On the Performance of String Search Algorithms for Deep Packet Inspection

Kieran Hunt

February 26, 2016

1 Introduction

In Deep Packet Inspection systems today are designed to operate on expensive custom hardware. Making changes to such a system (such as horizontally or vertically scaling) is often arduous, time consuming and expensive; the process requires adding or removing hardware that is often made-to-order from a manufacturer. Deep Packet Inspection via software means is slower but does provide some benefit over that of hardware inspection. Increasing the speed of software-based Deep Packet Inspection is often as simple as increasing the number of hosts performing the inspection. Modern data centres using hardware virtualisation enable the provisioning of extra processing capacity to done with very little lead time and to be completed quickly. This allows resources to only be used when needed and used elsewhere when not.

String search algorithms have long been of interest to the field of computer science - with the first paper being publish by Morris and Pratt in 1970 - and as a result a substantial number of search algorithms exist Charras and Lecroq (2004). The following paper asks which of these string search algorithms performs best at Deep Packet Inspection and how does their performance compare to that of their theoretical speed.

In order to properly test these string search algorithms, a system was designed to accurately compare each algorithm for inspection of both packet captures as well as textual inputs such as text files.

2 Background

In the context of Deep Packet Inspection, the following definitions are relevant:

- **Packet**: Binary data representing a unit of data carried on a packet-switched network . This includes information from the network to the application layer. The word *packet* is overloaded in computer networking: it either refers to the data unit containing Network Layer information in the OSI model, the data unit containing Network layer information in the TCP/IP model or an entire unit of data transmitted over a network. When referring to a packet we imply the latter meaning.
- **Packet Capture File (PCAP)**: A file containing packets. These packets were generally captured by recording a network interface using the libpcap library (tcp, 2016).
- **Deep Packet Inspection (DPI)**: The process by which a PCAP file or stream of packets from a network interface, is analysed for the presence of predefined patterns or rules.

2.1 String Search Algorithms

A vast collection of string search algorithms have been amassed by Charras and Lecroq (2004) and from that a selection of algorithms was chosen to implement, benchmark and then compare. These algorithms share a number similarities:

- **Exact String Matching**: Each of the algorithms that have been chosen to compare exactly match a given rule in an input. Some string search algorithms are designed to show partial matches but none of these have been selected for comparison.
- **Single Rule Matching**: Each of these algorithms searches for just a single rule throughout the entire input. For multiple rules another solution is needed.

Table 1 has a list of each algorithm, the year it was publish, its author(s) and the time complexity of searching with that algorithm.

Algorithm	Year	Author(s)	Time Complexity
Naive			O(mn)
MorrisPratt	1970	Morris and Pratt	O(n + m)
KnuthMorrisPratt	1977	Knuth et al.	O(n + m)
BoyerMoore	1977	Boyer and Moore	O(nm)
Horspool	1980	Horspool	O(n + m)
ApostolicoGiancarlo	1986	Apostolico and Giancarlo	O(n)
RabinKarp	1987	Karp and Rabin	O(mn)
ZhuTakaoka	1987	Feng and Takaoka	O(mn)
QuickSearch	1990	Sunday	O(mn)
Smith	1991	Smith	O(mn)
ApostolicoCrochemore	1991	Apostolico and Crochemore	O(n)
Colussi	1991	Colussi	O(n)
Raita	1991	Raita	O(nm)
GalilGiancarlo	1991	Galil and Giancarlo	O(n)
Bitap (Shift Or)	1992	Baeza-Yates	O(n)
NotSoNaive	1993	Hancart	O(nm)
Simon	1994	Simon	O(n + m)
TurboBoyerMoore	1994	Crochemore et al.	O(n)
ReverseColussi	1994	Colussi	O(n)

Table 1: Implemented string search algorithms for the purpose of comparison against a packet dataset. Under time complexity, *n* represents the length of the Input and *m* represents the length of a Rule. The time complexity is multiplied by a factor equal to the number of Rules.

Each of the string search algorithms has a known theoretical performance (known as algorithm complexity or Big-O (presented in the *Time Complexity* column of Table 1)) This value is generally related in some way to both the length of the input and rule. The results of the following experiments should follow the predicted complexity of string search algorithms. Algorithm complexity often only provides insight into processing speed where large variations in Input length (differing orders of magnitude) are present. In packet data a limited range of input lengths is possible - this value is set by the maximum transmission unit of the transport medium - and so it may come down to minutiae within the algorithms themselves rather than their overall complexity.

As the implemented algorithms were designed to search for a single rule, their Big-O notations only reflect that property. For many rules, and a serial-style programming approach (where rules are searched for sequentially), these algorithms need to run back-to-back; this style of implementation increases their time to process by a factor about equal to the number of rules. For a parallel-style approach (where many rules are searched for at the same time) the processing time is affected by a factor lower than that of the serial implementation. The speed at which a search for a complete set of rules with a parallel implementation completes is then both related to the algorithmic complexity of the search as well as the number of rules which are being searched for at the same time.

An upper bound for the number of concurrent searches for different rules with the same algorithm and input does exist. That upper bound is defined by the number of processor cores that are available for use as how much a single thread is able to saturate a processor core. For a system with 10 cores. Running a search in a single thread (and therefor on a single core) is only using 10% of the available processing power. Splitting the search across 10 should serve to give close to an order of magnitude speed increase. If those threads were saturating the processing speed of all 10 cores (using 100% 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.

3 Method

For the purpose of these experiments, a system was developed to allow for the accurate running and performance measurement of each of the implemented algorithms (in Table 1). The following terminology is relevant to that system:

- **Input**: The system interacts with all kinds of input through a common interface. Both textual- and packet-style are interacted with through this interface. The system is able to request a single byte of data at a given location from an input as well as obtain the length of the input.
- **Rule**: Rules are the patterns searched for by the system. Rules mimic inputs in that the system may also retrieve a single byte from the rule or get the rule's length.
- **Algorithm**: This represents a string search algorithm (from Table 1) and is the means by which the system interacts with all of the implemented algorithms. An algorithm provides a search function to the system with common inputs and outputs.
- **Result**: For a single input, set of rules and a particular run, a result is created. This result contains all of the information about the outcome of the search and the configuration surrounding it.

The test system was designed to accept a JSON file (jso, 2016) as configuration (shown in Listing 1).

```
{
  "algorithms": ["Naive", "MorrisPratt", ..., "ReverseColussi"],
  "rules": ["time", "person", ..., "msn"],
  "inputs": [
    {
      "type": "pcap",
      "location": "smallcapture.pcap"
    },
    {
      "type": "text",
      "location": "alice.txt"
    }
  ],
  "times": 20,
  "threadCount": 18
}
```

Listing 1: Sample testConfiguration.json

For the test, the following configuration was chosen:

- `algorithms` - a list of all implemented algorithms (See Table 1)
- `rules` - string-based rules covering the twenty most popular websites (ale, 2016) - where just their domain names were taken - as well as the twenty most popular words in the English language (oed, 2016).
- `inputs` - a dataset of DNS traffic was selected and a subset of 10000 packets was extracted. This was labeled `smallcapture.pcap` and known hereafter as *Dataset A*. The complete *Alice in Wonderland* by Lewis Carol, labeled as labeled `alice.txt`, was selected as the second input and known hereafter as *Dataset B*. The test system treats textual and PCAP input files differently. Text input files transformed into a single input object representing all information in that file. PCAP input files are split up into individual packet objects and each packet represents a single input.
- `times` - each algorithm searched all inputs for every rule. We chose to repeat the test 20 times to ensure that transient fluctuations caused by other processes on the test hardware could be identified.
- `threadCount` - drastic speed increases are made possible by splitting the work of the Algorithms across multiple threads. The machine used to perform the test has 24 useable cores and so a max `threadCount` of 18 was chosen to best make use of those cores.

For *Dataset A*, 2 gives a statistical summary of the lengths of the inputs. As *Dataset A* is comprised of DNS requests and responses it is unsurprising that the average length of a packet is about 110 bytes. This is far below the ethernet maximum transmission unit of 1500 bytes (Hornig, 1984).

Minimum	First Quartile	Median	Mean	Third Quartile	Maximum
54.0	81.0	85.0	109.8	99.0	585.0

Table 2: *Dataset A* packet length statistical summary (in bytes).

4 Results

The test generated 3400340 result objects. Result objects look similar to Listing 2. Each result object contains the following information about a search: the start time and end time, the elapsed time $(end\ time - start\ time)$, the list of rules, the locations that the rules were found in the input, the algorithm used to perform the search, the input file (as mentioned earlier many inputs can share an input file), the unique ID of the input, the run number (between 1 and the number of times the test was set to run), and the ID of that run.

```
[
    {
        "start": 206079193307938,
        "end": 206079207132342,
        "elapsed": 13824404,
        "rules": ["time", "person", ... "msn"],
        "locations": [1, 2, ..., n],
        "algorithm": "Smith",
        "inputFile": "smallcapture.pcap",
        "inputID": "bf75faa5",
        "runNumber": 1,
        "runId": "d9b4a28aefbd"
    },
    ...
]
```

Listing 2: Example result

4.1 Analysis

4.1.1 Which Algorithms are Fastest? Which are Slowest?

Figure 1b shows a comparison of processing times for each algorithm over each of the packets in *Dataset A*. Figure 1b gives the same comparison albeit for *Dataset B*.

The following is a list of algorithms, ordered by their mean processing times for both *Dataset A* and *Dataset B*: 1. Horspool, 2. QuickSearch, 3. Raita, 4. ReverseColussi, 5. BoyerMoore, 6. ZhuTakaoka, 7. Smith, 8. Colussi, 9. KnuthMorrisPratt, 10. TurboBoyerMoore, 11. ApostolicoGiancarlo, 12. MorrisPratt, 13. Naive, 14. Simon, 15. Bitap, 16. NotSoNaive, 17. RabinKarp.

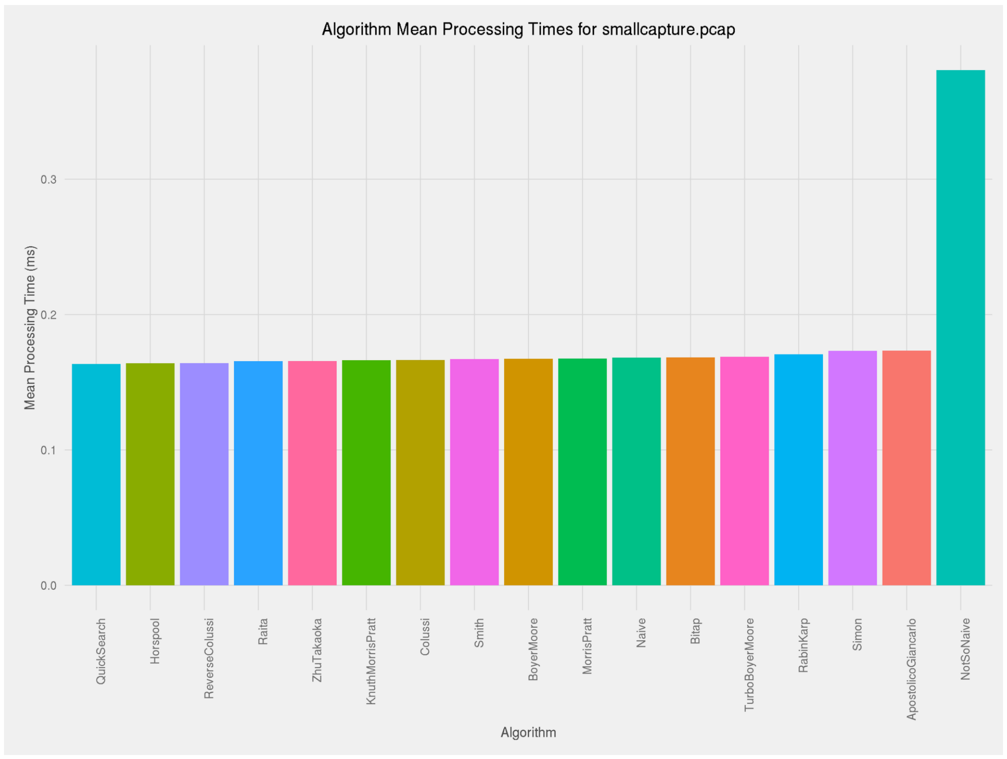
Although the order according to mean processing speed changes between Figure 1a and 1b, the position of the algorithms varies by less than four places.

The relative speed difference of each algorithm in Figure 1a appears to be noticeably small. The first 16 algorithms appear to have nearly identical mean processing speeds. By increasing the length of the input data (in this case by using the *Dataset B* in Figure 1b) we are able to accentuate the differences between the algorithms.
Table 3 gives a full statistical summary of every algorithm searching through *Dataset A*.

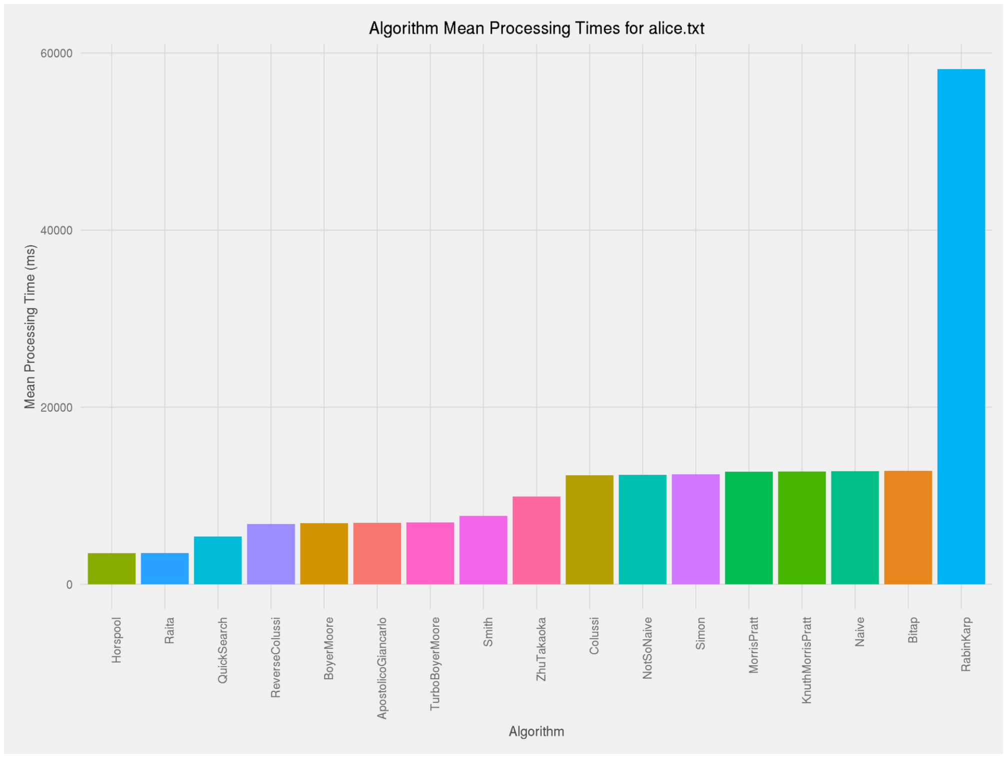
The *fastest* algorithms were QuickSearch and Horspool.

QuickSearch (Sunday, 1990) makes use of the BoyerMoore (Boyer and Moore, 1977) algorithm's bad-character shift table. The algorithm is known to perform well for short rules (Lecroq, 1995) and because our test used relatively short words as its input this may be why QuickSearch has performed so well.

The Horspool (Horspool, 1980) algorithm also makes use of the bad-character shift table from the BoyerMoore (Boyer and Moore, 1977) algorithm. Although the Horspool algorithm has a O(mn) complexity (Table 1), it can be shown that the average number of comparisons made with a character in the input is α, where $\frac{1}{c} \leq \alpha \leq \frac{2}{c+1}$ and c is the number of characters in the alphabet (Baeza-Yates and Régnier, 1992). In our case our alphabet is 256 characters long (the size of one byte) making the average number of comparisons to a character in our text $0.0039 \leq \alpha \leq 0.0079$. That is a very low number of comparisons per input character.



(a) Mean Input processing times of each Algorithm for *PCAP Dataset*.



(b) Mean Input processing times of each Algorithm for *Alice in Wonderland*.

Figure 1: Mean input processing times for each algorithm.

Algorithm	Minimum	First Quartile	Median	Mean	Third Quartile	Max
QuickSearch	0.08321	0.10750	0.11430	0.16350	0.12320	259.7
Horspool	0.0926	0.1098	0.1166	0.1640	0.1246	5.377
ReverseColussi	0.09139	0.11180	0.11910	0.16410	0.12740	5.7
Raita	0.09035	0.10710	0.11600	0.16560	0.12440	164.1
ZhuTakaoka	0.09288	0.11200	0.11900	0.16560	0.12740	4.645
KnuthMorrisPratt	0.09113	0.11180	0.11930	0.16630	0.12780	25.96
Colussi	0.09178	0.11130	0.11840	0.16640	0.12670	29.69
Smith	0.09102	0.11170	0.11880	0.16710	0.12660	5.334
BoyerMoore	0.09287	0.11070	0.11740	0.16730	0.12540	33.22
MorrisPratt	0.08543	0.10910	0.11540	0.16750	0.12360	297.8
Naive	0.08845	0.11180	0.11830	0.16820	0.12620	63.84
Bitap	0.09277	0.11160	0.11900	0.16840	0.12780	50.14
TurboBoyerMoore	0.09101	0.11220	0.11900	0.16880	0.12680	41.68
RabinKarp	0.09562	0.11580	0.12230	0.17070	0.13200	245.6
Simon	0.08248	0.11140	0.11890	0.17320	0.12750	117.2
ApostolicoGiancarlo	0.09368	0.11420	0.12170	0.17340	0.13020	5.067
NotSoNaive	0.1092	0.1698	0.2277	0.3806	0.2957	552.3

Table 3: Algorithm processing speed summary for *Dataset A*

The *slowest* algorithms were NotSoNaive and RabinKarp.

RabinKarp's (Karp and Rabin, 1987) slowness can easily be attributed to both its O(mn) complexity as well as its constant recalculation of the input hash. The RabinKarp algorithm works by maintaining a hash of the rule and calculating a hash on a sliding window of the input. The theory is that the comparison between the hashes is faster than that of the entire rule with the sliding window each time.

NotSoNaive's (Hancart, 1993) slowness is surprising at first. It performs worse than the Naive algorithm when it was designed as an improvement on it. The NotSoNaive algorithm attempts to improve upon the Naive algorithm by adding a check to see if the input ahead shows signs of matching the rule; this extra step seems to cause the speed decrease when compared to Naive. Figure 3 may make you may think that the mean processing time for NotSoNaive was heavily influenced by the maximum speed (that is to say that that maximum value is an outlier. Table 3) but by checking the median processing time - again in 3 - it is clear that the entire set of results is shifted towards a longer processing time.

4.1.2 Do Some Algorithms Perform Differently Depending on the Input Length?

Figure 2 shows a scatter plot of mean input processing time versus packet length. This plot shows processing times for all algorithms. Note that the vertical axis has been plotted with a logarithmic scale. From the plot it is clear that there is an upward trend of processing time as the input length increases. This is expected as every algorithm in Table 1 has an algorithmic complexity related to the length of the input.

Algorithms best suited to deep packet inspection would be expected to perform better for smaller input lengths and the performance at input lengths larger than the maximum length of a packet would not matter. Algorithms with a bounded maximum processing time would be especially enticing to a deep packet inspection system implementor as it could guarantee a minimum throughput.

Figure 3 shows similar graphs to Figure 2 but this time just the two fastest and two slowest algorithms were chosen.

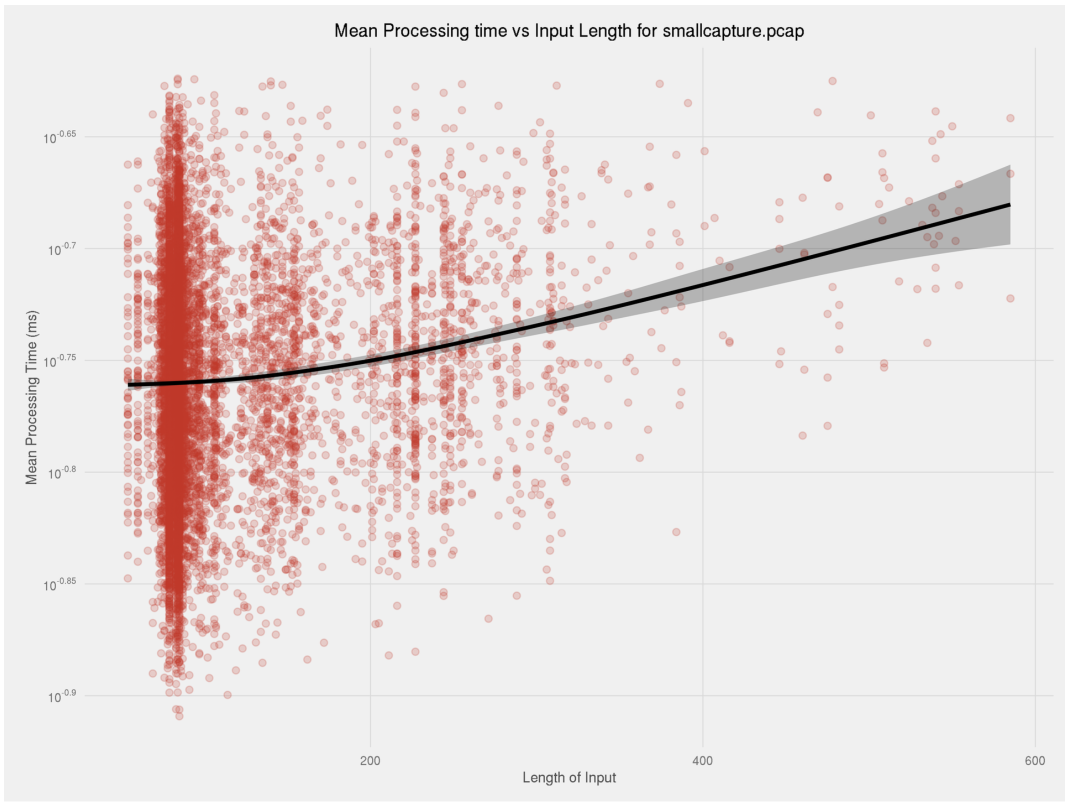


Figure 2: Input processing speeds vs input length for *Dataset A*.

4.1.3 Which Algorithms Have the Least Fluctuating Processing Times?

It is important that algorithms have consistent performance. Algorithms that have highly fluctuating processing speeds lead to unpredictable processing times and unwanted slowdowns of realtime packet inspection.

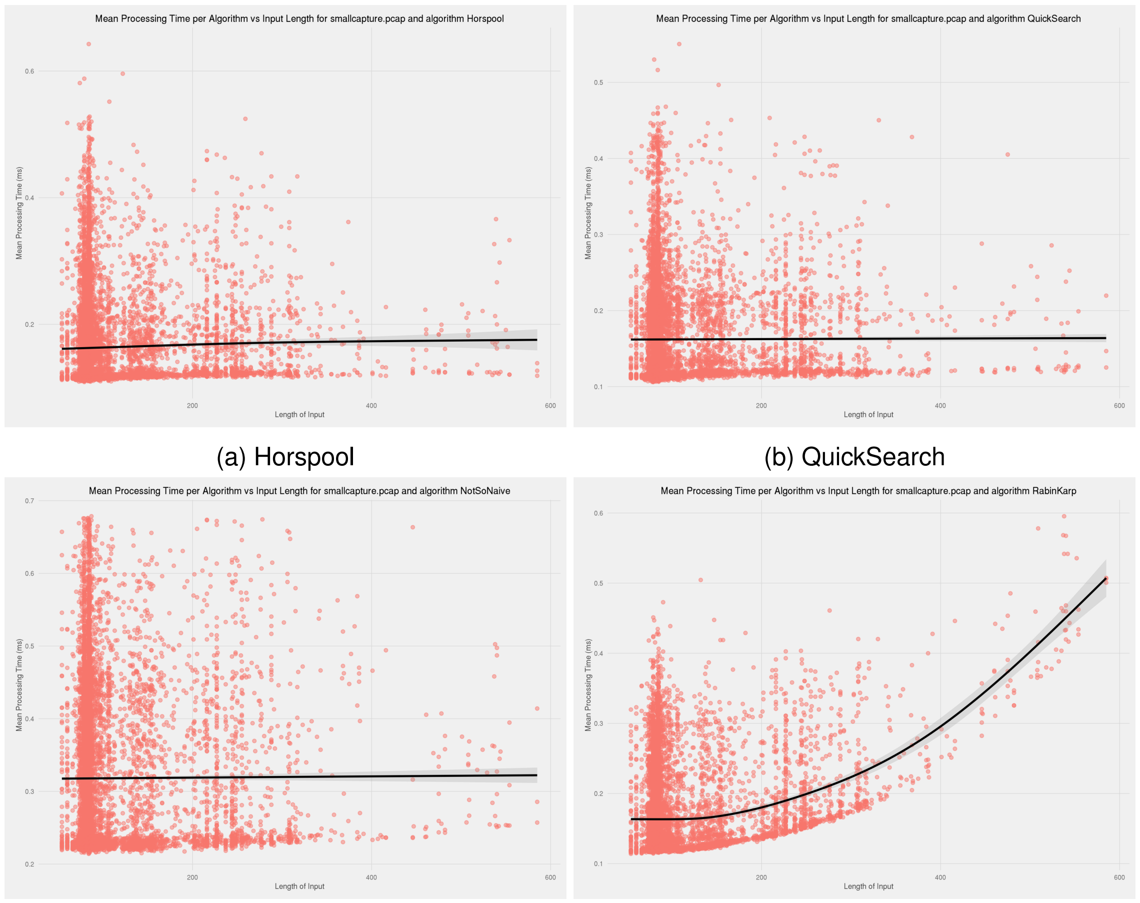
Figure 4 shows the standard deviations of the mean processing time for each packet each of the algorithms for *Dataset A*. Most algorithms appear to have relatively low fluctuation on their processing times whilst some are a bit higher. The Horspool (Horspool, 1980) algorithm appears to have a very low standard deviation which, combined with its speed shown in Table 3, makes it a very strong packet processing algorithm compared to the other algorithms.

Interestingly in Figure 4, QuickSeach has a notably higher standard deviation than Horspool. It may not be all that well suited to packet inspection as a result.

4.1.4 Real World Packet Processing Speeds

For the packet data, most algorithms have a mean packet processing speed of anywhere between 0.1 and 0.2 milliseconds.

The average processing speed of about 0.15ms (Table 3) as well as an average packet length of about 110 bytes (Table 2) means that a rough line speed of around 6 megabits per second is achievable. This speed is obviously much less than what would be required of a modern Deep Packet Inspection system - it is a few orders of magnitude smaller than what would be required. This research is not that concerned with the absolute speeds of each of these algorithms but their speeds when compared with each other.



(a) Horspool

(b) QuickSearch

(c) NotSoNaive

(d) RabinKarp

Figure 3: Mean processing times versus input length for the Horspool, QuickSearch, NotSoNaive, and RabinKarp algorithms for *Dataset A*.

5 Conclusion

Of the original algorithms chosen to test, Horspool (Horspool, 1980) seems to perform best when it comes to packet analysis. Although software-based deep packet inspection is not as popular as its hardware counterpart, it does provide an easier-to-scale environment that can better cope with the fluctuations of periodic network load. Bootstrapping a software-based packet inspection system could be done with any consumer-grade computer equipment without the need to purchase expensive hardware.

References

02 2016. URL `http://www.alexa.com/topsites`.

02 2016. URL `http://www.json.org/`.

02 2016. URL `https://www.oxforddictionaries.com/words/the-oec-facts-about-the-language`.

02 2016. URL `http://www.tcpdump.org/`.

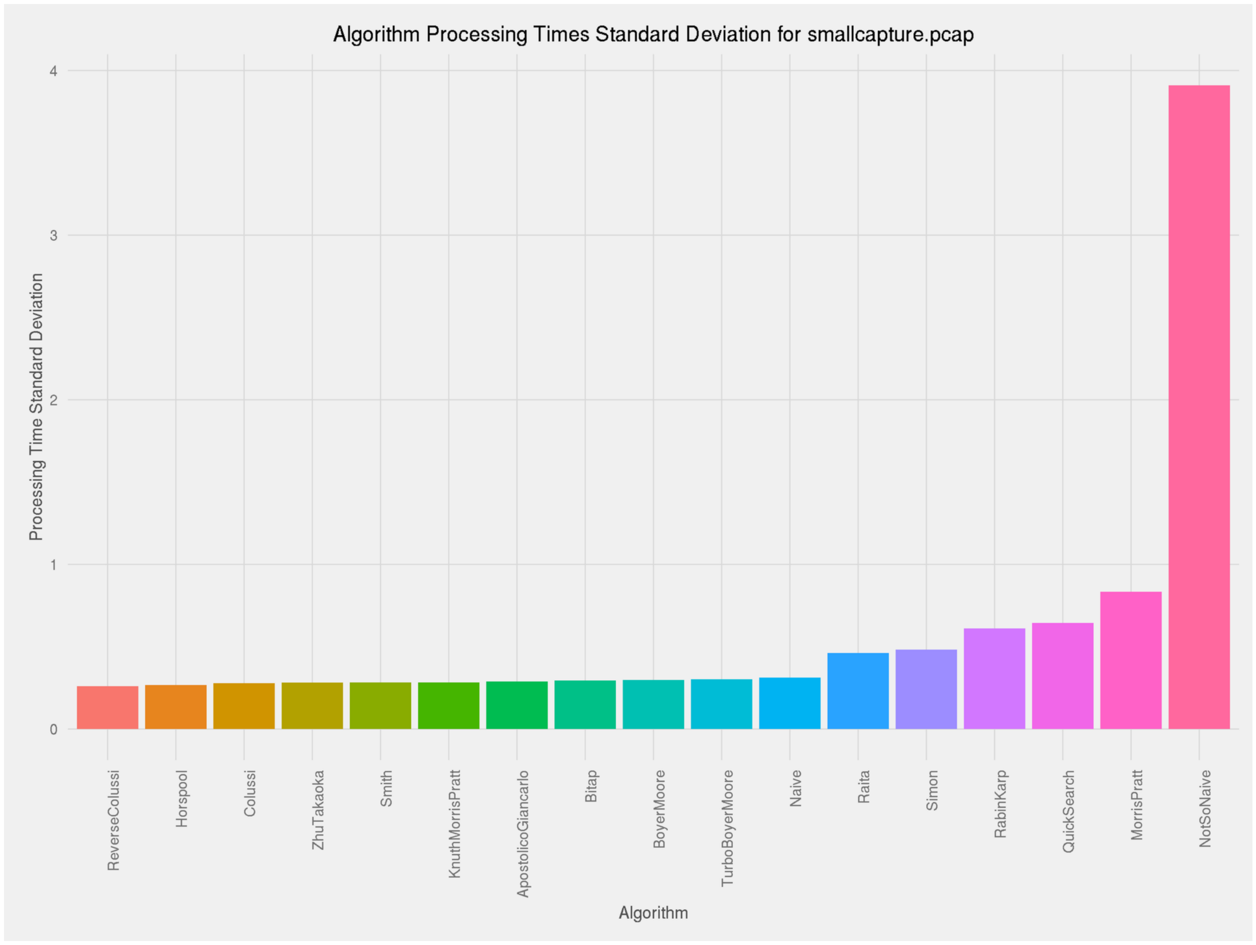


Figure 4: The standard deviation of processing time per algorithm for *Dataset A*.

A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Information and Computation*, 1991.

A. Apostolico and R. Giancarlo. The boyer moore galil string searching strategies revisited. *SIAM Journal on Computing*, 1986.

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

R. A. Baeza-Yates and M. Régnier. Average running time of the boyer-moore-horspool algorithm. *Theoretical Computer Science*, 1992.

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

C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. 2004.

L. Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 1991.

L. Colussi. Fastest pattern matching in strings. *Journal of Algorithms*, 1994.

M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 1994.

Z. R. Feng and T. Takaoka. On improving the average case of the boyer-moore string matching algorithm. *Journal of Information Processing*, 1987.

Z. Galil and R. Giancarlo. On the exact complexity of string matching: Upper bounds. *SIAM Journal on Computing*, 1991.

C. Hancart. *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*. PhD thesis, Université Paris Diderot, 1993.

Charles Hornig. A standard for the transmission of ip datagrams over ethernet networks. Technical report, Network Working Group, 1984.

R. N Horspool. Practical fast searching strings. *Software: Practice and Experience*, 1980.

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

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

T. Lecroq. Experimental results on string matching algorithms. *Software: Practice and Experience*, 1995.

J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

T. Raita. Tuning the boyer-moore-horspool string searching algorithm. *Software: Practice and Experience*, 1991.

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*, 1994.

D. Smith, P. Experiments with a very fast substring search algorithm. *Software: Practice and Experience*, 1991.

D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 1990.