E^2XB: A DOMAIN-SPECIFIC STRING MATCHING ALGORITHM FOR INTRUSION DETECTION

K. G. Anagnostakis*, S. Antonatos, E. P. Markatos, M. Polychronakis†
Institute for Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH)
P.O. Box 1385 - Heraklio, Crete, GR-711-10 GREECE
{kanag,antonat,markatos,mikepo}@ics.forth.gr

Abstract We consider the problem of string matching in Network Intrusion Detection Systems (NIDSes). String matching computations dominate in the overall cost of running a NIDS, despite the use of efficient general-purpose string matching algorithms. Aiming at increasing the efficiency and capacity of NIDSes, we have designed E^2xB, a string matching algorithm that is tailored to the specific characteristics of NIDS string matching. We have implemented E^2xB in `snort`, a popular open-source NIDS, and present experiments comparing E^2xB with the current best alternative solution. Our results suggest that for typical traffic patterns E^2xB improves NIDS performance by 10%-36%, while for certain ruleset and traffic patterns string matching performance can be improved by as much as a factor of three.

Keywords: network security, intrusion detection, string matching, network monitoring, network performance

1. Introduction

Network Intrusion Detection Systems (NIDSes) are receiving considerable attention as a mechanism for shielding against "attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer network" (2). The typical function of a NIDS is based on a set of *signatures*, each describing one known intrusion threat. A NIDS examines network traffic and determines whether any signatures indicating intrusion attempts are matched.

The simplest and most common form of NIDS inspection is to match string patterns against the payload of packets captured on a network link. The use

*Author is with the CIS Department, University of Pennsylvania, Email: *anagnost@dsl.cis.upenn.edu*
†Authors are also with the Computer Science Department, University of Crete

of existing efficient string matching algorithms for this purpose, such as (3, 1), bears a significant cost: recent measurements of the `snort` NIDS (13)on a production network show that as much as 31% of total processing is due to string-matching (6). The same study also reports that in the case of Web-intensive traffic, this cost is increased to as much as 80% of the total processing time. At the same time, NIDSes need to be highly efficient to keep up with increasing link speeds, and, as the number of potential threats (and associated signatures and rules) is expected to grow, the cost of string matching is likely to increase even further.

These trends motivate the study of new string matching algorithms tailored to the particular requirements and characteristics of Intrusion Detection, much like domain-specific algorithms were developed for efficient routing lookups and packet classification in IP forwarding (10, 7).

In this context, we present E^2xB, a string matching algorithm that is designed specifically for the relatively small input size (in the order of packet size) and small expected matching probability that is common in a NIDS environment. These assumptions allow string matching to be enhanced by first testing the input (e.g., the payload of each packet) for *missing* fixed-size sub-strings of the original signature string, called *elements*. The false positives induced by E^2xB, e.g., cases with all fixed-size sub-strings of the signature showing up in arbitrary positions within the input, can then be separated from actual matches using standard string matching algorithms, such as the Boyer-Moore algorithm (3). Experiments with E^2xB implemented in `snort`, show that in common cases, E^2xB is more efficient than existing algorithms by up to 36%, while in certain scenarios, E^2xB can be three times faster. This improvement is due to an overall reduction in executed instructions and, in most cases, a smaller memory footprint than existing algorithms.

2. Background

The general problem of designing algorithms for string matching is well-researched. One of the most widely used algorithms was first proposed in (3). The Boyer-Moore algorithm compares the search string with the input starting from the rightmost character of the search string. This allows the use of two heuristics that may reduce the number of comparisons needed for string matching (compared to the naive algorithm). Both heuristics are triggered on a mismatch. The first heuristic, called the *bad character heuristic*, works as follows: if the mismatching character appears in the search string, the search string is shifted so that the mismatching character is aligned with the rightmost position at which the mismatching character appears in the search string. If the mismatching character does not appear in the search string, the search string is shifted so that the first character of the pattern is one position past the mismatching character in the input. The second heuristic, called the *good suffixes heuristic*, is also triggered on a mismatch. If the mismatch occurs in the middle of the search string, then there is a non-empty suffix that matches. The heuristic then shifts the search string up to the next occurrence of the suffix in the string. Horspool (1980) improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic.

Aho and Corasick (1975) provided an algorithm for concurrently matching multiple strings. The set of strings is used to construct an automaton which is able to search for all strings concurrently. The automaton consumes the input one character at-a-time and keeps track of patterns that have (partially) matched the input.

Fisk and Varghese (2002) were the first to consider the design of NIDS-specific string matching algorithms. They proposed an algorithm called Set-wise Boyer-Moore-Horspool, adapting the Boyer-Moore algorithm to simultaneously match a set of rules. This algorithm is shown to be faster than both Aho-Corasick and Boyer-Moore for medium-size pattern sets. Their experiments suggest triggering a different algorithm depending on the number of rules: Boyer-Moore-Horspool if there is only one rule; Set-wise Boyer-Moore-Horspool if there are between 2 and 100 rules, and Aho-Corasick for more than 100 rules. This heuristic has been incorporated in `snort` and provides the baseline for our comparison in Section 4. Independently of Fisk and Varghese, Coit et al. (2002) implemented a similar algorithm in `snort`, adapting Boyer-Moore for simultaneously matching multiple strings, derived from the exact set matching algorithm of Gusfield (1977) .

Recently, we have proposed ExB, a precursor of E^2xB, providing quick negatives when the search string does not exist in the packet payload (11). E^2xB provides several improvements on ExB , the most important being a faster pre-processing phase, removing much of the overhead associated with initializing the occurrence map, and a wider set of experiment results, that also highlight NIDS properties that are interesting beyond the scope of the specific algorithm.

3. E^2xB: Exclusion-based string matching

We present an informal description of E^2xB, first in its simplest and most intuitive form and then in its more general form. E^2xB is based on the following simple observation:

> Suppose that we want to check whether an input string I contains a small string s. If there exists at least one character of string s that is not contained in I, then s is not a substring of I.

The above simple observation can be used to quickly determine several cases where a given string s does *not* appear in the input string I: **if s contains at least one character that is not in I, then s is not a substring of I.** However, this observation cannot be used to determine the cases where s *is* a substring of I. Indeed, if every character of string s belongs to input string I, then we should use a standard string matching algorithms (e.g., Boyer-Moore-

Horspool) to confirm whether s is actually a substring of I or not. The cases where every character of s is in I, but s is not a substring of I are called *false matches*, or *false positives*.

This method is effective only if there is a fast way of checking whether a given character c belongs in I or not. We perform this check with the help of an *occurrence map*. Specifically, we first *pre-process* the input string I, and for each (8-bit) character c that appears in string I, we mark the corresponding (i.e. c_{th}) *cell* on the (256-cell) map. Although we could use a binary value to mark the mentioned cells (i.e. if the c_{th} position of the cell map is 1, then the character c appears in I, otherwise it does not), our experiments in (11)suggest that the cost of cleaning (i.e. filling with zeros) the cell map for each new packet can be very high. To reduce this cost, we decided to mark the cell with the (index) number of the current network packet. Thus, if the c_{th} position of the cell map contains the number of the current network packet, the character c appears in I, otherwise it does not [1].

In order to reduce the percentage of false matches, the above algorithm can be generalized for *pairs* of (8-bit) characters: instead of recording the occurrence of single characters in string I, it is possible to record the appearance of each *pair* of consecutive characters in string I. In the matching process, instead of determining whether each character of s appears in I, the algorithm then checks whether each pair of consecutive characters of s appears in I. If a pair is found that does not appear in I, E^2xB knows that s is not in I.

Generalizing further, instead of using 8-bit characters, or 16-bit pairs of characters, E^2xB can use bit-strings of arbitrary length (hereafter called *elements*). That is, E^2xB records all (byte-aligned) bit-strings of length x. The element size exposes a trade-off: larger elements are likely to result in fewer false matches, but also increase the size of the occurrence map, which could, in turn, increase capacity misses and degrade performance.

The pseudo-code for pre-processing `input` and for matching a string `s` on `input` is presented in Figure 1.

The main difference between E^2xB and ExB is the use of cells: ExB assumed an occurence *bitmap* where each element was marked by setting the 1-bit cell to 1. This required the bitmap to be cleared for each packet, adding unnecessary overhead. A second difference lies in the way the two bytes forming an element are *hashed* together. E^2xB uses OR while ExB uses XOR. Although in theory XOR does provide a better hash than OR, the difference in the number of collisions was found to be negligible. The value of using XOR lies more in that XOR instructions were found to result in slightly better performance. Finally, an important implementation detail that has been addressed in E^2xB is support for *case-insensitive matching*, as many NIDS signatures

[1]To reduce the number of bits needed to store the cell map, the numbers of network packets are limited to a predefined number of bits, which we call *cell_size*. If the number of network packets exceed 2^{cell_size}, then the next packet gets the number 0.

```
pre_process(char *input, int len)
{
  pktid=pktno & (1<<cellsize - 1);

  for (int idx = 0 ; idx < len-1 ; idx++) {
       element = s[idx]<< ( elementsize-8 ) ^ s[idx+1];
       occurence_map [ element ] = pktid;
  }
}
search(char *s, char *input, int len_s, int len)
{
  for (int idx = 0 ; idx < len_s-1 ; idx++) {
      element= s[idx]<< ( elementsize-8 ) ^ s[idx+1];
      if ( occurence_map [ element ] != pktid)
          return DOES_NOT_EXIST ;
  }
  return boyer_moore(s, len_s, input, len);
}
```

Figure 1. Pseudo-code for E^2xB pre-processing and search.

are case-insensitive. This is done by modifying the *search* procedure to test for the occurence of all four combinations of upper- and lower-case for each of the two bytes used to compute the element index.

4. Experimental evaluation

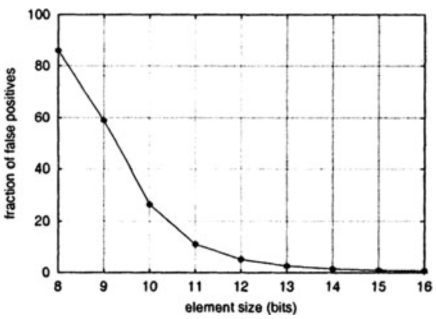
Using trace-driven execution, we evaluate the performance of E^2xB against the heuristic of (6)(denoted as FVh in the rest of this paper) and the implementation of (3)in `snort` .

4.1 Environment

For all the experiments we used a PC with a Pentium 4 processor running at 1.7 GHz, with a L1 cache of 8 KB and L2 cache of 256 KB, and 512 Mbytes of main memory. The measured memory latency is 1 ns for the L1 cache, 10.9 ns for the L2 cache and 170.4 ns for the main memory , measured using `lmbench` (12). The host operating system is Linux (kernel version 2.4.14, RedHat 7.3). We use `snort` version 1.9.0 (build 205) compiled with `gcc` version 2.96.

Each packet is checked against the "default" rule-set of the `snort` distribution. The ruleset is organized as a two-dimensional chain data-structure, where each element - called a *chain header* - tests the input packet against a packet header rule. When a packet header rule is matched, the chain header points to a set of signature tests, including payload signatures that trigger the execution of the string matching algorithm. The default rule-set consists of 187 chain headers with a total of 1661 rules, 1575 of which are string matching rules.

We use packet traces from four different sources:



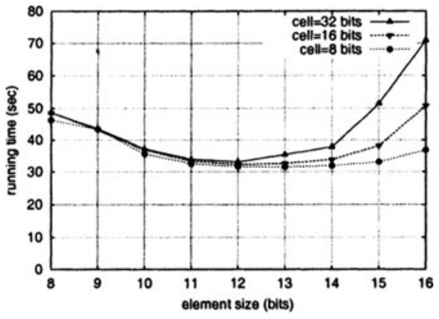


Figure 2. Effect of element and cell size parameters on the fraction of false positives

Figure 3. Effect of element and cell size parameters on running time

- A set of full-packet traces from the DEFCON "capture the flag" dataset. [2] These traces contain numerous intrusion attempts.
- A full packet trace containing Web traffic, generated by concurrently running a number of recursive `wget` requests on popular portal sites.
- Three header-only traces from the NLANR archive. These packet traces were taken on backbone links. Because these are header-only traces, for our experiments we added random payloads. We argue that the results are representative after determining that random payloads do not significantly alter NIDS performance.
- A set of header-only traces collected on the OC3 link connecting the University of Crete campus network (UCNET) to the Greek academic network (GRNET)(5), with random payloads.

For the experiments of Sections 4.2 and 4.3, we use the DEFCON `eth0.dump2` trace containing 1,035,736 packets. For simplicity, traces are read from a local file by using the appropriate `snort` option, which is passed to the underlying `pcap(3)` library. (Replaying traces from a remote host provided similar results.)

4.2 Element and cell size

We first determine the optimal size for E^2xB elements and cells. In Figure 2 we show the fraction of false positives for different element and cell sizes, and in Figure 3 the corresponding running time of `snort`, obtained using the `time(1)` facility of the host operating system. We observe that the fraction of false positives is well below 2% when using elements 13 bits or more. Completion time decreases with increasing element size, as the fraction of false

[2] Available at http://www.shmoo.com/cctf/

positives that have to be searched using Boyer-Moore is reduced. However, it is not strictly decreasing: it is minimized at 13 bits but exhibits a slight increase for more than 13 bits, apparently because of the effect of data-structure size (8 KB for 13-bit elements, 64 KB for 16 bits, for a cell size of 8 bits) on cache performance. For our specific configuration, 13-bit elements and 8-bit cells appear to offer the best performance.

4.3 Experiments with the default rule-set

We determine if E^2xB offers any overall improvement compared to FVh and BM using the `eth0.sump2` trace. The completion time for E^2xB , BM and FVh are 30.20, 47.31 and 47.36 seconds, respectively. We observe that using E^2xB , `snort` is 36% faster than both known algorithms. E^2xB is faster because, in the common case, it can quickly decide that a given set of strings is not contained in a packet. More specifically, in this experiment, the string matching function was invoked 22,716,676 times. Out of those, E^2xB was able to quickly state that the considered string was not a substring of the input packet in 22,395,210 of the invocations (or 98.4%). Thus, in 98.4% of all invocations, E^2xB was able to deliver the correct answer without actually searching for the pattern in the packet. In the remaining 1.6%, E^2xB used the Boyer-Moore string searching algorithm to find whether the string is really in the packet.

4.4 Other packet traces

We repeated the experiments with the three algorithms on the full set of traces. The results are summarized in Table 1. We first confirm that random payloads behave similarly to real payloads for the DEFCON `eth0.dump2` trace: the difference in performance between the original trace and the trace with the payload replaced with random data is negligible for all three algorithms. Based on this observation, we can argue that using random payloads on the NLANR and UCNET traces provides a reasonably accurate estimate on how the algorithms would perform with real payloads.

Comparing the performance of the string matching algorithms, we observe that E^2xB performs better than FVh and BM on all traces except for one and that the relative improvement varies. It is also interesting to see that FVh, reported in (6)to perform better than BM, sometimes performs worse for the traces examined. Although the improvement of E^2xB is typically between 25% and 35%, and can be as high as 36.17%, there are cases where the gain is only around 8% or, even in the case of the NLANR AIX trace, worse than BM by 8%. This appears to relate, at least in part, to differences in the packet size distribution: the average packet size is 835 bytes for the DEFCON `eth0.dump2` trace and 364 bytes for the NLANR AIX trace. For larger packets, `snort` spends more time in string matching, and E^2xB offers significant benefits, while for smaller packets, `snort` spends less time in string match-

Trace characteristics				Running time			
trace name	ID	nr. of packets	avg.pkt (bytes)	BM (sec)	FVh (sec)	E^2xB (sec)	%
eth0.dump2	D.02	1035736	835	47.31	47.36	30.20	+36.17
eth0.dump2.r	D.02.R			46.35	46.60	29.77	+35.77
eth0.dump4	D.04	595267	1481	14.11	56.24	9.81	+30.47
eth0.dump8	D.08	497302	1111	9.79	41.51	6.74	+31.15
webtrace	W.0	1188660	761	345.60	300.86	274.51	+8.76
NLANR IND	N.IND	2254931	703	93.53	83.8	62.04	+25.97
NLANR MRA	N.MRA	2760531	760	137.39	122.40	89.07	+27.23
NLANR AIX	N.AIX	1624223	364	13.17	14.00	14.26	-8.28
UCNET 0000	UC.00	1564131	422	103.93	82.35	66.84	+18.83
UCNET 0100	UC.01	2245938	413	108.69	84.20	62.54	+25.72

Table 1. Completion time of `snort` with different string matching algorithms – all traces

trace	rules	% pkts	% bytes	avg pkt
D.02	60	21.13	35.53	1336
	62	21.18	36.20	1358
	66	54.09	26.45	388
D.04	13	24.71	24.90	1472
	32	73.98	74.60	1473
D.08	13	24.82	24.84	1093
	32	74.83	74.91	1092
N.AIX	28	87.63	92.20	330
	36	5.56	2.84	160
N.IND	36	4.98	5.25	692
	38	40.07	30.31	495
	60	30.82	36.90	785
	62	8.38	9.22	721
N.MRA	60	43.72	44.04	713
	61	9.82	9.96	718
	62	13.89	14.17	722
	63	14.04	13.90	701
	101	6.16	5.80	667

trace	rules	% pkts	% bytes	avg pkt
W.0	103	56.47	33.41	419
	107	0.53	0.31	410
	820	42.99	66.28	1092
UC.00	36	15.81	13.52	316
	38	7.44	6.42	320
	60	18.85	16.63	326
	62	5.35	6.58	456
	68	12.71	10.13	295
	101	16.89	24.86	545
	102	9.62	10.47	402
	820	4.79	3.76	290
UC.01	36	11.54	9.51	296
	38	5.75	4.78	299
	60	42.71	39.91	336
	61	5.35	4.76	320
	68	7.35	5.70	279
	101	10.42	17.33	599
	102	7.68	10.09	473

Table 2. Analysis of rule-set invocations (rules rarely triggered are not presented)

ing, and E^2xB is less useful. On the other hand, results can be very different for traces with similar packet size statistics. For example, the average packet size for webtrace and MRA are 761 and 760 bytes, respectively, but the gain of E^2xB is 8.76% and 27.23%, respectively. More detailed analysis is therefore needed to understand the benefits of our approach.

We obtain processor-level statistics of executed instructions and L2 data cache misses for each trace using the `brink/abyss` toolkit which collects data from the Pentium performance counters (14). The results are presented in Figures 4 and 5. We observe that the number of instructions for E^2xB is significantly smaller in all cases except for the AIX trace. The reduction in

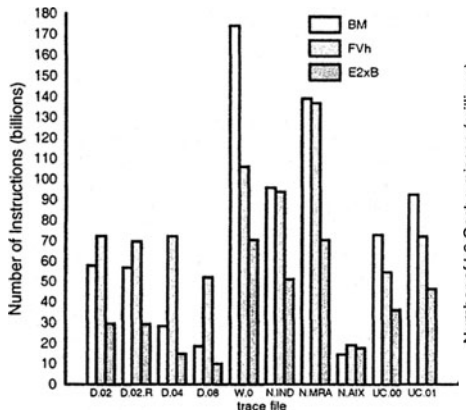


Figure 4. Executed instructions.

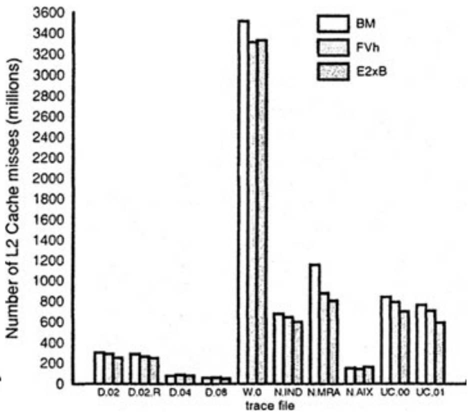


Figure 5. L2 data cache misses.

L2 data cache misses is relatively small compared to the reduction in executed instructions. For example, for the *W*.0 trace (Web-traffic) E^2xB has 30% less instructions but a slighly higher number of cache misses. This explains the relatively small overall performance gain (rougly 8%) for E^2xB on this trace.

To further understand the differences in the results, we instrumented `snort` to provide a trace of the chain headers and content rules invoked for each packet. The results for all packet traces are presented in Table 2. We observe that the string matching workload for different traces varies significantly. For instance, for the AIX trace 87.6% of the packets are checked against only 28 rules, while for the Web trace 56.4% of the packets are checked against 103 rules, and 43% against 820 rules. Considering these statistics, it appears that E^2xB offers larger improvements in cases where a large fraction of packets are checked against 30 to 100 content rules (as in the IND, MRA and all DEFCON traces). This also indicates that it may be necessary to consider hybrid algorithms, especially in cases where there is either a very small or very large number of rules applying to a significant fraction of packets. In such cases, E^2xB may not perform as well as BM when the number of content rules in a chain header is very small or the Aho-Corasick algorithm used in FVh when the number of content rules per chain header is large.

Although the details of such a hybrid algorithm are beyond the scope of this paper, we run a simple experiment to confirm that the cost of E^2xB is higher than BM for small sets of content rules and higher than Aho-Corasick for large sets of content rules. For this, we measure algorithm performance *off-line* e.g., as an isolated standalone program, with random inputs checked against a set

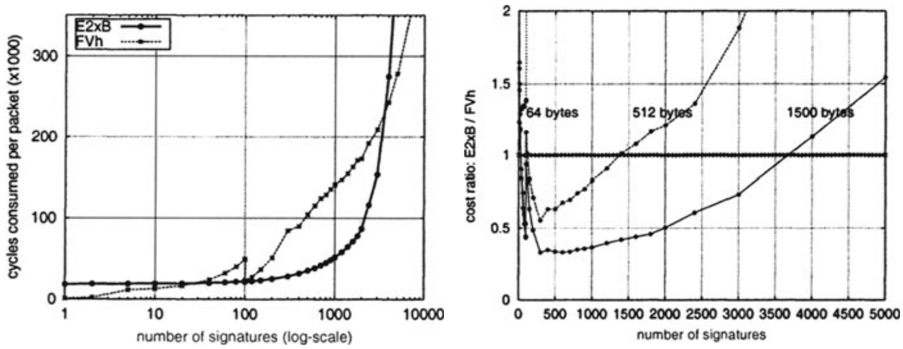


Figure 6. Off-line performance measurements of E^2xB and FVh.

of random rules. We fix the input size at 1500 bytes and obtain the average number of cycles for each input "packet" for different numbers of rules. Each rule is assumed to be a 20-bytes string. The results are presented in Figure 6 (left). We see that E^2xB is indeed more expensive than FVh for less than 20 rules, and that the relative performance benefits are maximized at around 700-1000 rules. After a certain point, the cost of E^2xB rises sharply, possibly due to the joint effect of increasing false-match rates *per-packet* and capacity misses (due to the size of the rule-set). We also run the same experiment with the input size set to 64 and 512 bytes, and compute the ratio of the average number of cycles consumed per-packet of E^2xB over FVh. These results are presented in Figure 6 (right). As expected, the relative benefits of the two algorithms and the ranges in which they perform better depend a lot on packet size. Experimentation with the actual NIDS and a more realistic traffic model and rule-set (or rule-set model) is, therefore, required to obtain the right thresholds for such a hybrid algorithm. Beyond the hybrid algorithm, these results also provide some insights on the scalability of different algorithms: E^2xB appears to cover a reasonable range of rule-set sizes that is likely to be sufficient as NIDS rulesets continue to increase in size.

4.5 Different architectures

We repeat the experiments on a system with a 1 GHz Pentium 3 processor and a 512 KB L2 cache. The results for the Pentium 3 are presented in Figure 7. We see that the gain for E^2xB is slighly higher on the Pentium 3 compared to the Pentium 4, with the proportion of the gain roughly consistent for the different traces. When comparing the performance of the P3 vs. the P4 system, the results may appear somewhat surprising: the P3 is almost always faster than the P4, as shown in Figure 8. This happens because the P3 has a 512 KB cache and the P4 we used has a 256 KB cache. For the Webtrace which has the highest memory usage among all traces, the P3 is almost 4 times faster than the P4. Besides highlighting the importance of considering the underlying

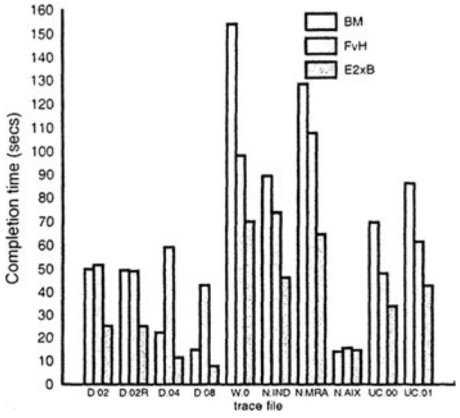


Figure 7. Performance on P3 processor.

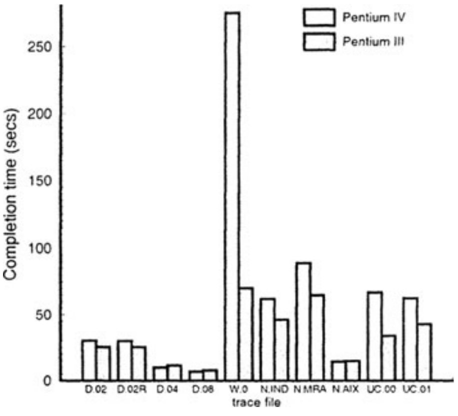


Figure 8. Performance on P3 vs P4.

system architecture when designing (and deploying) NIDSes, this experiment also demonstrates the great care needed in evaluating NIDS performance, as the results can be very sensitive to the environment.

5. Summary and concluding remarks

We have studied the performance of NIDS string matching algorithms, and presented the design E^2xB , a new algorithm for NIDS string matching. Using an extensive set of packet traces, we have evaluated E^2xB against existing algorithms. Our results show that in most cases E^2xB offers significant overall improvement in NIDS performance. We have shown realistic cases in which our approach improves performance by as much as 37%. The impact of E^2xB appears to relate to the packet size distribution and the number of string matching rules invoked per packet: small packets and very small or very large sets of rules per packet reduce the effectiveness of E^2xB . For medium-size rule-sets, E^2xB appears to be much faster than existing algorithms. These results point to the need for a hybrid algorithm, with E^2xB covering a range of medium-size rulesets. Determining the details of such a hybrid algorithm, including exact thresholds will be the subject of future work.

Our results also allow for some more general observations to be made on the performance as well as modelling, analysis and benchmarking of NIDSes: we have found that results are very sensitive to traffic and NIDS host processor and that random payloads behave similarly to real payloads. We expect these results to be useful towards more effective NIDS benchmarking and design.

Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. Work of the first author is also supported in part by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795, and by the USENIX/NLnet Research Exchange Program (ReX). We would also like to thank Dionisis Pnevmatikatos for his constructive comments, and Vasilis Siris for providing the UCnet traces.

References

[1] A.V. Aho and M.J. Corasick. Fast pattern matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] R. Bace and P. Mell. *Intrusion Detection Systems*. National Institute of Standards and Technology (NIST), Special Publication 800-31, 2001.

[3] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[4] C. Jason Coit, S. Staniford, and J. McAlerney. Towards faster pattern matching for intrusion detection, or exceeding the speed of snort. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2002.

[5] C. Courcoubetis and V. A. Siris. Measurement and analysis of real network traffic. In *Proceedings of the 7th Hellenic Conference on Informatics (HCI'99)*, August 1999.

[6] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. Technical Report CS2001-0670 (updated version), University of California - San Diego, 2002.

[7] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 147–160. ACM Press, 1999.

[8] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. University of California Press, 1997.

[9] R.N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10(6):501–506, 1980.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 203–214. ACM Press, 1998.

[11] Evangelos P. Markatos, Spyros Antonatos, Michalis Polychronakis, and Kostas G. Anagnostakis. ExB: Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN)*, pages 146–152, November 2002.

[12] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of the 1996 Usenix Technical Conference*, pages 279–294, January 1996.

[13] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. (available from *http://www.snort.org/*).

[14] Brinkley Sprunt. Brink and abyss: Pentium 4 performance counter tools for linux, February 2002. Available from *http://www.eg.bucknell.edu/˜bsprunt/*.