On the Performance of String Search Algorithms for Deep Packet Inspection - Followup

Kieran Hunt

March 11, 2016

1 Introduction

In an earlier paper, Hunt (2016) compares various string search algorithms for use in Deep Packet Inspection (DPI). This paper serves as an extension to that work; it follows up on a few select algorithms.

From that earlier paper the following algorithms were selected: Horspool (Horspool, 1980), QuickSearch (Sunday, 1990), NotSoNaive (Hancart, 1993), and RabinKarp (Karp and Rabin, 1987).

For each of the selected algorithms the following questions were posed:

- How does the length of a packet's payload affect the speed at which that packet is processed?
- How does the number of matches found within a packet affect the speed at which that packet is processed?
- How does the number of threads used when performing the matching affect the speed at which a packet is processed?

From these questions the following paper has been created.

1.1 Algorithms

In (Hunt, 2016), the authors compare a number string search algorithms against each other. From their results we have chosen the two best and two worst performing algorithms. The two best performing algorithms - based on their mean processing times - were the Horspool (Horspool, 1980) and QuickSearch (Sunday, 1990); the two worst performing - based on the same criterion - were NotSoNaive (Hancart, 1993) and RabinKarp (Karp and Rabin, 1987). Table 1 presents these algorithms along with the year they were published, their authors, and the time complexity associated with each one.

Algorithm	Year	Author(s)	Time Complexity
Horspool	1980	Horspool	O(n + m)
RabinKarp	1987	Karp and Rabin	O(mn)
QuickSearch	1990	Sunday	O(mn)
NotSoNaive	1993	Hancart	O(nm)

Table 1: String search algorithms. Under time complexity, *n* represents the length of the input and *m* represents the length of a rule.

1.2 Datasets

Table 2 presents the datasets used throughout the rest of the paper. In Table 2, 'μ Length' refers the mean length of the inputs, in bytes, within that dataset and '# Inputs' refers to the number of inputs in that dataset. An input is defined as a Both *Dataset A* and *Dataset B* have been reused from the earlier datasets used in Hunt (2016).

Name	Description	μ Length	# Inputs
Dataset A	Real-world DNS traffic	109.61	10000
Dataset B	Full text of *Alice in Wonderland* by Lewis Carroll	163780	1
Dataset C	Randomly generated DNS traffic with a payload size between 0 and 1500 bytes	770.89	10000
Dataset D	Dataset C edited so that the payload just contains matches to the required rules	770.89	10000
Dataset E	Dataset C edited so that each packet is filled with a random number of matches	769.92	10000
Dataset F	Packets of fixed length - filled with a random number of matches	1500	10000

Table 2: All datasets used in the tests

Dataset C was created using Wireshark's[1] `randpkt` tool (ran, 2016). Listing 1 shows how that Dataset was created.

```
$ randpkt -b 1500 -c 10000 -t dns random_dns.pcap
```

Listing 1: Creating 10000 random DNS packets

The packets in *Dataset C* had an average length of 770 bytes as can be seen in Table 3. As *Dataset C* was completely comprised of random bytes, the number of matches within it was near zero. This dataset was useful when comparing processing speed with input length as the number of matches within each packet would not affect the processing speed.

Min	First Quartile	Median	Mean	Third Quartile	Max
44	407	763.5	770.9	1136	1500

Table 3: A summary of the input lengths in *Dataset C*

Dataset D was a modification of *Dataset C*. To make *Dataset D*, *Dataset C* was edit such that each packet's payload was combination of each the rules. For each packet, a list of the rules would have its order randomised, the list would then be turned into a string and then string multiplied until its length was greater than that of the packet's payload, the string would then be truncated to the length of the packet's payload and the packet would be edited so that its payload is that string. The result of these changes to *Dataset C* means that each packet - although of random length - is completely filled with matches to the rules used in the tests.

Dataset E was also a modification of *Dataset C*. Unlike *Dataset D*, *Dataset E* contains a random amount of matches to the rules and the rest of the space is filled with random bytes. The construction of *Dataset E* is similar to that of *Dataset D*. Prior to changing the packet's payload to the string of matches

[1] https://www.wireshark.org/

a random number is calculated. That random number is between zero and the length of the packet's payload. The string of matches is then truncated to the length of that random number, the packet's payload is updated to contain that string and then the subsequent bytes from the random number to the original length of the packet's payload is filled randomly.

Dataset F was constructed as a way to isolate the number of matches from the length of the packet's payload. For this dataset all packets were created with a length of 1500 bytes. Like with *Dataset E*, each packet would be filled with a random number of matches and the rest of the space in the payload would be filled with random bytes. The result is a dataset with packets of 1500 bytes in length and a random number of matches within each.

2 Processing Speed vs Input Length

In Hunt (2016), the Dataset used by the authors (*Dataset A*) contains matches to the rules being searched for. For some algorithms the number of matches may affect the processing speed of each packet. Furthermore, *Dataset A*'s shortfall is that - because it contains real-world DNS data - the average length of a packet is very short (See Table 2). *Dataset C* was created for for the purpose of this test as it contains packets of much longer length with zero matches to the rules used in the tests.

For this test the processing speed of each packet is now independent of the number of matches as the number of matches has been reduced to zero.

Each of the four algorithms were tested for the same rules as in Hunt (2016) and with *Dataset C* as input. Figure 1 shows a comparison between each of the algorithms.

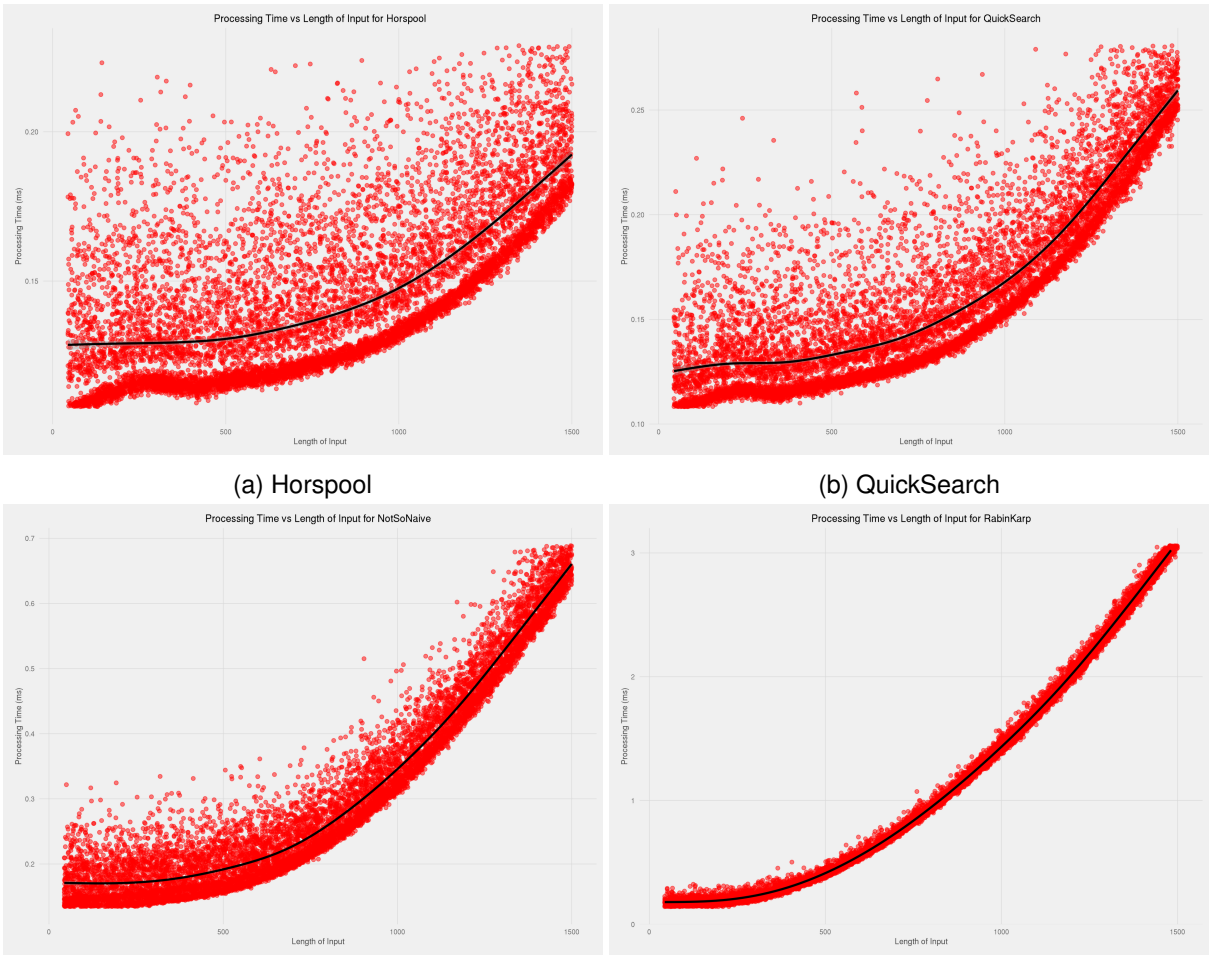
From Figure 1, it is clear that the processing times for Horspool and QuickSearch (Subfigures 1a and 1b respectively) do not rise as quickly as those of NotSoNaive and RabinKarp (Subfigures 1c and 1d respectively) for inputs of greater length. Both NotSoNaive and RabinKarp show an exponential rise in processing times. Another point to note is that the maximum processing time for NotSoNaive is almost 3 times that of Horspool and QuickSearch, and the maximum processing speed of Horspool is almost 5 times that of NotSoNaive.

Horspool and QuickSearch also show initial humps in processing speed around the 250 byte mark. This can be attributed to the overheard associated with each packet. For packets of more than 250 bytes, the processing speed relies more on the length of the packet then the initial - per-packet - overhead. This initial hump is not evident on either NotSoNaive nor RabinKarp; the reason for this is twofold. First, the speed of these algorithms is more affected by the input length, because of this an initial hump would be surpassed by the time it takes to process each packet. Second, the vertical scale of these two graphs is much larger than that Horspool or QuickSearch. Owing to that a small hump earlier on is less visible on a larger scale.

3 Processing Speed vs Number of Matches

For Section 2, *Dataset C* was created. It was designed in such a way that the number of matches in the input would not affect the processing speed. The processing speed was thus only affected by the length of the input and the algorithm used. In order to compare the processing speed against different numbers of matches, another dataset needed to be created. *Dataset F* was created in such a way that the processing speed would only be affected by the number of matches and not the length of the packet. Each packet in this data set is 1500 bytes long.

Figure 2 presents the results for each algorithm of a comparison between the processing speed vs the number of matches in the packet.



(a) Horspool

(b) QuickSearch

(c) NotSoNaive

(d) RabinKarp

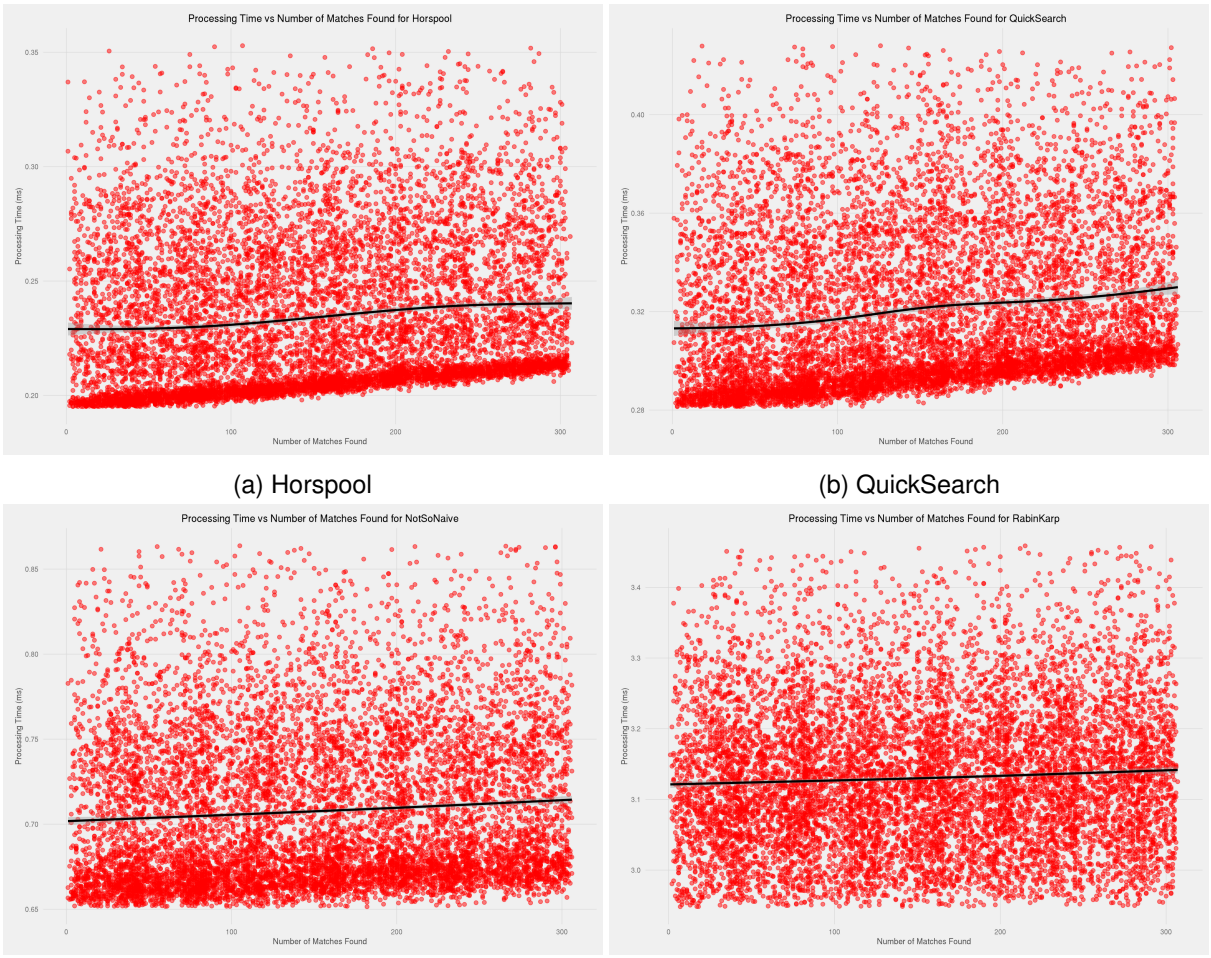
Figure 1: Mean processing times versus input length for the Horspool, QuickSearch, NotSoNaive, and RabinKarp algorithms for *Dataset C*.

As can be seen in Figure 2 the processing speed of both NotSoNaive and RabinKarp seem to not be affected by the number of matches in each packet whereas the number of packets does seem to affect the processing speeds of the Horspool and QuickSearch algorithms. These results make sense.

The NotSoNaive and RabinKarp algorithms incur little computational penalty when comparing against a substring that matches a rule compared with one that does not. Both the Horspool and QuickSearch algorithms make use of BoyerMoore's (Boyer and Moore, 1977) bad-character shift table. This table is most effective when few or no matches are present and it allows the algorithms to skip forward over characters which cannot form part of a match. Because *Dataset F* contains so many matches bad-character shift table does not prove to be very useful.

4 Processing Speed vs Number of Threads

In order to test how each of the algorithms performs under different levels of parallelism, a simple set of tests was devised. *Dataset C* was selected as the input for the tests as it would not contain any matches to the rules used. The configuration file mentioned in Hunt (2016) allows the number of threads to be



(a) Horspool

(b) QuickSearch

(c) NotSoNaive

(d) RabinKarp

Figure 2: Mean processing times for each algorithm vs number of matches. *Dataset F*.

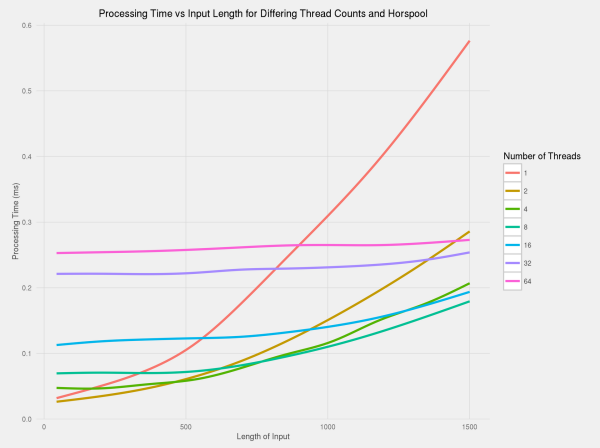
specified. The test hardware has a total of 24 cores available and the test searches for each of the rules in parallel. In this test's case there were 40 rules. The following numbers of threads were selected to be tested: 1, 2, 4, 8, 16, 32, 64.

An overall speed increase should take place for thread numbers between one and sixteen. This is still below the twenty four cores available and so each thread can occupy a separate core. At thirty two threads there are now more threads than available cores and so threads have to share processing time on cores. This could lead to further overhead as the processor switches contexts depending on which thread has been scheduled.

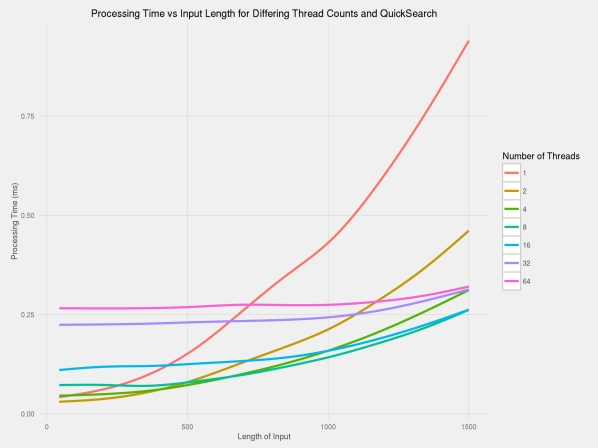
Figure 3 provides some very interesting results. As expected, single core implementations are the slowest at longer input lengths and speed increases for more cores are realised as the length of the packets grow. This result is unsurprising as longer packets mean that the overhead of switching threads is diminished as more of the processing time is spent on the searching and less on switching between threads. Also as expected, the lower thread counts are more efficient for smaller packets but they lose efficiency as the length increases.

On our particular test hardware, between eight and sixteen threads proves to be the most efficient for longer packet lengths. Table 4 shows the outcome of the comparison at various input lengths.

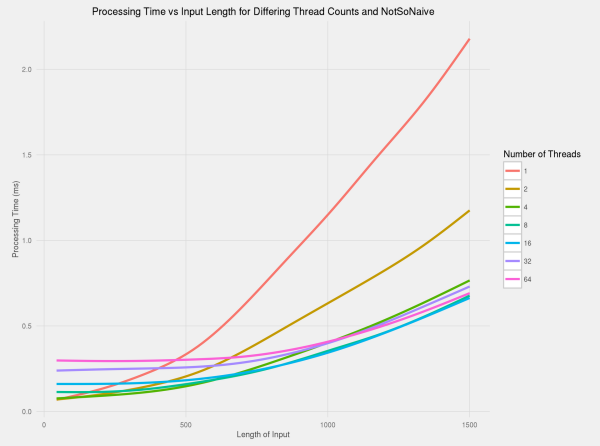
Table 4 shows that for shorter inputs, fewer threads are more efficient. Furthermore, as is apparent



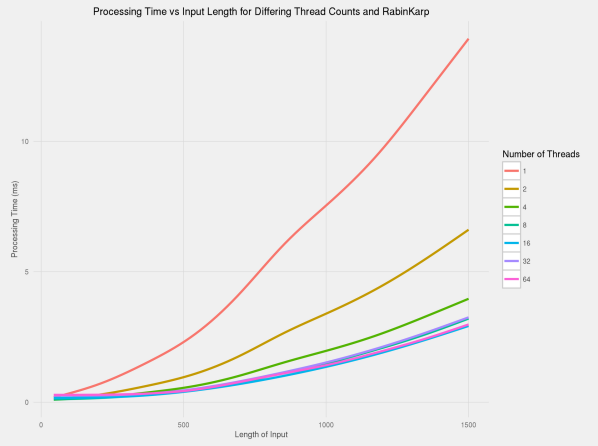
(a) Horspool



(b) QuickSearch



(c) NotSoNaive



(d) RabinKarp

Figure 3: Mean processing times for each algorithm vs number of matches. *Dataset D*.

	25%	50%	75%
Horspool	2	4-8	8
QuickSearch	2	8	8
NotSoNaive	4	8	16
RabinKarp	16	16	16

Table 4: Most efficient number of threads at various percentages of 1500 bytes for each algorithm

in both Figure 3 and Table 4, the faster algorithms tend to be more efficient at lessor numbers of threads for the same input lengths as the slower algorithms.

References

March 2016. URL `https://www.wireshark.org/docs/man-pages/randpkt.html`.

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

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

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

K. R. Hunt. On the performance of string search algorithms for deep packet inspection. *who knows*, 2016.

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

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