

# Performance Analysis of Content Matching Intrusion Detection Systems

S. Antonatos\*, K. G. Anagnostakis<sup>†</sup>, E. P. Markatos\*, M. Polychronakis\*

\* Institute of Computer Science  
Foundation for Research & Technology – Hellas  
P.O.Box 1385 Heraklio, GR-711-10 GREECE  
{antonat,markatos,mikepo}@csi.forth.gr

<sup>†</sup>Distributed Systems Laboratory  
CIS Department, Univ. of Pennsylvania  
200 S. 33rd Street, Phila, PA 19104  
anagnost@dsl.cis.upenn.edu

## Abstract

*A central question in the design and evaluation of a Network Intrusion Detection System (nIDS) is whether it is possible to define a practical, accurate and meaningful performance evaluation methodology. In this direction, we examine how nIDS performance interacts with experiment parameters such as traffic characteristics, nIDS rulesets, string matching algorithms and processor architecture. Our results indicate that nIDS performance is sensitive to the both packet and ruleset content, yet this sensitivity seems to be bounded, allowing us to craft and experiment with synthetic traces and rulesets. Our experiments also show that experiments on a single trace and processor architecture are likely to be misleading; effective nIDS evaluation therefore requires careful consideration of a fairly extensive set of scenarios.*

**Keywords:** security, intrusion detection, workload characterization and generation

## 1 Introduction

Network Intrusion Detection is 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” [3]. The typical function of a Network Intrusion Detection System (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 of existing efficient string matching algorithms for this purpose, such as [4, 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 [7]. 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, a nIDS needs to be highly efficient to keep up with increasing link speeds. Finally, 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 string matching algorithms in light of the particular requirements and characteristics of Intrusion Detection. Several algorithms have been used recently, some of which were designed specifically for the particular application domain. The evaluation of nIDS algorithms and systems is usually based on circumstantial evidence: there is no established or standard workload model for nIDS performance measurement, and it is difficult to obtain, use and share full packet traces because of (understandable) privacy issues.

In this paper we present a study of nIDS string matching algorithms and their performance evaluation. The central question that we attempt to answer is whether it is possible to define a practical, reasonably accurate and meaningful methodology for evaluating nIDS performance. Such a methodology would be helpful both as a benchmark for comparing different systems as well as an aid for designing better systems and algorithms. To this end, we examine how nIDS performance interacts with changing experiment parameters – traffic characteristics, nIDS rulesets, string matching algorithms and processor architecture – and determine if these parameters can be captured in a simpler model, and at what cost. Our results so far allow us to make the following observations:

- nIDS performance is sensitive to packet and ruleset content. Adding random content to the widely-available traffic header traces is thus, at least on first sight, questionable as a method for nIDS evaluation.

However, our analysis shows that the sensitivity exhibits certain patterns in over- or under-estimating performance, depending on the string matching algorithm and the characteristics of the traffic. Regarding rule-set content, using random rule patterns for determining nIDS performance and scalability also requires extreme care: our results suggest that a more accurate way of creating synthetic rulesets is to use permutations of existing patterns. As with packet content, the sensitivity follows some predictable pattern depending on algorithm and traffic.

- There are large differences in measured performance depending on traffic characteristics: the highest measured mean cost per-packet is up to four times as much as the lowest cost in the traces we examined, mostly due to differences in the distribution of packets to the different subsets of the nIDS ruleset.
- The choice of processor architecture has a dramatic effect, both on overall system performance as well as the relative performance of different string matching algorithms. There are cases where one string matching algorithm is faster than another algorithm on one processor but slower on another processor. Our results show that algorithm performance improves (unusually) well with processor technology, partly due to increasing cache size.
- No single algorithm performs best in all cases; a hybrid string matching engine triggering different algorithms depending on ruleset and packet size appears to be the best approach, but the parameters may also vary depending on the processor architecture.

The rest of the paper is organized as follows: in Section 2 we briefly review how a string matching nIDS operates and what algorithms are being used. In Section 3 we present the results of our investigation; Section 4 presents a summary of our results and issues for further experimentation.

## 2 Background

We describe a (simplified) model of how a nIDS operates and summarize the key characteristics of string matching algorithms that have been recently used in the nIDS context. A nIDS is designed as a passive monitoring system that reads packets from a network interface through `libpcap`[11]. After a set of normalization passes (such as IP fragment reassembly, TCP stream reconstruction, etc.) each packet is checked against the nIDS ruleset. The ruleset is typically organized as a two-dimensional chain data-structure, where each element - often 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 remainder of this Section presents string matching algorithms used in a nIDS.

### 2.1 Boyer-Moore

The most well-known algorithm for matching a single pattern against an input was proposed by Boyer and Moore[4]. 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 [9] improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic.

### 2.2 Aho-Corasick

Aho and Corasick [1] provide 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.

### 2.3 Set-wise Boyer-Moore

Fisk and Varghese [7] designed an algorithm for nIDS string matching. The algorithm, called Set-wise Boyer-Moore-Horspool, is an adaptation of Boyer-Moore 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. A similar algorithm was implemented independently by Coit et al.[5], derived from the exact set matching algorithm of [8].

## 2.4 Exclusion-based Signature Matching

We have recently proposed  $E^2xB$ , a string matching algorithm that is designed for providing quick negatives when the search string does not exist in the packet payload, assuming a relatively small input size (in the order of packet size). [10, 2]. As mismatches are by far more common than matches, string matching can 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 string 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 [4]. The small input assumption ensures that the rate of false positives is reasonably small – our experiments demonstrate false positive rates of 10% in the worst case. In the common case, negative responses can be obtained without resorting to general-purpose string matching algorithms.

## 2.5 Wu-Manber

The most recent implementation of `snort` uses a simplified variant of the Wu-Manber multi-pattern matching algorithm [15], as discussed in [14]. The "MWM" algorithm is based on the bad character heuristic similar to Boyer-Moore but uses a one or two-byte bad shift table constructed by pre-processing all patterns instead of only one. MWM performs a hash on the two-character prefix of the current input to index into a group of patterns, which are then checked starting from the last character, as in Boyer-Moore. The results of [14] show that `snort` is much faster than previous versions that used the Set-Wise Boyer-Moore and Aho-Corasick, although it is not clear how much of the performance improvement is because of the new string matching algorithm.

## 3 Analysis

For most 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 MB 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 2.0-beta2 compiled with `gcc` version 2.96 (optimization flags `O2` – results with `O3` were found to be similar).

We use packet traces from four different sources:

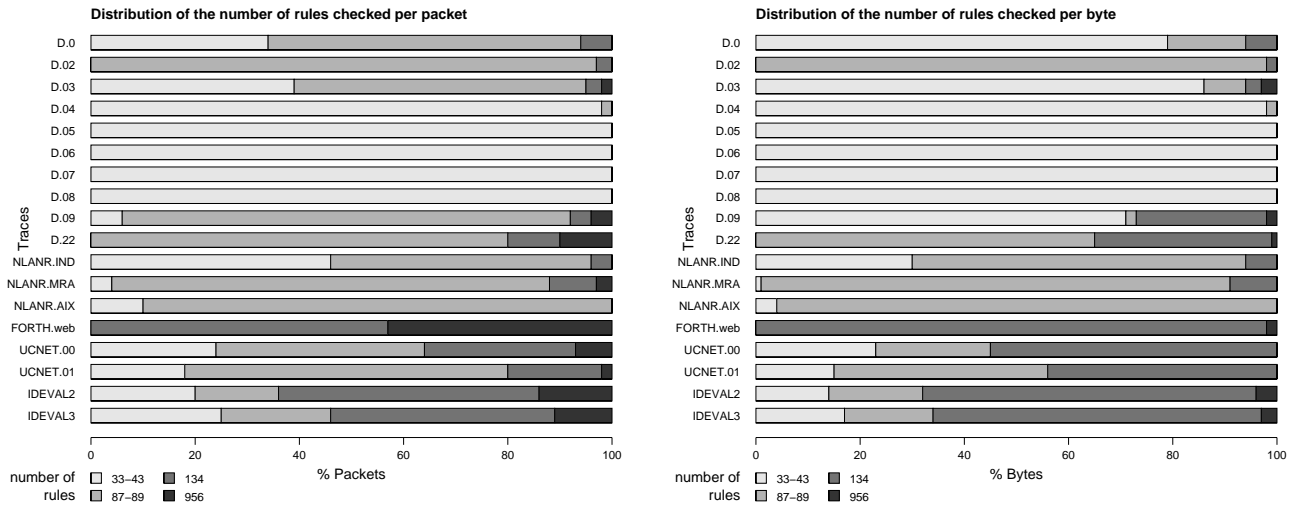
- A set of full-packet traces from the DEFCON “capture the flag” data-set. These traces contain numerous intrusion attempts.
- A full packet trace containing Web traffic, generated by concurrently running a number of recursive `wget` requests on popular portal sites from a host within the FORTH network.
- 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.
- 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)[6], with random payloads.

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.

### 3.1 Effect of traffic characteristics

We instrumented `snort` to record the chain header triggered for each packet for each trace. The results are summarized in Figure 1. We observe that the use of different chain groups varies significantly: roughly 42% of packets in the FORTH.web trace trigger 956 rules, only 1-5% for the other traces. For the NLANR backbone traces, 77-93% of the packets trigger at most 87 rules. Although such differences should be expected given that the traces represent traffic in different settings (backbone, campus, a hacker contest and a web-only environment) there are visible differences even between traces of the same kind. We can hereby argue that no single workload is sufficient for evaluating “overall” nIDS performance.

We also measure the performance of  $E^2xB$  and MWM string matching on each of these traces. For header-only traces we add random payloads, assuming this provides representative results – in Section 3.2 we will discuss in more detail whether and to what degree this assumption is reasonable. The measured running time of `snort` on each trace is presented in Figure 2. The most striking observation is that the mean processing time per-packet is between 5.04-19.71  $\mu s$  for MWM and 5.17-14.01  $\mu s$  for  $E^2xB$ , that is, the highest cost is 3 to 4 times the lowest cost, and only



**Figure 1. Distribution of the number of rules checked per packet and byte (rules rarely triggered are not presented)**

slightly lower if we ignore the DEFCON traces (as not representative of “real” network traffic). We also observe that in all traces except for DEFCON.0-9  $E^2xB$  performs better than MWM; in most cases the improvement seems to be between 18-36%. The per-byte cost for DEFCON.0.9 seems to be unusually high: 83.1% of the packets in this trace have a payload size of one byte - obviously not representative of common scenarios. The relatively small improvement of  $E^2xB$  over MWM for the NLANR.AIX trace seems to coincide with the small average packet size (340 bytes) in this case compared to other traces with a similar distribution of packets to different chain header rules.

The highest cost is observed in the FORTH.web trace, as expected, given that 43% of the packets trigger 956 rules, much higher than any other trace. What is less obvious is that NLANR.MRA and DEFCON.0-4 are not much cheaper than FORTH.web, although the average number of rules triggered per packet is much smaller: 99% of packets in DEFCON.0-4 and 77% of packets in NLANR.MRA trigger 33 and less than 89 rules respectively. The explanation lies in the average packet size measured for the dominant chain headers: the average packet size for the rules triggered by DEFCON.0-4 is 1480 bytes and between 736 and 929 for NLANR.MRA.

In summary, these results indicate that nIDS string matching performance varies significantly for different traces; simply using one source/type of workload therefore leads to questionable and potentially misleading results. Note that almost half of the traces used in this set of experiments contain random payloads. We will discuss next how this affects results.

### 3.2 Packet payloads: real vs. synthetic

We attempt to quantify the value of using real payloads in nIDS evaluation. For this we use the DEFCON and FORTH traces that contain real payloads, and for each trace we construct a new one with the original payload of each packet replaced by random characters. We obtain running times (in seconds) for MWM and  $E^2xB$  and present the mean over 10 runs (the results are accurate within 0.01 sec) in Figure 3.

The following observations can be made. The basic result is that synthetic, random payloads give quite different results than real payloads for the same trace. For roughly half of the traces the difference seems to be consistent: 14-18% for  $E^2xB$  and 24-30% for MWM. Although this result can be regarded as negative, it appears that running times for  $E^2xB$  are generally overestimated while for MWM the running times are underestimated. If this trend is persistent then it is possible to use random payloads for evaluating nIDS performance, bearing in mind the expected measurement error to be in the direction and at the order of the results presented here.

### 3.3 Rulesets: real vs. synthetic

IDS rulesets are frequently updated with new signatures as new threats are identified. It is therefore necessary to predict performance for larger rulesets. This could be done by synthesizing rule-sets, but it is not clear what kind of synthesis method is suitable for producing representative rule-

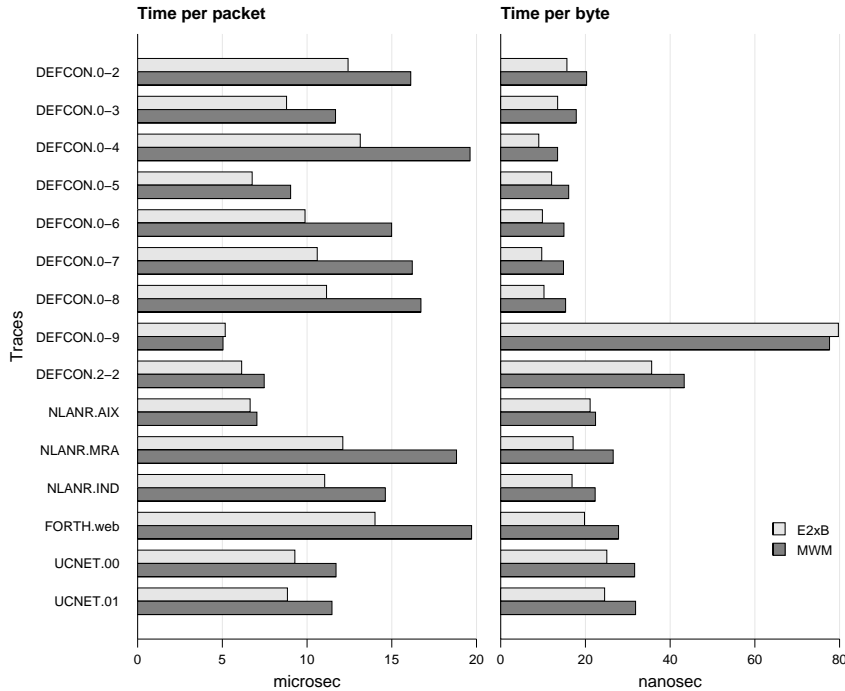


Figure 2. Running time, per-packet and per-byte

sets. We consider two methods: creating random strings and using permutations of the existing ruleset to generate new rules. In both cases, the length of the generated rules follows the distribution of string lengths in the seed ruleset.

To determine the accuracy of this approach we use the two methods for creating a new ruleset based on the default `snort` ruleset with all other parameters (number of headers, number of signatures per header, length of each signature) unchanged. This allows us to estimate how good the prediction would be if we wanted to predict the performance of the current ruleset. We obtain the running times for the original ruleset as well as the synthetic ruleset with each of the two methods, and present the results in Table 1.

We observe that both methods generally result in wrong performance estimates. The errors appear to be much smaller for  $E^2xB$  than for MWM: except for the FORTH.web trace all other traces actually have very small errors, although the 19.6% error in FORTH.web suggests that the method is not generally error-free. The use of permutations is generally (but not always) much better than the use of random characters. Unlike the errors produced by random payloads, there is no clear trend towards under- or over-estimating performance. Considering permutations only, the maximum error is 6.71% for  $E^2xB$  and 24% for MWM. Any results using rules synthesized with this

method should therefore expect the prediction error to be close to these values (assuming the ruleset structure will not change significantly).

### 3.4 Analysis using synthetic rulesets

We use the ruleset synthesis method to analyze the performance of  $E^2xB$  and MWM as a function of ruleset-size as well as packet-size. We create synthetic rulesets containing a single chain header with 1 to 10,000 rules with an average size of 14 bytes. A simple traffic generator creates packets with random payloads and a given packet size. We measure the performance of  $E^2xB$  and MWM for packet sizes of 50, 500 and 1500 bytes. The results are presented in Figure 4: we present the running time of  $E^2xB$  and MWM for a fixed packet size of 1500 bytes and between 1 and 10,000 rules. We see that MWM performs better than  $E^2xB$  when the number of rules is less than 6 or more than 2,500. For values between 6 and 2,500  $E^2xB$  is better, almost twice as fast when the number of rules is between 9 and 100. As the number of rules increases, the cost of string matching increases as well – for more than 2,000 the growth is super-linear. This is not surprising given that the growing footprint of string matching also generates more L2 cache misses. Such results indicate that nIDS load bal-

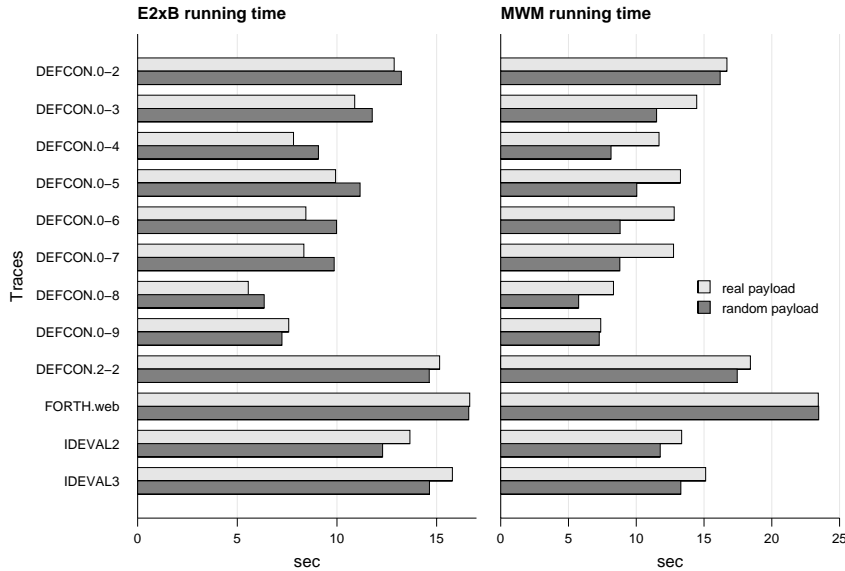


Figure 3. Random vs. real payloads

ancing needs to carefully consider sensor locality issues in choosing load balancing policies. For example, if the number of rules increases dramatically a policy that allocates half of the flows to each of two sensors may be less efficient than a policy that sends all flows to both sensors and uses half the ruleset on each sensor.

In Figure 5 we present the ratio of the running time of  $E^2xB$  over MWM for different packet sizes. We see that the “turning points” where  $E^2xB$  exceeds the performance of MWM depend on packet size. These results suggest that a hybrid approach, triggering different algorithms depending on ruleset and packet size is likely to result in the best overall performance. This is similar in principle to the hybrid algorithm suggested by Fisk and Varghese [7] although the effect of packet size had not been considered in their hybrid Boyer-Moore / Set-Wise Boyer-Moore / Aho-Corasick heuristic.

### 3.5 The effect of processor architecture

We measure the performance of  $E^2xB$  and MWM on three different systems: in addition to the 1.7 GHz P4 used in all the other experiments we run the same experiment on a 1.0 GHz P3 (256 MB cache) and a 2.4 GHz P4 (512 MB cache). The results are shown in Figure 6.

Interestingly, string matching seems to scale well with increasing processor speed:  $E^2xB$  is 4.7 to 7.4 times faster on the 2.4 GHz P4 than on the 1.0 GHz P3 while MWM is 2.2 to 4.3 times faster. The difference in cache size seems

to provide some explanation for this behavior: the 2.4 GHz P4 has a 512 MB cache while the other two processors have 256 MB. On the other hand, although the improvement from P3 to 1.7 GHz P4 is much less – 1.62-2.71 times faster for 1.05 to 1.49 times faster for MWM – it is often close to or even higher than the relative improvement in processor clock rate. A second observation relates to the relative improvement of the two algorithms:  $E^2xB$  performs worse on the P3 than MWM, but much better than MWM on the 2.4 GHz P4, with the gap increasing as we move from slower to faster machine.

## 4 Summary

In this paper we have studied the performance of string matching in a Network Intrusion Detection System (nIDS). One of the main problems with nIDS performance studies so far has been the difficulty in obtaining and sharing full packet traces and uncertainty about the representativeness of any trace. The primary goal of this paper was therefore to determine what experiment parameters are important in nIDS evaluation and whether it is possible to develop a practical yet accurate workload model.

Our experiments so far have led to three main observations. First, we have found that nIDS performance measurements are sensitive to both packet content and ruleset content. Although this may sound discouraging, we have

trace	original		random				permutations			
	E2xB	MWM	E2xB		MWM		E2xB		MWM	
			time	error	time	error	time	error	time	error
NLANR.AIX	10,78	11,43	10,86	<b>+0,74</b>	10,46	<b>-8,49</b>	11,32	<b>+5,01</b>	11,92	<b>+4,29</b>
NLANR.IND	24,89	32,96	24,57	<b>-1,29</b>	29,37	<b>-10,89</b>	25,49	<b>+2,41</b>	34,61	<b>+5,01</b>
NLANR.MRA	33,42	51,94	33,47	<b>+0,15</b>	46,21	<b>-11,03</b>	34,60	<b>+3,53</b>	55,24	<b>+6,35</b>
DEFCON.0-2	12,86	16,69	12,82	<b>-0,31</b>	14,83	<b>-11,14</b>	13,05	<b>+1,48</b>	17,60	<b>+5,45</b>
DEFCON.0-2(r)	12,50	15,46	12,67	<b>+1,36</b>	14,50	<b>-6,21</b>	12,91	<b>+3,28</b>	16,55	<b>+7,05</b>
DEFCON.0-4	7,82	11,68	7,54	<b>-3,58</b>	6,45	<b>-44,78</b>	7,56	<b>-3,32</b>	8,95	<b>-23,37</b>
DEFCON.0-8	5,54	8,31	5,30	<b>-4,33</b>	4,72	<b>-43,20</b>	5,42	<b>-2,17</b>	6,49	<b>-21,90</b>
UCNET.00	14,50	18,31	14,55	<b>+0,34</b>	17,45	<b>-4,70</b>	14,88	<b>+2,62</b>	19,14	<b>+4,53</b>
UCNET.01	19,85	25,77	20,62	<b>+3,88</b>	24,08	<b>-6,56</b>	21,28	<b>+7,20</b>	27,25	<b>+5,74</b>
FORTH.web	16,65	23,43	14,43	<b>-13,33</b>	19,12	<b>-18,40</b>	15,72	<b>-5,59</b>	22,75	<b>-2,90</b>
IDEVAL2	13,66	13,35	12,50	<b>-8,49</b>	11,12	<b>-16,70</b>	13,12	<b>-3,95</b>	13,09	<b>-1,95</b>
IDEVAL3	15,80	15,29	15,30	<b>-3,16</b>	12,69	<b>-17,00</b>	15,32	<b>-3,04</b>	14,79	<b>-3,27</b>

Table 1. Synthetic rulesets: real vs. random vs. permutations

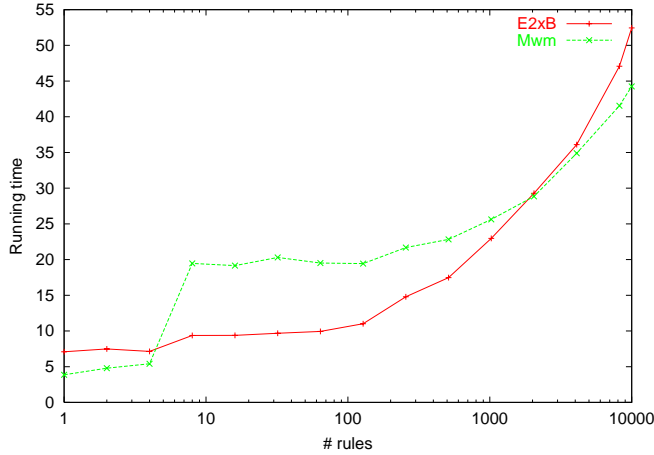


Figure 4. Running time of  $E^2xB$  and  $MWM$  vs. ruleset size (for a synthetic trace containing 1500-byte packets and synthetic rulesets)

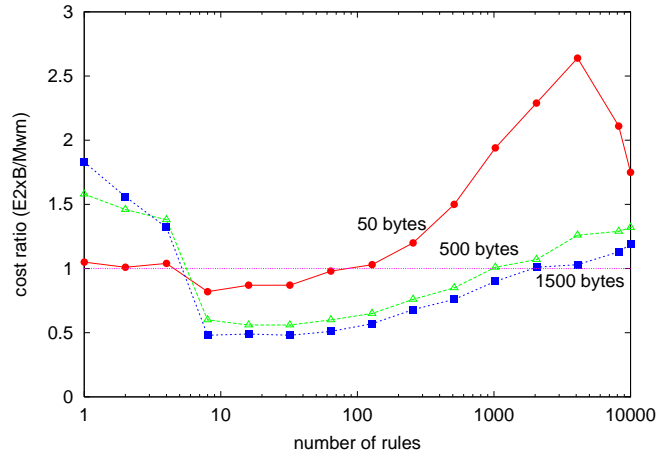


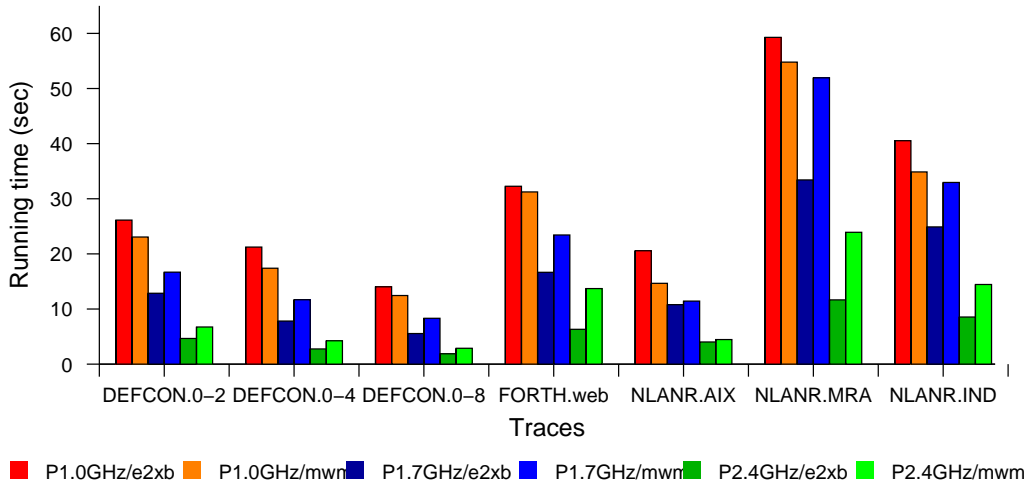
Figure 5. Relative performance (ratio of running times) of  $E^2xB$  and  $MWM$  vs. ruleset- and packet-size (synthetic rules+content)

found that the level of sensitivity appears to be in most cases bounded. Although quantifying this sensitivity has been beyond the purpose of our work, our results so far suggest that it is possible to experiment with random payloads and random rulesets, at the cost of some measurement error. We believe this to be a useful result: taking such errors into account, one can create and use representative nIDS traces by simply extracting the timestamp, packet size and matching chain header identifiers from (real or captured) nIDS traffic. Such traces hide all privacy-sensitive information while also being significantly smaller and therefore more portable than full packet traces – an estimate for some of the traces used in this paper are presented in Table 2. We believe that this approach will make it easier for researchers to use (and share) nIDS traffic traces.

Second, results are sensitive to traffic characteristics and

processor architecture. We have compared the  $E^2xB$  and  $MWM$  algorithms on many different traces and have found significant variation in both the average per-packet cost for each trace as well as in the relative benefits of each algorithm. We also evaluated the two algorithms on on three different processor architectures: a 1.0 GHz P3, a 1.7 GHz P4 and a 2.4 GHz P4. Our results show that algorithm performance improves (unusually) well with processor technology. The improvement is in part (only) due to the larger data cache on the high-end P4. Our results also show that the processor can significantly affect the comparison of two algorithms, and that the relative improvement in performance when comparing  $E^2xB$  to  $MWM$  seems to increase with processor clock rate.

Third, we compared the two algorithms using the synthetic ruleset generation method, and found that no algo-



**Figure 6. Measurements of  $E^2xB$  and MWM on different processors**

trace	normal			bzip2		
	original	compressed	ratio	original	compressed	ratio
UCNET.20.06-00	683.8 MB	14.2 MB	1:48	619.44 MB	4.82 MB	1:128
UCNET.20.06-01	961.6 MB	21.2 MB	1:45	872.30 MB	7.42 MB	1:117
DEFCON.0-2	900.0 MB	6.9 MB	1:129	795.37 MB	1.04 MB	1:762
DEFCON.0-4	900.0 MB	7.0 MB	1:127	5.60 MB	0.04 MB	1:125
DEFCON.0-8	900.0 MB	5.9 MB	1:151	161.81 MB	0.32 MB	1:4948
NLANR.IND	1,623.7 MB	22.7 MB	1:71	1530.43 MB	7.73 MB	1:197
NLANR.MRA	2,147.4 MB	22.6 MB	1:95	2038.79 MB	7.57 MB	1:268
FORTH.web	926.2 MB	8.0 MB	1:115	45.86 MB	2.67 MB	1:17

**Table 2. Original vs. compressed trace size**

rithm is better in all cases. Performance depends on ruleset and packet size, suggesting that a hybrid method invoking a different algorithm depending on these parameters is likely to offer better performance than the exclusive use of a single algorithm.

Our study is admittedly incomplete given the limited set of traces we could use. Nevertheless we believe that we have exposed some of the key issues, trade-offs and parameters that need to be carefully considered in the experimental analysis of nIDS performance. We believe that our results are valuable towards more effective nIDS evaluation and design.

## Acknowledgements

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union.

Work of the second 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, Sotiris Ioannidis and Stefan Miltchev for their constructive comments, and Vasilis Siris for providing the UCnet traces.

## Availability

Source code for the workload generation tools discussed in this paper as well as a patch for snort containing the  $E^2xB$  algorithm can be found at <http://www.ics.forth.gr/carv/ids.html>

## References

- [1] A. Aho and M. Corasick. Fast pattern matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.
- [2] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis.  $E^2xB$ : A domain-specific string matching algorithm for intrusion detection. In *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, May 2003. (to appear).
- [3] R. Bace and P. Mell. *Intrusion Detection Systems*. National Institute of Standards and Technology (NIST), Special Publication 800-31, 2001.
- [4] R. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.
- [5] C. J. 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.
- [6] 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.
- [7] 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.
- [8] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. University of California Press, 1997.
- [9] R. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10(6):501–506, 1980.
- [10] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis. ExB: Exclusion-based signature matching for intrusion detection. In *Proceedings of CCN'02*, November 2002.
- [11] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to ftp.ee.lbl.gov.
- [12] L. McVoy and C. Staelin. Imbench: Portable tools for performance analysis. In *Proc. of the 1996 Usenix Technical Conference*, pages 279–294, Jan. 1996.
- [13] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. (software available from <http://www.snort.org/>).
- [14] Sourcefire. *Snort 2.0 - Detection Revisited*. [http://www.snort.org/docs/Snort\\_20\\_v4.pdf](http://www.snort.org/docs/Snort_20_v4.pdf), October 2002.
- [15] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.