Exclusion-based Signature Matching for Intrusion Detection

Evangelos P. Markatos, Spyros Antonatos, Michalis Polychronakis, Kostas G. Anagnostakis *
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385 Heraklio, Crete, GR-711-10 GREECE
{markatos,antonat,mikepo}@csi.forth.gr
appears in IASTED International Conference on Communications and Computer Networks (CCN 2002)

ABSTRACT
We consider the problem of efficient string-based signature matching for 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 ExB, a string matching algorithm tailored to the specific characteristics of NIDS string matching. We have implemented ExB in `snort` and present experiments comparing ExB with the current best alternative solution. Our preliminary experiments suggest that ExB offers improvements in overall system performance by as much as a factor of three.

KEY WORDS
network security, intrusion detection, string matching, network performance

1 Introduction

Network Intrusion Detection Systems (NIDSes) are receiving considerable attention as a mechanism of last resort for shielding computer systems and networks against attackers [2]. The typical function of a NIDS is based on a set of signatures (or rules), each describing one known intrusion threat. A NIDS examines network traffic and determines whether any signatures indicating intrusion attempts are matched. To detect such activity, NIDSes often need to inspect the payload of incoming packets for such signatures.

The simplest and most common form of inspection is brute-force string matching against the packet payload. For instance, consider the (simplified) signature shown in Figure 1, taken from `snort`, a widely-used open-source NIDS [13]. This signature matches all TCP/IP packets originating from computers outside the monitored domain (i.e., the `$EXTERNAL_NET`), destined to the web servers of the monitored domain (i.e., the `$HTTP_SERVERS` at port `80`), and containing the string "`/usr/bin/perl`" in the payload. If the NIDS determines that a packet matches this rule, it infers that a malicious client may be trying to make the web server execute the perl interpreter, hoping to gain unauthorized access. To decide whether a packet matches the signature, the NIDS needs to check the (TCP/IP) packet header for the specified values (i.e., `$EXTERNAL_NET`, `$HTTP_SERVERS`, `80`). In addition, the NIDS needs to check whether the payload contains the string "`/usr/bin/perl`".

String matching is generally expensive: finding a single pattern in an input string imposes computation which is at least linear to the size of the input string [12], and NIDS rule-sets often contain hundreds of such strings. Most known NIDS implementations use general-purpose string matching algorithms that are known to perform well. The computational burden of string matching using those algorithms is significant: recent measurements on a production network suggest that `snort` spends roughly 30% of its total processing time in string matching, while for Web-intensive traffic, this cost is increased to as much as 80% [6]. Furthermore, NIDSes need to be highly efficient to keep up with increasing link speeds. For instance, a 10 Gbit/s network link delivers roughly one byte every 0.8 nanoseconds. Considering a state-of-the-art processor operating at 2 GHz, this allows for no more than roughly 1.6 instructions for each incoming byte. Finally, as the number of potential threats (and associated signatures and rules) is expected to grow, the string matching workload is likely to increase even further.

In this paper, we present ExB, a multiple-string matching algorithm designed specifically for NIDSes. The basic idea is to determine if the input (e.g., each packet received) contains *all* fixed-size bit-strings of the signature string, *without* considering if the bit-strings appear in-sequence, as done by existing algorithms. If at least one bit-string of the signature does not appear in the packet, then ExB determines that the signature does not match. The small size of the input ensures that ExB matches correlate well with actual matches. This approach also allows for a straightforward and efficient implementation: for each packet, ExB first creates an occurrence bitmap marking each fixed-size bit-string that exists in the packet. The bit-strings for each signature are then matched against the occurrence bitmap. As packets are rarely expected to match any signature, ExB performs better in the common case compared to existing algorithms. In the case of false matches (e.g., when all fixed-size bit-strings show up, but in arbitrary positions

*E. P. Markatos, S. Antonatos and M. Polychronakis are also with the University of Crete. Kostas G. Anagnostakis is with the CIS Department, University of Pennsylvania. Email: anagnost@dsl.cis.upenn.edu

```
alert tcp $EXTERNAL_NET any ->
$HTTP_SERVERS 80 (con-
tent:``/usr/bin/perl'')
```

Figure 1: A simple intrusion detection rule.

within the input), ExB falls back to standard algorithms (such as the Boyer-Moore algorithm [3]).

To validate our approach, we have implemented ExB in `snort`. Experiments using full packet traces show that the rate of false matches is reasonably small, and that ExB offers significant performance benefits; in certain cases, ExB makes `snort` up to three times faster.

The rest of the paper is organized as follows. In Section 2 we review previous work and place our algorithm in context, and in Section 3 we informally describe our algorithm. Section 4 presents experiments with ExB as implemented in `snort` and compares its performance with the current best alternative. Finally, Section 5 outlines open issues for further investigation, and Section 6 summarizes our results.

2 Previous work

The general problem of designing algorithms for string matching is well-researched. The most widely used algorithm is due to Boyer and Moore [3]. The Boyer-Moore algorithm compares the string with the input starting from the rightmost character of the 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 improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic[8].

Aho and Corasick provide an algorithm for concurrently matching multiple strings [1]. 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. Algorithms based on Aho-Corasick are widely used in current compiler technology, and several improvements have been presented [5, 9, 15].

Fisk and Varghese were the first to consider the design of NIDS-specific string matching algorithms. They proposed an algorithm called Set-wise Boyer-Moore-Horspool [6], adapting the Boyer-Moore algorithm to simultaneously match a set of rules. This algorithm is shown to be faster than both Aho-Corasick [1] and Boyer-Moore[3] 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.*[4] implemented a similar algorithm in `snort`, adapting Boyer-Moore for simultaneously matching multiple strings, derived from the exact set matching algorithm of Gusfield[7].

NIDSes are unlikely to be able to track increasing network speeds, regardless of whether more efficient algorithms can be designed, as the cost of string matching appears to be orders of magnitude higher than the cost of IP forwarding. A straightforward approach for scaling such systems is to consider distributed architectures, such as the one examined by Kruegel *et al.*[10]. The architecture splits incoming traffic into several Intrusion Detection Sensors that work in parallel to identify intrusion attempts [10]. Such efforts are orthogonal to improving string matching – better algorithms will require less sensors in a distributed NIDS architecture.

3 ExB: Exclusion-based string matching

We present an informal description of ExB, first in its simplest and most intuitive form and then in its more general form. ExB is based on the following simple reasoning:

> Suppose that we want to check whether a small input I contains a string s. Then, if there is at least one character in s that is not in I, then s is not in I.

This can be used to determine when a given string s does *not* appear in the input string I. If, on the other hand, every character of s belongs to I, then we use a standard string searching algorithms (e.g., Boyer-Moore-Horspool) to confirm whether s is a substring of I. This is needed in case of *false matches*, e.g., cases where every character of s is in I, but not in the sequence they appear in s. This simplifies the matching problem, since we can efficiently determine whether a character c belongs to I by means of an *occurrence bitmap*. Specifically, we first *pre-process* the input I, and, for each character c that appears in string I, we mark the corresponding element on the (256-element) bitmap. After pre-processing, we know that string I contains the j_{th} character only if the j_{th} element of the bitmap is marked. The pseudo-code for pre-processing

```
boolean exists[256];

pre_process(char *input, int len)
{
  bzero(exists, 256/8); // clear array

  for (int idx = 0 ; idx < len ; idx++) {
       exists[input[idx]] = 1;
  }
}
search(char *s, char *in-
put, int len_s, int len)
{
  for (int idx = 0 ; idx < len_s ; idx++) {
      if (exists[ s[idx] ] == 0)
          return DOES_NOT_EXIST ;
  }
  return boyer_moore(s, len_s, input, len);
}
```

Figure 2: Pseudo-code for ExB pre-processing and search.

`input` and for matching a string `s` on `input` is presented in Figure 2.

The algorithm can be generalized for *pairs* of characters, with the intention of reducing the probability of false matches. 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, ExB knows that s is not in I.

Generalizing further, ExB can use bit-strings of arbitrary length, instead of just 8-bit characters. That is, ExB records all (byte-aligned) bit-strings of length x. The size of the bit-string exposes a trade-off: larger bit-strings are likely to result in fewer false matches, but also increase the size of the occurrence bitmap, which could, in turn, increase capacity misses and degrade performance. For most of our experiments we shall use $x = 13$ which our experimental analysis has shown to maximize performance (in the particular experiment setup).

4 Experimental evaluation

We evaluate the performance of ExB against the Fisk-Varghese heuristic (denoted as FVh in the rest of this paper) as implemented in `snort` , using trace-driven execution.

4.1 Environment

All experiments were run on a PC with a Pentium 4 processor running at 1.7 GHz, with a L1 data 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 data cache, 10.9 ns for the L2 cache and 170.4 ns for the main memory (measured using `lmbench`[11]). The host operating system is Linux (kernel version 2.4.14, RedHat 7.3).

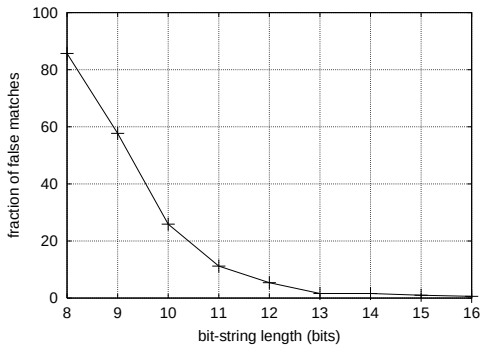

Figure 3: Fraction of false matches vs. bit-string length.

We use `snort` version 1.9.0 compiled with `gcc` version 2.96 (optimization flags `O2` – results with `O3` were found to be similar). Each packet is checked against the "default" rule-set of the `snort` distribution. This rule-set is composed of 1243 rules, of which 90.3% requires examining the packet payload. `snort` organizes these rules in 152 "chain headers". Chain headers in `snort` are used to associate each packet header rule with a suitable set of string matching rules.

To drive the execution of `snort`, we use full-packet traces from the "capture the flag" data-set[1]. The "capture the flag" contest is held every year at DEFCON: the "largest underground Internet security gathering on the planet". These traces contain a significant number of intrusion attempts[2]. For most of the experiments, we used the `eth0.dump2` trace containing 1,035,736 packets. For simplicity, traces are read from a file. Replaying traces from a remote host provided similar results.

4.2 Experiments with the default rule-set

Before comparing the two algorithms, we first determine the optimal size for the fixed-size bit-string used by ExB. In Figure 3 we show the fraction of false matches for different bit-string lengths, and in Figure 4 the corresponding running time of `snort` , obtained using the `time(1)` facility of the host operating system. We observe that the fraction of false matches is well below 2 % when using bit-strings of 13 bits and more. Completion time decreases with increasing bit-string size, as the fraction of false matches 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

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

[2] The most appealing reason for using these traces is less the number of intrusion attempts and more the availability of *real* payloads. One could, however, argue that neither of these two characteristics affects performance noticably, but this has not been confirmed experimentally.

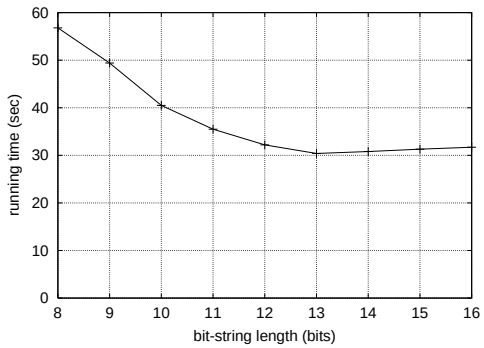

Figure 4: Completion time vs. bit-string length.

	FVh	ExB
completion time (sec)	41.42	30.34

Table 1: Completion time of `snort` using the default rule-set

because of the effect of data-structure size (1KB for 13 bits, 8 KB for 16 bits) on cache performance. For our specific configuration, 13 bits appear to offer the best performance.

We now examine whether ExB offers any overall improvement compared to FVh. The completion time for ExB and FVh are presented in Table 1. We see that using ExB, `snort` completes execution 27% faster compared to using FVh. ExB completes execution faster because in the common case it can quickly decide that a given string is not contained in a packet. In 98.4% of all invocations, ExB was able to terminate without actually invoking Boyer-Moore. In the remaining 1.6% of the cases, ExB used the Boyer-Moore to find whether the considered string was contained in the input packet.

To better understand the behavior of ExB , we obtain processor-level statistics using the Pentium performance counters[14]. We measure the total number of instructions executed, the number of L1 data cache misses, and the number of L2 cache misses. These statistics are reported in Table 2. We see that ExB achieves 27% improvement in completion time, uses 37% fewer instructions, induces 17% fewer L1 data cache misses, and 14% fewer L2 cache misses.

	FVh	ExB	improv.
completion time (sec)	41.42	30.34	27%
instructions ($\times 10^9$)	58.9	37.1	37%
L1 misses ($\times 10^9$)	1.83	1.51	17%
L2 misses ($\times 10^6$)	259	222	14%

Table 2: Performance of `snort` using the default rule-set

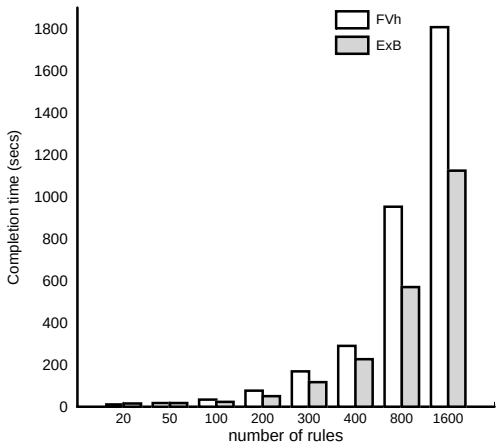

Figure 5: Performance of `snort` vs. number of rules.

4.3 Synthetic rules

4.3.1 Number of rules

To better understand the performance of ExB we construct a synthetic rule-set, where each rule checks every packet against a random 20-character string. To focus on the cost of string matching, rules are applied to all packets. Figure 5 shows the completion time of `snort` using ExB and FVh. We see that for a small number of rules, FVh slightly outperforms ExB , but as the number of rules increases, ExB clearly outperforms FVh. This is because ExB pays the price of pre-processing for each packet. When a packet is checked against several rules, the initial pre-processing overhead is amortized over a larger number of rules, and thus its effect on the total completion time is reduced.

4.3.2 String length

We examine the effect of string length on the performance of ExB and FVh. We use a set of 200 rules that match all TCP/IP headers and search the payload for a random string of given length. Note that the length of strings in the default rule-set are between 2 and 39 bytes, with an average of 14. Figure 6 summarizes the results: ExB outperforms FVh in all cases, but the effect of string length on completion time does not appear to have a very clear trend (and repeated experiments did not improve the picture). A noticable difference is, however, that the relative benefits of ExB are much higher in the case of 2-byte strings. We traced this to the surprisingly poor cache behavior of FVh, considering the large number of L1 data cache misses, as reported (in billions) in the following table:

string length	FVh misses	ExB misses
2	3.38	1.71
20	3.14	3.37

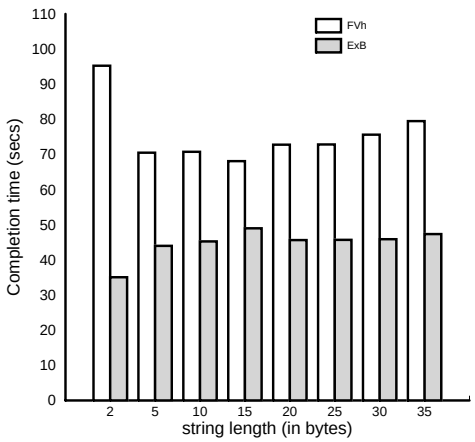

Figure 6: Completion time vs. string length (200 rules)

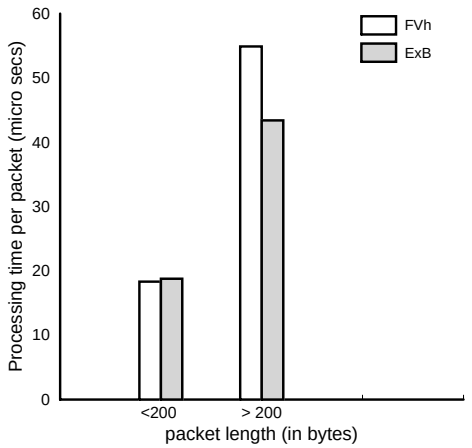

Figure 7: Performance as a function of packet size.

We observe that, although both approaches have a comparable number of cache misses (within 5%) for 20-byte strings, the number of cache misses is significantly reduced for 2-byte strings in ExB, but not in FVh. This is because FVh uses data structures that do not fit in the 8 KB L1 data cache. In contrast, ExB uses a 1 KB data-structure, thereby leaving much more space in the cache for rules and other data structures.

4.4 Packet size

To examine how ExB and FVh perform for different packet sizes, we divide the trace in two sets: one containing "small" packets of less than 200 bytes, and one containing "large" packets of more than 200 bytes. We run `snort` on these two sets and report the results in Figure 7.

We observe that FVh is marginally better for small packets, while for larger packets ExB outperforms FVh by roughly 20 %. This is understandable, as the relative cost of header processing is higher for small packets than for large packets. Therefore, an improved string matching algorithm is unlikely to provide any noticable improvements for small packets.

4.5 Other traces

All results reported so far are based on a single trace. For completeness, we repeated the experiments with other traces from the same data source, using the default rule-set. The results are summarized in Table 3. ExB performs better than FVh for all traces, but the relative improvement varies. The improvement is as high as 69% (three times faster), but there are a few cases where the improvement of ExB is as small as 2%. This coincides with differences in the packet size distribution: the average packet size for `eth2.dump2` is 111 bytes[3]. In contrast, the `eth0.dump7` trace has an average packet size of 1114 bytes. Thus, when processing large packets (as in `eth0.dump7`), `snort` spends a large fraction of its time in string matching, and ExB offers significant benefits. On the contrary, in the case of small packets (as in `eth.dump2`), `snort` spends only a small fraction of its time in string matching, and therefore any improvements in string matching do not affect total completion time.

Trace characteristics			Running time		
trace name (defcon_X)	number of packets	avg. pkt size (bytes)	FVh (sec)	ExB (sec)	improv. (%)
eth0.dump	1119212	658	55.52	34.66	38
eth0.dump2	1035736	835	41.42	30.12	27
eth0.dump3	1238761	693	54.87	30.70	44
eth0.dump4	595267	1481	41.87	13.39	68
eth0.dump5	1468543	582	45.47	19.30	58
eth0.dump6	853988	1023	49.38	15.67	68
eth0.dump7	786446	1114	49.62	15.41	69
eth0.dump8	497302	1111	31.95	9.80	69
eth0.dump9	1464704	111	31.81	31.12	2
eth2.dump2	2467168	241	85.54	83.04	3

Table 3: Completion time of `snort` using various traces

5 Future work

There are a number of questions that remain unanswered by our experiments so far. First, the set of traces used is rather limited; it would be interesting to examine the performance of ExB on a more diverse set of traces, including workloads from production networks, as in [6]. Due to privacy issues it has been generally difficult to obtain such traces for research purposes.

Second, a crucial dimension that has not been explored sufficiently is the effect of processor and memory architecture. Our results suggest that this parameter has a significant

[3]More specifically, these data-sets seem to contain an unusually large number of TCP SYN and ACK packets and ICMP, portscan and shellcode attacks, which tend to be small packets

effect on performance, but it remains to be shown that the benefits of ExB are pervasive.

Third, a theoretical analysis and comparison of ExB to existing algorithms is needed to better understand the relative benefits demonstrated here. Similarly, analyzing algorithm parameters such as the rule-set structure (e.g., distribution of rules per chain header) and the frequency at which different chain headers are invoked in different scenarios could offer valuable insights.

Finally, a more detailed experimental analysis, including a cost breakdown for the various operations of ExB, could lead to further optimizations.

6 Summary and concluding remarks

We have examined the problem of string matching for Network Intrusion Detection Systems, and presented the design of an efficient algorithm called ExB.

We have evaluated ExB against the set of algorithms currently implemented in `snort`using trace-driven execution with real packet traces. The experiments presented in this short paper are by no means exhaustive, and a number of questions remain unanswered, as discussed in the previous section. Nevertheless, based on our results so far we can make the following observations.

First, *ExB string matching appears to be more efficient than the set of algorithms currently used in* `snort` , resulting in a significant overall performance improvement for NIDSes; in some cases ExB makes `snort` three times faster. Second, *the performance benefits of our approach improve with packet size*. Third, *ExB scales well with increasing rule-sets*, consistently outperforming the FVh approach.

Finally, we must note that we expect the relative benefits of improved string matching algorithms such as ExB to be even more pronounced in the future, as network link speeds are likely to continue increasing faster than processor speeds.

Acknowledgements

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. Work of the last 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 Bart Samwel, Herbert Bos, Dionisis Pnevmatikatos, Vasilis Siris, Sotiris Ioannidis, and Stefan Miltchev for their constructive comments.

References

[1] A. Aho and M. Corasick. Fast pattern matching: an aid to bibliographic search. *Commun. 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. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[4] 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.

[5] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of ICALP'79*, pages 118–132, July 1979.

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

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

[9] S. Kim and Y. Kim. A fast multiple string pattern matching algorithm. In *Proceedings of 17th AoM/IAoM Conference on Computer Science*, August 1999.

[10] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.

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

[12] R. L. Rivest. On the worst-case behavior of string-searching algorithms. *SIAM Journal on Computing*, 6(4):669–674, December 1977.

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

[15] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.