On the Performance of String Search Algorithms for Deep Packet Inspection

Kieran Hunt

February 24, 2016

1 Introduction

In Deep Packet Inspection today, systems are generally built on top of expensive custom hardware. Making changes to such a system (such as horizontally scaling) is often arduous, time consuming and expensive. Deep Packet Inspection via software means is usually slower but does provide some benefit: adding or removing capacity to perform inspection is often as simple as adding or removing hosts doing the inspection. String search algorithms have long been of interest to the field of computer science and as a result a substantial number of search algorithms exist. This paper looks to ask which of these string search algorithms perform best at deep packet inspection and how does their performance compare to that of their intended design.

Relevant Deep Packet Inspection terminology is as follows:

- **Packet**: Data representing a TCP/IP stack packet. This includes information from the network to the application layer.
- **Packet Capture File (PCAP)**: A file containing packets. These packets were generally captured by recording a network interface.
- **Deep Packet Inspection (DPI)**: the process by which a packet, or stream of packets, is analysed for the presence of predefined patterns.

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.

In the system for comparing string search algorithms, the following terminology is relevant:

- **Input**: This is the interface through which the system reads in either the packets or textual Input. Algorithms can request a single byte from the input, the length of the Input or the entire Input itself.
- **Rule**: The system searches through the input for a given Rule. Rules are very similar to input in that a single byte of information, the length of the rule or the entire rule itself can be requested.
- **Algorithm**: This represents a string search algorithm and is the means by which the system interacts with all of the algorithms. It has a single interface for performing a search where the Input is supplied and a Result is returned.

- **Result**: This represents the result of the inspection of a single Input. In it is the start and end times of the inspection, the Rules, Input and the location (if any) in the Input where the Rules were matched.

Each of the string search algorithms have known performance (algorithm complexity known as big O) usually related to the length of the string being searched. The results of the following experiments should follow the predicted complexity of string search algorithms. Algorithm complexity often only provides insight into large variations in Input length (differing orders of magnitude) whereas in packet data has a limited range of Input lengths and so it may come down to minutiae within the algorithms themselves rather than their overall complexity.

2 String Search Algorithms

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

The time complexities in Table 1 should only serve as a basic indication of the speed of an algorithm. Big-O notations generally strip off any factors and so two algorithms may appear to

have the same time complexities but in practice their speeds differ greatly because of this unknown factor.

Another point to note is that because these algorithms were designed to search for a single Rule, their Big-O notations only reflect that property. For multiple rules, and a serial-style approach, these algorithms need to run back-to-back; this increases increases their time to process by a factor equal to the number of Rules. For a parallel-style approach searches for different rules using the same algorithm can be run at the same time. The speed at which a search for a complete set of Rules 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 exists. That upper bound is defined by the number of processor cores that are available for use.

3 Method

As mentioned previously, a system was developed to allow accurate running and performance measurement of each of the implemented algorithms (in Table 1). The system takes a json file as input (See Listing 1). 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) - 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`. A second Input was also selected, this is the complete *Alice in Wonderland* by Lewis Carol. This Input was labeled `alice.txt`. The system treats text and pcap input files differently. Text input files are used to create 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` - the tests were each run 20 times.
- `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.

Listing 1: Sample testConfiguration.json

```
{
  "algorithms": [
    "Naive",
    "MorrisPratt",
    ...
    "ReverseColussi"
  ],
  "rules": [
    "time",
    "person",
```

```
      ...
      "msn"
    ],
    "inputs": [
      {
        "type": "pcap",
        "location": "smallcapture.pcap"
      },
      {
        "type": "text",
        "location": "alice.txt"
      }
    ],
    "times": 20,
    "threadCount": 18
}
```

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 statistics.

4 Results

The test generated 3400340 result objects. Each result object looked very similar to Listing 2. Containing information about the start and end times (in nanoseconds), the elapsed time (the end time - the start time), the rules searched for, the locations that rules were matched at, the algorithm used, the input file, the Input ID, the run number and the ID of that run.

Listing 2: Sample test Result

```
[
    {
        "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"
```

```
    },
    ...
]
```

5 Analysis

5.1 Which Algorithms are Fastest? Which are Slowest?

Figure 1a shows a comparison of processing times for each algorithm over each of the packets in PCAP dataset. Figure 1b gives the same comparison albeit for the *Alice in Wonderland* input file.

Taking into account the processing speed of each algorithm from both Figure 1a and 1b we are able to list the algorithms in order of their processing speed - fastest first: Horspool, Quicksearch, Raita, ReverseColussi, BoyerMoore, ZhuTakaoka, Smith, Colussi, KnuthMorrisPratt, TurboBoyer-Moore, ApostolicoGiancarlo, MorrisPratt, Naive, Simon, Bitap, NotSoNaive, RabinKarp.

Although the order of speediness 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 quite small. By increasing the length of the input data (in this case by using the *Alice in Wonderland* text. Figure 1b) we are able to accentuate the differences.

Algorithm	Minimum	First Quartile	Median	Mean	Third Quartile	Max
QuickSearch	0.08321	0.10750	0.11430	0.16350	0.12320	259.70000
Horspool	0.0926	0.1098	0.1166	0.1640	0.1246	5.3770
ReverseColussi	0.09139	0.11180	0.11910	0.16410	0.12740	5.70000
Raita	0.09035	0.10710	0.11600	0.16560	0.12440	164.10000
ZhuTakaoka	0.09288	0.11200	0.11900	0.16560	0.12740	4.64500
KnuthMorrisPratt	0.09113	0.11180	0.11930	0.16630	0.12780	25.96000
Colussi	0.09178	0.11130	0.11840	0.16640	0.12670	29.69000
Smith	0.09102	0.11170	0.11880	0.16710	0.12660	5.33400
BoyerMoore	0.09287	0.11070	0.11740	0.16730	0.12540	33.22000
MorrisPratt	0.08543	0.10910	0.11540	0.16750	0.12360	297.80000
Naive	0.08845	0.11180	0.11830	0.16820	0.12620	63.84000
Bitap	0.09277	0.11160	0.11900	0.16840	0.12780	50.14000
TurboBoyerMoore	0.09101	0.11220	0.11900	0.16880	0.12680	41.68000
RabinKarp	0.09562	0.11580	0.12230	0.17070	0.13200	245.60000
Simon	0.08248	0.11140	0.11890	0.17320	0.12750	117.20000
ApostolicoGiancarlo	0.09368	0.11420	0.12170	0.17340	0.13020	5.06700
NotSoNaive	0.1092	0.1698	0.2277	0.3806	0.2957	552.3000

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. NotSoNaive's (Hancart, 1993) slowness is surprising at first. It performs worse than the Naive algorithm when it is supposed to be an improvement on it. The NotSoNaive algorithm tries to improve upon the Naive algorithm by adding a 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. By checking 3, you may think that the mean processing time for NotSoNaive was heavily influenced by the maximum speed (that

is to say that that value is an outlier) but by checking the median processing it is clear that the mean is not far from it.

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 rule (Lecroq, 1995) and since rules used in the experiment were mostly short, it performed well. The Horspool (Horspool, 1980) algorithm also makes use of the bad-character shift table from BoyerMoore (Boyer and Moore, 1977). Although the algorithm has a O(mn) complexity, 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 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) and so the average number of comparisons to a character in our text is $\frac{1}{256} \leq \alpha \leq \frac{2}{257}$. That is a very low number of comparisons per input character.

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

Figure 2a shows a scatter plot of Input processing speed versus packet length. This scatter 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.

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.

5.3 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.

5.4 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 6 megabits per second is achievable. This speed is obviously much less than what would be required of a modern Deep Packet Inspection system. The goal of this research is to develop an understand of the algorithms' relative speed.

6 Conclusion

References

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

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

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.

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.

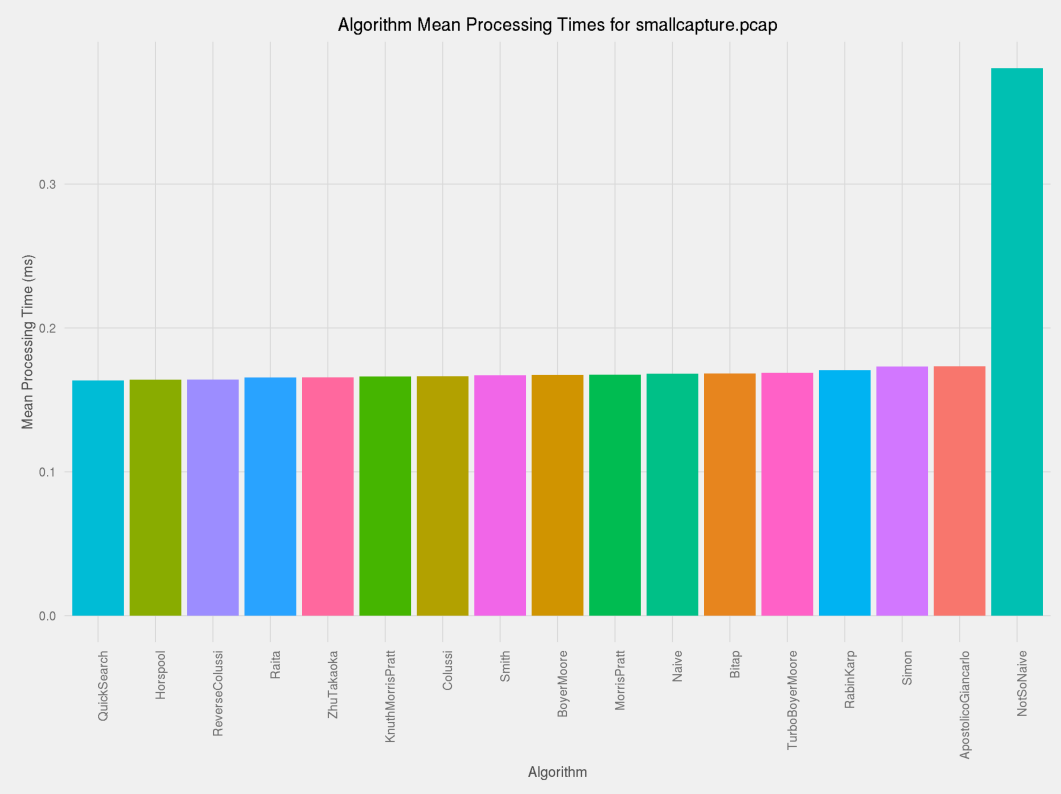
J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

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

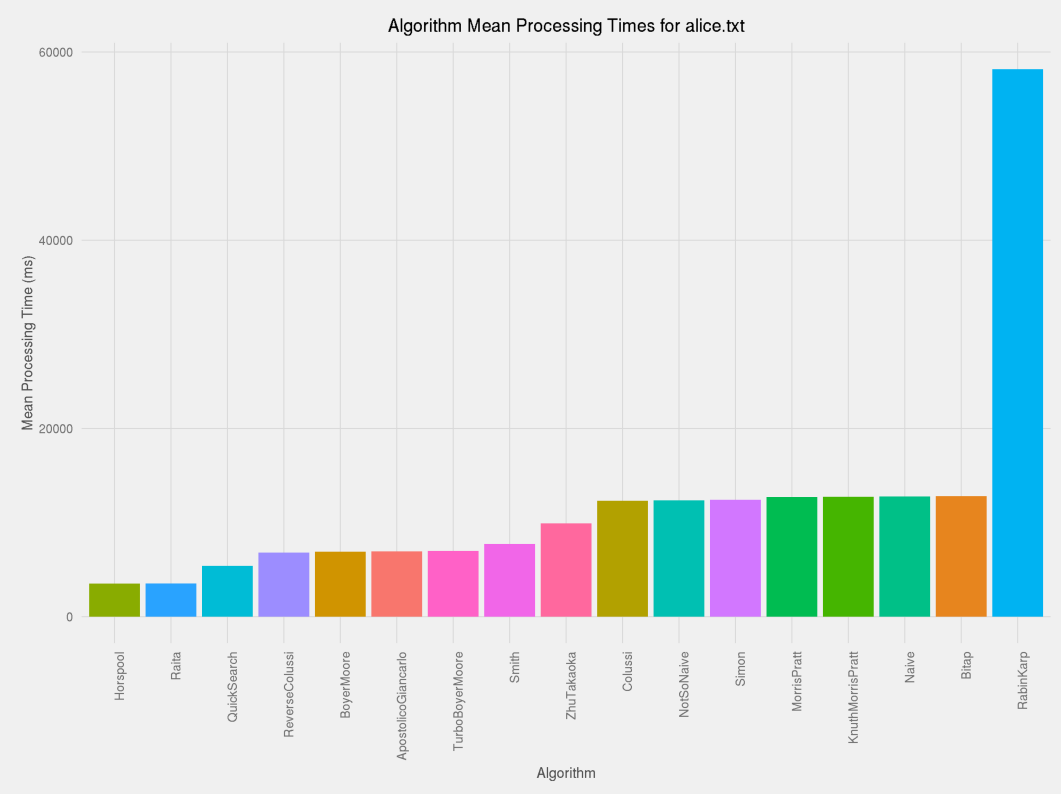
I. Simon. String matching algorithms and automata. In *Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science*, 1994.

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.

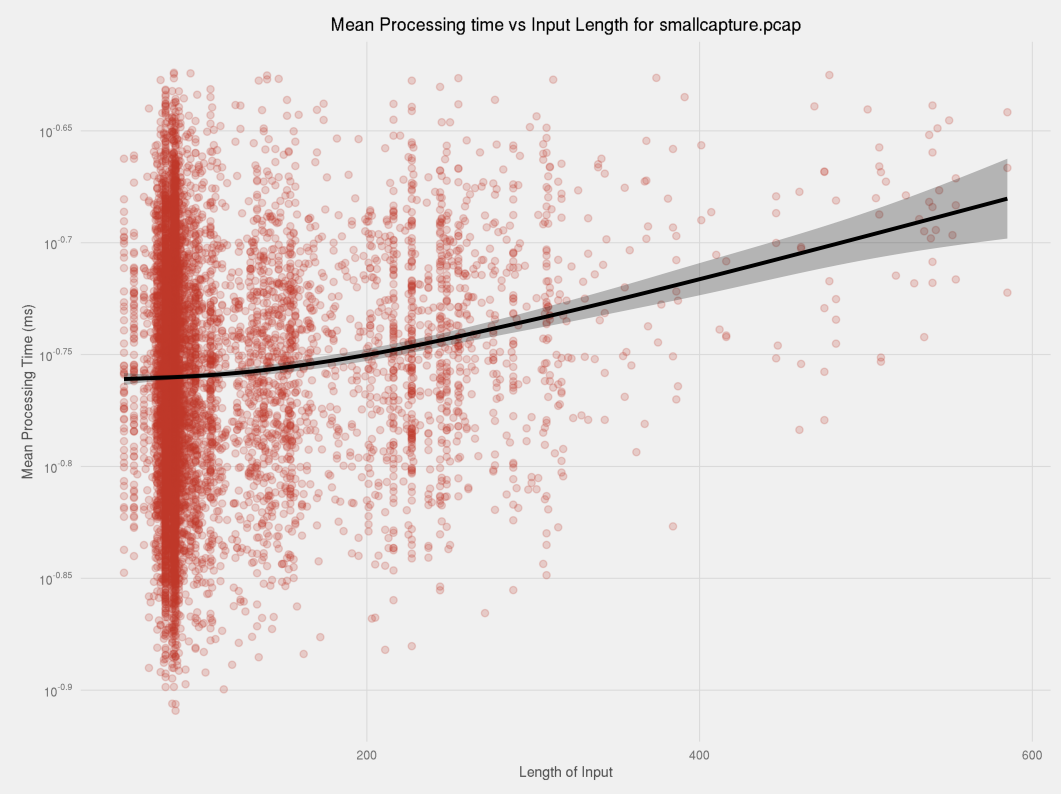


(a) Mean Input processing times of each Algorithm for *PCAP Dataset*.

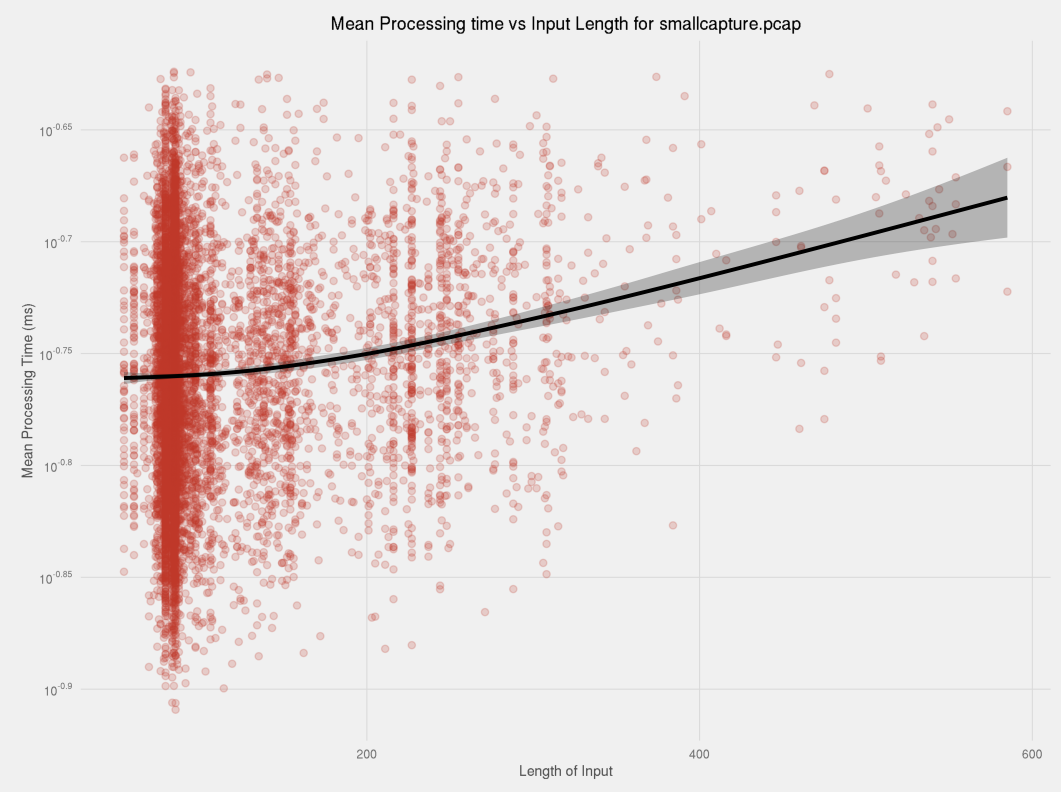


(b) Mean Input processing times of each Algorithm for *Alice in Wonderland*.

Figure 1: Comparison of the mean processing times for both the packet and textual datasets.



(a) Input processing speed vs Input length for *Dataset A*.



(b) Input processing speed per algorithm vs Input length for *Dataset A*.

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