# A Comparison of String Search Algorithms for Deep Packet Inspection

Kieran Hunt

October 15, 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.

## 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

A bit about each of the algorithms. Their time and space complexities. Talk about each of their strengths a weaknesses. Try to guess which might be good at DPI.

Talk about why we chose those algorithms. Talk about any that we didn't choose.

## 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.

This basic algorithm simply tests every possible substring of the search term against every possible substring of the packet payload.

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

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

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

## 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 ($P = \{he, she, his, hers\}$); the following trie can be constructed:

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.

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.

### 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 $g(q, a) = \emptyset$.

### 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.

### 2.2.3 The output function

The output function, $out(q)$ defines all search terms that have been recognised when entering $q$.

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 (reference?), the Bitap algorithm Baeza-Yates and Gonnet (1992) is an approximate string matching algorithm. That is to say that for a given search term 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

Bloom (1970)

### 2.5 Boyer-Moore

Boyer and Moore (1977)

### 2.6 Cuckoo

Fan et al. (2014)

### 2.7 Horspool

Horspool (1980)

### 2.8 Rabin-Karp

Karp and Rabin (1987)

### 2.9 Knuth-Morris-Pratt

Knuth et al. (1977)

## 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[4] library. This library provides a wrapper around libpcap[5]. Through pcap, we are able to easily retrieve the

[4] `https://github.com/ebfull/pcap`
[5] `http://www.tcpdump.org/`

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[6] Bloom (1970) and Cuckoo[7] 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.

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.

# 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*.

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

[7] `https://github.com/seiflotfy/rust-cuckoofilter`

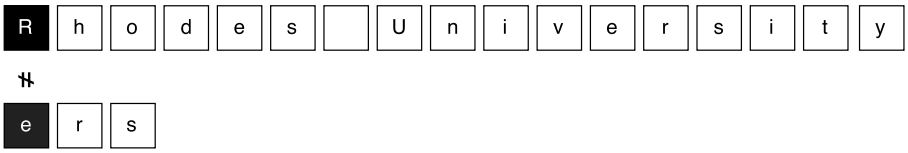
Boyer, R. and Moore, J. (1977). A fast string searching algorithm. *Communications of the ACM*.

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*.

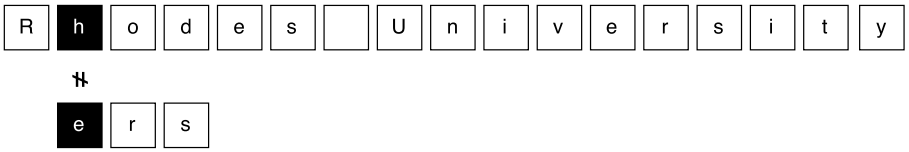
Horspool, N. (1980). Practical fast searching in strings. *Software: Practice and Experience*.

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

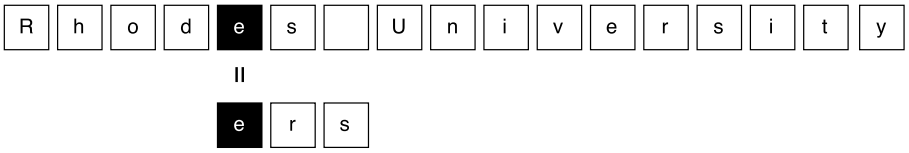
Knuth, D., Morris, J., and Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*.



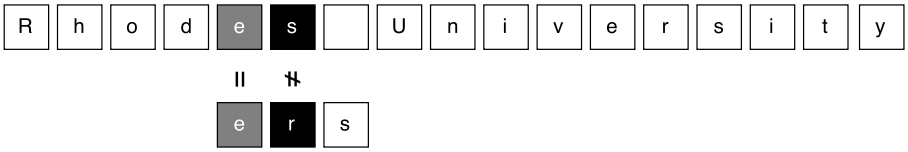

(a) The Naïve algorithm compares the first letter of the search term to the first letter of the text.



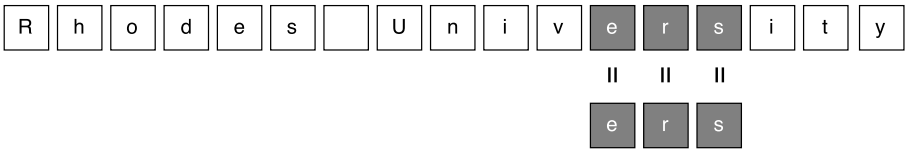

(b) Because the first letter of the search term did not match the first letter of the text, the Naïve algorithm now compares the second letter of the text to the first letter of the search term.




(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

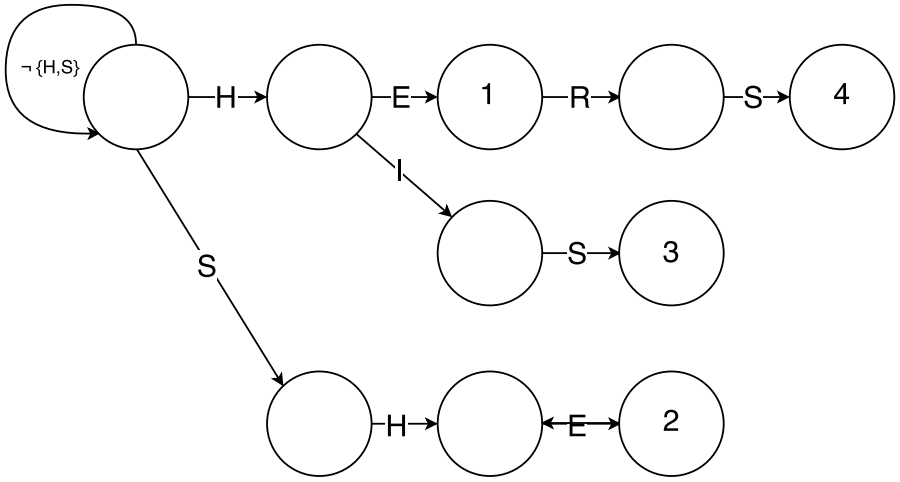



Figure 2: Trie (keyword tree) for the search terms $P = \{he, she, his, hers\}$

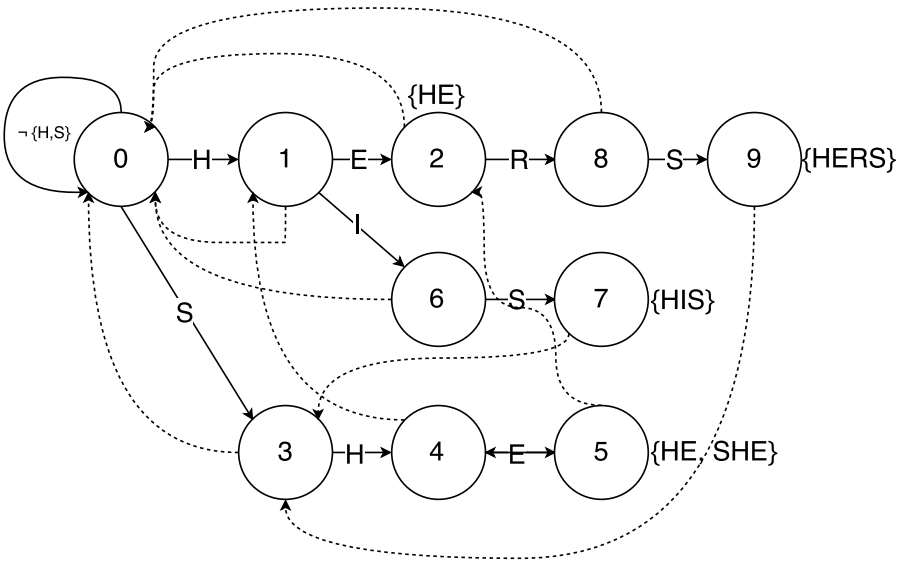



Figure 3: Aho-Corasick Automaton for $P = \{he, she, his, hers\}$