On the Performance of String Search Algorithms for Deep Packet Inspection

***Abstract*—As the speed of computer processors grows, so too does the speed at which networks communicate [32]. With more and more people placing increasing trust on computer networks, the requirement for safety within these networks is paramount. Traditionally, packet analysis in firewalls and intrusion detection systems has been achieved through expensive and custom hardware specifically designed for this purpose. Off-the-shelf hardware - although slower - makes it easier for implementers to quickly develop and deploy new packet inspection techniques and rules, and create systems that scale to meet a network's demands. This paper investigates and compares exact string matching algorithms - implemented on general purpose processors - for the purpose of deep packet inspection.**

***Index Terms*—Network Security, Firewalls, Intrusion Detection Systems, Deep Packet Inspection, Exact String Search Algorithms**

I. Introduction

Deep Packet Inspection (DPI) systems today are designed to operate on expensive custom hardware [5]. 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 [14]. Modern data centres using hardware virtualisation enable the provisioning of extra processing capacity with very little lead time. This allows resources to only be used when needed and employed elsewhere when not.

String search algorithms have long been of interest to the field of computer science - with the first paper written in 1970 [31] - and as a result a substantial number of search algorithms exist [13]. 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.

II. Background

Computer networks have for long been vulnerable to attacks [6, 19, 37]; both externally and internally. Vulnerabilities such as Eavesdropping, Viruses, Worms, Trojans, Phishing, IP Spoofing, and Denial of Service are prevalent today [18]. Network administrators wishing to mitigate these threats can employ a number of different measures. Such measures could include: Cryptographic Systems, Firewalls, Intrusion Detection Systems (IDSs), Anti-Malware Software, and Transport Layer Security (TLS) [18].

Many of those mitigating measures - notably Firewalls and Intrusion Detection Systems - make use of Deep Packet Inspection. These system scan network traffic in an effort to identify packets of interest.

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 [2].
- 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.

Deep Packet Inspection is an extremely resource intensive task. Systems designed to inspect packet data are often processing packets destined for hundreds of other devices and are compelled to match network line speeds in order to meet quality of service requirements.

Hardware implementations have, for long, been the method of choice for systems performing Deep Packet Inspection. Hardware methods of Deep Packet Inspection are provably fast [20], but suffer from high costs and difficult expansion [33]. Software implementations can be run on commodity hardware but are notably slower than their hardware counterparts [14].

The intention of this research is to compare the speed and behaviour of string search algorithms when performing Deep Packet Inspection.

A. String Search Algorithms

A vast collection of string search algorithms have been described by Charras and Lecroq [13], 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 or pattern 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.

Rules take the form of a string which is then search for in the input.

Table I gives a list of each algorithm, the year it was publish, a reference to its seminal paper and the time complexity of searching with that algorithm. 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'. The algorithmic complexity is used to describe the behaviour of an algorithm as its factors tend towards infinity [9, 28]. This value is generally related

TABLE I: Chosen and implemented exact string search algorithms.

Algorithm	Year	Cite	Big-0
Naïve			$O(nm)$
Morris-Pratt	1970	[31]	$O(n+m)$
Knuth-Morris-Pratt	1977	[27]	$O(n+m)$
Boyer-Moore	1977	[12]	$O(nm)$
Horspool	1980	[25]	$O(n+m)$
Apostolico-Giancarlo	1986	[8]	$O(n)$
Rabin-Karp	1987	[26]	$O(nm)$
Zhu-Takaoka	1987	[21]	$O(nm)$
Quick-Search	1990	[38]	$O(nm)$
Smith	1991	[36]	$O(nm)$
Apostolico-Crochemore	1991	[7]	$O(n)$
Colussi	1991	[15]	$O(n)$
Raita	1991	[34]	$O(nm)$
Galil-Giancarlo	1991	[22]	$O(n)$
Bitap (Shift Or)	1992	[10]	$O(n)$
Not So Naïve	1993	[23]	$O(nm)$
Simon	1994	[35]	$O(n+m)$
Turbo Boyer-Moore	1994	[17]	$O(n)$
Reverse Colussi	1994	[16]	$O(n)$

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 (each rule is 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.

III. 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 I). The following terminology is relevant to that system:

```
{
  "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

- **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 I) 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 [1] as configuration (shown in Listing 1).

For the test, the following configuration was chosen:

- `algorithms` - a list of all implemented algorithms (See Table I)
- `rules` - string-based rules covering the twenty most popular websites [4] - where just their domain names were taken - as well as the twenty most popular words in the English language [3].
- `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.

```
[
    {
        "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

For *Dataset A*, II 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 [24].

TABLE II: *Dataset A* packet length statistical summary (in bytes).

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

IV. Results

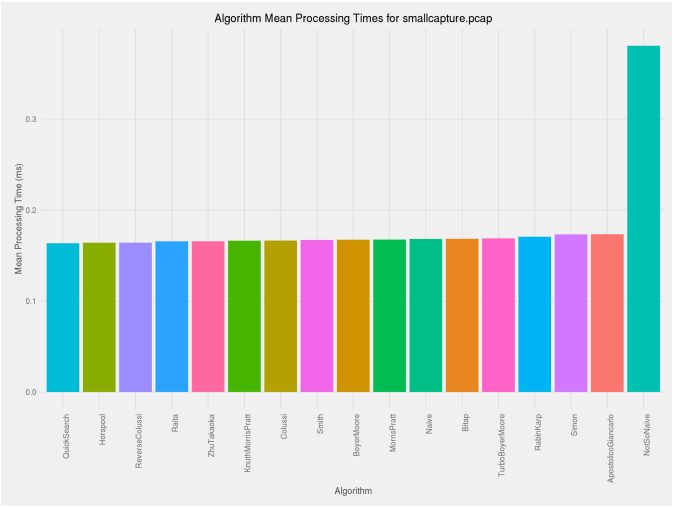
The test generated 340 0340 result objects. Result objects look similar to Listing 2. Each result object contains the following information about a search: the start time and end time, the elapsed time ($end\ time - start\ time$), the list of rules, the locations that the rules were found in the input, 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.

A. Analysis

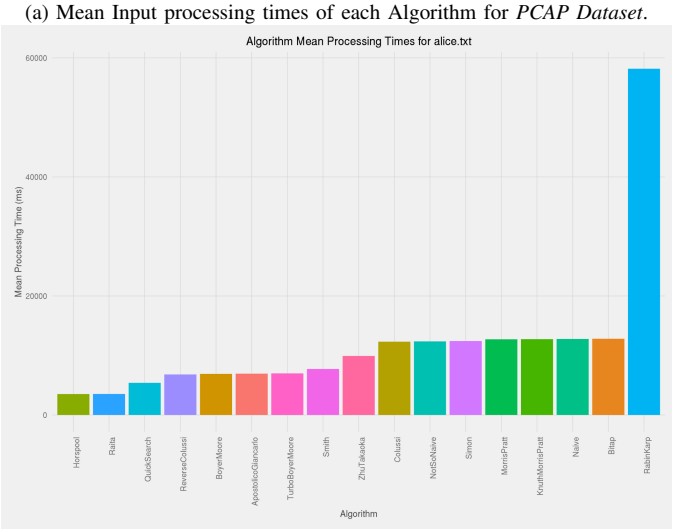
In order to accurately judge the performance of the string search algorithms in the context of Deep Packet Inspection, a number of questions relating to their behaviour were posed. First and foremostly is the question of the algorithms' raw speed when processing the datasets. Next, an indication of the algorithms' behaviour for inputs of varying length was needed. Algorithms should be able to process each packet - regardless of length - in a reasonable amount of time. Finally, algorithms must have deterministic processing times for good candidacy in packet processing. An indication of determinism is the standard deviation of each algorithm.

1) Which Algorithms are Fastest? Which are Slowest?: Figure 1a shows a comparison of processing times for each algorithm over each of the packets in *Dataset A*. Figure 1b gives the same comparison, this time where *Dataset B* is the input.

The order of the algorithms in Figures 1a and 1b is inconsistent. The relative position of each algorithm, from one figure to the other varies by very little. The relative speed difference of each algorithm in Figure 1a is noticeably small in comparison with the slowest algorithm. The first sixteen algorithms appear to have nearly identical mean processing speeds. By increasing the length of the input data



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



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

Fig. 1: Mean input processing times for each algorithm.

(in this case by using the *Dataset B* in Figure 1b) we are able to accentuate the differences between the algorithms.

Table III gives a full statistical summary of every algorithm searching through *Dataset A*.

TABLE III: Algorithm processing time statistics: *Dataset A*

Algorithm	Min	Q_1	$\widetilde{\Delta t}$	μ	Q_3	Max
Quick Search	0.08321	0.10750	0.11430	0.16350	0.12320	259.7
Horspool	0.0926	0.1098	0.1166	0.1640	0.1246	5.337
Reverse Colussi	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
Zhu-Takaoka	0.09288	0.11200	0.11900	0.16560	0.12740	4.645
Knuth-Morris-Pratt	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
Boyer-Moore	0.09287	0.11070	0.11740	0.16730	0.12540	33.22
Morris-Pratt	0.08543	0.10910	0.11540	0.16750	0.12360	297.8
Naïve	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
Turbo Boyer-Moore	0.09101	0.11220	0.11900	0.16880	0.12680	41.68
Rabin-Karp	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
Apostolico-Giancarlo	0.09368	0.11420	0.12170	0.17340	0.13020	5.067
Not So Naïve	0.1092	0.1698	0.2277	0.3806	0.2957	552.3

In Table III, the the fastest algorithms from Figures 1a and 1b are highlighted in green whilst the two slowest algorithms from those figures are highlighted in blue.

The slowest algorithms from those tests were the Not So Naïve [23] and Rabin-Karp [26] algorithms. The fastest algorithms were the Quick Search [38] and Horspool [25] algorithms.

The QuickSearch [38] algorithm makes use of the Boyer-Moore [12] algorithm's bad-character shift table [13]. The algorithm is known to perform well when the rule is much shorter than the length of the alphabet [30]. Our test consisted primarily of fairly short rules which may indicate why the algorithm performed so well.

The Horspool [25] algorithm also makes use of the bad-character shift table from the BoyerMoore [12] algorithm. Although the Horspool algorithm has a $O(nm)$ complexity (Table I), 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 [11]. 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.

The Rabin Karp algorithm's [26] slowness can easily be attributed to both its O(mn) complexity as well as its constant recalculation of the input hash. The Rabin-Karp 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.

The Not So Naïve algorithm's [23] 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 III 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 III) but by checking the median processing time - again in Table III - it is clear that the entire set of results is shifted towards a longer processing time.

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

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 7 shows the standard deviations of the mean processing time for each packet each of the algorithms for *Dataset A*. Most algorithms

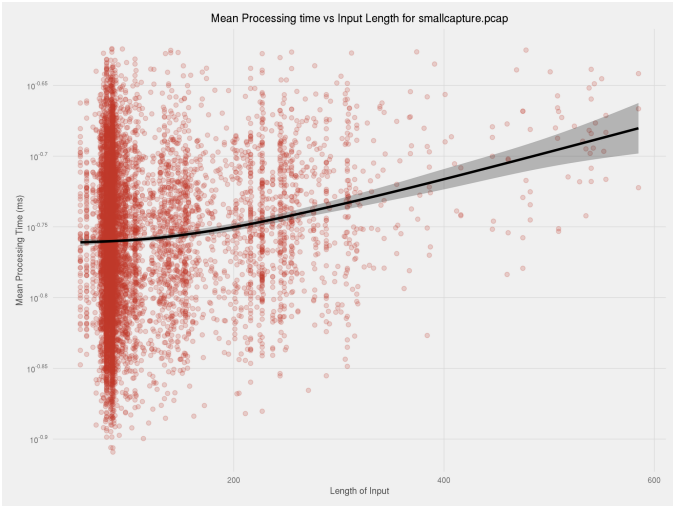


Fig. 2: Input processing speeds vs input length for *Dataset A*.

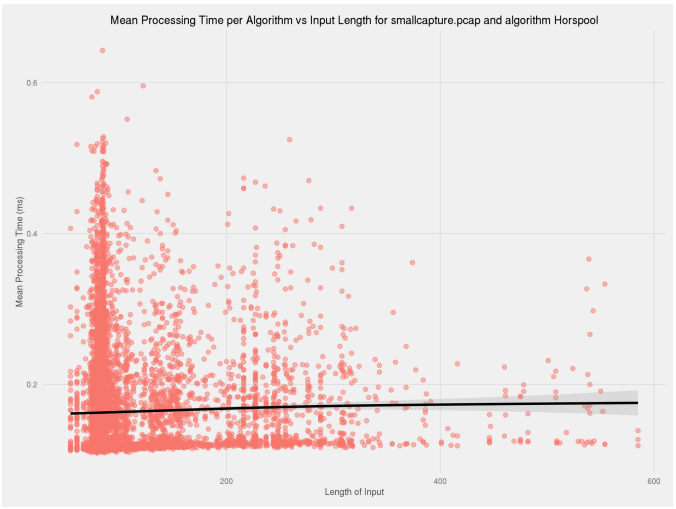


Fig. 3: Horspool

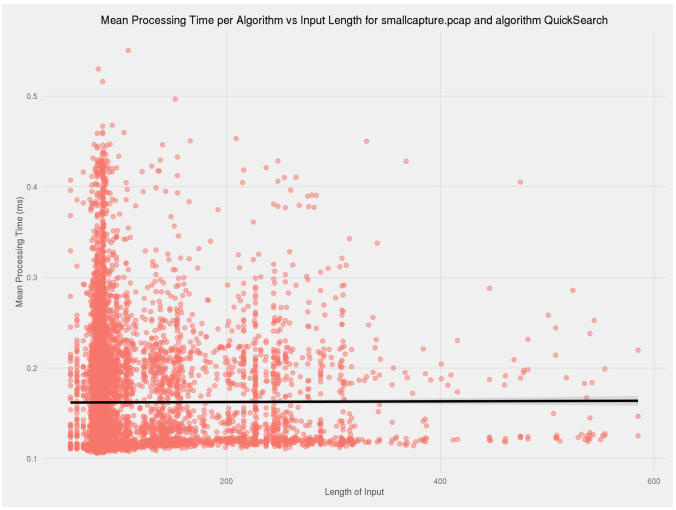


Fig. 4: Quick Search

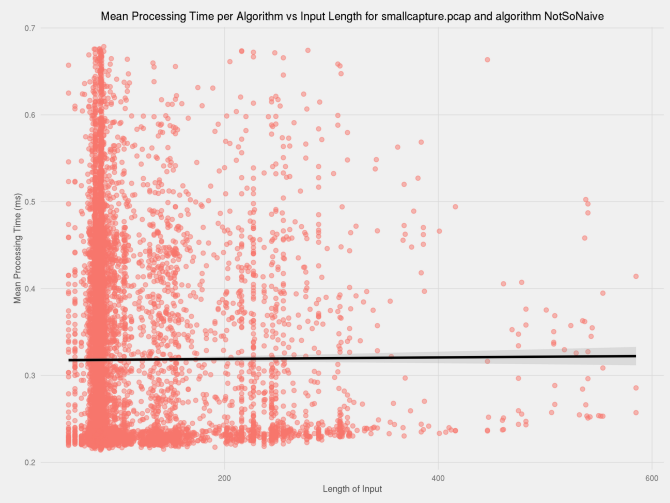


Fig. 5: Not So Naïve

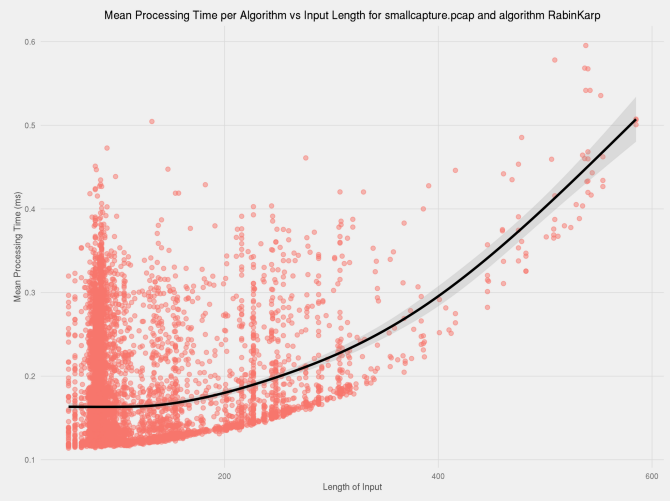


Fig. 6: Rabin-Karp

appear to have relatively low fluctuation on their processing times whilst some are a bit higher. The Horspool [25] algorithm appears to have a very low standard deviation which, combined with its speed shown in Table III, makes it a very strong packet processing algorithm compared to the other algorithms.

Interestingly in Figure 7, the Quick Search algorithm has a notably higher standard deviation than that of the Horspool algorithm. It may not be all that well suited to packet inspection as a result.

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 III) as well as an average packet length of about 110 bytes (Table II) 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.

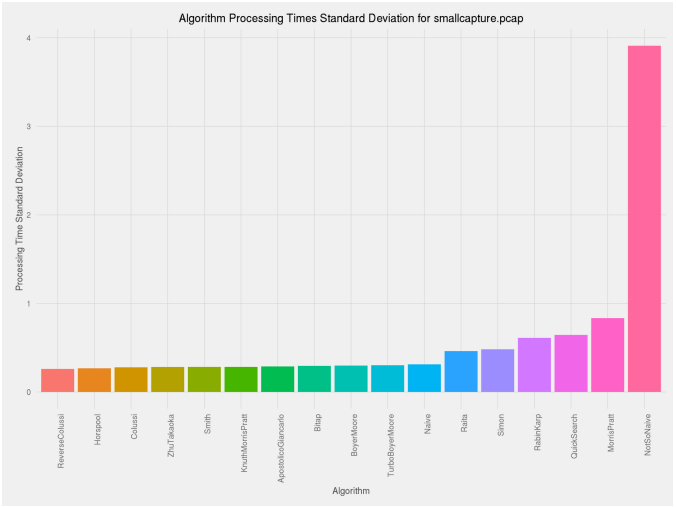


Fig. 7: The standard deviation of processing time per algorithm for *Dataset A*.

V. Conclusion

Of the original algorithms chosen to test, Horspool [25] 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.

VI. Future Work

The two datasets used in these tests are good examples of datasets from their domains. *Dataset A*, which contains DNS data, represents real-world DNS traffic. The drawback of *Dataset A* is that, although of normal length of DNS data, the average length of each packet is very short. The average length of each packet in *Dataset A* is about 110 bytes. The maximum length of an ethernet frame is 1500 bytes [29]. This discrepancy could serve to conceal the true nature of the algorithms.

Further work on this subject could include tests using datasets more akin the the entire spectrum of traffic seen on networks today.

References

[1] "Introducing JSON," Online http://www.json.org/, 02 2016.
[2] "TCPDUMP & LIBPCAP," Online http://www.tcpdump.org/, 02 2016.
[3] "The Oxford English Corpus' Facts about the Language," Online https://www.oxforddictionaries.com/words/the-oec-facts-about-the-language, 02 2016.
[4] "The Top 500 Sites on the Web," Online http://www.alexa.com/topsites, 02 2016.
[5] T. AbuHmed, A. Mohaisen, and D. Nyang, "A Survey on Deep Packet Inspection for Intrusion Detection Systems," *Journal of Korean Communications (Information and Communication)*, vol. 24, no. 11, pp. 25–36, 2007.
[6] J. Anderson, "Computer Secuirty Threat Monitoring and Surveillance," James P. Anderson Company, Fort Washington, Pennsylvania, Tech. Rep., 1980.

[7] A. Apostolico and M. Crochemore, "Optimal Canonization of all Substrings of a String," *Information and Computation*, vol. 95, no. 1, pp. 76–95, 1991.

[8] A. Apostolico and R. Giancarlo, "The Boyer Moore Galil String Searching Strategies Revisited," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 98–105, 1986.

[9] P. Bachman, *Die Analytische Zahlentheorie*. Teubner, 1894, vol. 2.

[10] R. Baeza-Yates and G. Gonnet, "A New Approach to Text Searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[11] R. Baeza-Yates and M. Régnier, "Average Running Time of the Boyer-Moore-Horspool Algorithm," *Theoretical Computer Science*, vol. 92, no. 1, pp. 19–31, 1992.

[12] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[13] C. Charras and T. Lecroq, *Handbook of Exact String Matching Algorithms*, 2004.

[14] A. Chaudhary and A. Sardana, "Software Based Implementation Methodologies for Deep Packet Inspection," in *2011 International Conference on Information Science and Applications*. Jeju Island, Republic of Korea: IEEE, 2011, pp. 1–10.

[15] L. Colussi, "Correctness and Efficiency of Pattern Matching Algorithms," *Information and Computation*, vol. 95, no. 2, pp. 225–251, 1991.

[16] ——, "Fastest Pattern Matching in Strings," *Journal of Algorithms*, vol. 16, no. 2, pp. 163–189, 1994.

[17] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Speeding up Two String-Matching Algorithms," *Algorithmica*, vol. 12, no. 4-5, p. 247, 1994.

[18] B. Daya, "Network Security: History, Importance, and Future," University of Florida Department of Electrical and Computer Engineering, Tech. Rep., 2013.

[19] D. Denning, "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, no. 2, pp. 222–232, 1987.

[20] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," in *2003 IEEE 11th Annual Symposium on High-Performance Interconnects*, D. Azada, Ed. Stanford University, Stanford, California: IEEE, 2003, pp. 44–51.

[21] Z. R. Feng and T. Takaoka, "On Improving the Average Case of the Boyer-Moore String Matching Algorithm," *Journal of Information Processing*, vol. 10, no. 3, pp. 173–177, 1987.

[22] Z. Galil and R. Giancarlo, "On the Exact Complexity of String Matching: Upper Bounds," *SIAM Journal on Computing*, vol. 21, no. 3, pp. 407–437, 1991.

[23] C. Hancart, "Exact Analysis and Average Analysis of String Searching Algorithms," Ph.D. dissertation, Université Paris Diderot, 1993.

[24] C. Hornig, "A standard for the transmission of ip datagrams over ethernet networks," Network Working Group, Tech. Rep., 1984.

[25] R. N. Horspool, "Practical Fast Searching Strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[26] R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[27] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[28] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*. Teubner, 1909, vol. 1.

[29] D. Law, W. Diab, A. Healy, S. Carlson, V. Maguire, O. Anslow, and M. Hajduczenia, "IEEE Standard for Ethernet," IEEE Standards Association, Tech. Rep., 2012.

[30] T. Lecroq, "Experimental Results on String Matching Algorithms," *Software: Practice and Experience*, vol. 25, no. 7, pp. 727–765, 1995.

[31] J. H. Morris and V. R. Pratt, "A Linear Pattern-Matching Algorithm," University of California, Berkeley, Tech. Rep. 40, 1970.

[32] J. Neilsen, "Nielsen's Law of Internet Bandwidth," Online https://www.nngroup.com/articles/law-of-bandwidth/, 1998.

[33] C. Parsons, "Deep Packet Inspection and Its Predecessors," 2012.

[34] T. Raita, "Tuning the Boyer-Moore-Horspool String Searching Algorithm," *Software: Practice and Experience*, vol. 22, no. 10, pp. 879–884, 1991.

[35] 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, pp. 386–388.

[36] D. Smith, P., "Experiments with a Very Fast Substring Search Algorithm," *Software: Practice and Experience*, vol. 21, no. 10, pp. 1065–1074, 1991.

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

[38] D. M. Sunday, "A Very Fast Substring Search Algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132–142, 1990.