INFORMATION SOCIETY TECHNOLOGIES (IST) PROGRAMME

A Scaleable Monitoring Platform for the Internet
Contract No. IST-2001-32404

E2.1 "Distributed Security Applications for the Internet using SCAMPI"

Abstract: This document describes the security applications that have been developed as part of the SCAMPI project, namely, the *Intrusion Detection Application* and the *Denial of Service Attack Detection Application*. We present the implementation of the applications, the algorithms and techniques they incorporate, and several aspects regarding their extension to a distributed security infrastructure.

Contractual Date of Delivery	Not Applicable
Actual Date of Delivery	4 May 2004
Deliverable Security Class	Public

The SCAMPI Consortium consists of:

TERENA	Coordinator	The Netherlands
IMEC	Principal Contractor	Belgium
FORTH	Principal Contractor	Greece
LIACS	Principal Contractor	The Netherlands
NETikos	Principal Contractor	Italy
Uninett	Principal Contractor	Norway
CESNET	Principal Contractor	Czech Republic
FORTHnet	Principal Contractor	Greece
Masaryk University	Principal Contractor	Czech Republic
Siemens	Principal Contractor	Germany

Contents

1 Introduction **7**

2 Intrusion Detection Application **9**
2.1 String Searching Algorithm 10
2.1.1 Background 10
2.1.2 E^2xB: Exclusion-based string matching 12
2.1.3 Experimental evaluation 14
2.2 Implementation 20
2.3 Demonstrator 22

3 Denial of Service Attack detection Application **25**
3.1 High-level description 26
3.1.1 Anomaly detection algorithms 26
3.1.2 DoS attack detection application 27
3.2 Performance evaluation 28
3.2.1 High intensity attacks 29
3.2.2 Low intensity attacks 30

4 Distributed Security Applications **35**
4.1 Motivation 35
4.2 Recent Efforts in Cooperative Security Analysis. 37
4.2.1 Distribution and Collection of IDS Data. 37
4.2.2 Cooperative Intrusion Detection Systems 38
4.3 Extending Current Security Applications 38
4.3.1 Data Sharing and Privacy 38
4.3.2 Data Correlation 43

List of Figures

2.1 Pseudo-code for E^2xB pre-processing and search. 13
2.2 Effect of element and cell size parameters on the fraction of false positives 14
2.3 Effect of element and cell size parameters on running time 14
2.4 Executed instructions. 16
2.5 L2 data cache misses. 16
2.6 Distribution of the number of rules checked per packet and byte (rules rarely triggered are not presented) 17
2.7 Running time, per-packet and per-byte 18
2.8 Uniformly random vs. real payloads 20
2.9 Comparison between Snort NIDS and MAPI-based NIDS. 21

3.1 Adaptive threshold algorithm. An alarm is signaled when the measurements exceed a threshold, for a number of consecutive intervals k. 27
3.2 CUSUM algorithm. An alarm is signaled when the accumulated volume of measurements g_i that are above some traffic threshold, exceed some aggregate volume threshold h. 27
3.3 High intensity attacks. Both the adaptive threshold and the CUSUM algorithm have very good performance. 30
3.4 Low intensity attacks. The performance of the adaptive threshold algorithm has deteriorated significantly compared to its performance for high intensity attacks. On the other hand, the performance of the CUSUM algorithm remains very good. 31
3.5 Detection probability and false alarm ratio tradeoff for low intensity attacks. The CUSUM algorithm has better performance compared to the adaptive threshold algorithm (better performance corresponds to points towards the lower-right). 32
3.6 False alarm ratio and detection probability for CUSUM algorithm and for the algorithm in [25]. 33
3.7 False alarm ratio and detection delay tradeoff for the adaptive threshold and the CUSUM algorithms for low intensity attacks. Better performance corresponds to points towards the lower-left. 33

4.1 Example of the most popular packets seen from different sensors. Packet `D` is among the popular packets of each sensor, thus it may belong to a self-replicating virus. 40
4.2 Data structure to find the most popular packets with the same content. Each field holds the count of the packets with the corresponding payload digest. 40
4.3 Data structure to find the most popular packets with size greater than 200 bytes. The structure of Figure 4.2 is enhanced with a BPF-like rule. 41
4.4 Data structure to find the most popular packets with size greater than 200 bytes and with ten different source and/or destination IP address pairs. The structure of Figure 4.3 is enhanced with an additional hash table for each field, which holds the occurences of different source-destination pairs. 42
4.5 Most popular packets captured in the network of ICS-FORTH. From all the packets captured in a subset of the internal network of ICS-FORTH, we printed the most popular ones that had at least 10 different source-destination pairs and the same destination port. The packets belonged to the Welchia worm. 43

Chapter 1

Introduction

Security applications are an important part of any modern network infrastructure. Depending on the method used to detect attacks, security applications are either *host-based* or *network-based*. Host-based applications observe the activity on a specific host in order to detect malicious attempts by monitoring system call traces, system files integrity and application or firewall logs. Network-based applications constantly monitor network traffic in passive mode, trying to detect cyber-attacks such as intrusion attempts, Denial of Service (DoS) attacks, worms, or port scans. Network attack detection applications perform either *misuse detection*, by comparing the traffic against well-defined signatures of attacks, or *anomaly detection*, by reacting to statistical anomalies of the traffic observed under normal conditions. As part of the SCAMPI project we have developed two security applications, namely, an *Intrusion Detection Application* and a *Denial of Service Attack Detection Application*.

Network Intrusion Detection Systems (NIDS) constantly inspect network traffic, trying to detect attacks by matching packet data against a set of specific patterns. Each of these patterns, or rules, describes one known intrusion threat. To detect malicious activity, NIDSes often need to inspect the payload of incoming packets for such patterns. For example, a packet directed to port 80 containing in its payload the string /bin/perl.exe is probably an indication of an attacker trying to compromise a web server. This attack can be detected by a rule which checks the destination port number, and defines a string search for /bin/perl.exe in packet body. String matching is a generally expensive operation which constitutes up to 80% of the total processing time of modern NIDSes [8], and bounds their efficiency and capability to keep up with the increasing link speeds. In this context we have designed E^2xB, a string matching algorithm that is tailored to the specific characteristics of NIDS string matching.

Denial of Service attack detection applications have the goal of detecting attacks that attempt to disrupt the ability of a provider to offer service, at a particular performance level, to legitimate users. DoS attacks typically involve flooding a particular network node or server with traffic at a rate much higher than the node

or server can handle. There are also low-bandwidth DoS attacks, known as *algorithmic attacks*, which use deliberately crafted packets to exploit algorithmic deficiencies in the data structures of widely used applications [7]. A common feature of DoS attacks is that they lead to changes in measurable characteristics of network traffic. Such characteristics can include the type and size of packets, the number of half open connections, and the rate of packets associated with a particular application or port number. Based on this property of changes in network characteristics, DoS attack detection applications are commonly based on anomaly detection models, where the behavior of a measurable network characteristic is compared to its normal behavior, in order to detect significant and abrupt deviations. An advantage of anomaly detection systems is that they do not require any a priori specification of attack signatures, hence they can detect new types of attacks. The DoS attack detection application is based on time series models, which take into account the time correlations of network traffic measurements.

The increasing spread of new sophisticated types of cyber-attacks such as polymorphic worms, coordinated DoS attacks or distributed port scans, and the verbosity and ambiguity of the alerts of existing network security applications, give rise for more advanced and scalable solutions through the use of *distributed* security applications. In the following sections we present the two security applications that have been developed as part of the SCAMPI project, the algorithms and techniques they incorporate, and several aspects regarding their extension to a distributed environment.

Chapter 2

Intrusion Detection Application

Network-based Intrusion Detection is a research and development area that aims to improve the security of our cyberinfrastructure through the early detection of intrusion attempts. Network-based Intrusion Detection is usually deployed in the core (or the edge) of the Internet in order to identify possible intrusions as they are being launched. After a possible intrusion is identified, all the information regarding the intrusion is being logged, and the administrators of the system are (optionally) being alerted. The administrators, in turn, may take corrective measures to reduce the effects of this intrusion and possibly patch the security hole that led to the intrusion.

Network-based Intrusion Detection Systems (nIDSes) are usually based on a set of rules (also called signatures). Each possible type of intrusion is described by one or more rules. For example, the following rule describes an attempt by an outsider to become super-user (i.e. *root*) in one of the local systems: *OUTSIDE_NETWORK → LOCAL_NETWORK TCP 23 content "su root"*. The above rule states that if a packet is sent from a computer located in the *OUTSIDE_NETWORK* (an alias for all computers outside the monitored organization) towards a computer in the *LOCAL_NETWORK* (an alias for all computers in the monitored organization) on *port 23* (the telnet port) using the protocol TCP, and the payload of the packet contains the substring *"su root"*, then this is a possible intrusion attempt. In a Network-based Intrusion Detection System each packet is checked against every rule. If the packet matches a rule, it is logged and the administrators may be notified.

We have implemented an Intrusion Detection application using the MAPI interface. We capitalize on our experience with similar publicly available applications, such as `snort`[19], in order to optimize the application for high-speed networks. We have also incorporated into MAPI advanced string searching algorithms targeted to improve the content inspection of network packets, a core function of Intrusion Detection systems.

2.1 String Searching Algorithm

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 traditional efficient string matching algorithms for this purpose, such as [1, 4], bears a significant cost: recent measurements of the `snort` nIDS [19] on a production network show that as much as 31% of total processing is due to string-matching [8]. 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 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 [9, 13]. In this context, we have designed E^2xB, a string matching algorithm that is targeted 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 [4]. 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.

2.1.1 Background

The general problem of designing algorithms for string matching is well-researched. We summarize the key characteristics of string matching algorithms that have been recently used in the nIDS context.

2.1.1.1 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 [11] improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic.

2.1.1.2 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.

2.1.1.3 Set-wise Boyer-Moore

Fisk and Varghese [8] 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 [10].

2.1.1.4 Wu-Manber

The most recent implementation of `snort` uses a simplified variant of the Wu-Manber multi-pattern matching algorithm [26], as discussed in [21]. 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 [21] 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.

2.1.2 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 algorithm (e.g., Boyer-Moore-Horspool) to confirm whether s is actually a substring of I or not. [1]

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 bitmap*. Specifically, we first *pre-process* the input string I, and for each character c that appears in string I, we mark the corresponding (i.e. c_{th}) *cell* on the (256-cell) bitmap. After pre-processing, we know that if the c_{th} position of the cell map is 1, then the character c appears in I, otherwise it does not.

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 character 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.

Our experiments in [14] suggest that the cost of cleaning (i.e. filling with zeros) the cell map during the pre-processing 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 [2]. This

[1] 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*.

[2] 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)
{
  /* if pktno is greater than 2^cellsize - 1, set it to 0 */
  pktid=pktno & (1<<cellsize - 1);

  for (int idx = 0 ; idx < len-1 ; idx++) {

      /* xor the character at the current position (idx)
       * with the next character (idx+1) to form an element */
      element = input[idx]<< (elementsize-8) ^ input[idx+1];

      /* set the corresponding position at the occurrence map */
      occurrence_map[element] = pktid;
  }
}

search(char *s, char *input, int len_s, int len)
{
  for (int idx = 0 ; idx < len_s-1 ; idx++) {

      /* xor the character at the current position (idx)
       * with the next character (idx+1) to form an element */
      element= s[idx]<< (elementsize-8) ^ s[idx+1];

      /* check if this element exists in input */
      if ( occurrence_map[element] != pktid)
          return DOES_NOT_EXIST ;
  }
  return boyer_moore(s, len_s, input, len);
}
```

Figure 2.1: Pseudo-code for E^2xB pre-processing and search.

is a major improvement to the ExB algorithm [14], a precursor of E^2xB, which assumed an occurrence *bitmap* where each element was marked by setting the 1-bit cell to 1. E^2xB avoids the unnecessary overhead of clearing the bitmap for each new packet by using the "epoch" approach. The pseudo-code for pre-processing `input` and for matching a string `s` on `input` is presented in Figure 2.1.

A second difference between E^2xB and ExB 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 are case-insensitive. This is done by modifying the *search* procedure to test for the occurrence of all four combinations of upper- and lower-case for each of the two

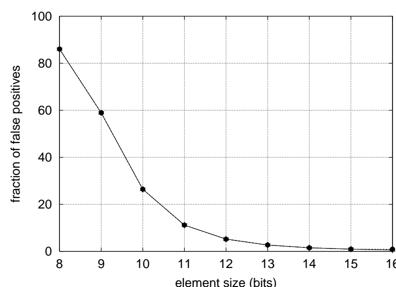


Figure 2.2: Effect of element and cell size parameters on the fraction of false positives

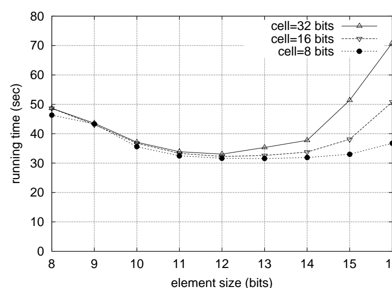


Figure 2.3: Effect of element and cell size parameters on running time

bytes used to compute the element index.

2.1.3 Experimental evaluation

Using trace-driven execution, we evaluate the performance of E^2xB against the Fisk and Varghese algorithm [8] (denoted as FVh in the rest of this document) and the implementations of the Boyer-Moore [4] and the Wu-Manber multi-pattern matching algorithm in `snort` [26].

2.1.3.1 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`[16]. 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:

- A set of full-packet traces from the DEFCON "capture the flag" data-set. [3] These traces contain numerous intrusion attempts.

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

Trace characteristics				Running time			
trace name	ID	nr. of packets	avg.pkt (bytes)	BM (sec)	FVh (sec)	E^2xB (sec)	% improvement
eth0.dump2	D.02	1,035,736	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	595,267	1481	14.11	56.24	9.81	+30.47
eth0.dump8	D.08	497,302	1111	9.79	41.51	6.74	+31.15
webtrace	W.0	1,188,660	761	345.60	300.86	274.51	+8.76
NLANR IND	N.IND	2,254,931	703	93.53	83.8	62.04	+25.97
NLANR MRA	N.MRA	2,760,531	760	137.39	122.40	89.07	+27.23
NLANR AIX	N.AIX	1,624,223	364	13.17	14.00	14.26	-8.28
UCNET 0000	UC.00	1,564,131	422	103.93	82.35	66.84	+18.83
UCNET 0100	UC.01	2,245,938	413	108.69	84.20	62.54	+25.72

Table 2.1: Completion time of `snort` with different string matching algorithms – all traces

- 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)[6], with random payloads.

For the experiments of Sections 2.1.3.2 and 2.1.3.3, 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.)

2.1.3.2 Element and cell size

We first determine the optimal size for E^2xB elements and cells. In Figure 2.2 we show the fraction of false positives for different element and cell sizes, and in Figure 2.3 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 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,

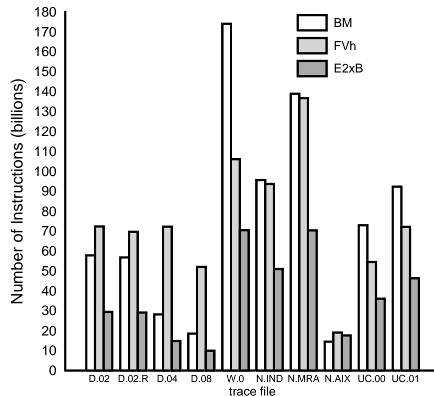


Figure 2.4: Executed instructions.

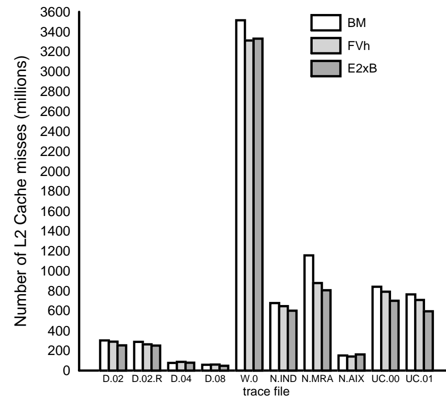


Figure 2.5: L2 data cache misses.

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.

2.1.3.3 Experiments with the default rule-set

We determine if E^2xB offers any overall improvement compared to FVh and BM using the `eth0.dump2` 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.

2.1.3.4 Other packet traces

We repeated the experiments with the three algorithms on the full set of traces. The results are summarized in Table 2.1. 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.

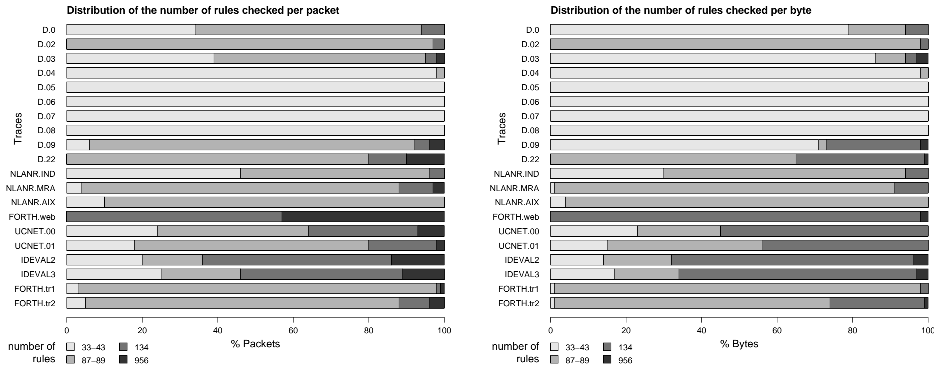


Figure 2.6: Distribution of the number of rules checked per packet and byte (rules rarely triggered are not presented)

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 [8] 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 matching, 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 [22]. The results are presented in Figures 2.4 and 2.5. We observe that the number of instructions for E^2xB is significantly smaller in all cases except for the AIX trace. The reduction in 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.

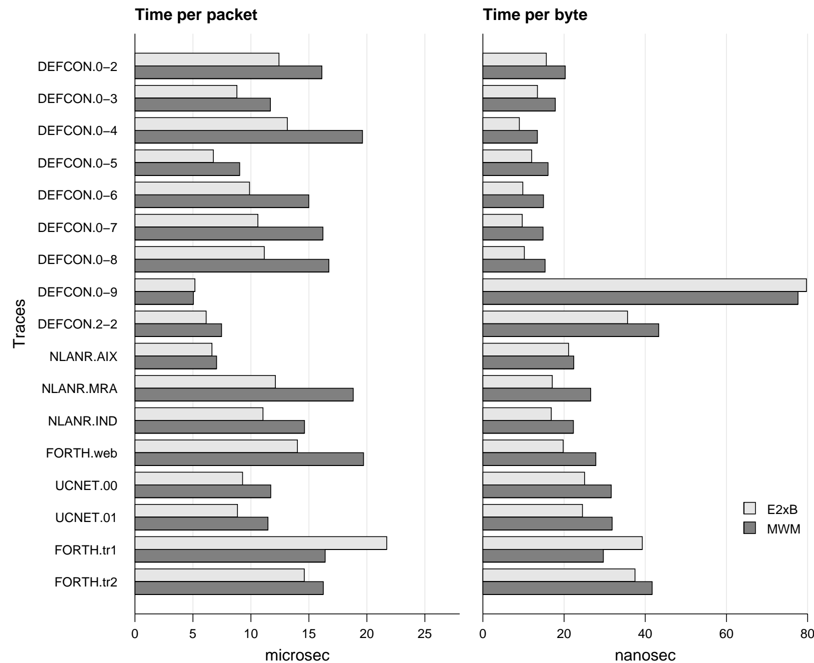


Figure 2.7: Running time, per-packet and per-byte

2.1.3.5 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 2.6. We observe that the use of different chain groups varies significantly. Roughly 42% of packets in the FORTH.web trace trigger 956 rules, while this is true only for the 1-5% of the packets 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 measure the performance of E^2xB and the variant of the Wu-Manber multi-pattern matching algorithm used in `snort` [21], on each of the above mentioned traces. For header-only traces we add uniformly random payloads, assuming this provides representative results – in Section 2.1.3.6 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 2.7. The most striking observation is that the mean processing time per-packet is between 5.04 and 19.71 μs for MWM and 5.17 to 21.71 μs for E^2xB that is, the highest cost is

about 4 times the lowest cost, and only 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 and FORTH.tr1 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. However, 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 uniformly random payloads. We will discuss next how this affects the results.

2.1.3.6 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 2.8.

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 uniformly 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.

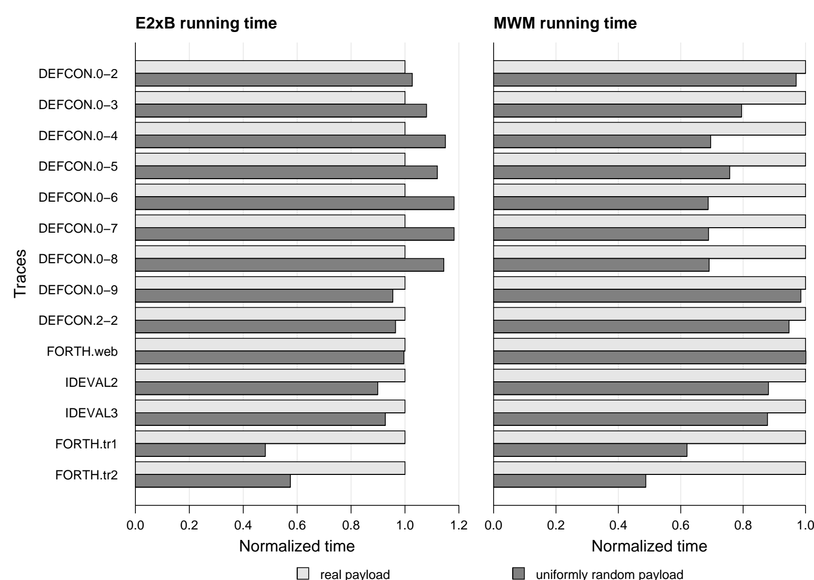


Figure 2.8: Uniformly random vs. real payloads

2.2 Implementation

Implementing a NIDS is rather a complicated task. Several basic operations like packet decoding, filtering, and classification, TCP/IP stream reconstruction, and string searching, must be crafted together to form a fully functional system. Each one of these operations alone requires deliberate decisions for its design, and considerable programming effort for its implementation. Furthermore, the resulting system is usually targeted to a specific hardware platform. For instance, the majority of current NIDSes are built on top of `libpcap` [15] packet capture library using commodity network interfaces set in promiscuous mode. As a result, given that `libpcap` provides only basic packet delivery and filtering capabilities, the programmer has to provide considerable amount of code to implement the large and diverse space of operations and algorithms required by a NIDS.

In contrast, MAPI inherently supports the majority of the above operations in the form of functions which can be applied to network flows, and thus, can be effectively used for the development of a complete NIDS. Consequently, a great burden is released from the programmer who has now a considerably easier task. As a matter of fact, we have developed a Network Intrusion Detection System based on MAPI. Based on the observation that a rule which describes a known intrusion threat can be represented by a corresponding network flow, overall implementation is straightforward. As an example, consider the following rule taken from the popular Snort [19] NIDS, which describes an IMAP buffer overflow attack:

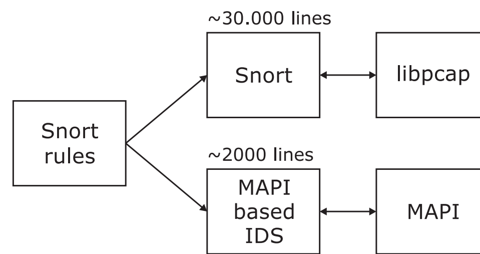


Figure 2.9: Comparison between Snort NIDS and MAPI-based NIDS.

```
alert tcp any any -> 139.91/16 143 (flags: PA;
      content:"|E8C0FFFFFF|/bin";
      msg: "IMAP buffer overflow";)
```

All packets that match this particular rule can also be returned by the following network flow, after the application of the appropriate MAPI functions:

```
fd = mapi_create_flow(dev);
mapi_apply_function(fd, BPF_FILTER,
      "tcp and dst host 139.91 and dst port 143");
mapi_apply_function(fd, TCP_FLAGS, "PA");
mapi_apply_function(fd, STR_SEARCH, "|E8C0FFFFFF|/bin");
```

Our MAPI-based NIDS operates as follows: During program start-up, the files that contain the set of rules are parsed, and for each rule, a corresponding network flow is created. Rules are written in the same description language used by Snort. Snort rules are converted by a separate module to the appropriate MAPI function elements, which are then applied to the related network flow. The rest of the functionality is left to MAPI, which will optimize the functional components of all the defined rules and deliver the packets that match any of them.

Implementing the above intrusion detection application using `libpcap` would have resulted in longer code and higher overheads. As shown in Figure 2.9, our implementation takes no more than 2000 lines of code, while the core functionality of other popular NIDSes, such as Snort, consists of roughly 30.000 lines of code[4]. For example, `libpcap` does not provide any string searching facility, and thus the programmer would have to provide a significant chunk of code for the implementation of the chosen string searching algorithm. Instead of forcing the programmer to provide all this mundane code, MAPI already provides this frequently used functionality.

[4]Although the functionality of the two systems is not identical, it is clearly depicted a difference in code length of at least one order of magnitude.

Note that a NIDS based on MAPI is not restricted to a specific hardware platform. MAPI operates on top of a diverse range of monitoring hardware, including more sophisticated lower level components like network processors, and thus, can further optimize overall system performance, considering that certain MAPI functions can be pushed to the hardware. Additionally, the functionality of the MAPI daemon can be shared by multiple concurrently running applications. For example, along with the intrusion detection application, one can develop a firewall application in the same fashion (i.e., in a few lines of code), adding this way extra capabilities to the overall system. Again, instead of providing code for the whole firewall operations, the programmer can use MAPI to reduce the development effort, and to effectively share resources by pushing the core firewall functionality into the MAPI daemon.

2.3 Demonstrator

Security applications such as Intrusion Detection Systems, usually need to analyze full packet payloads in order to detect malicious content. The intrusion detection application will be driven with full packet workload from:

Synthetic traces: Traces gathered on controlled networks which contain instances of intentionally reproduced attacks.

Real traces: Publicly available traces containing actual attacks.

Being written on top of MAPI, the application does not heavily depend on the underlying hardware: any monitoring system that supports MAPI is a suitable platform for our application. For the purpose of the demonstration we will run the application on top of the phase I SCAMPI adapter with 1000BASE-TX interface. There are several techniques to fed the test traffic to the application:

Off-line trace: Read traffic off-line from disk at the monitoring device. The intrusion detection application is running at the same device.

Replay trace: Replay the trace from another machine to produce traffic which is captured by the monitoring device. The traffic can be sent to this monitoring device using a switch configured with port mirroring or using an optical splitter. The traffic is analyzed on-the-fly by the intrusion detection application.

Real traffic: The application is fed with real traffic on production networks. However, since network-based intrusion detection applications inspect both the headers and the payload of the network packets, privacy concerns may be raised.

The detection of intrusions is based on a well-known set of rules, such as the one distributed by snort. Since the implemented intrusion detection application

uses the same syntax for the signatures, the latest set of snort rules will be used. In order to evaluate the operation of the application we will compare the its performance against the widely used snort IDS driven with the same set of signatures. Particularly, we will consider the following performance aspects:

Correct alerts: Successful identification of attacks that exist in the test traffic.

False positives: Positive responses for attacks that do not exist in the test traffic.

Sustained traffic: The maximum traffic throughput that the application can sustain without dropping packets.

Chapter 3

Denial of Service Attack detection Application

Over the past few years many sites on the Internet have been the target of denial of service (DoS) attacks, among which TCP SYN flooding is the most prevalent [18]. Indeed, recent studies[1] have shown an increase of such attacks, which can result in disruption of services that costs from several millions to billions of dollars.

The aim of denial of service attacks are to consume a large amount of resources, thus preventing legitimate users from receiving service with some minimum performance. TCP SYN flooding exploits the TCP's three-way handshake mechanism and its limitation in maintaining half-open connections. Any system connected to the Internet and providing TCP-based network services, such as a Web server, FTP server, or mail server, is potentially subject to this attack. A TCP connection starts with the client sending a SYN message to the server, indicating the client's intention to establish a TCP connection. The server replies with a SYN/ACK message to acknowledge that it has received the initial SYN message, and at the same time reserves an entry in its connection table and buffer space. After this exchange, the TCP connection is considered to be half open. To complete the TCP connection establishment, the client must reply to the server with an ACK message. In a TCP SYN flooding attack, an attacker, from a large number of compromised clients in the case of distributed DoS attacks, sends many SYN messages, with fictitious (spoofed) IP addresses, to a single server (victim). Although the server replies with SYN/ACK messages, these messages are never acknowledged by the client. As a result, many half-open connections exist on the server, consuming its resources. This continues until the server has consumed all its resources, hence can no longer accept new TCP connection requests.

A common feature of DoS attacks is that they lead to changes in a measured statistic of a network traffic flow. Such statistics can include the type and size of packets, the number of half open connections, and the rate of packets associated

[1] 2002 and 2003 CSI/FBI Cybercrime Survey Report. The 2003 report indicates that DoS attacks alone were responsible for a loss of $65 million.

with a particular application or port number; in the case of TCP SYN flooding the statistic is the number of TCP SYN packets. Based on the aforementioned property, DoS attack detection applications are commonly based on anomaly detection models, where the behavior of a measurable network characteristic is compared to its normal behavior, in order to detect deviations. An advantage of anomaly detection systems is that they do not require any a priori specification of attack signatures, hence they can detect new types of attacks. One approach for describing normal behavior is to use a static characterization; such an approach has the disadvantage of not adapting to trends and periodic behavior of normal traffic, e.g. the load of a networking system is much higher during peak hours compared to non-peak hours, which may eventually lead to an increased false alarm rate. Hence, anomaly detection systems should adaptively learn normal behavior, in order to track trends and periodic behavior.

We present a DoS attack detection application that implements two statistical anomaly detection algorithms which adaptively learn the normal behavior of the system to which they are applied: an adaptive threshold algorithm and a particular application of the cumulative sum (CUSUM) algorithm for change point detection. Then we present an evaluation of the two algorithms for detecting TCP SYN attacks. Our focus is on investigating the tradeoffs between the detection probability, the false alarm ratio, and the detection delay, and how these tradeoffs are affected by the parameters of the detection algorithm and the characteristics of the attacks. Such an investigation can assist in tuning the parameters of the detection algorithm to satisfy specific performance requirements. Our results show that although simple and straightforward algorithms, such as the adaptive threshold algorithm, can exhibit good performance for high intensity attacks, their performance deteriorates for low intensity attacks. On the other hand, algorithms based on a strong theoretical foundation can exhibit robust performance over various attack types, and without necessarily being complex or costly to implement. Detection of low intensity attacks is particularly important since this would enable the early detection of attacks whose intensity slowly increases, and the detection of attacks close to the sources, thus facilitating the identification of compromised hosts that are participating in distributed DoS attacks [25].

3.1 High-level description

In this section we present a high level description of the DoS attack detection application, and the two anomaly detection algorithms that it implements.

3.1.1 Anomaly detection algorithms

The two anomaly detection algorithms that are implemented by the DoS attack detection application try to detect changes in some statistic of the traffic flow, based on measurements of the statistic in consecutive intervals of the same duration. In

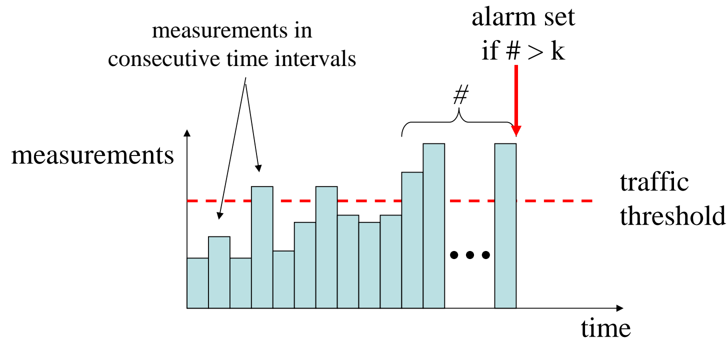


Figure 3.1: Adaptive threshold algorithm. An alarm is signaled when the measurements exceed a threshold, for a number of consecutive intervals k.

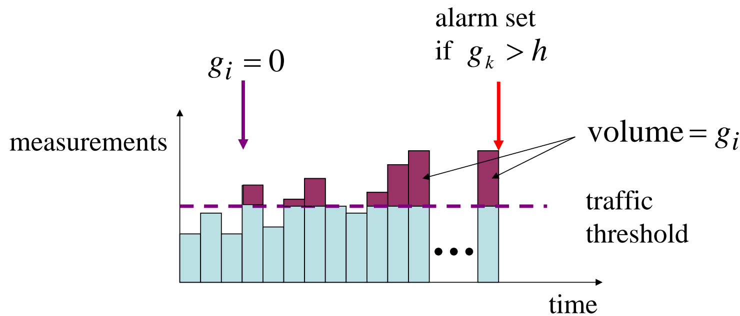


Figure 3.2: CUSUM algorithm. An alarm is signaled when the accumulated volume of measurements g_i that are above some traffic threshold, exceed some aggregate volume threshold h.

both algorithms, the normal behavior is adaptively estimated from measurements of the mean rate.

The adaptive threshold algorithm signals an alarm when the measurements exceed some threshold $(\alpha + 1)\mu$, where μ is the measured mean rate, for a number of consecutive intervals k, Figure 3.1.

The CUSUM algorithm signals an alarm when the accumulated volume of measurements g_i that are above some traffic threshold, which as with the adaptive threshold algorithm is given by $(\alpha + 1)\mu$, where μ is the measured mean rate, exceeds an aggregate volume threshold h, Figure 3.2.

3.1.2 DoS attack detection application

The DoS attack detection application works online, and takes as input load measurements in consecutive intervals of the same duration; this duration is typically

a	percentage above which the measurement is potentially considered to be due to an attack, default value 0.5
b	parameter of the moving average algorithm used to estimate the mean rate, default value 0.98
k	number of consecutive intervals above which, if the measurements exceed the traffic threshold, an alarm is signaled
i	duration (in seconds) of the intervals in which traffic measurements are taken, default value 10 seconds

Table 3.1: Parameters of the adaptive threshold algorithm

of the order of tens of seconds. The measurements in consecutive time intervals are taken continuously using the MAPI interface, using its "PKT_COUNT" function. In its first version, the application detects SYN flooding attacks, hence the measurements consist of the aggregate number of SYN packets in consecutive time intervals; later versions will include other statistics, such as the number of UDP packets and the number of ICMP packets. The measurements are processed according to the two anomaly detection algorithms described in the previous subsections, based on which an alarm is signaled when a change of the normal behavior is detected.

Two different programs implement the two anomaly detection algorithms: the adaptive threshold algorithm is implemented by the program `adaptive`, whereas the CUSUM anomaly detection algorithm is implemented by the program `CUSUM`. The user is allowed to set specific values for the parameters of each algorithm. The command line for executing the program `adaptive` is

```
adaptive <a> <b> <k> <i>
```

where the different parameters are explained in Table 3.1. The command line for executing the program `cusum` is

```
cusum <a> <b> <h> <i>
```

where the different parameters are explained in Table 3.2.

The sensitivity of the two algorithms, as discussed in the experiments we present later, is determined mainly by the parameter k for the adaptive threshold algorithm, and h for the CUSUM algorithm. For this reason, the algorithms can be executed by giving only this parameter, in which case the other parameters obtain the default values shown in Tables 3.1 and 3.2.

3.2 Performance evaluation

In this section we investigate the performance of the two algorithms presented in the previous section for detecting TCP SYN flooding attacks. The performance metrics considered include the detection probability, the false alarm rate, and the detection delay. In addition to investigating the tradeoffs between these metrics,

a	percentage above which the measurement is potentially considered to be due to an attack, default value 0.5
b	parameter of the moving average algorithm used to estimate the mean rate, default value 0.98
h	aggregate volume threshold above which an alarm is signaled
i	duration (in seconds) of the intervals in which traffic measurements are taken, default value 10 seconds

Table 3.2: Parameters of the CUSUM algorithm

we seek to investigate how the parameters of the detection algorithm and the characteristics of the attack affect the performance.

Our experiments used actual network traffic taken from the MIT Lincoln Laboratory[2]. We used trace data taken during two days, with the trace from each day containing 11 hours of collected packets (08.00-19.00). The first investigations that we present considered SYN packet measurements in 10 second intervals.

The attacks were generated synthetically; this allowed us to control the characteristics of the attacks, hence to investigate the performance of the detection algorithms for different attack types. The duration of one attack was normally distributed with mean 60 time intervals (10 minutes assuming 10 second intervals) and variance 10 time intervals. We consider both attacks whose intensity increases abruptly, i.e. reaches its peak amplitude in one time interval, and attacks whose intensity increases gradually. The inter-arrival time between consecutive attacks was exponentially distributed, with mean value 460 time intervals (approximately 77 minutes assuming 10 second intervals); this results in approximately 8 attacks in an 11 hour period.

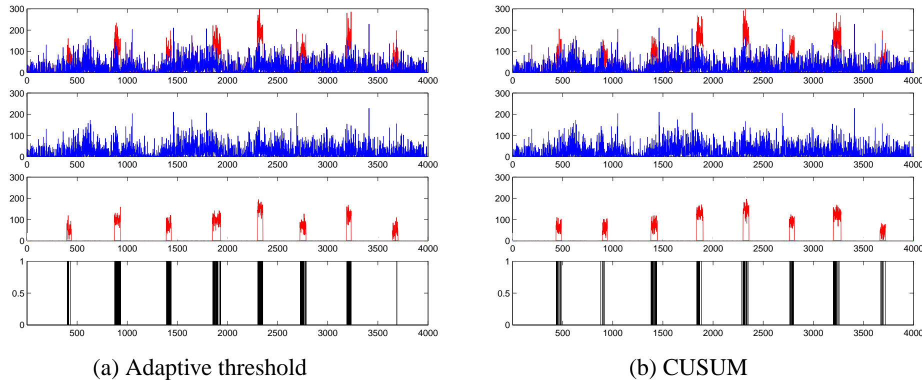
The detection probability is the percentage of attacks for which an alarm was raised, and the false alarm ratio (FAR) is the percentage of alarms that did not correspond to an actual attack. Unless otherwise noted, the parameters we considered for the adaptive threshold algorithm were $\alpha = 0.5$, $k = 4$, and $\beta = 0.98$, and the parameters for the CUSUM algorithm were $\alpha = 0.5$, $h = 5$, and $\beta = 0.98$.

3.2.1 High intensity attacks

Our first experiment considered high intensity attacks, whose mean amplitude was 250% higher than the mean traffic rate, which was approximately 31.64 SYN packets in one time interval; the length of the time interval was 10 seconds.

Figures 3.3(a) and 3.3(b) show the results for the adaptive threshold and the CUSUM algorithm, respectively. The horizontal axis in these figures is the number of time interval, with 0 and 4000 corresponding approximately to 8:00 and 19:00, respectively. In each graph, from top to bottom, we have the traffic trace with attacks, the original traffic trace without attacks, the attacks only, and finally the bottom graph shows the time intervals where an alarm was raised.

[2]DARPA intrusion detection evaluation: `http://www.ll.mit.edu/IST/ideval`



(a) Adaptive threshold (b) CUSUM

Figure 3.3: High intensity attacks. Both the adaptive threshold and the CUSUM algorithm have very good performance.

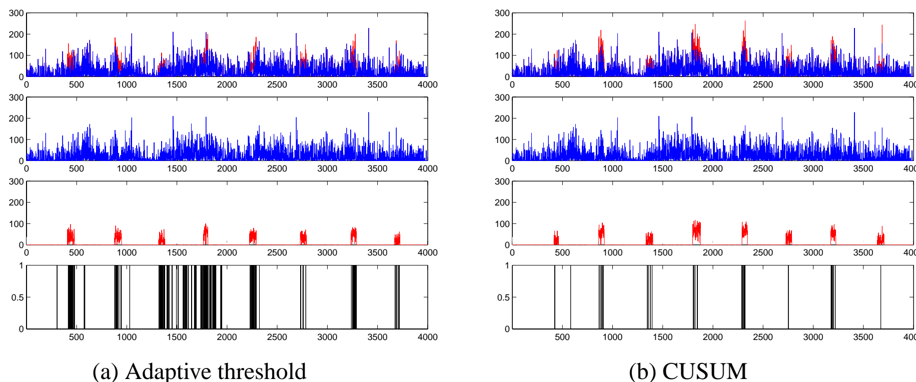
The above graphs show that both the adaptive threshold and the CUSUM algorithm have excellent performance in the case of high intensity attacks, since they both yielded a detection probability of 100% and a false alarm ratio (FAR) of 0%. The detection delay was very close: 3.01 and 2.75 time intervals, respectively.

3.2.2 Low intensity attacks

Next we investigate the performance of the attack detection algorithms in the case of low intensity attacks, whose mean amplitude is 50% of the traffic's actual mean rate. Detection of low intensity attacks is important for two reasons: First, early detection of DoS attacks with increasing intensity would enable defensive actions to be taken earlier. Second, detection of low intensity attacks would enable the detection of attacks close to the sources, since such a placement of detection algorithms can facilitate the identification of stations that are participating in a distributed DoS attack.

Figure 3.4(a) shows that for low intensity attacks the performance of the adaptive threshold algorithm has deteriorated significantly, giving a very high FAR equal to 32%. On the other hand, Figure 3.4(b) shows that the performance of the CUSUM algorithm remains close to its performance in the case of high intensity attacks, namely the FAR was less than 9%. Nevertheless, the detection delay of the CUSUM algorithm has increased to 10.25 time intervals, from only 2.75 time intervals in the case of high intensity attacks. Note that the detection probability for both algorithms was 100%.

The difference in the performance of the adaptive threshold and the CUSUM algorithms lies in the way each maintains memory: the adaptive threshold algorithm has memory of whether the threshold was violated or not in the previous $k-1$ time intervals. On the other hand, the CUSUM algorithm maintains finer information on the amount of data exceeding the amount expected based on some



(a) Adaptive threshold (b) CUSUM

Figure 3.4: Low intensity attacks. The performance of the adaptive threshold algorithm has deteriorated significantly compared to its performance for high intensity attacks. On the other hand, the performance of the CUSUM algorithm remains very good.

estimated mean rate.

3.2.2.1 Tradeoff between detection probability and false alarm ratio

The above results were for specific values of the parameters of the two detection algorithms. Next we investigate the tradeoff between the detection probability and the false alarm ratio (FAR) for different values of k (with range 1 - 10) for the adaptive threshold algorithm, and h (with range 1 - 10, for the MIT trace) for the CUSUM algorithm.

Figures 3.5(a) and 3.5(b) show the results in the case of low intensity attacks for the adaptive threshold and the CUSUM algorithm, respectively. Each point in the graph corresponds to a different value of the tuning parameter, k or h. The data for each point was the average of 50 runs. An algorithm has better performance when the points corresponding to the detection probability and FAR pair are located towards the lower-right of the graph. Observe that the CUSUM algorithm exhibits better performance, supporting our observation in the previous section.

Figure 3.6(a) and 3.6(b) shows the performance of the CUSUM and of the algorithm in [25], for traces from the University of Crete (the range of h was now 10-100). The algorithm of [25] is given by

$$g_n = \left[g_{n-1} + (X_n - a')\right]^+ ,$$

where X_n is the (# of SYN pkts - # of FIN pkts)/(average # FIN pkts). The graph in Figure 3.6(b) was obtained for an alarm threshold $h = 9$, and for $a' = 1 - 10$. Observe that the CUSUM algorithm discussed in this paper has better performance than the algorithm in [25].

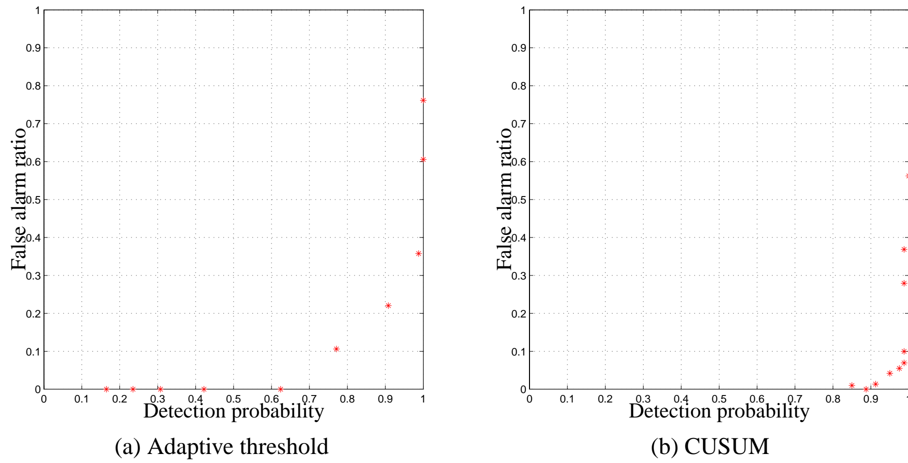
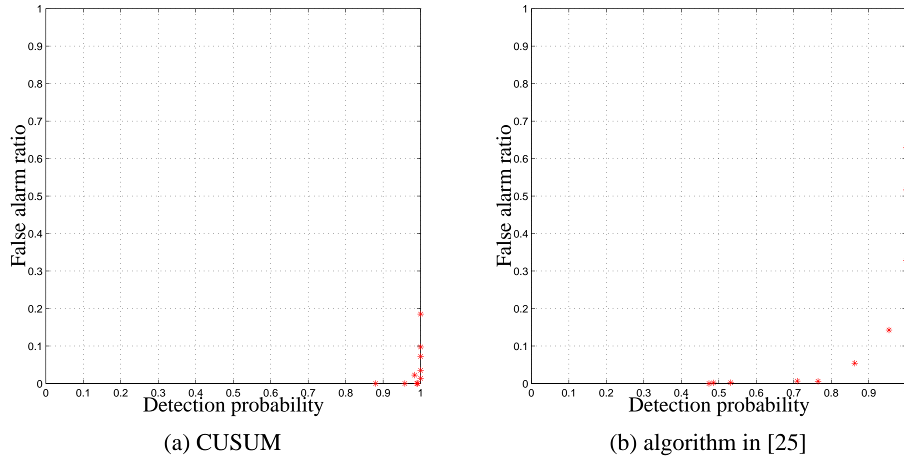


Figure 3.5: Detection probability and false alarm ratio tradeoff for low intensity attacks. The CUSUM algorithm has better performance compared to the adaptive threshold algorithm (better performance corresponds to points towards the lower-right).

Graphs such as those in Figure 3.5 and Figure 3.6 can assist in the tuning of the parameters of the detection algorithm. Indeed, note that the alarm threshold h is different for different traces, and controls the sensitivity of the attack detection.

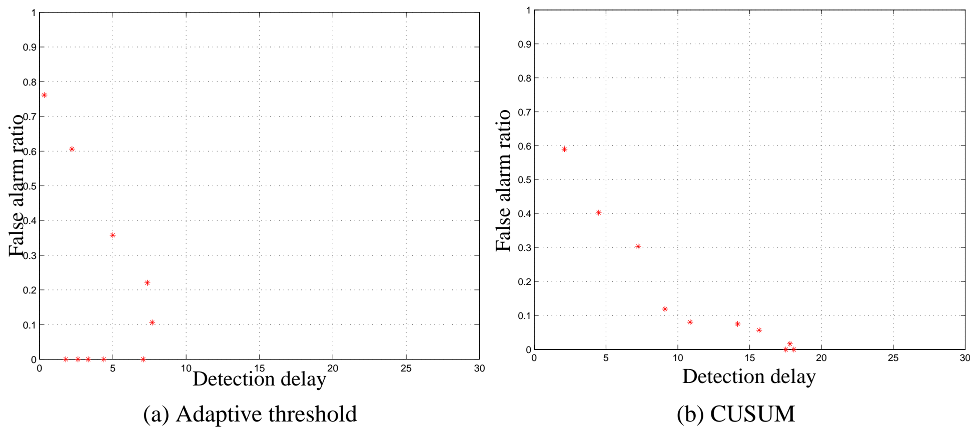
3.2.2.2 Tradeoff between false alarm ratio and detection delay

Next we investigate the tradeoff between the false alarm ratio and the detection delay. Figures 3.7(a) and 3.7(b) show the results in the case of low intensity attacks for the adaptive threshold and the CUSUM algorithm, respectively. Each point in the graph corresponds to a different value of the tuning parameter, k or h. An algorithm has better performance when the points corresponding to the detection probability and FAR pair are located towards the lower-left of the graph. Observe that there is a tradeoff between false alarm ratio and detection delay. Note that in Figure 3.7(a), which is for the adaptive threshold algorithm, the values on the lower-left correspond to low detection delay, but have a small detection probability.



(a) CUSUM (b) algorithm in [25]

Figure 3.6: False alarm ratio and detection probability for CUSUM algorithm and for the algorithm in [25].



(a) Adaptive threshold (b) CUSUM

Figure 3.7: False alarm ratio and detection delay tradeoff for the adaptive threshold and the CUSUM algorithms for low intensity attacks. Better performance corresponds to points towards the lower-left.

Chapter 4

Distributed Security Applications

4.1 Motivation

Network Intrusion Detection is receiving considerable acceptance as a mechanism to ensure the reliable and secure operation of computer networks. As a response to the growing number of cyberattacks, system operators are increasingly deploying intrusion detection sensors at diverse locations across their administrative domains, seeking a more complete coverage against potential threats. Such systems inspect passively the network traffic, trying to identify patterns of malicious activity (intrusion detection), or deviations from the normal network traffic characteristics (anomaly detection).

As described in Chapter 2, intrusion detection systems are based on a predefined set of rules, also known as signatures. Each signature is crafted to match a specific pattern of a known, malicious event. Upon the detection of suspicious activity, the IDS issues an alert to warn the system operator about the emerging threat. Since the signature that was triggered is known and the malicious packet(s) that triggered this particular signature have been logged, the operator has accurate information regarding the source and the type of the attack. The composition and selection of the signatures, which constitute the heart of an IDS, is a difficult task which must be accomplished carefully. Complex signatures can precisely describe advanced attacks, but can easily miss even their slight variations. Simplistic signatures tend to trigger too often and result to declare benign traffic as malicious. Besides the inherent inability of NIDSes to detect novel attacks, the major drawbacks of current NIDSes are the high rate and verbosity of the alerts, and the increased number of false positives, i.e. events that look like attacks, but in reality are completely legitimate traffic.

Anomaly detection systems, on the other hand, are based on heuristics that look for changes in the traffic patterns, such as a sudden increase in the number of TCP SYN packets per second, as described in Chapter 3. After establishing statistical models of the network traffic based on the normal system operation, any traffic that deviates more than a threshold level is considered anomalous and triggers an

alert. The major advantage of anomaly detection systems is their ability to recognize novel attacks for which no signature exists, or that cannot be described by signatures at all, such as port scans and worm outbreaks. However, although abrupt changes in traffic patterns may indicate attacks, they may also be due to legitimate reasons, including interesting breaking news, popular software updates, and flash crowds. Using only information available to anomaly detection sensors, it may be difficult to distinguish an attack from a legitimate traffic increase. Therefore, anomaly detection systems may incur a higher false positives rate than signature-based intrusion detection systems, which makes them rather ineffective as stand-alone security mechanisms.

Either way, the increasing use of both intrusion and anomaly detection systems produces vast amounts of logs and alerts that are difficult, if not impossible, to manage. Indeed, it is hard to separate the false alarms from the real attacks among the hundreds of reported alerts, and to bring out the truly critical threats. In addition, it seems very difficult to identify among the alerts sophisticated new types of attacks which involve many different hosts, such as Distributed Denial of Service (DDos) attacks, polymorphic worms, and distributed port scans. A security system based only on the sensors deployed at the network of a single autonomous system, has still a restricted point of view which limits its ability to detect such coordinated attacks.

The shortcomings of existing network security applications give rise for more extended and scalable solutions through the use of *distributed* intrusion detection systems. This promising approach involves the *cooperation* of many, possibly heterogeneous, monitoring sensors, distributed over a large network or several collaborating autonomous systems. In this environment, data gathered at each sensor are correlated and processed either in a distributed manner or after being collected at a central server. The analysis of the information that is provided by the several deployed intrusion and anomaly detection sensors, gives a broader perspective in which related incidents become apparent. The major benefits of distributed security applications include:

- **Reduced false alarms.** A distributed infrastructure can cover a larger and more diverse monitoring space than traditional single-node security applications. Events seen by each sensor can be correlated with the information collected from the rest of the sensors in order to increase the confidence of the reported incidents [24]. Studies, such as [2] have shown that cooperative detection is more effective in suppressing "false alarms" that may be caused, for example, by mechanisms with attack-like characteristics like popular downloads, flash crowds and peer-to-peer applications. For example, a sudden increase of identical packets at different sensors may imply the outbreak of a new worm. The same effect seen by a single node could be due to legitimate requests for a popular page that just went on-line. Thus, distributed intrusion detection systems may have considerably reduced false alarm rate.

- **Enhanced detection capability.** With the deployment of numerous monitoring sensors, there is an increased possibility that distributed and coordinated attacks will cross several of the cooperating sensors which may be able to identify them. An individual sensor may not be able to observe such kinds of attacks, since it will get only a small portion of the malicious traffic, while a cooperative security system exploits the complementary coverage from several monitoring nodes. A recent study [27] suggests that a network of 40 distributed sensors is sufficient for detecting "top-offenders"; as the number of sensors increases, the system can detect attacks at an earlier stage and can react more effectively. The correlation of data from multiple sources enhances the detection capability of current anomaly detection systems, enabling them to perform more accurate detection of sophisticated new types of attacks which intrusion detection systems cannot cope with, such as polymorphic worms or slow port scanners.

- **Reduced number of alerts.** Current intrusion and anomaly detection systems tend to be verbose and produce vast amounts of alerts which are difficult to manage and interpret. In a distributed intrusion detection system, alerts from various sensors are aggregated and may be prioritized according to the number of different sensors that have reported a particular alert. Hence, they provide condensed alerts and logs which enables the operator to make informed decisions about the emerging threats.

- **Resilience and Scalability.** A distributed infrastructure makes it harder for attackers to evade detection by spotting and blacklisting specific monitoring hosts or portions of the address space that are being monitored for random attacks. Additionally, such a system comprises numerous hosts, and thus, withstands denial of service attacks targeted to harm its availability. At the same time, a distributed infrastructure is easy to expand and reduces the cost per-participant, considering the related economies of scale.

4.2 Recent Efforts in Cooperative Security Analysis.

4.2.1 Distribution and Collection of IDS Data.

Having realized the importance of information sharing, several Computer Security Incident Response Teams (CSIRTs) have developed ECSIRT, the European CSIRT network that aims to facilitate the exchange of security incident-related data[1] and to generate warnings and emergency alerts based on these data. The project has successfully created a language for incident data exchange, and has established the collection and analysis of incident statistics. ECSIRT has even been publishing statistics about daily and weekly attacks seen by each security sensor.

[1] http://www.ecsirt.net/

The DShield project[2], supported by the SANS Institute, provides an attack correlation engine based on data about malicious activity from all over the Internet. Currently the system is tailored to data from simple packet filter logs from firewall systems. The SANS Institute also supports the Internet Storm Center[3] which is a collection site for intrusion detection data that is being collected from numerous volunteer sites.

The Intrusion Detection Working Group (IDWG) of the Internet Engineering Task Force (IETF) is developing IDMEF, the Intrusion Detection Message Exchange Format[4], which defines a standard format based on XML for information sharing among intrusion detection and response systems. There is already an IDMEF XML plugin for Snort IDS which outputs alert events in the form of IDMEF messages[5].

4.2.2 Cooperative Intrusion Detection Systems

Having recognized the benefit of correlating data gathered from different security-related sensors, several researchers have started investigating distributed security systems. COVERAGE is a cooperative virus and worm response mechanism that facilitates the defense against these types of attacks [2]. Symantec has developed DeepSight, a system that collects and correlates alerts from firewalls and intrusion detection systems in order to provide early warning that gives notifications of global attacks and best practice countermeasures [23]. EMERALD[6] is a distributed scalable intrusion detection tool suite for tracking malicious activity across large networks, which utilizes distributed high-volume event correlation methodologies. Indra is a distributed security system that runs on top of a peer-to-peer network of Intrusion Detection Systems [12]. Indra correlates information gathered by individual IDSs in order to detect new types of attacks.

4.3 Extending Current Security Applications

4.3.1 Data Sharing and Privacy

One of the core operations of a distributed security infrastructure is the dissemination and exchange of the information gathered by each sensor. The correlation and processing of this data can reveal potential threats and sophisticated attacks. A simple case of information sharing which enhances the detection accuracy, for example, is the exchange of "blacklists", namely lists of observed IP addresses which are potential sources of malicious activity. By the correlation of the malicious addresses contributed by each sensor, the likelihood that they are indeed

[2] http://www.dshield.org/
[3] http://www.incidents.org/
[4] http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-11.txt
[5] http://www.silicondefense.com/idwg/snort-idmef/
[6] http://www.sdl.sri.com/projects/emerald/

sources of malicious activity can be increased. Sometimes it is also necessary to disseminate more detailed information about the captured packets, such as several header fields, or even full packet payloads. For example, the payload of a suspicious packet which may carry malicious content, may have to be delivered to other sensors or the central server, for further processing and correlation with payloads from other sensors. However, data exhange in a distributed security environment brings on some drawbacks:

- **Data volume:** Although simple IP address lists seem to be rather easy and affordable to transfer using simple compression, they may require a considerable amount of traffic, given the substantial link speeds and considering the cumulative number of alerts from the numerous deployed sensors. On the other hand, the more detailed the exchanged data, the more efficient and effective their correlation can be. Under these conditions, the exchange of whole packets seems to be rather impracticable since it demands significant amounts of bandwidth. However, the identification of sophisticated new types of attacks still requires the exchange of information based on the header and payload data of the captured packets.

- **Privacy:** The dissemination of packet header or payload data is often contrary to the privacy policies of many organizations. Even simple "blacklists" of suspicious IP addresses may reveal legitimate customers or partners who usually communicate with the organization, in case of false positives. The concerns about sharing full packet payloads are presumably even more sound. Moreover, the manipulation of such sensitive data across the sensors or the central server might lure attackers to attack them and gain access on the gathered data.

In order to cope with the aforementioned problems, the relevant information must be encoded in a compact manner, which yet allows for further correlation and processing for the identification of potential threats. Such an encoding would reduce the amount of data transfered between the sensors, while preserving the privacy and anonymity of each participant, and enabling the sensors deployed at diverse locations to securely share attack information. These requirements can be fulfilled through the use of content *digests*. A digest, sometimes also called a "signature", can be thought of as a "summary" of a packet. Instead of transferring the actual packet contents, we can transfer digests for all the necessary parts of each packet. Digests are computed using appropriate hash functions which take as input the corresponding parts of the packet. The one-way encoded data have reduced size compared with the original data and preserve the confidentiality of their source, while can still be used for correlation with other data encoded in the same fashion. Thus, the traffic required for the exchanged of the gathered data is reduced considerably. For example, instead of sharing human-readable IP addresses, we can compute a digest for each 4-byte IP address which can then be used for correlation

sensor 1 packet	quantity	sensor 2 packet	quantity	sensor 3 packet	quantity	sensor 4 packet	quantity
A	100	E	90	H	100	K	110
B	80	F	85	I	90	D	90
C	70	D	70	J	70	L	80
D	50	G	60	D	60	M	70

Figure 4.1: Example of the most popular packets seen from different sensors. Packet D is among the popular packets of each sensor, thus it may belong to a self-replicating virus.

packet payload → hash function → packet count: 1, 3, 0, 5, 8, 0, 0

Figure 4.2: Data structure to find the most popular packets with the same content. Each field holds the count of the packets with the corresponding payload digest.

with other IP data. A similar approach is used by the Worminator project[7] which supports the the correlation of potential malicious IP addresses via the replication and use of Bloom filters [3]. Digests should be computed over the invariant parts of the packet, excluding fields like TTL, TOS, and checksums, in order to avoid otherwise similar packets to produce different digests [20].

This technique can form the basis for a more advanced data structure which holds all the necessary information for the correlation and processing of complex requirements. Additional information can be stored in a hash table which is indexed by an appropriate digest. Using separate digests for each required part to index several hash tables, we can build a compound data structure which holds even more information, but also consumes limited space and can be easily disseminated.

[7]http://worminator.cs.columbia.edu/

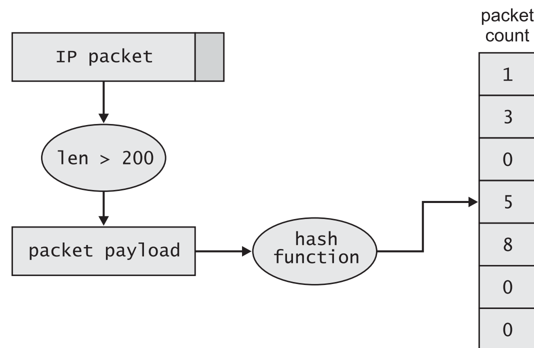


Figure 4.3: Data structure to find the most popular packets with size greater than 200 bytes. The structure of Figure 4.2 is enhanced with a BPF-like rule.

A usual way for the identification of self-replicating worm outbreaks is through the finding of the most popular packets seen among the deployed sensors. Indeed, if several sensors observe a sudden increase of similar packets, then, this may be an indication of the spread of a new self-replicating worm. This is illustrated in Figure 4.1, which shows the most popular packets seen by four different sensors. Besides packet D, all the remaining packets are different. However, packet D is among the popular packets of *all* the sensors, thus it may belong to a self-replicating virus.

In order to find out the globally most popular packets seen among the deployed sensors, each sensor must contribute the full payload of each popular packet it has seen so far, i.e., packets from different sources and/or destinations which have the same payload. Figure 4.2 shows an appropriate structure for the identification of *the most popular packets* at each host. A hash function which takes as input the payload of every packet that passes through the sensor, produces a digest which corresponds to the packet payload. This digest is used to index a hash table which holds the number of packets with the same payload so far. To identify the ten most popular packets, by the end of the measured interval, the fields with the ten highest values will represent the ten most popular packets.

The globally most popular packets may include very small packets commonly seen in the substantial network traffic. Packets like TCP acknowledgements, which have no payload, or telnet packets, each carrying a single character, add noise to the observed trafffic and harm the correlation process. On the other hand, even the smallest worms, which are usually contained in a single packet, carry a considerable amount of code necessary for their spread. For example, the Sapphire worm, one of the smallest self-replicating worms so far, comprises simple compact code of 376 bytes, which with the requisite headers forms a single 404-byte UDP packet [17]. Given that Sapphire is a rather simple worm, we do not expect future

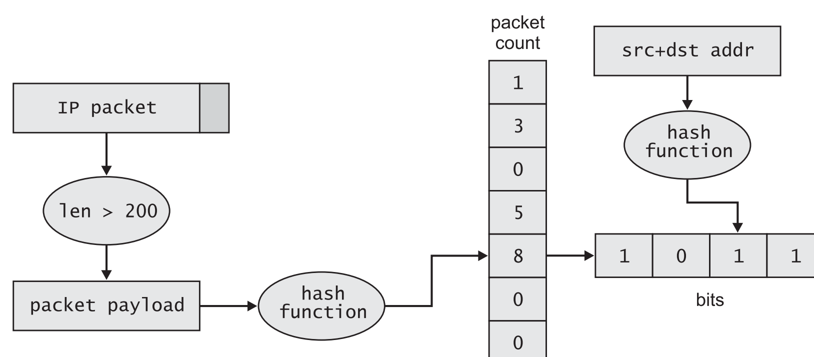


Figure 4.4: Data structure to find the most popular packets with size greater than 200 bytes and with ten different source and/or destination IP address pairs. The structure of Figure 4.3 is enhanced with an additional hash table for each field, which holds the occurences of different source-destination pairs.

sophisticated worms of smaller size[8].

Consequently, it would be useful to keep away from the correlation process very small packets, which are considered harmful for the correlation process, and it is rather impossible to carry any malicious payload. Thus, the rather simple monitoring requirement of the most popular packets can be refined to express *the most popular packets with size greater than 200 bytes*. The structure of Figure 4.2 can be extended with a BPF-like rule, which expresses the additional requirement of keeping away small packets. The resulting structure is shown in Figure 4.3.

A potential source of false results in worm detection is traffic with attack-like characteristics, like flash crowds, due to breaking news, critical software pacthes, or popular downloads. Such traffic usually comes from a single or a few web servers, and spreads across numerous end hosts. In order to filter out legitimate traffic, we may refine the query to denote the *the most popular packets with size greater than 200 bytes, and with ten different source and/or destination IP address pairs*. Indeed, this requirement is appropriate for the monitoring experiment presented in Section 4.3.2.

In order to find the most popular packets with at least ten different IP address combinations, each field of the hash table for the packet counts, points to an additional hash table, as shown in Figure 4.4. This time, the hash table is implemented as a bit vector which holds the occurences of the different address combinations. Each bit vector is indexed by a second digest which is computed by both the source and destination IP address of the packet. Note that packets with the same payload digest, may have several different header fields, including different IP addresses.

[8]Note that worms may split their code into many small packets, in order to evade detection. This evasion can be easily avoided with the appropriate packet defragmentation and reassembly according to the protocol used.

```
(A)
source port: *, destination port: 135
payload:
 05 00 0B 03 10 00 00 00 48 00 00 00 7F 00 00 00 ........H.......
 D0 16 D0 16 00 00 00 00 01 00 00 00 01 00 01 00 ................
 A0 01 00 00 00 00 00 00 C0 00 00 00 00 00 00 46 ...............F
 00 00 00 00 04 5D 88 8A EB 1C C9 11 9F E8 08 00 .....]..........
 2B 10 48 60 02 00 00 00                         +.H`....

(B)
source port: *, destination port: 135
payload:
 05 00 00 03 10 00 00 00 A8 06 00 00 E5 00 00 00 ................
 90 06 00 00 01 00 04 00 05 00 06 00 01 00 00 00 ................
 00 00 00 00 32 24 58 FD CC 45 64 49 B0 70 DD AE ....2$X..EdI.p..
 74 2C 96 D2 60 5E 0D 00 01 00 00 00 00 00 00 00 t,..`^..........
 70 5E 0D 00 02 00 00 00 7C 5E 0D 00 00 00 00 00 p^......|^......
 ....
```

Figure 4.5: Most popular packets captured in the network of ICS-FORTH. From all the packets captured in a subset of the internal network of ICS-FORTH, we printed the most popular ones that had at least 10 different source-destination pairs and the same destination port. The packets belonged to the Welchia worm.

Thus, each new value at a particular bit vector will indicate another packet with the same payload, but with a different address combination. A bit vector with ten bits set shall indicate that the corresponding payload has been found in packets from ten different address combinations.

Using the appropriate combination of hash tables, such a scheme can facilitate the sharing of detailed information within a compact structure. At the same time, the one-way computation of the digests satisfies privacy needs. However, in order to minimize potential noise due to hash table collisions and ensure the resiliency of the data, the respective thresholds and sizes must be chosen carefully.

4.3.2 Data Correlation

One of the major benefits of distributed security systems is their enhanced detection capability which enables the identification of novel attacks. Data collected by different sensors are correlated using heuristic-based techniques, in order to reveal possible spread of new attacks. Such heuristic techniques may capitalize on similarities in the fields of the packet header and the packet payload of attack packets. For example, if several different sensors start to receive lots of identical packets destined for the same destination port, then this is probably a strong indication that a new worm is spreading on the Internet. A similar event observed from a single sensor, or the sensors deployed within a single autonomous system, could be due to legitimate reasons, such as a flash crowd to an internal web server.

Our preliminary results with correlating header and payload information of network packets suggest that such heuristic techniques may be very effective in identifying attacks. For example, we monitored part of the internal network of ICS-FORTH and recorded the most popular network packets. Actually, we recorded those network packets that had identical payloads, identical destination port, originated from several source IP addresses and destined to several destination IP addresses. Figure 4.5 shows the two most popular packets found. We see that both packets were heading to destination port 135 and carry code that exploits the DCOM vulnerability [9]. This is the same vulnerability exploited by Blaster and Welchia worms. Indeed, the packets belonged to the Welchia worm.

It is interesting to note that out of millions of network packets monitored, the two packets that met our heuristic criteria were both worm packets. We believe that these attack detection methods are already effective and they will be even more powerful within a distributed security infrastructure. The correlation of the information found in different sensors can imporve the detection capability and the accuracy of the results. Furthermore, the identification of the malicious packets required inspection of the packet payloads, which denotes the need for a robust and secure solution for the dissemination of such relevant information in a distributed environment, as presented in Section 4.3.1.

[9] http://www.counterpane.com/alert-v20030801-001.html and http://www.linklogger.com/msblast.htm

Bibliography

[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] K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the 11th IEEE Internation Conference on Networking (ICON)*, September/October 2003.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

[5] 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.

[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] Scott Crosby. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[8] 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.

[9] 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.

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

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

[12] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *Proceedings of IEEE WETICE 2003*, June 2003.

[13] 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.

[14] 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.

[15] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to ftp.ee.lbl.gov.

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

[17] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, 2003.

[18] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial of service activity. In *Proc. of USENIX Security Symposium, 2001.*

[19] Martin 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/*).

[20] Alex C. Snoeren. Hash-based ip traceback. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14. ACM Press, 2001.

[21] Sourcefire. *Snort 2.0 - Detection Revisited.* http://www.snort.org/docs/Snort_20_v4.pdf, October 2002.

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

[23] Symatec. Symantec deepsight threat management system. Technology brief, http://www.symantec.com/.

[24] Alfonso Valdes and Keith Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pages 54–68. Springer-Verlag, 2001.

[25] H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *Proc. of IEEE INFOCOM'02, 2002*.

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

[27] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of NDSS 2004*, 2004.