A Comparison of String Search Algorithms for Deep Packet Inspection

Kieran Hunt

October 27, 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 or 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 Bloom filter was the fastest overall, Rabin-Karp was the most memory efficient and that the Nave algorithm was the slowest. Furthermore we found that, although the Bloom and Cuckoo filters have the same theoretical time complexity, the cuckoo filter was almost twice as slow as its counterpart. 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 a computer network. For the general routing 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 obvious intrusions.

Deep packet inspection is very prevalent today. From Internet Service Providers (ISPs) to Corporate Networks. Packets are inspected for a plethora of reasons.

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 such as Field Programmable Gate Arrays (FPGAs). Hardware is generally faster than software (should cite) 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.

Bro and Snort are two platforms designed for network analysis.

Deep packet inspection is traditionally done within an intrusion detection system (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.

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.

2 Algorithms

The following section gives some information on the different algorithms used.

2.1 Naïve

In order to objectively compare algorithms we implemented a control algorithm. Named 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.

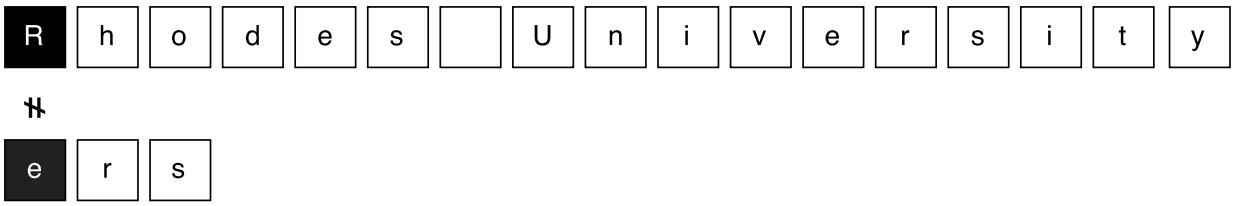
2.2 Aho-Corasick

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

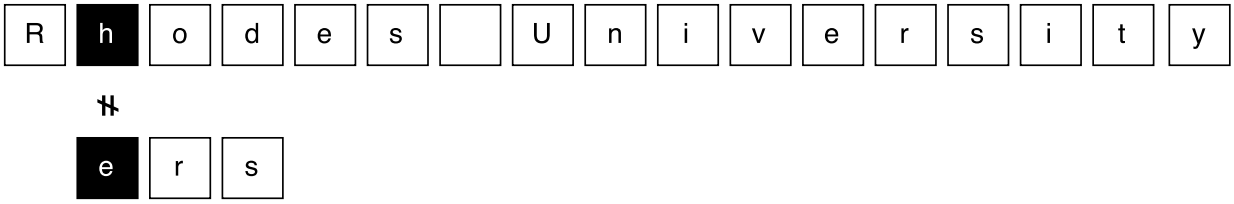
The Aho-Corasick algorithm was designed as an improvement on the trie (or keyword tree). A trie works by constructing a tree where each edge is labeled by a character and each node is the concatenation of edges leading up to the node. Nodes are then labeled with the index of the corresponding search 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.

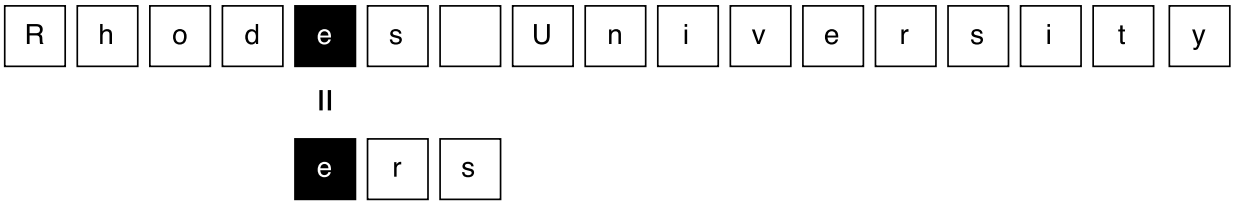
[1] `https://www.bro.org/`
[2] `https://www.snort.org/`
[3] `https://github.com/KieranHunt/dpi_algorithms/blob/master/src/dpi_interface/naive.rs`



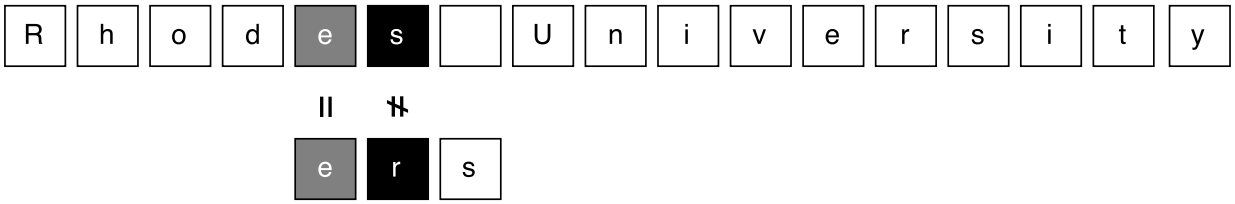
(a) The Naïve algorithm compares the first letter of the search term to the first letter of the text.



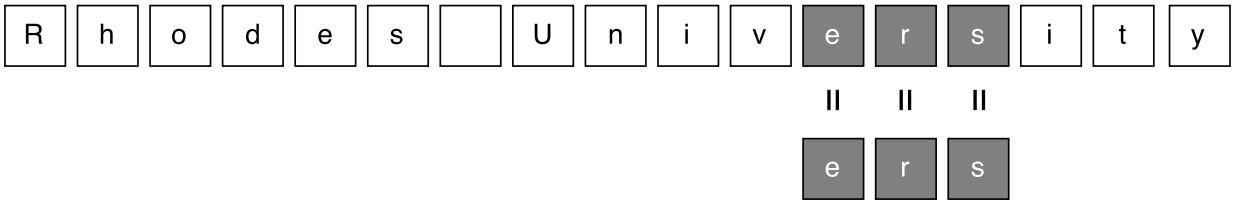
(b) Because the first letter of the search term did not match the first letter of the text, the Naïve algorithm now compares the second letter of the text to the first letter of the search term.



(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.

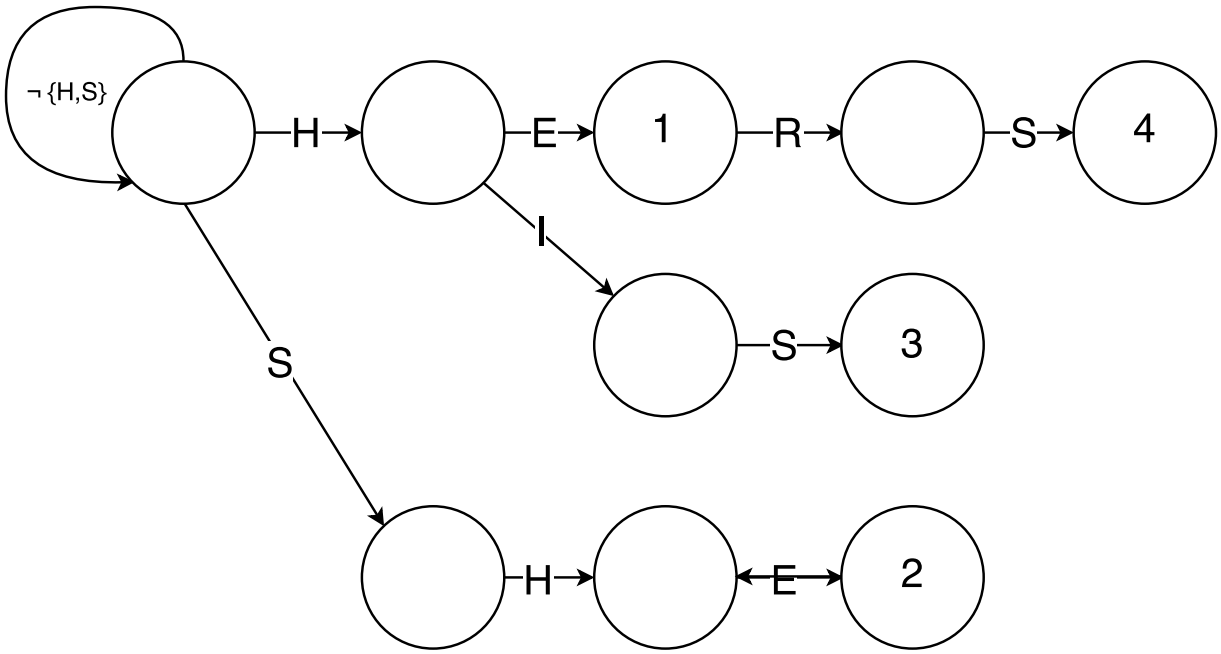


Figure 2: Trie (keyword tree) for the search terms $P = \{he, she, his, hers\}$

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`

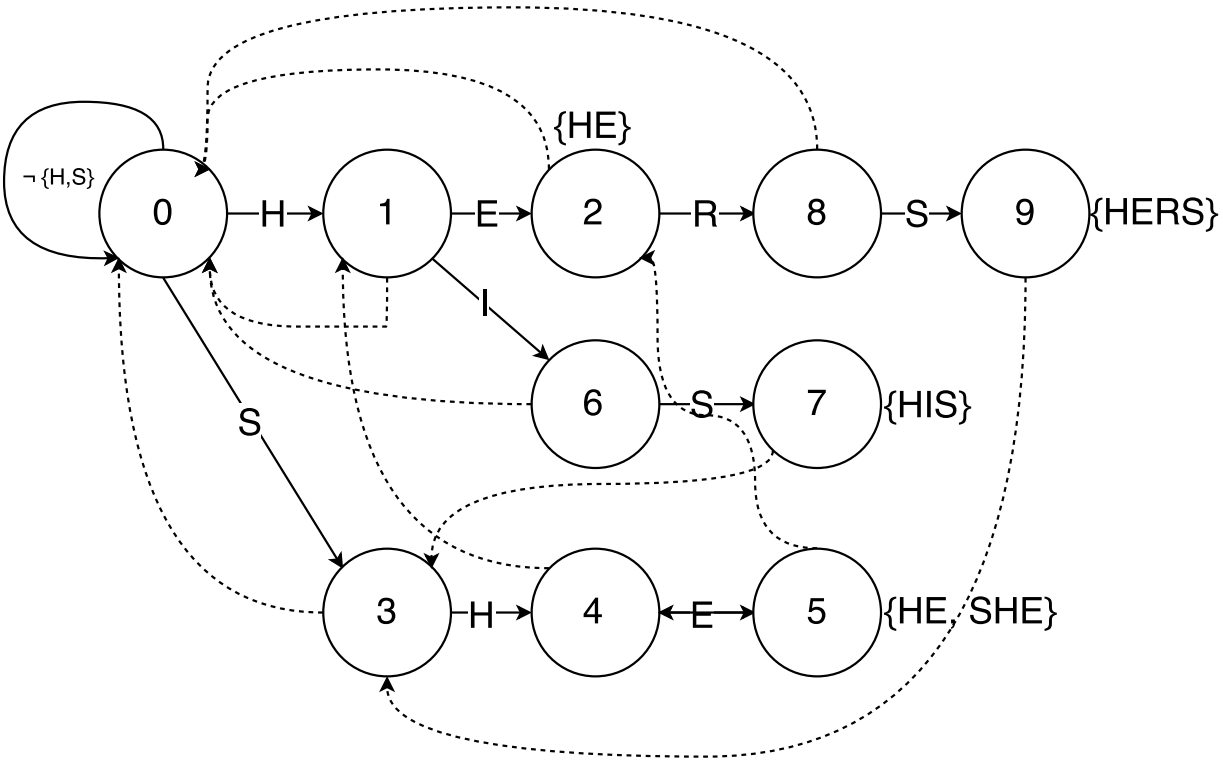


Figure 3: Aho-Corasick Automaton for $P = \{he, she, his, hers\}$

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

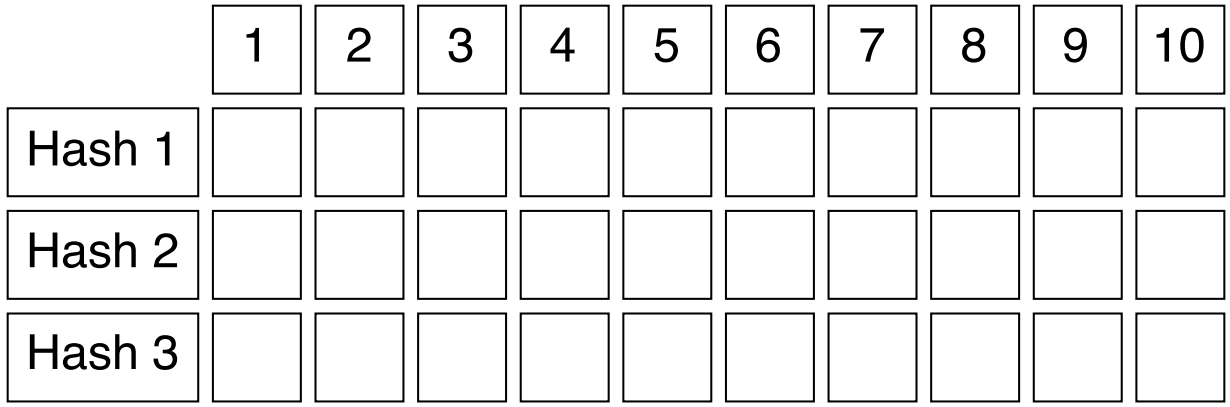
A Bloom filter, first proposed by Bloom (1970), is a data-structure used to determine whether something is a member of a set. Traditionally Bloom filters and other hash table-based algorithms have been used in networking applications such as routing, shallow packet inspection and network monitoring (Song 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.

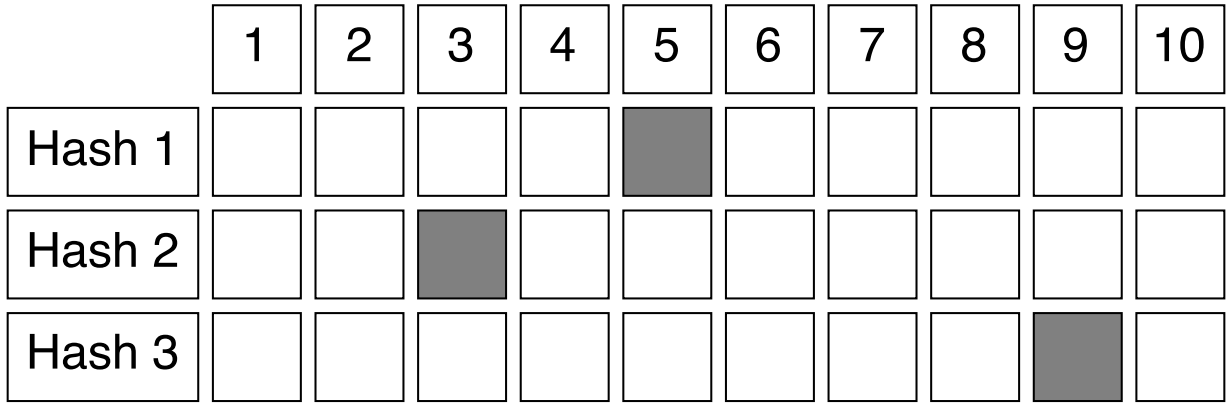
2.5 Boyer-Moore

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.

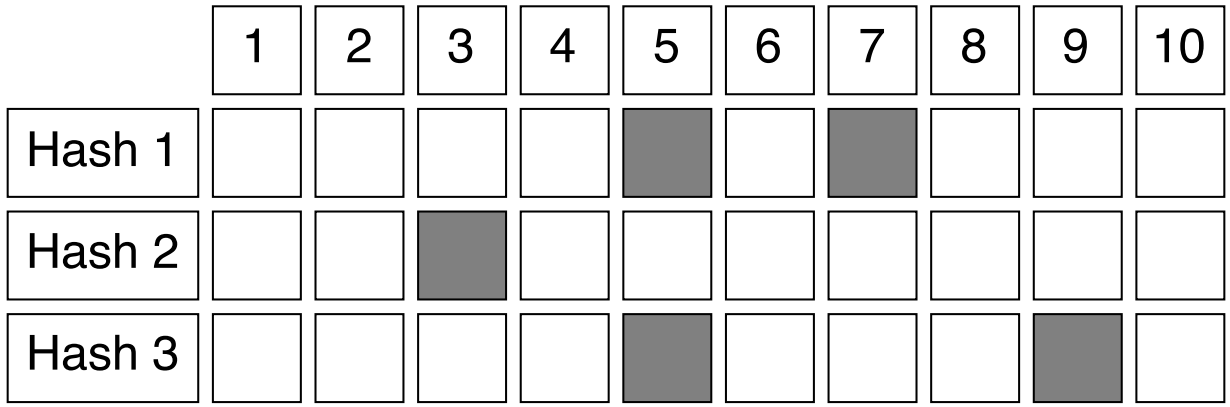
In the Naïve algorithm, if during the matching of a word it fails halfway through, the algorithm simply starts at the next character. The Boyer-Moore uses its pre-processing step to determine if and how far along in the text the algorithm can jump. Thus making fewer comparisons and increasing the overall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



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

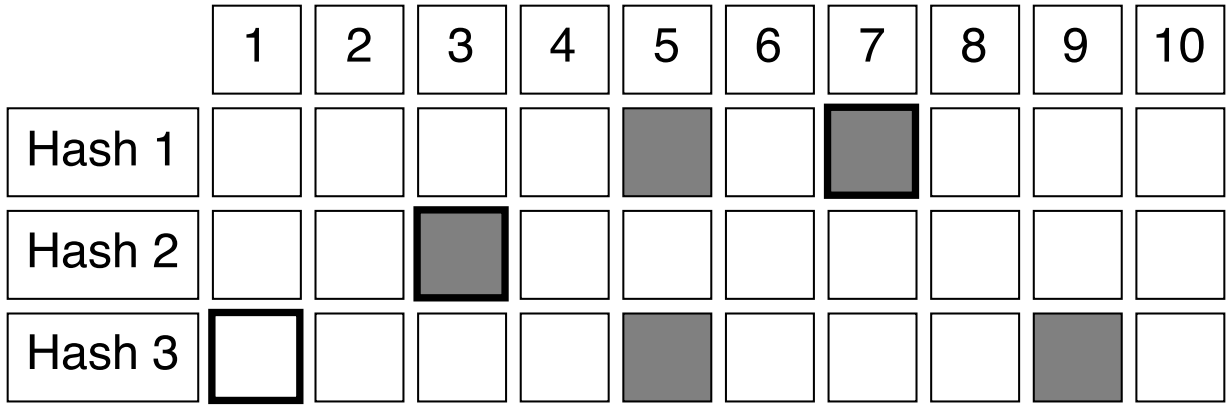


(b) The first search term is hashed by each of the hashing algorithms, for each algorithm a single value is returned. The place in the hash table corresponding to the value is set. Here 5 is set for Hash 1, 3 for Hash 2 and 9 for Hash 10.

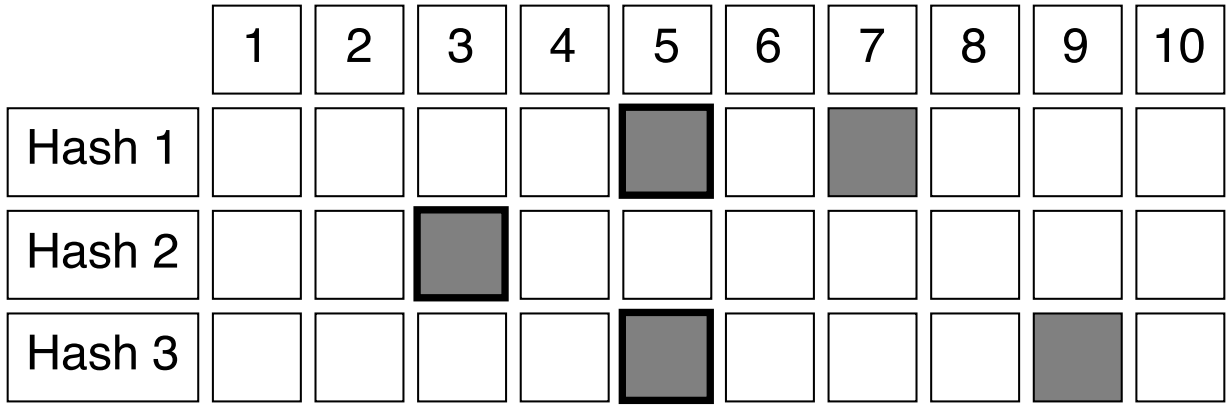


(c) A second search term is put through the three hashing functions. For this term Hash 1 gives the value of 7, Hash 2 gives 3 and Hash 3 gives 5. Note that Hash 2 has given the same values for both search terms.

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.

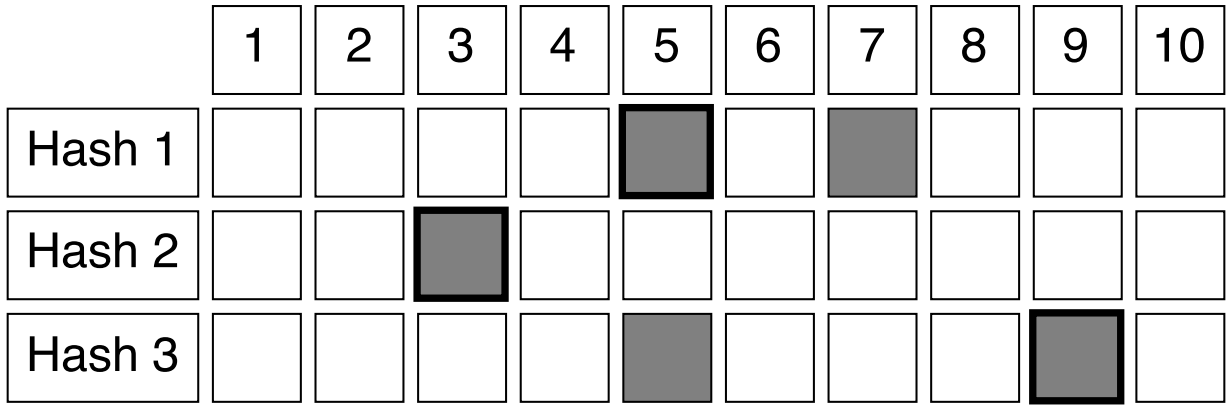
2.6 Horspool

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 2.5), 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). 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(mn)	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)

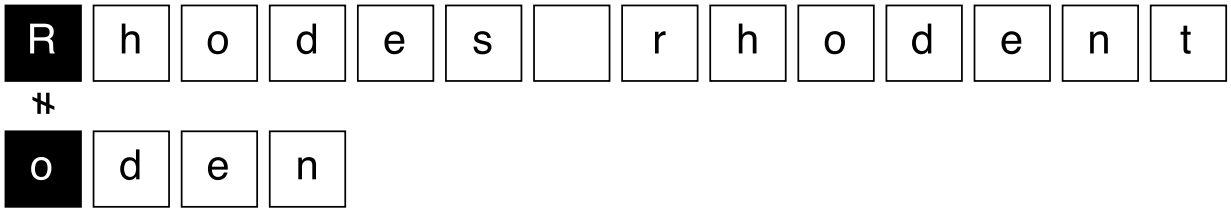
Table 1: Average algorithm time and space complexity

3 Testing Environment

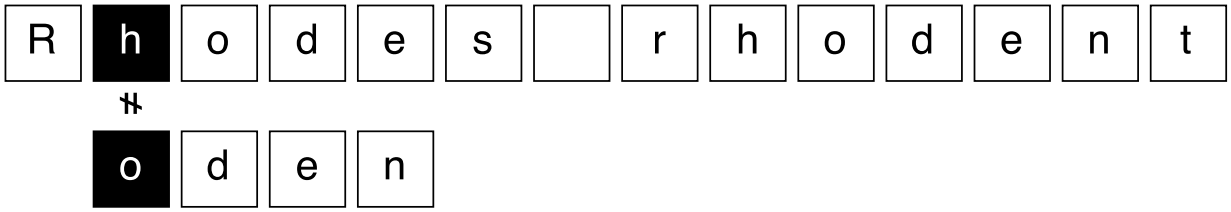
Give a run-down of the system used.

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

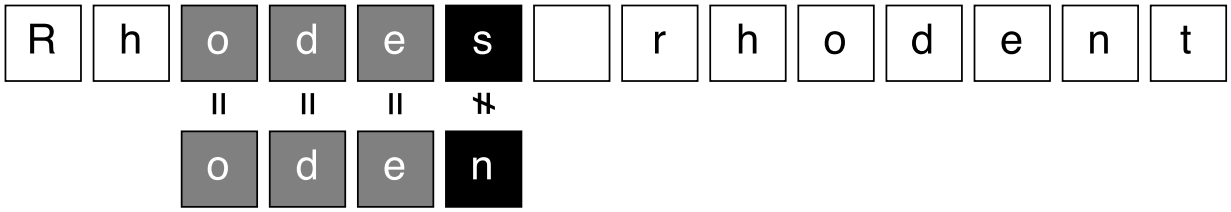
[5] `https://github.com/ebfull/pcap`



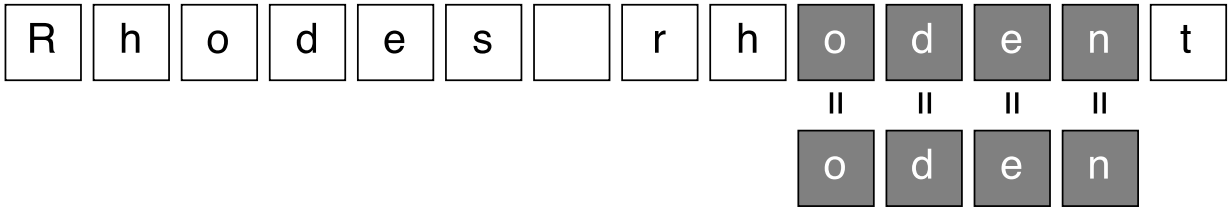
(a) The KMP algorithm compares the first letter of the search string with the first letter of the text. No match is found.



(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”

wrapper 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 Cuckoo[8] Fan et al. (2014) 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/seiflotfy/rust-cuckoofilter

Each algorithm was separately run over the dataset for a variety of search terms.

Search terms: google, amazonaws.com

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

Have some tables of general speeds across the various algorithms/datasets. Same for memory.

	Alices-Adventures-in-Wonderland	Bird-Watching	Frankenstein-or-the-Modern-Prometheus	Pride-and-Prejudice	The-Adventures-of-Sherlock-Holmes	The-Adventures-of-Tom-Sawyer-Complete	The-Yellow-Wallpaper
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 2: Mean processing time of DPI when inspecting plain text for the term “google”

5 Analysis

Compare results with each other.

Compare results against the theoretical outcomes.

6 Conclusion

Talk about going forward. Future work.

References

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*.

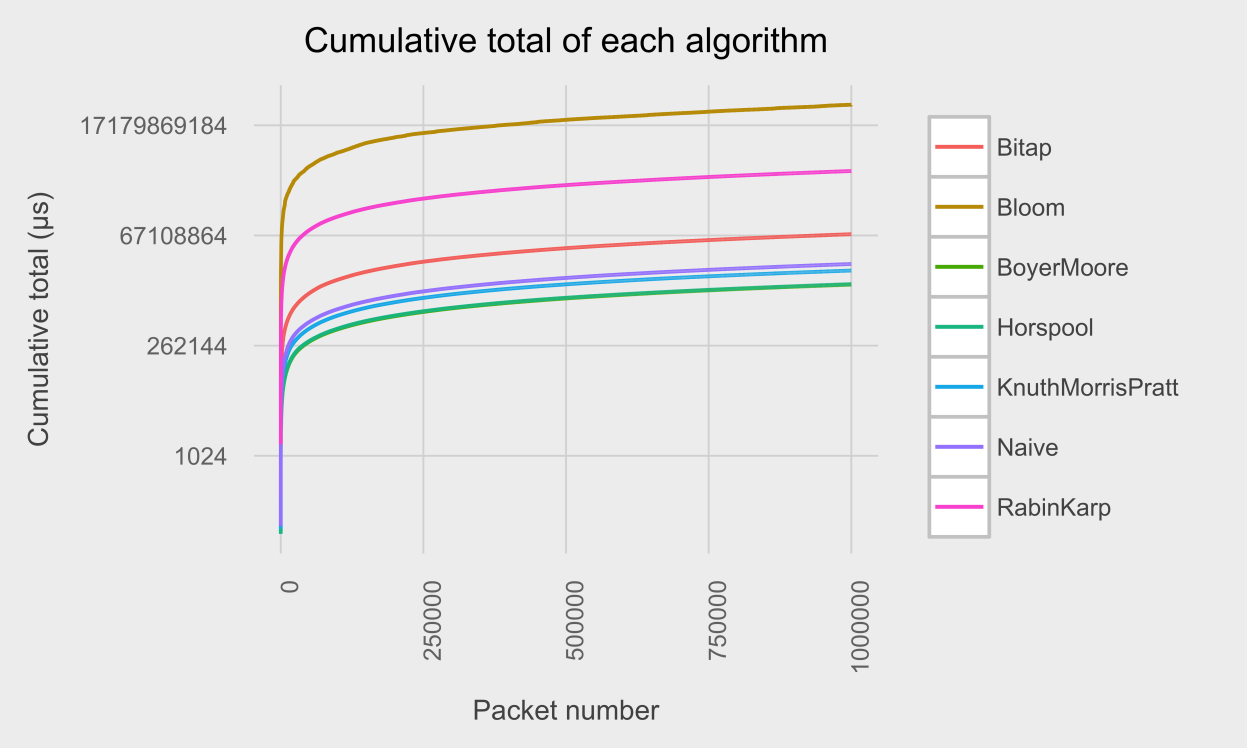


Figure 6: Cumulative sum of processing times of packets in Dataset A for the term “google”

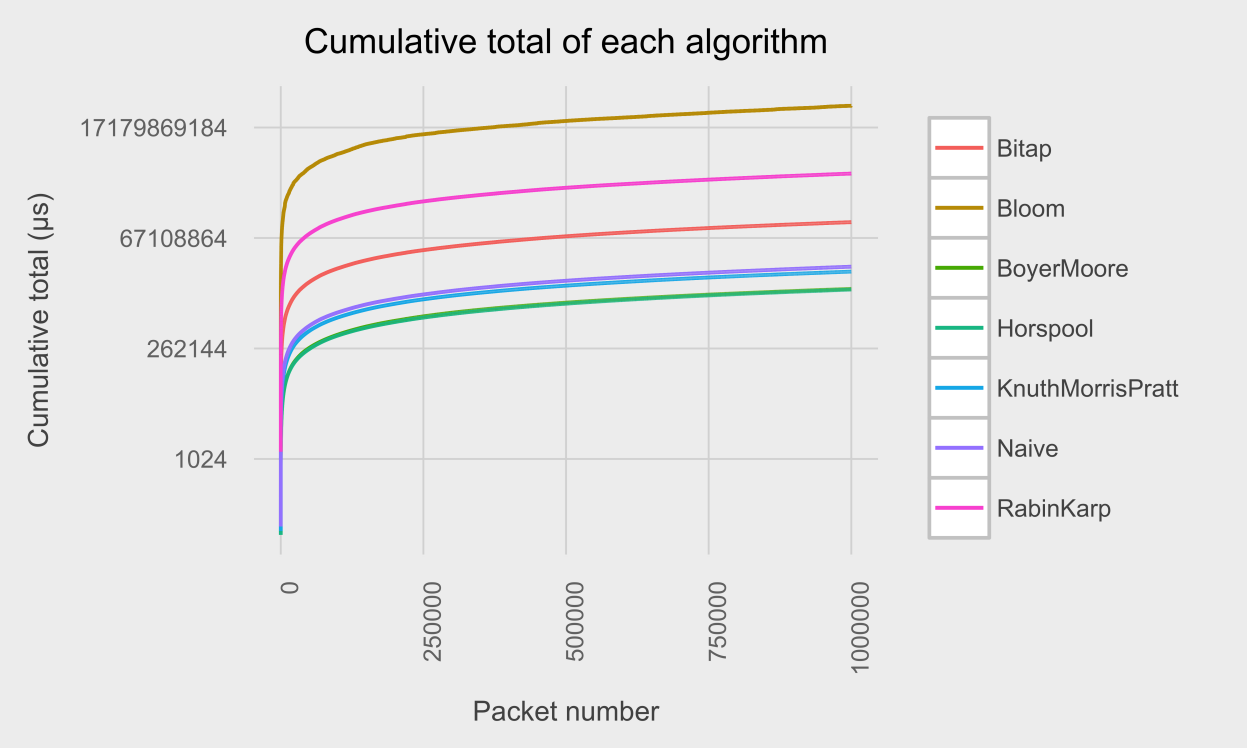


Figure 7: Cumulative sum of processing times of packets in Dataset A for the term “amazonaws.com”

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

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*.