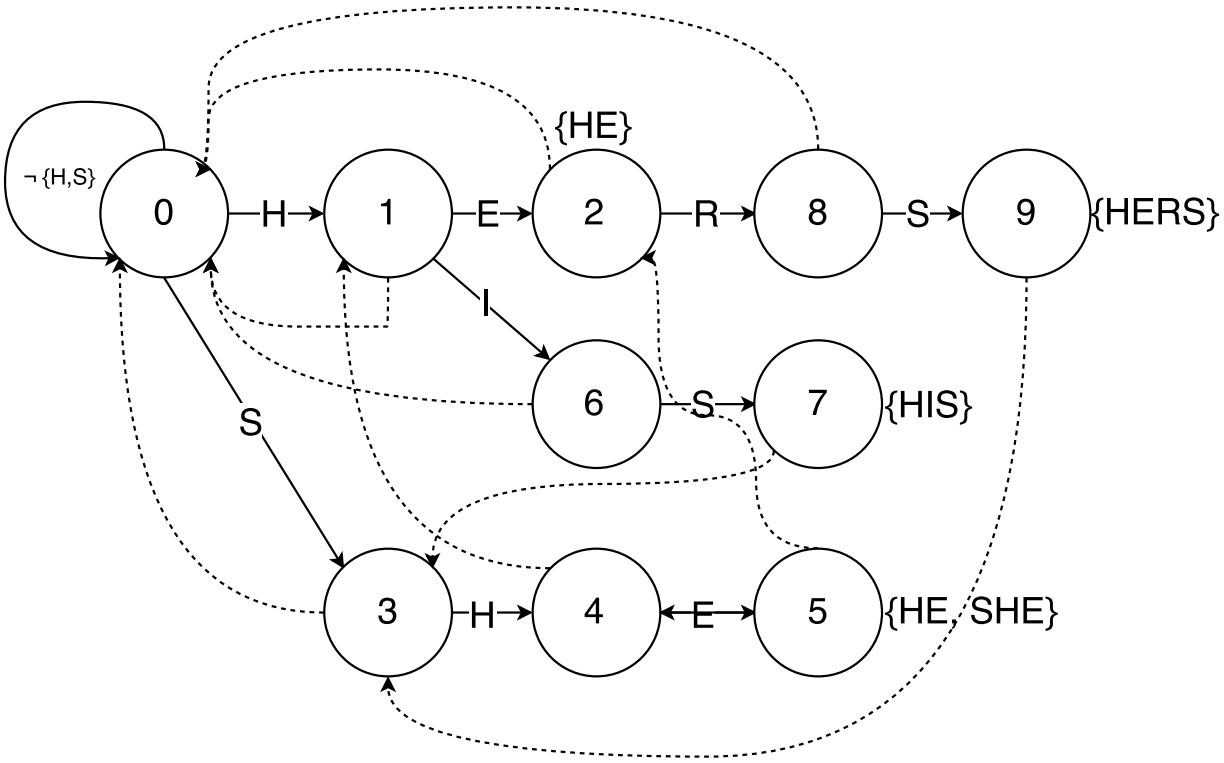


Figure 3: Aho-Corasick Automaton for $P = \{he, she, his, hers\}$

and substring of text, the Bitap algorithm can tell you if those elements are approximately equal. The algorithm's speed is due to its use of bitwise operations on precomputed bitmasks .

2.4 Bloom

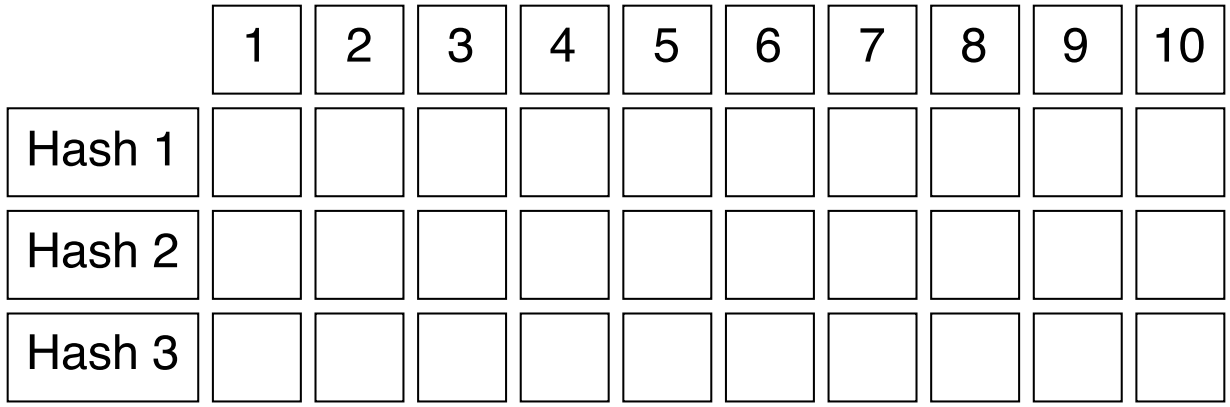
A Bloom filter, first proposed by Bloom (1970), is a data-structure used to determine whether something is a member of a set. Traditionally Bloom filters and other hash table-based algorithms have been used in networking applications such as routing, shallow packet inspection and network monitoring (Song et al., 2005). In the context of packet inspection a Bloom filter is used to see if a substring of the packet payload is a member of the set containing all search terms.

A bloom filter works by hashing each of the search terms a number of times, for each hash a bit is set in a corresponding bit vector. Lookup in a bloom filter involves hashing the string that you want to look up each time for each bit vector and checking to see if the bits are set. The price for efficient such efficient lookup is that a bloom filter can only tell you if a string in definitely not in the set. It can only say with a non-100 percent certainty if the string matches a search term.

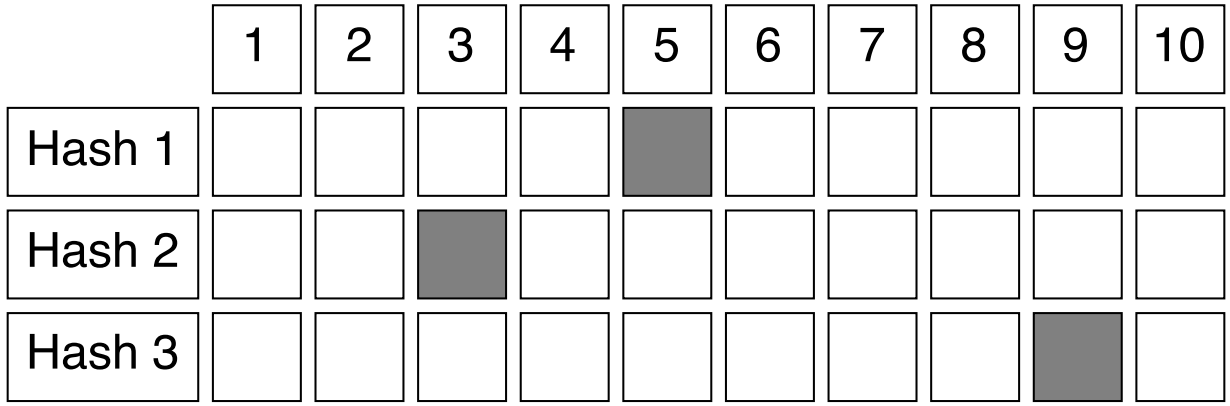
2.5 Boyer-Moore

The Boyer-Moore algorithm (Boyer and Moore, 1977) is an improved version of the Naïve algorithm. It too uses a sliding window to compare the search string with substrings in the search text. Boyer-Moore's improvement on the Naïve algorithm involves a pre-processing step whereby the algorithm determines certain ways that it can jump over text.

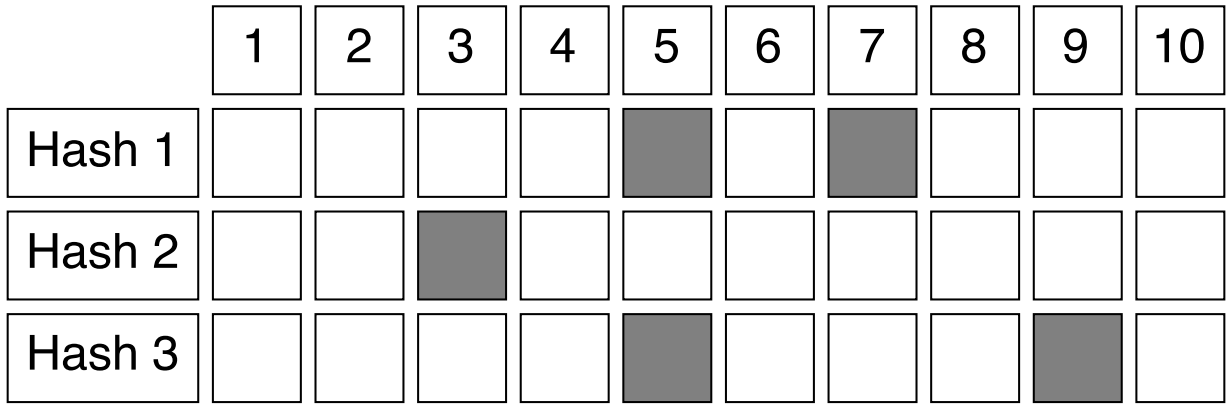
In the Naïve algorithm, if during the matching of a word it fails halfway through, the algorithm simply starts at the next character. The Boyer-Moore uses its pre-processing step to determine if and how far along in the text the algorithm can jump. Thus making fewer comparisons and increasing the overall speed.



(a) The bloom filter initially starts with hash tables initialised to zero, one for each of the hash functions being used.

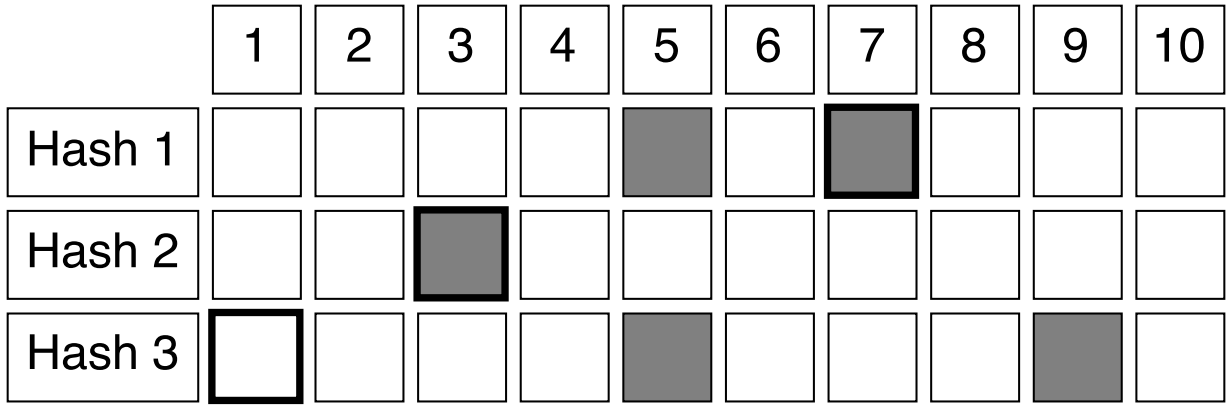


(b) The first search term is hashed by each of the hashing algorithms, for each algorithm a single value is returned. The place in the hash table corresponding to the value is set. Here 5 is set for Hash 1, 3 for Hash 2 and 9 for Hash 10.

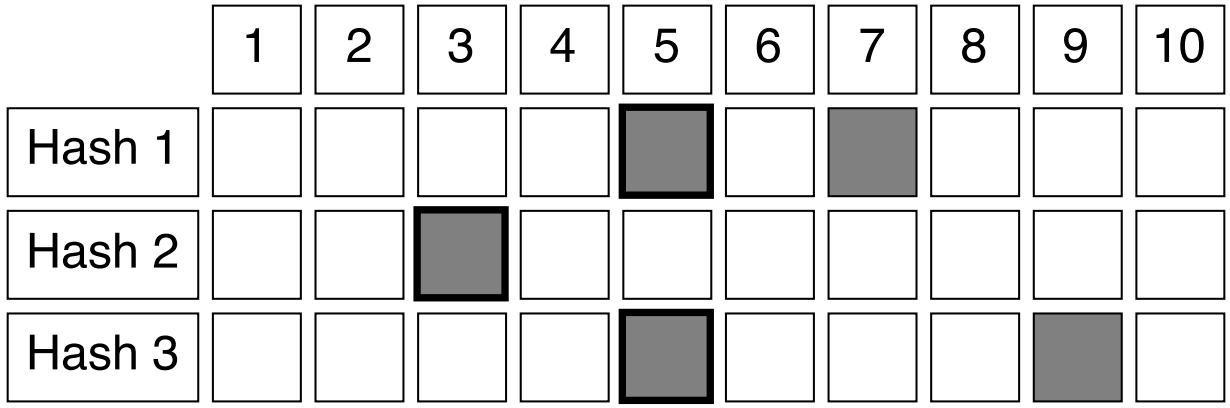


(c) A second search term is put through the three hashing functions. For this term Hash 1 gives the value of 7, Hash 2 gives 3 and Hash 3 gives 5. Note that Hash 2 has given the same values for both search terms.

The Boyer-Moore algorithm works by comparing text from the rear to the front of the search pattern. If the character in the text does not match any character in the search pattern then the pattern can jump the length of the pattern forward. If the character is found somewhere within the search string then the pattern



(d) When doing a lookup in a Bloom table, a similar approach is used to insertion. The term being looked up is hashed by the various hashing algorithms and the hash tables are check to see if the corresponding bits are set. Here the first two hash tables show that the bits are set but the last hash table does not find a match. No match is declared.



(e) Another being of text is being evaluated. Here the text is put through the same three hashing algorithms and for each one a corresponding entry in the hash table is found. The Bloom filter declares this a likely match. The careful reader may notice that this is, in fact, not a match. Figures **??** and **??** show each term being added to the Bloom filter and the results of this search do not match the insertions.

is shifted to where the characters line up and the process is then repeated.

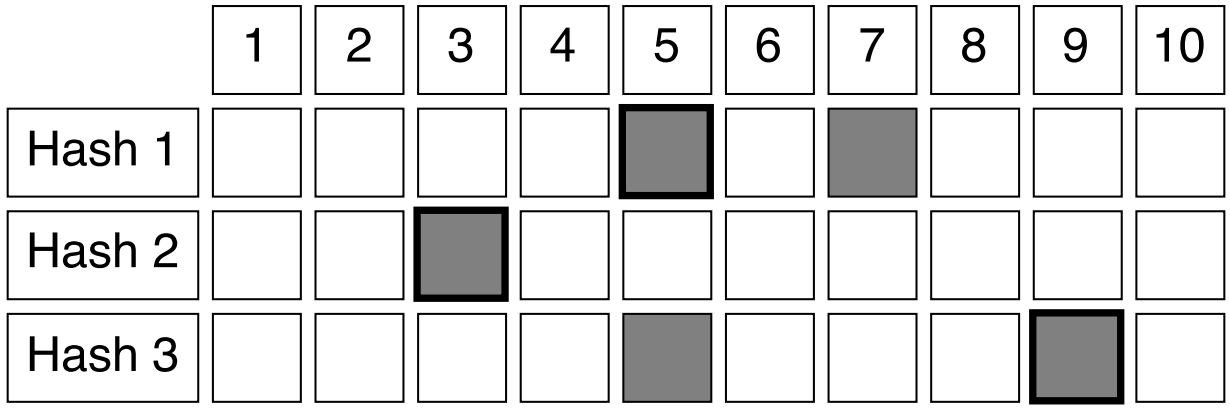
2.6 Horspool

The Horspool algorithm (Horspool, 1980) (also known as the Boyer-Moore-Horspool algorithm) is a string search algorithm which is extremely fast. It has a average time complexity of O(n) and a worst case complexity of O(mn) (Horspool, 1980).

The average time complexity is met when

Like the Boyer-Moore algorithm (Boyer and Moore, 1977) (See **??**), the Horspool algorithm has a pre-process step in which a table is created representing each symbol of the alphabet and where the algorithm can skip to.

The algorithm encounters its worst case time complexity behaviour when the search term very closely matches the many substrings in the text being searched through.



(f) Another piece of text is put through the bloom filter with matches again appearing. These matches are not false positives as they correspond to the initially placed bit during the insertion phase.

Figure 4: The process of inserting into and doing a lookup on a Bloom filter.

2.7 Rabin-Karp

The Rabin-Karp algorithm (Karp and Rabin, 1987) is very similar to the Naïve algorithm (See 2.1). It too compares substrings of the text to the search terms. The Rabin-Karp algorithm does, however, use hashing to faster compare search terms to the text. Rabin-Karp is well suited to searching for many search terms at the same time by simply taking an initial hash of the search terms and then doing multiple comparisons against the hashes of the substring of the text.

2.8 Knuth-Morris-Pratt

The Knuth-Morris-Pratt (KMP) algorithm Knuth et al. (1977) is too an extension the the Naïve algorithm (See 2.1) but improves on the performance by skipping characters (AbuHmed et al., 2007). The KMP algorithm starts by comparing the each character of both the search string and the text being searched through. If a match is found the second character of both the search string and text are compared. Each successive character is compared. If the end of the search term is reached then a match is declared. The power of the KMP algorithm is apparent when only part of the search term is matched. By keeping a record of the previous comparisons, if the algorithm has noted that the start of the pattern has occurred elsewhere then a jump to that place can be made without the interim comparisons.

2.9 Summary

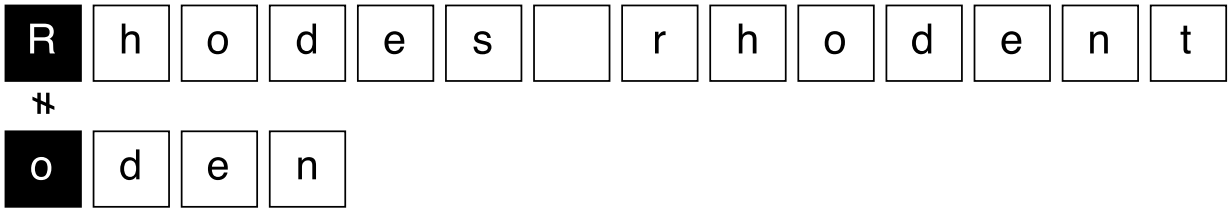
	Aho-Corasick	Bitap	Bloom	Boyer-Moore	Horspool	Knuth-Morris-Pratt	Naïve	Rabin-Karp
Time	O($m+n+o$)	O(mn)	O(n^2)	O($m+n$)	O(n)	O(k)	O(n^2)	O($n+m$)
Space	O(m)	O(m)	O()	O(m+?)	O(*alphabet*)	O(m)	N/A	O(p)

Table 1: Average algorithm time and space complexity

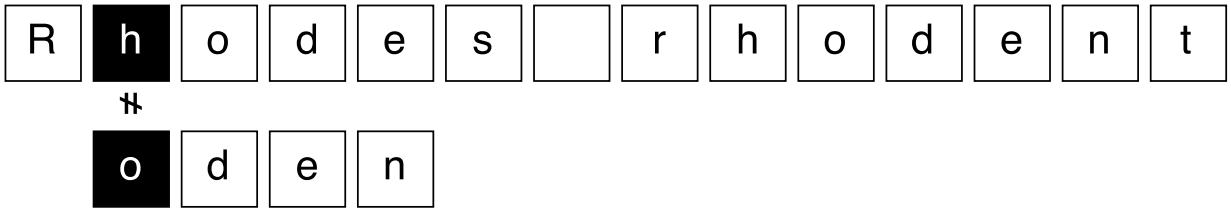
3 Testing Environment

The Rust language was used to implement all of the search algorithms. Rust was selected because it is fast and safe. Both qualities are important in a real-time packet inspection scenarios. For interacting with both packet captures and live capture handles we used the Rust pcap[5] library. This library provides a wrapper

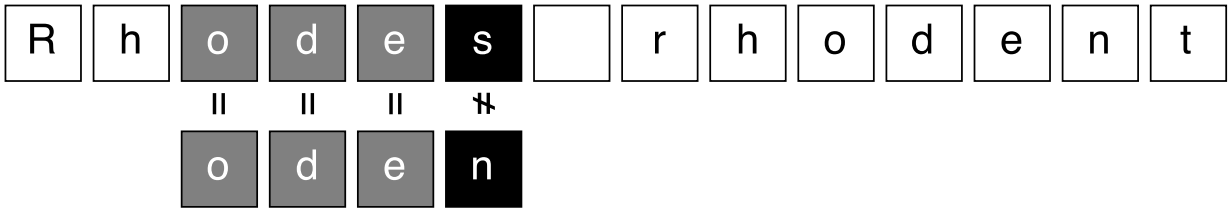
[5] `https://github.com/ebfull/pcap`



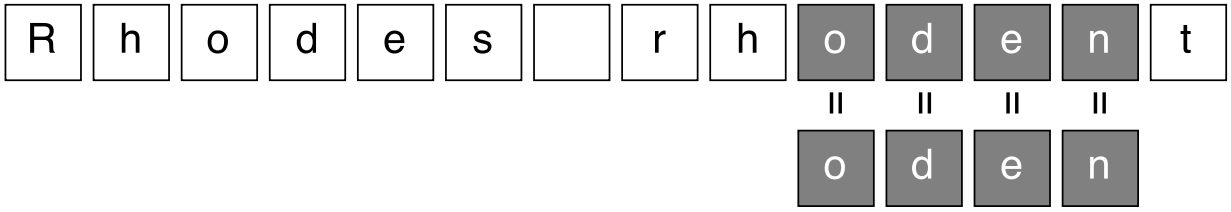
(a) The KMP algorithm compares the first letter of the search string with the first letter of the text. No match is found.



(b) The algorithm moved on to the second letter of the text and compares it with the first letter of the search term. No match is found.



(c) Now the KMP algorithm compares the first letter of the search string with the third letter of the text. A match is found. The fourth and fifth letters of the text are compared with the second and third of the search term and they too match. Finally the the fourth letter of the search term is compared with the



(d)

Figure 5: The KMP algorithm performing a search for the term “oden” in the string “Rhodes rhodent”

around libpcap[6]. Through pcap, we are able to easily retrieve the packet data which can in turn be run through the various algorithms (See part 2).

Each algorithm was implemented in Rust. All but the Bloom[7] Bloom (1970) and Aho-Corasick[8] Aho and Corasick (1975) filters were were implemented by the authors.

(Talk about the pcap format.)

The data used in the tests was real-world packet data collected the DNS servers of local schools. The libpcap library, in conjunction with tcpdump was used to collect and save the packet information. The total number of captured packets was culled to a reasonable one million packets to allow for reasonable processing and interpretation times.

[6] http://www.tcpdump.org/

[7] https://github.com/nicklan/bloom-rs

[8] https://github.com/BurntSushi/aho-corasick

Each algorithm was separately run over the dataset for a variety of search terms.
Search terms: google, amazonaws.com

Book	Short Name	Size	Number of Characters
Alices Adventures in Wonderland	AAW	269K	167516
Bird Watching	BW	653K	609763
Frankenstein or the Modern Prometheus	FMP	525K	448687
Pride and Prejudice	PP	780K	717573
The Adventures of Sherlock Holmes	ASH	653K	594916
The Adventures of Tom Sawyer Complete	ATSC	525K	421872
The Yellow Wallpaper	YW	58K	52079

Table 2: Books, their shorthand names, size on disk (in bytes) and the number of characters in each.

In dataset A, "google" occurs 15223 times whilst "amazonaws.com" occurs 6389 times.

	"amazonaws.com"		"google"	
Index	Prefix	Suffix	Prefix	Suffix
1.	a	m	g	e
2.	am	om	go	le
3.	ama	com	goo	gle
4.	amaz	.com	goog	ogle
5.	amazo	s.com	googl	oogle
6.	amazon	ws.com	google	google
7.	amazona	aws.com		
8.	amazonaw	naws.com		
9.	amazonaws	onaws.com		
10.	amazonaws.	zonaws.com		
11.	amazonaws.c	azonaws.com		
12.	amazonaws.co	mazonaws.com		
13.	amazonaws.com	amazonaws.com		

Table 3: Search term prefix and suffix index reference

	"amazonaws.com"		"google"	
Index	Prefix Count	Suffix Count	Prefix Count	Suffix Count
1.	2783497	812339	465303	1061226
2.	101726	290968	19007	92113
3.	55808	266672	15456	15432
4.	9111	266645	15231	15223
5.	9094	25810	15223	15223
6.	9094	6980	15223	15223
7.	7285	6389		
8.	7285	6389		
9.	7285	6389		
10.	6389	6389		
11.	6389	6389		
12.	6389	6389		
13.	6389	6389		

Table 4: Search term prefix and suffix frequency for the packet dataset: Dataset A.

"amazonaws.com"	"google"

Index	Prefix Count	Suffix Count	Prefix Count	Suffix Count
	AAW			
1.	9081	2245	2750	15082
2.	189	334	197	790
3.	5	84	28	9
4.	0	79	0	0
5.	0	14	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		
	BW			
1.	34859	10713	10695	57776
2.	741	1948	266	2983
3.	18	556	50	93
4.	0	516	0	0
5.	0	100	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		
	FMP			
1.	26315	10295	5761	45717
2.	694	1475	172	2239
3.	14	393	35	40
4.	2	364	0	0
5.	0	64	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		
	PP			
1.	42154	13401	10160	70342
2.	1330	2247	527	3617
3.	27	710	199	347
4.	18	655	0	0
5.	0	144	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		

11.	0	0		
12.	0	0		
13.	0	0		
	ASH			
1.	35301	11398	8033	54581
2.	962	2002	484	2872
3.	15	507	152	52
4.	2	459	0	0
5.	0	50	0	0
6.	0	2	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		
	ATSC			
1.	24160	7397	6904	37791
2.	516	1968	601	2008
3.	16	340	108	46
4.	1	318	0	0
5.	0	74	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		
	YW			
1.	2874	883	780	4488
2.	55	157	54	213
3.	5	45	17	5
4.	1	43		0
5.	0	8	0	0
6.	0	0	0	0
7.	0	0		
8.	0	0		
9.	0	0		
10.	0	0		
11.	0	0		
12.	0	0		
13.	0	0		

Table 5: Search string prefix and suffix frequency in the sample books. For reference to the index, see Table 2

Books: Alices-Adventures-in-Wonderland.txt, Pride-and-Prejudice.txt, The-Prince.txt, Bird-Watching.txt, The-Adventures-of-Sherlock-Holmes.txt, The-Yellow-Wallpaper.txt, Frankenstein-or-the-Modern-Prometheus.txt, and The-Adventures-of-Tom-Sawyer-Complete.txt.

4 Results

The following section provides the results obtained from running the myriad of tests designed for the project. A large amount of raw data (nearly 400 million datapoints) has been condensed into what is presented in the following pages.

Initially every algorithm was run over the packet sample dataset, described in section 3. The time that each algorithm took to process each packet was recorded and logged. This is repeat for all algorithms. The entire test is then run twenty times over to smooth over potential anomalies caused by the server.

The first of these tests was for the string: "google". The mean processing time across all packets and the twenty tests is presented in Table 6. The results of which can be found in Table 6.

Bitap	Bloom	Boyer-Moore	Horspool	Knuth-Morris Pratt	Naïve	Rabin-Karp
71.51	48300	5.642	5.804	11.48	15.95	1720.0

Table 6: Mean packet processing times for the various algorithms with the search term: "google". All values are in microseconds (μs).

After searching for the keyword "google", the tests were rerun using the keyword "amazonaws.com". Again, each algorithm inspected every packet in dataset A and those inspections were run twenty times. A total of 180 million data points were gathered. The mean processing times for the algorithms, calculated from those datapoints, with the keyword "amazonaws.com" can be found in Table 7.

Bitap	Bloom	Boyer-Moore	Horspool	Knuth-Morris Pratt	Naïve	Rabin-Karp
48.6	51190	5.226	5.041	12.42	15.91	1701.0

Table 7: Mean packet processing times for the various algorithms with the search term: "amazonaws.com". All values are in microseconds (μs).

Following the packet testing, the test system was altered so that it could more easily test plain text files as opposed to the pcaps. The plain text files were tested in a slightly different way as they were searched through in their entirety, unlike the pcap files which were searched through packet by packet. Each book was inspected by each algorithm and the tests were repeated 1000 times each. First, the word "google" was searched for. The mean packet processing times of which can be found in Table 8.

	AAW	BW	FMP	PP	ASH	ATSC	YW
Bitap	657900	656200	657200	659000	654900	659500	656600
Boyer-Moore	12700	13470	13100	13660	13410	13120	12580
Horspool	19510	20730	20250	20990	20610	20080	19120
Knuth-Morris-Pratt	55150	55970	55540	55930	55730	55940	55150
Naïve	124400	125200	124500	124900	124900	125100	124200
Rabin-Karp	17840000	17840000	17820000	17840000	17860000	17850000	17810000

Table 8: Mean processing time of DPI when inspecting plain text for the term "google"

After completing the search for "google", the same exact tests were run again with the term "amazonaws.com". See Table 9 for mean processing time across the data.

5 Analysis

The results (Section 4) of the testing have proven to be quite interesting. From Tables 7 and 6 we are able to fairly easily see which algorithms are the most performant. By comparing the mean packet processing times against each other we can arrange the algorithms by speed. Table 10 shows that, from the test results, the Horspool and Boyer-Moore (Subsections 2.6 and 2.5) are the fastest performing of the algorithms.

	AAW	BW	FMP	PP	ASH	ATSC	YW
Bitap	1262000	1260000	1258000	1259000	1260000	1260000	1259000
Boyer-Moore	5598	6120	5951	6300	6129	5927	5468
Horspool	8502	9392	9065	9620	9395	9035	8321
Knuth-Morris-Pratt	51980	54180	53420	54610	54160	53260	51450
Naïve	115200	118000	117200	118800	118000	116900	114500
Rabin-Karp	426800	427400	427000	427200	427100	427200	426200

Table 9: Mean processing time of DPI when inspecting plain text for the term "amazonaws.com"

Rank	Algorithm
1	Horspool
	Boyer-Moore
3	Knuth-Morris-Pratt
4	Naïve
5	Bitap
6	Rabin-Karp
7	Bitap

Table 10: Algorithms ranked by their mean packet processing speeds.

5.1 Comparing Search Terms

What is interesting to note is that, on average, the searching for the term "google" is slower than searching for "amazonaws.com" (See Tables 6, 7, 8 and 9). This result is oddly counter-intuitive on the surface as one would expect, for most of the algorithms, that a longer string would take longer to find as a greater number of comparisons need to take place to assure correctness. We have formulated the following hypothesis for why searching for "amazonaws.com" takes less time than searching for "google":

- popular substrings
- term frequency

5.1.1 Popular Substrings

A couple of the algorithms used rely heavily on comparing progressively large slices of the search term to pieces of the text. Some of these work from the last character to the first and these are called suffix matchers. Others work from the first character to the last, these are called prefix matchers.

For prefix and suffix matchers, text with a high frequency of substrings matching substrings within the search terms means that much processing time is wasted on evaluating string similar but ultimately often not the same as the original search string. As can be seen from Table 5, substrings of "amazonaws.com" are generally found more frequently than those of "google". In Table 4, google substrings are generally more popular than those of amazon however it becomes apparent that "google" leads to a match more often more early than "amazonaws.com". "google" is guaranteed a match after the fourth character (for both prefix and suffix matching) whilst "amazonaws.com" is only guaranteed a match after the ninth character.

Algorithms which make use of string suffixes include:

- Aho-Corasick (Subsection 2.2)
- Bitap (Subsection 2.3)
- Boyer-Moore (Subsection 2.5)
- Horspool (Subsection 2.6)
- Knuth-Morris-Pratt (Subsection 2.8)

Algorithms which make use of string prefixes include:

- Naïve (Subsection 2.1)
- Rabin-Karp (Subsection 2.7)

	Bitap	Bloom	Boyer-Moore	Horspool	Knuth-Morris Pratt	Naïve	Rabin-Karp
Suffix vs Prefix	Suffix	N/A	Suffix	Suffix	Suffix	Prefix	Prefix
Fastest	A	G	A	A	G	A	A

Table 11: Comparison of mean packet processing time between "google" and "amazonaws.com". Tests in which "google" were faster are marked by a "G" and tests in which "amazonaws.com" finished earlier are marked with a "A".

Figure 6 gives a better visual representation of the data shown in Table 4. As you can see from Table 11, searching for "amazonaws.com" is mostly faster than searching for "google". This is possibly because of the steep drop off seen in the number of matching suffixes after the suffix "s.com".

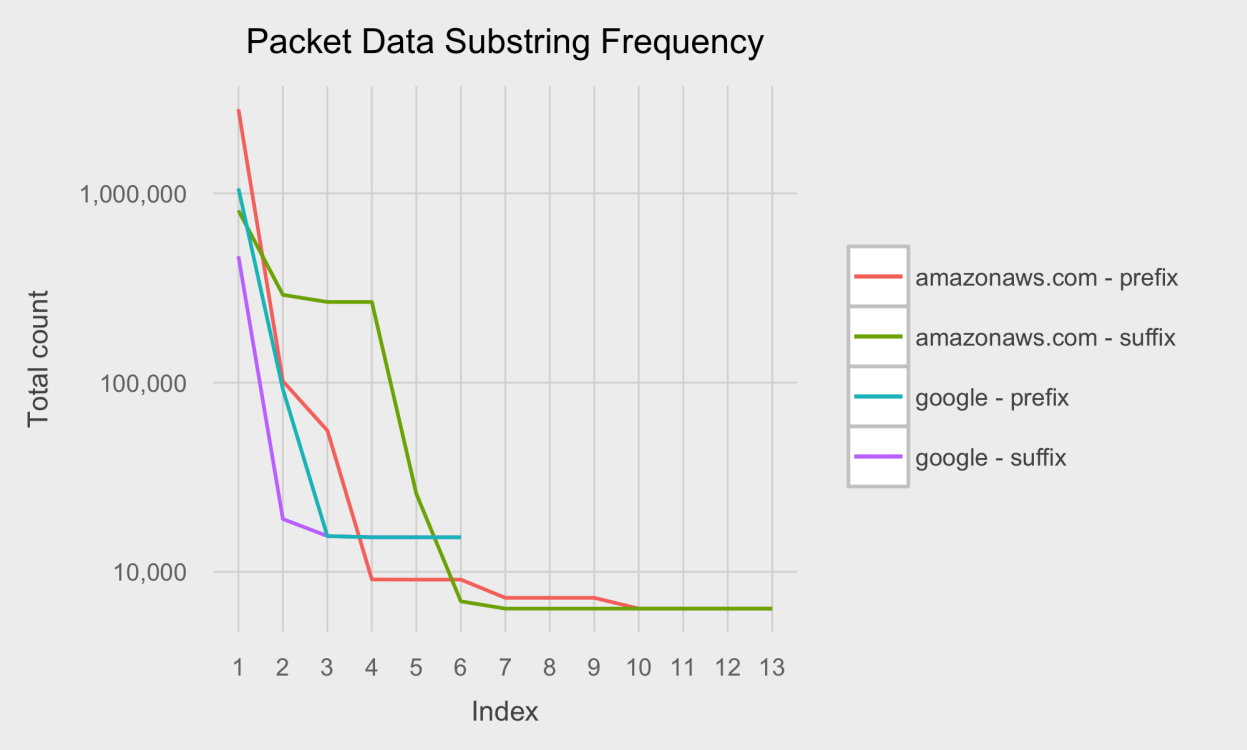


Figure 6: Substring frequency for both prefix and suffix

5.1.2 Term Frequency

The search terms "google" and "amazonaws.com" do not appear at the same frequency in dataset A. "google" is found 15223 times whilst "amazonaws.com" can be found only 6389. We believe that this is the main factor changing the speed of most of the algorithms. Because dataset A contains more occurrences of "google" than "amazonaws.com" some of the algorithms will spend more time matching and less time skipping over text that clearly does not have a match.

By adjusting the mean times according to frequency, we are able to better understand the relationship between the frequency of the term and how it affects the processing speed. Table 12 contains those adjustments.

Algorithms	“google”	“amazonaws.com” - adjusted	Percentage Difference
Bitap	71.51	115.79	61.93%
Bloom	48300	121969.84	152.53%
Boyer-Moore	5.642	12.45	120.70%
Horspool	5.804	12.01	106.95%
Knuth-Morris-Pratt	11.48	29.59	157.78%
Naïve	15.95	37.90	137.67%
Rabin-Karp	1720	4052.95	135.64%

Table 12: Packet processing speeds relative to search term frequency.

Table 12 shows that, by adjusting for search term occurrence, searching for the term “amazonaws.com“ is slower per occurrence of the term in the packet. The percentages shown in the table may correlate with the data from Table 1.

5.2 Comparing Algorithms

Figures 7 and 8 provide insight into the comparative of each algorithm for both the “google” search (Figure 7) and for “amazonaws.com” (Figure 8). These graphs were created by plotting the cumulative sum of the time it takes for each algorithm to processes dataset A. The y-axis for each has been set to a log scale so that all of the lines would fit onto a reasonably sized space.

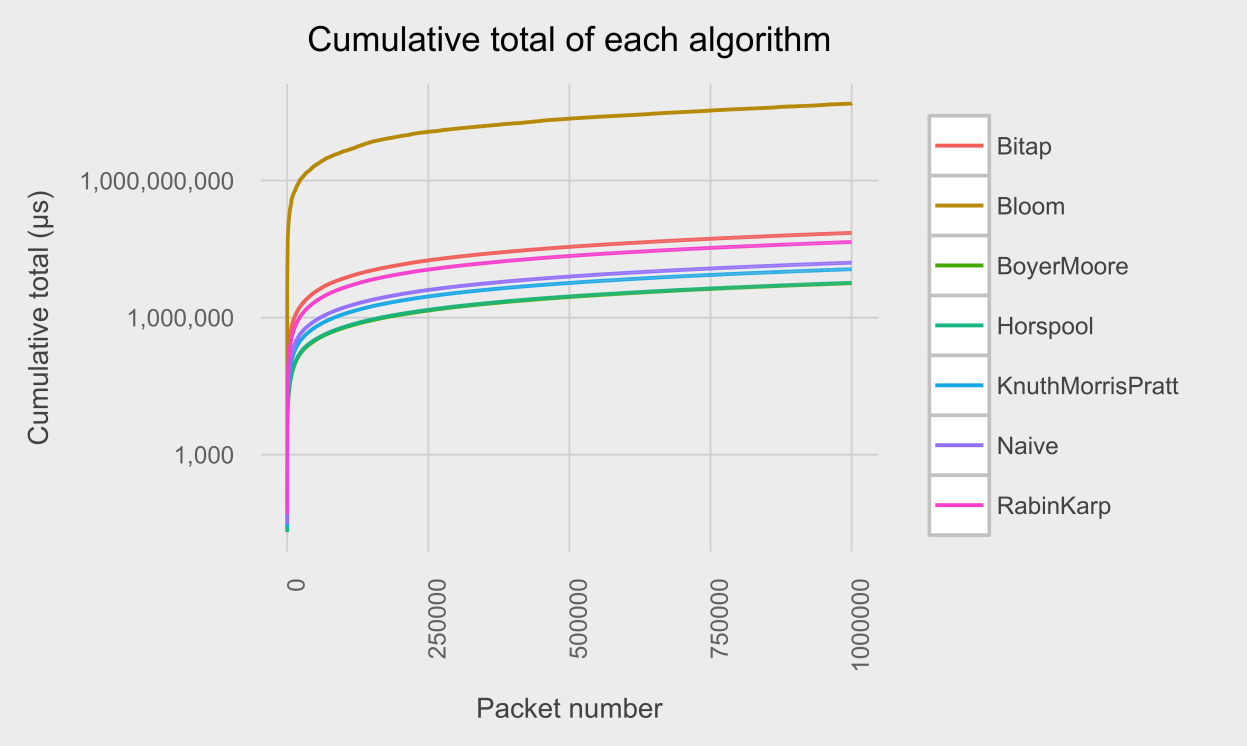


Figure 7: Cumulative sum of processing times of packets in Dataset A for the term “google”

From these Figures we can group the algorithms into three distinct sections based on speed. First and fastest is group one which contains Boyer-Moore, Horspool, Knuth-Morris-Pratt and Naïve. Group two, who are averagely fast, contain Rabin-Karp and Bitap. The third and slowest group contains just Bloom.

The results of the Bloom filter (Bloom, 1970) (Subsection 2.4) show that it is many orders of magnitude slower than any of the other algorithms. This is expected behaviour. Bloom filters were never designed with

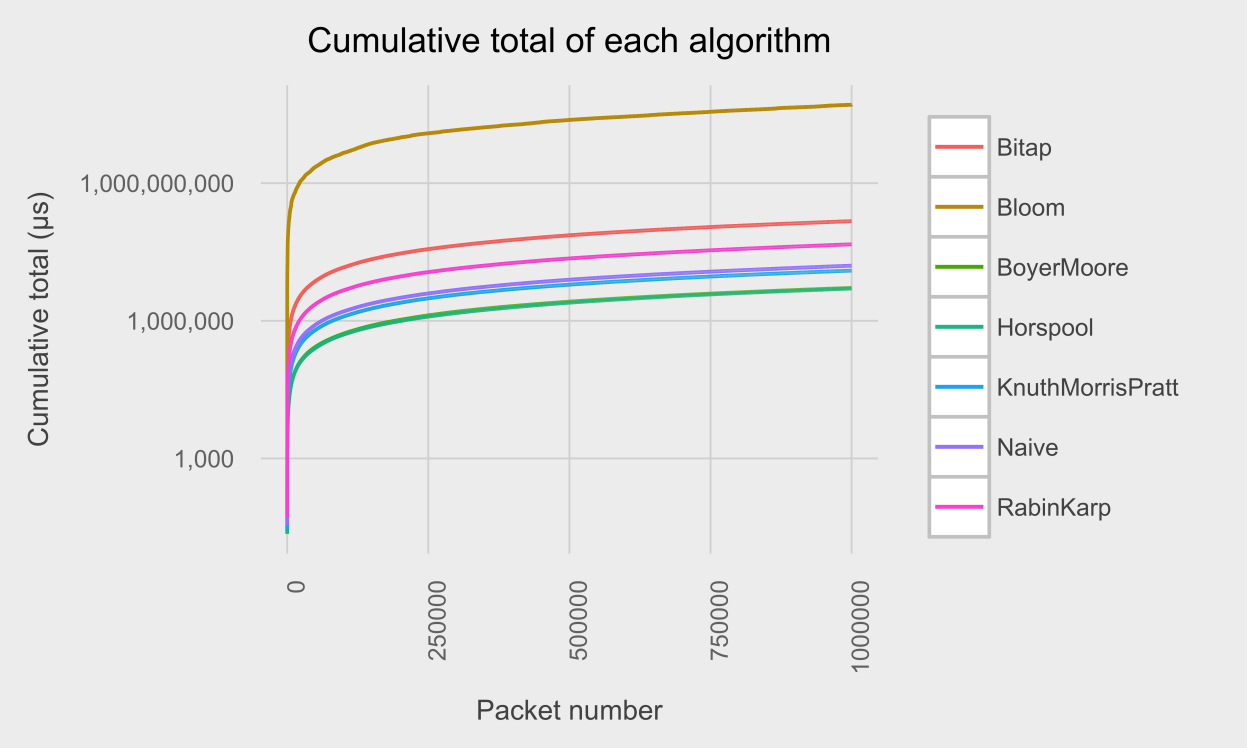


Figure 8: Cumulative sum of processing times of packets in Dataset A for the term "amazonaws.com"

string search in mind. A bloom filter performs best when performing a lookup on a known length string. An example of such would be routing traffic based on IP address. Some trie-based solutions exist but a Bloom filter is used when space efficiency is needed. Since the Bloom filter is performing many different hashing algorithms on the same input to compare it against the hash table generated by the preprocessing step, it is comparable to the Rabin-Karp algorithm (Karp and Rabin, 1987) (Subsection 2.7). The Rabin-Karp algorithm performs much better for two reasons. Firstly, it uses an extremely fast hashing algorithm known as FarmHash[9]. Secondly, it hashes the input string exactly once and compares it to the hash of the string being searched for whereas the Bloom filter hashes the input string many times and then checks for a match in the many hash tables.

Boyer-Moore (Boyer and Moore, 1977) (Subsection 2.5) and Horspool (Horspool, 1980) (Subsection 2.6) are, rather unsurprisingly, the fastests algorithms. Both algorithms average between five and six milliseconds to process a packet (Tables 6 and 7). According to Table 1, both algorithms have similar time complexities. Boyer-Moore having $O(n + m)$ and Horspool having $O(n)$. These algorithms are, unsurprising, related in that the Horspool algorithm is an refinement on the work done by Boyer and Moore (1977).

6 Conclusion

Talk about going forward. Future work.

References

AbuHmed, T., Mohaisen, A., and Nyang, D. (2007). A survey on deep packet inspection for intrusion detection systems. *Korea Communications Society (Information and Communication)*.

[9] `https://github.com/google/farmhash`

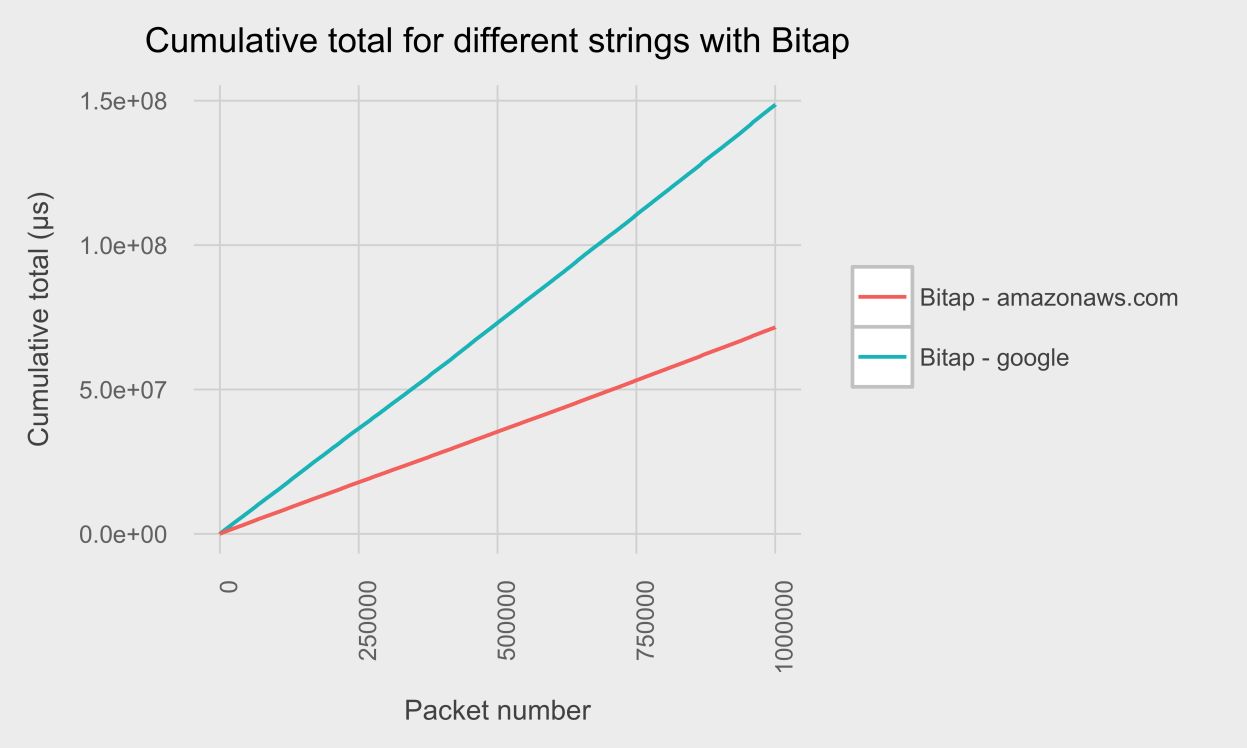


Figure 9: Comparison between the cumulative processing times of the Bitap algorithm for the terms "amazonaws.com" and "google"

Aho, A. and Corasick, M. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*.

Baeza-Yates, R. and Gonnet, G. (1992). A new approach to text searching. *Communications of the ACM*.

Bloom, B. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*.

Boyer, R. and Moore, J. (1977). A fast string searching algorithm. *Communications of the ACM*.

Cascarano, N., Ciminiera, L., and Risso, F. (2011). Optimizing deep packet inspection for high-speed traffic analysis. *Journal of Network and Systems Management*.

Fan, B., Andersen, D., Kaminsky, M., and Mitzenmacher, M. (2014). Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*.

Horspool, N. (1980). Practical fast searching in strings. *Software: Practice and Experience*.

Karp, R. and Rabin, M. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*.

Knuth, D., Morris, J., and Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*.

Song, H., Turner, J., Dharmapurikar, S., and Lockwood, J. (2005). Fast hash table lookup using extended bloom filter: An aid to network processing. *ACM SIGCOMM Computer Communication Review*.

Wu, S. and Udi, M. (1992). Agrep a fast approximate pattern-matching tool. In *Usenix Winter 1992*.

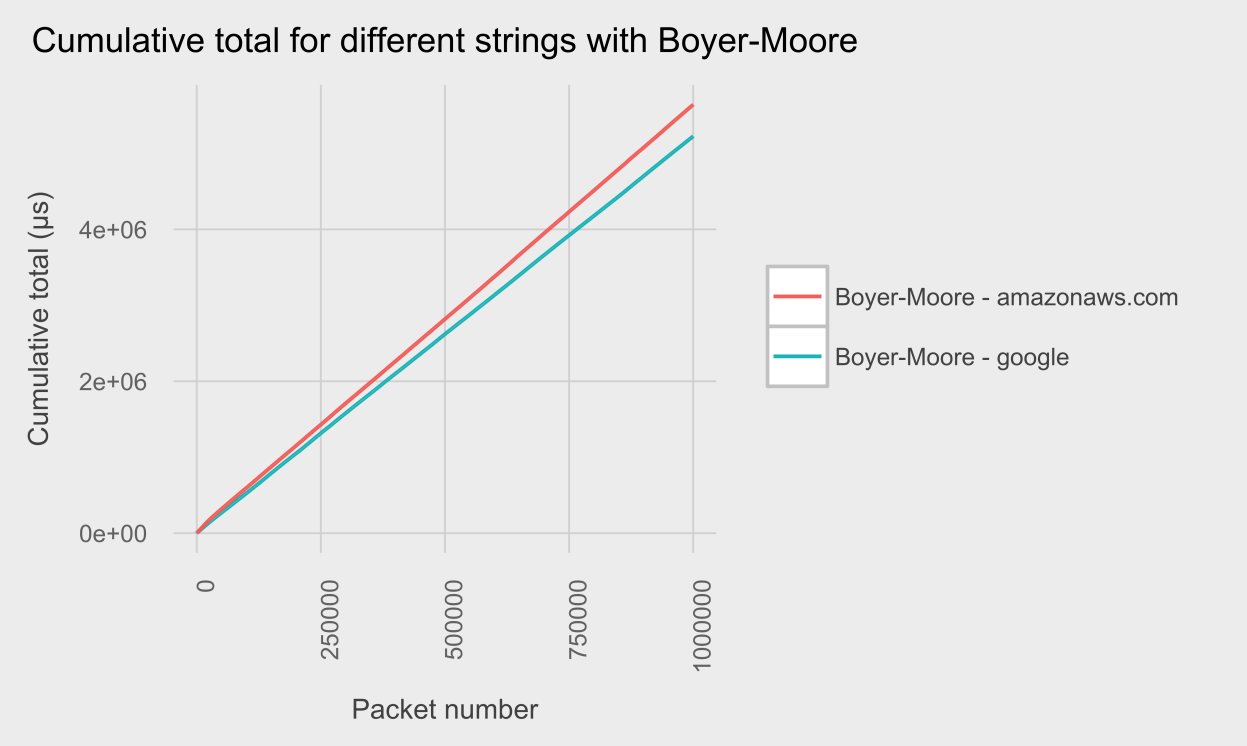


Figure 10: Comparison between the cumulative processing times of the Bitap algorithm for the terms "amazonaws.com" and "google"

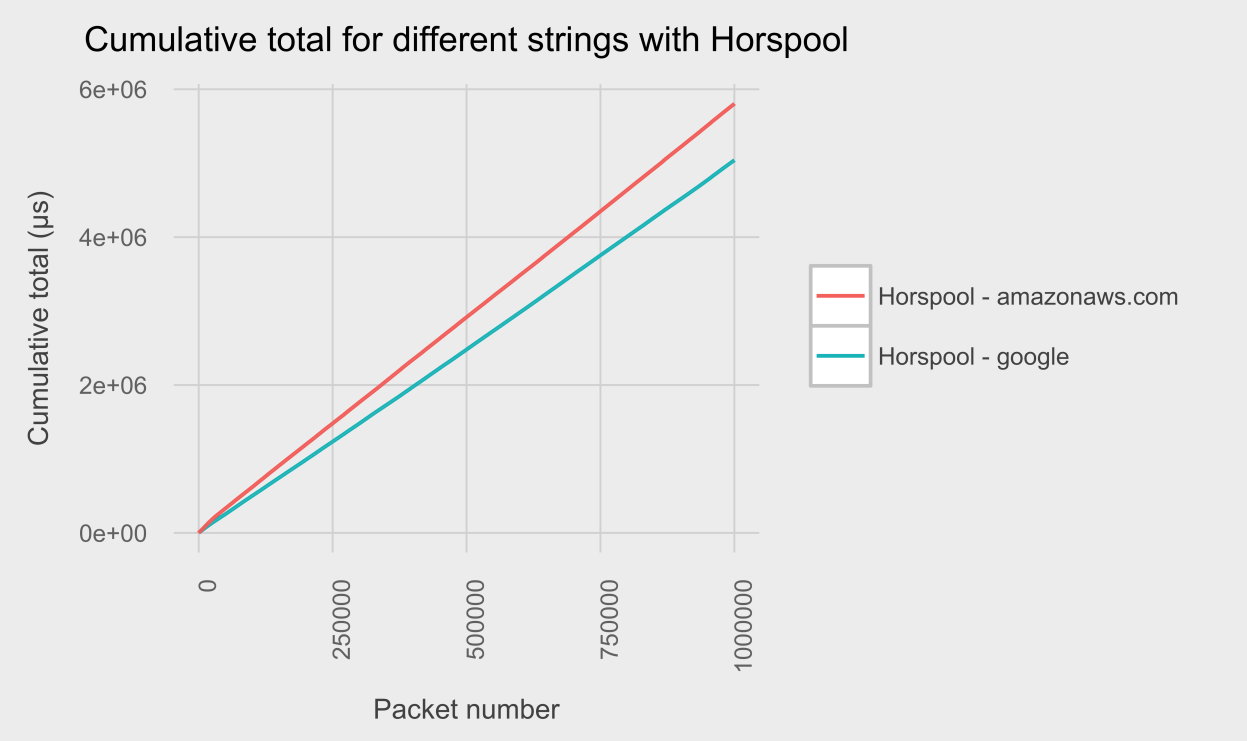


Figure 11: Comparison between the cumulative processing times of the Horspool algorithm for the terms "amazonaws.com" and "google"

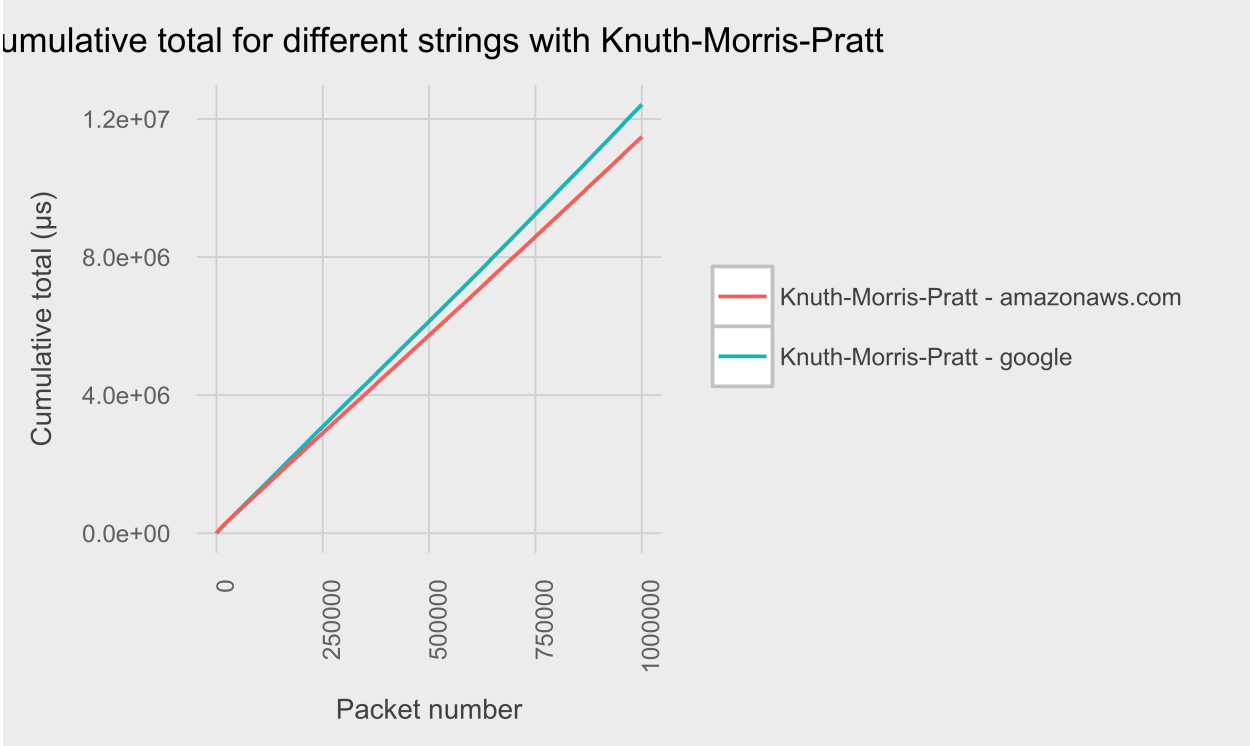


Figure 12: Comparison between the cumulative processing times of the Knuth-Morris-Pratt algorithm for the terms "amazonaws.com" and "google"

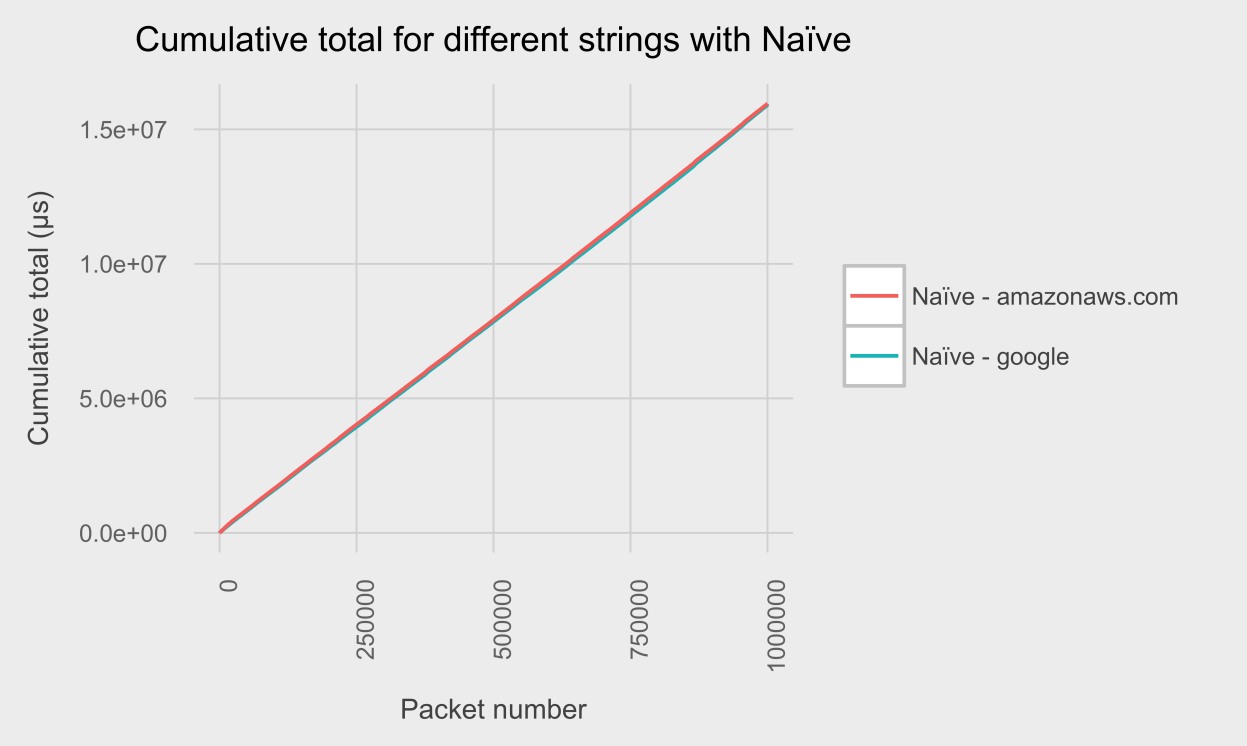


Figure 13: Comparison between the cumulative processing times of the Naïve algorithm for the terms "amazonaws.com" and "google"

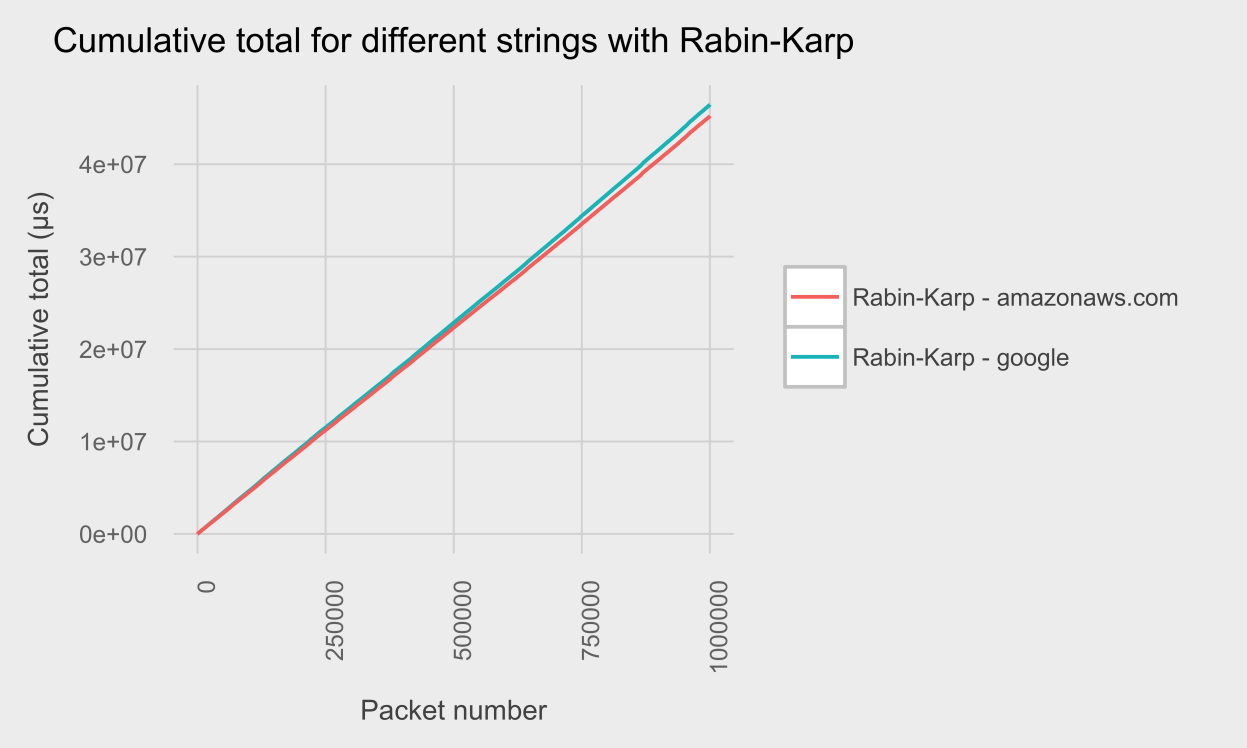


Figure 14: Comparison between the cumulative processing times of the Rabin-Karp algorithm for the terms "amazonaws.com" and "google"