Generating Realistic Workloads for Network Intrusion Detection Systems *

Spyros Antonatos
Institute of Computer Science
Foundation for Research & Technology – Hellas
antonat@ics.forth.gr

Kostas G. Anagnostakis
Distributed Systems Lab
CIS Department, Univ. of Pennsylvania
anagnost@dsl.cis.upenn.edu

Evangelos P. Markatos
Institute of Computer Science
Foundation for Research & Technology – Hellas
markatos@ics.forth.gr

Appears in Proceedings of the Fourth International Workshop on Software and Performance (WOSP),2004

ABSTRACT

While the use of network intrusion detection systems (nIDS) is becoming pervasive, evaluating nIDS performance has been found to be challenging. The goal of this study is to determine how to generate realistic workloads for nIDS performance evaluation. We develop a workload model that appears to provide reasonably accurate estimates compared to real workloads. The model attempts to emulate a traffic mix of different applications, reflecting characteristics of each application and the way these interact with the system. We have implemented this model as part of a traffic generator that can be extended and tuned to reflect the needs of different scenarios. We also present an approach to measuring the capacity of a nIDS that does not require the setup of a full network testbed.

Keywords

security, intrusion detection, workload characterization and generation

1. INTRODUCTION

Intrusion detection is receiving considerable attention as a mechanism for keeping administrators informed on potential security breaches and suspicious network activity. The typical function of a Network Intrusion Detection System (nIDS) is based on a set of *signatures*, each describing one known intrusion threat. A nIDS examines network traffic and determines whether any signatures indicating intrusion attempts are matched. The simplest and most common form of nIDS inspection is to match string patterns against the payload of packets captured on a network link. This form of detection is often called *content matching*.

This paper considers the problem of determining efficient and accurate methods for evaluating the performance of a content-matching nIDS. In such systems, the primary measure of interest is the *capacity* of the system. Accurately measuring the capacity of a nIDS is worth examining because of two main reasons. First, administrators need to determine if a given configuration (*e.g.,* system hardware, nIDS software, ruleset, and traffic characteristics) is suitable for deployment in terms of being able to cope with the workload without the risk of missing intrusion attempts. Recent work[5, 27] has shown that performance is highly dependent on these parameters, and thus each configuration needs to be evaluated on a case-by-case basis. Second, nIDS technology continues to evolve, and there is significant ongoing effort to improve efficiency[8, 10, 4, 28]. Such engineering efforts require accurate tools for analyzing existing systems, evaluating new ideas, and comparing different approaches. Previous work has shown that nIDS evaluation is not as easy as measuring the capacity of other networking systems (such as routers, or Web-servers) and simplistic benchmarks can easily become misleading[5, 19, 28].

The approach proposed in this paper is based on a modular traffic generator that can be tuned to closely resemble different types of nIDS workloads. The modules create application-specific traffic for protocols such as DNS, SMTP and HTTP, and can be configured for different levels of detail in the workload model. At the finest level, modules can emulate different sessions between sets of clients and servers following the characteristics of each protocol. At a coarser level, a module can synthesize traffic based on statistical properties of each protocol including the fraction of traffic per protocol, packet size and payload characteristics. If necessary and appropriate, these models can also be combined with header-only packet traces, including widely available datasets available for Internet research [24, 13]. Finally, there are modules for emulating different types of attacks to the nIDS itself such as those that attempt to overload the nIDS in order to evade detection.

*This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union and in part by the EAR GSRT Project (US-022) funded by the Greek General Secretariat for Research and Technology. The second author is also supported in part by ONR under Grant N00014-01-1-0795. S. Antonatos and E. P. Markatos are also with the University of Crete. The work of K. Anagnostakis was done while at ICS-FORTH.

The proposed approach offers four main advantages. First, it can be easy to use as it does not necessarily require the use of a live testbed or real traffic. Second, it provides more accurate results when compared to existing methods. Third, it is extensible, allowing the introduction of new protocols and models. Finally, it is tunable so that users can generate different types of workloads for different environments.

1.1 Paper organization

The remainder of this paper is organized as follows. In Section 2 we provide a brief overview of how intrusion detection systems work, with particular emphasis on the pattern matching algorithms used. We also discuss some of the methods employed in nIDS performance evaluation and discuss their problems. In Section 3 we present an analysis of nIDS workloads and their impact on performance and determine the accuracy of different workload models. Based on this analysis, in Section 4.1 we present the architecture and implementation of a nIDS traffic generator, and provide details on the different modules implemented so far. In Section 5 we summarize the main results and present our conclusions.

2. BACKGROUND

In this Section we describe a - rather simplified model - of how a content matching nIDS operates and summarize the key characteristics of pattern matching algorithms that have been recently used in intrusion detection. We also discuss previous work in nIDS evaluation to place our work in context.

2.1 Basic nIDS model

A nIDS is designed as a passive monitoring system that reads packets from a network interface through standard system facilities such as `libpcap`[18]. After a set of normalization passes (such as IP fragment reassembly, TCP stream reconstruction, etc.) each packet is checked against the nIDS ruleset. The ruleset is typically organized as a two-dimensional data-structure chain, where each element - often called a *chain header* - tests the input packet against a packet header rule. When a packet header rule is matched, the chain header points to a set of signature tests, including payload signatures that trigger the execution of the pattern matching algorithm. pattern matching is the single most expensive operation of a nIDS in terms of processing cost. In order to understand interaction between pattern matching algorithm, ruleset and experimental workload paper, we briefly present, further on this paper, some of the pattern matching algorithms that are commonly used in intrusion detection systems.

2.2 Pattern matching algorithms

A number of algorithms has been proposed for pattern matching in a nIDS. Performance of each algorithm may vary according to the case in which it is applied. The multi-pattern approach of Boyer-Moore is fast for a few rules but does not perform well when used for a large set. On the contrary, Wu-Manber behaves well on large sets but its performance is degraded when short patterns appear in rules. E^2xB is based on the idea that in most cases we have a mismatch and tries to filter out patterns that do not match. However, E^2xB introduces additional preprocessing cost per packet, which is amortized only after a certain number of rules. In the following subsections a more detailed description for each algorithm is provided.

2.2.1 The Boyer-Moore algorithm

The most well-known algorithm for matching a single pattern against an input was proposed by Boyer and Moore[7]. The Boyer-Moore algorithm compares the search pattern with the input starting from the rightmost character of the search pattern. This allows the use of two heuristics that may reduce the number of comparisons needed for pattern 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 pattern, the search pattern is shifted so that the mismatching character is aligned with the rightmost position at which the mismatching character appears in the search pattern. If the mismatching character does not appear in the search pattern, the search pattern 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 pattern, then there is a non-empty suffix that matches. The heuristic then shifts the search pattern up to the next occurrence of the suffix in the pattern. Horspool [12] improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic. Fisk and Varghese[10] recently developed Set-Wise Boyer-Moore (SWBM), an algorithm based on Boyer-Moore concepts and operating on a set of patterns. SWBM was integrated in `snort` and tested using a single traffic trace from an enterprise Internet connection.

2.2.2 The E^2xB algorithm

E^2xB is a pattern matching algorithm designed for providing quick negatives when the search pattern does not exist in the packet payload, assuming a relatively small input size (in the order of packet size)[17, 4]. As mismatches are by far more common than matches, pattern matching can be enhanced by first testing the input (e.g., the payload of each packet) for *missing* fixed-size sub-strings of the original signature pattern, called *elements*. The false positives induced by E^2xB , e.g., cases with all fixed-size sub-strings of the signature pattern showing up in arbitrary positions within the input, can then be separated from actual matches using standard pattern matching algorithms, such as the Boyer-Moore algorithm [7]. The small input assumption ensures that the rate of false positives is reasonably small – our experiments demonstrate false positive rates of 10% in the worst case. In the common case, negative responses can be obtained without resorting to general-purpose pattern matching algorithms. The E^2xB algorithm was evaluated on traffic traces from diverse environments, like traces containing attacks, others with normal web traffic and WAN traffic traces from a local ISP.

2.2.3 The Wu-Manber algorithm

The most recent implementation of `snort` uses a simplified variant of the Wu-Manber multi-pattern matching algorithm [32], as discussed in [29]. 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 performance of MWM was originally measured using text files and various sets of patterns. The first attempt to measure MWM as the basic algorithm for pattern matching in a nIDS was performed in [29]. The results of [29] show that `snort` is much faster than previous versions that used Set-Wise Boyer-Moore and Aho-Corasick, although a large part of this improvement is due to a substantially re-engineered nIDS engine.

2.3 Previous work in nIDS evaluation

One of the first and most well-known efforts at developing a methodology for nIDS evaluation is presented in [14]. The study combined a synthesized stream that includes both normal traffic and a number of attacks. The principal goal of this work was to measure the ability of systems to identify new attacks without any knowledge of prior attacks, and estimate detection and false alarm rates. The study also determines the receiver operating characteristic (ROC) curves, comparing the percentage of detected attacks against the false alarm rate. A similar approach is described in[25], with the development of a testbed simulating the behavior of a large network, tracing the traffic on the testbed, and using that as input to the nIDS for evaulation.

The NSS Group[30] evaluated 15 commercial nIDS and open-source Snort. The testbed used was a 100 Mbps network with no real traffic. The primary metrics of interest were attack detection rate and correct labeling of attacks. The attacks were 66 commonly available exploits like portscans, web, FTP and finger attacks and were generated with specialized tools like `snot`[2] and `stick`[1]. Besides attacks, background traffic was also generated in order to test nIDS under different network loads. Background traffic was consisted of small (64 byte) and large (1514 byte) packets that consumed variable percentage of the network bandwidth (between zero and 100%). Other approaches like [22] tested the nIDS by injecting attacks into a stream of real background traffic and measuring the fraction of attacks that could be detected by the nIDS.

Mell *et al.* [20] studied past evaluation efforts and listed a number of problems related to nIDS evaluation. The use of sanitized traffic, the effect of background traffic and the difficulties in generating traffic on a testbed network are some of the problems spotted. Furthermore, they presented a set of recommendations for nIDS testing. These recommendations included shared datasets between multiple organizations, like widely available traffic traces, the use of traces containing attacks instead of real attacks and finally real data cleansing, that is the removal of confidential data from traffic logs in order to overcome privacy issues. Athanasiades *et al.* in [6] also provided a study of past evaluation efforts and proposed an environment suitable for nIDS evaluation. This environment uses synthetic background traffic and controlled injection of attacks in order to emulate a real network. Furthermore, it is equipped with the ability to respond to traffic in real time and generate traffic at gigabit speed so as to provide more realistic traffic scenarios.

Hall *et al.* [11] spotted a number of limitations applying to nIDS. Fixed resources, such as memory size and memory bandwidth, set an upper bound to performance of packet capture and packet flow architecture. Difficulties are also presented in generation of real traffic at high peak rates. Finally, packet analysis and state tracking may infer performance penalties. They proposed a test suite for nIDS evaluation, each component of which is used to measure different portion of a nIDS. This test suite included testing the maximum bandwidth a nIDS can inspect without packet loss, testing the alarm capabilities, stressing the state tracking engine and finally a set of tests with configurable metrics, based mainly on HTTP traffic.

Schaelicke *et al.* [27] studied the performance bottlenecks on `snort` and the impact of different architectures and operating systems. In their testbed, they used the `ttcp` utility to generate traffic between a pair of hosts. Measurements were performed for four fixed payload sizes and for two types of rules: rules that check only headers and rules that perform pattern matching. In their experiments, the basic metric of interest was the number of rules that can be processed without packet loss. An extended set of six machines with different architecture was examined. They also studied the impact of different operating systems on nIDS operation, specifically Linux and OpenBSD, and the effect of multiprocessor systems to the performance of packet capturing.

A recent study[5] illustrates some of the difficulties in evaluating nIDS performance. First, it shows that nIDS performance is sensitive to packet and ruleset content. Adding random content to the widely-available traffic header traces is thus, at least on first sight, questionable as a method for nIDS evaluation. However, our analysis shows that the sensitivity exhibits certain patterns in over- or under-estimating performance, depending on the pattern matching algorithm and the characteristics of the traffic. Regarding ruleset content, using random rule patterns for determining nIDS performance and scalability also requires extreme care: our results suggest that a more accurate way of creating synthetic rulesets is to use permutations of existing patterns. As with packet content, the sensitivity follows some predictable pattern depending on algorithm and traffic.

Second, it demonstrates large differences in measured performance depending on traffic characteristics: the highest measured mean cost per-packet is up to four times as much as the lowest cost in the traces we examined, mostly due to differences in the distribution of packets to the different subsets of the nIDS ruleset.

Third, it is shown that the choice of processor architecture has a dramatic effect, both on overall system performance as well as the relative performance of different pattern matching algorithms. There are cases where one pattern matching algorithm is faster than another algorithm on one processor but slower on another processor. As no single algorithm performs best in all cases a hybrid pattern matching engine triggering different algorithms depending on ruleset and packet size appears to be the best approach, but the parameters may also vary depending on the processor architecture.

3. ANALYSIS

3.1 Metrics

There are two basic metrics in evaluating a nIDS: *attack detection rate*, and *false alarm rate*. The attack detection rate can be defined as the fraction of attacks successfully detected by the nIDS, while the false alarm rate is the fraction of alarms that did not indicate a real intrusion over the total number of alarms produced by the nIDS. For a nIDS that uses statistical anomaly detection methods, both metrics reflect the quality of the algorithm, although the attack detection rate is also affected by the ability of the system to process the input stream without sustaining packet loss. For content matching systems such as `snort`, the detection process is more exact than in statistical methods. Therefore, the attack detection rate is more closely related to the *capacity* of the system: if all packets are processed then all attacks specified in the ruleset will be detected. For content matching systems the false alarm rate is dependent on ruleset configuration and alarm filtering that is usually done outside the critical path. We therefore focus on capacity or *maximum loss-free rate* (MLFR) as the basic metric of interest in this study.

The capacity of a nIDS also relates to the *processing time* as reflected in the measured number of instructions and processor cycles devoted to the operation of the nIDS. Running time can provide a reasonable estimate of nIDS capacity in many cases, and can be easily measured without setting up a full testbed (*e.g.,* through trace-driven execution). However, burstiness in packet arrival rates and variable processing needs on the nIDS may cause transient overload and packet loss that may not be reflected in processing time. Capacity, in terms of MLFR, is more accurate, but normally requires a testbed which makes experimentation and measurement more time-consuming. We discuss a method for approximating the capacity of a nIDS without the complexities of setting up and using a testbed in Section 4.1.

3.2 Single-stream, trace-based and synthetic workload models

A straightforward approach for measuring nIDS performance would be to use a simple packet generator like `ttcp`[23] or `iperf`[3]. Such generators can create a stream of packets with certain characteristics (e.g., UDP or TCP, different packet sizes, etc.). This approach appears to be acceptable for evaluating systems such as routers, mainly because the processing involved in forwarding packets is very simple. A content matching nIDS is more complex: a modern nIDS need to check each packet against hundreds of rules, and almost all of them involve deep payload inspection. Figure 1 presents the fraction of nIDS processing time spent on pattern matching for three traces containing real payloads. We notice that pattern matching accounts for 40-70% of the total processing time, and 60-85% if we consider the number of processor instructions issued. Most of the rest of the processing time is spent on header checking except for packet decoding which is approximately 2-3%. Because of the complexity of pattern matching it is unlikely that a single-stream model would produce accurate performance estimates.

To determine the sensitivity of nIDS performance to traffic we analyzed the behavior of `snort` using network traces from

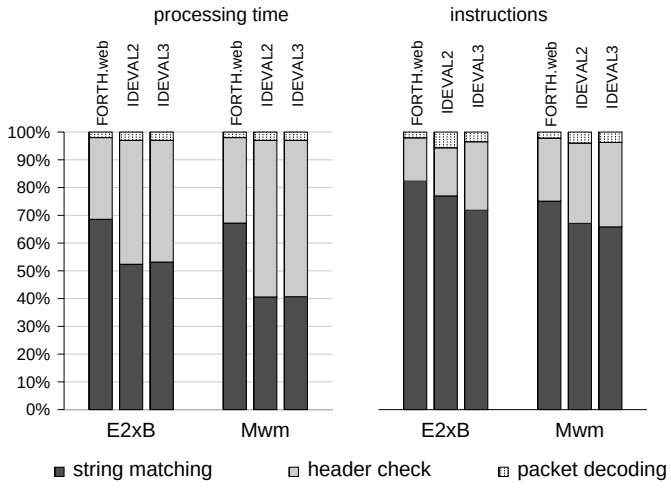

Figure 1: The cost of nIDS pattern matching in terms of running time and instructions

.

different environments. In Figure 3 we present the processing time per packet and per byte for different traces. Traces labeled `NLANR.IND`, `NLANR.MRA` and `NLANR.AIX` contain wide-area network traffic captured on a set of peering points. The set of `DEFCON` traces was collected on a LAN during a hack festival[31]. `FORTH.web` is a trace containing a fair number of concurrent Web browsing sessions captured at FORTH and `UCNET` are traces from the ATM link between the University of Crete and GRNET[9]. We observe that the highest measured per-packet or per-byte costs are 3 to 4 times more than the lowest measured cost. To investigate the causes for this sensitivity we instrumented `snort` to record the subset of the nIDS ruleset (*e.g.,* the "chain header" in `snort`) triggered for each packet in the trace. The results are shown in Figure 2. We observe that the use of different subsets of the nIDS ruleset varies significantly: roughly 42% of packets in the `FORTH.web` trace trigger a set of 956 rules while the same set is only triggered by 1-5% of the packets for all other traces. For the NLANR traces, a large fraction of packets (77-93%) trigger at most 87 rules. Although such differences should be expected given that the traces represent traffic in different settings (wide-area, hacker contest and a web-only environment) there are visible differences even between traces of the same type. Thus, simplified single-stream workloads or the use of a single traffic trace does not seem to be sufficient for evaluating "overall" nIDS performance.

Given that using a single traffic trace is not sufficient, a reasonable strategy would be to use a carefully selected set of traces as a benchmark. There are, however, three main problems with such an approach. First, there are very few full-packet traces available, mainly because of privacy issues. In most publicly available traces, packet payloads are removed and IP addresses are sanitized. Second, traces require significant storage requirements and cannot be easily distributed. For example, a 60 second trace of a fully-utilized gigabit link requires roughly 7.5 GB of storage. Furthermore, traces offer practically no flexibility in experimentation. Even a simple change in traffic characteristics, as for

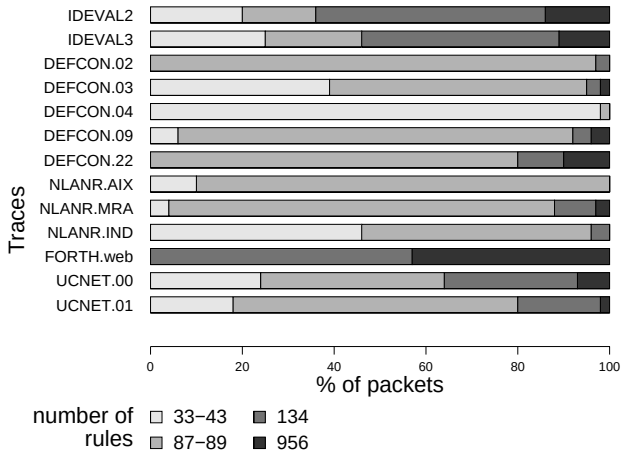

Figure 2: Distribution of the number of rules checked per packet

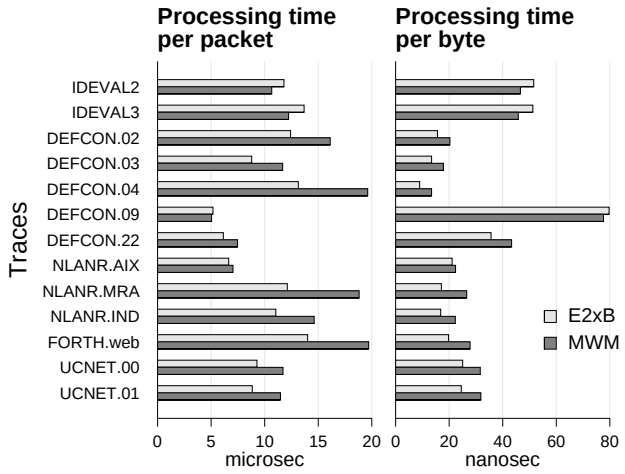

Figure 3: Processing time, per packet and per byte

example different payload sizes, would require the existence or creation of a different trace. Finally, trace replay is expensive and limits the maximum sending rate of the generator. Although lightweight tools like `tcpfire` [21] have been developed to deal with this, replaying a trace at very high rates may be constrained by storage and bus performance limits. This can be addressed using multiple senders but adds cost and complexity to the testbed and experiment setup.

To make nIDS evaluation easier, it would be desirable to design synthetic workloads based on statistical properties of network traffic. Synthetic workloads can be used to generate traffic similar to real traffic. Modeling packet headers can be achieved by using statistical information collected from available traces [16, 26] or coarse time-scale aggregate measurements[15]. However, the issue of payload modeling still remains open.

3.3 Modeling packet payloads

In our effort to find the most suitable approach to model payload generation we used only traffic traces that contained packets with real payload: `FORTH.web`, `IDEVAL2` and `IDEVAL3`. We evaluated three approaches in order to find a way for generating payload that gives results similar to the real ones. In each approach, packet headers were left unchanged and only payloads were altered.

A first approach for modeling payloads is the use of uniformly random payloads. Usually, a payload would contain exclusively either printable characters (like HTTP requests and FTP command) or binary content (images, compressed files, etc.). We evaluated the performance of `snort` , using either E^2xB or MWM as the main pattern matching algorithm. In our experiments we used two rulesets: 2.0.1 was the latest available ruleset at the time of the experiments and 2.0.0 was the ruleset coming with the latest official release of `snort` .

Figure 4 shows that uniformly random content introduces an error of 14% - 38% in the measured processing time of the nIDS, and the error depends on both the algorithm and the ruleset used. Considering the E^2xB algorithm, patterns consisting of sequences of the same character are the main reason behind this difference. One such pattern is "`0000`" (four adjacent zeros) which is part of 14 rules in ruleset version 2.0.1. While in real payloads the sequences "`00`" appears in about 10% of packets (because many protocols, such as GIF headers, use null data), in uniformly random payloads its occurrence is more rare (e.g., at a rate of 1 in 65536) When E^2xB sees the pair "`00`" inside the payload cannot decide for exclusion of pattern "`0000`" and fall-back pattern matching routine is invoked. In case of uniformly random payloads, the number of fall-back routine invocations is significantly smaller than the equivalent number for real payloads. Table 1 shows the times these repetitive patterns fall-back routine was invoked for actual searching. We notice that searches for `FORTH.web` containing uniformly random payloads are nearly the 40% of searches triggered real payloads, while in `IDEVAL` traces the reduction rate of searches reaches 94%.

	E^2xB		MWM	
	uniform	**real**	**uniform**	**real**
`FORTH.web`	4,574,786	7,712,983	315,763	12,001,484
`IDEVAL2`	964,033	16,074,186	140,658	6,730,802
`IDEVAL3`	1,066,654	19,004,703	363,014	7,163,140

Table 1: Number of searches for E^2xB and MWM

On the other hand, the difference in running time between uniformly random and real payloads for MWM is caused by the algorithm itself. MWM searches first for a two-byte prefix in the payload that is common prefix to patterns. For example, patterns "`http://`" and "`http://cgi-bin?..`" have the common prefix "`ht`". MWM searches for such common prefixes and whenever they are found the suffix is checked (the algorithm is constrained by the minimum pattern length for the determination of the suffix). If there is also a match in the suffix then the text between suffix and prefix is checked. In real payloads such common prefixes are often found as for example the prefix "`CO`". This prefix is located in several patterns like "`COMMAND=REGISTER`" or

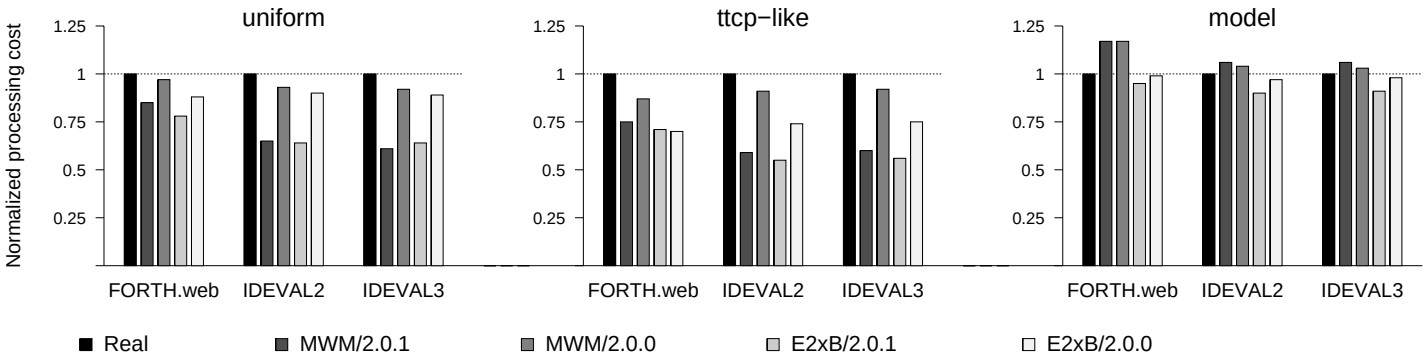

Figure 4: Normalized time for traces with uniformly random, ttcp-like, and modeled payloads

"CONF/HTTPD.CONF" and its occurrence in a payload would trigger many checks. However, in uniformly random payloads, its occurrence is less likely, thus leading to less checks and consequently diverging from the real case. Table 1 shows the number of patterns checked for each trace. When uniformly random payload is used, MWM performs approximately 96% less checks in comparison to the case of traces with real payload.

The difference in performance between real and uniformly random payloads also depends on the ruleset used. In snort 2.0.0 ruleset some rules of 2.0.1 that added overhead to the detection engine, e.g for E^2xB rules searching for pattern "0000", are absent. Figure 4 summarizes the results for ruleset 2.0.0. The difference between real and uniformly random payloads varies from 3% up to 26%, still remaining algorithm-dependent. Although in some cases error seems small, results are not consistent throughout different rulesets.

Many approaches use ttcp as a basic tool for generating traffic in order to evaluate nIDS systems like [27]. By default, ttcp injects repetitively into packets the pattern:

```
~!'()*+,-./0123456789@ABCDEFGHIJKLMNOP
QRSTUVWXYZ`abcdefghijklmnopqrstuvwxyz|
```

Testing whether such a payload type is appropriate for evaluating an nIDS, we replaced the payload of our traces with the one placed by ttcp. In Figure 4, we see that this type of payload declines from real trace at non-acceptable rates. For IDEVAL traces with ttcp-like payloads, difference in running time fluctuates between 10% and 40% while for FORTH.web ranges from 22% to 30%. As the pattern used by ttcp is consisted of a limited set of characters and its content is fixed, only a few rules are triggered and the detection engine is offloaded.

In our approach, we try to generate synthetic payloads similar to the real ones based on statistical data. The key term in our method is clustering. Term cluster defines an upper level protocol. Each packet in a trace belongs to a cluster. For example UDP packets arriving at port 53 belong to the DNS cluster. The clusters we used were HTTP packets containing text, HTTP packets containing images, HTTP packets with application data (like PDF, sounds and videos), HTTP requests, DNS queries, FTP data, SSH, Telnet, mail, netbios and other (traffic not belonging to previous clusters). The choice of clustering was made based by the popularity of the protocols. Figure 5 shows a cluster analysis of the two out of three traces we used in our experiments and justify our choice of clusters. The cluster analysis for IDEVAL3 is similar to the one of IDEVAL2. As FORTH.web is a trace containing web traffic, percentage of HTTP packets reaches 92%. We observe that most traffic originating from web servers belongs to application data cluster and is expected as size of multimedia elements (like a video) is larger than HTML pages. The remaining 8% is TCP control data (mostly ACKs). As IDEVAL traces contain WAN traffic, we can identify several clusters, like SSH, FTP, mail and telnet. A large portion of traffic (nearly 30%) belongs to the telnet cluster, while a significant portion of packets (20%) belongs to DNS cluster. Trace also contained packets belonging to HTTP application data cluster in a percentage less than 1% (was ommited in the plot to avoid congestion). Although incoming web traffic could be considered as one cluster, that approach would be misleading. Packets containing HTML pages have different impact on the nIDS detection engine from the packets containing a PDF file, in terms of rules being examined and the amount of work done by the pattern matching engine.

For each cluster we maintain a set of empirical distributions, one distribution for each position in the payload. Values for empirical distributions are obtained using information provided by payloads of traces. In these distributions we record for each one of the ASCII characters the probability of appearance in the corresponding position. In our approach we used FORTH.web to extract information for clusters concerning web traffic (HTTP text, images, application data and requests) and IDEVAL2 trace for the rest of the clusters. To illustrate how our model works we provide an example. Consider the cluster of HTTP requests. For this cluster we maintain 1500 distributions (as an Ethernet packet cannot exceed 1500 bytes), each one containing 256 values (one value for each ASCII character). As all HTTP requests begin with the string "GET /", the first distribution would contain a single value "G 100%", second one would contain "E 100%", the third one would contain "T 100%" and so on. The dis-

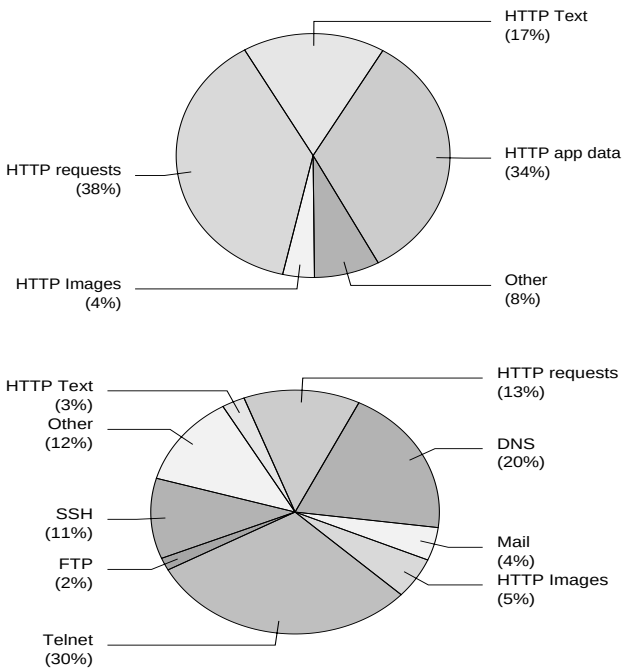

Figure 5: Protocol breakdown `FORTH.web` and `IDEVAL2` traces

.

tribution for sixth position, however, would contain several values. The name of the requested object begins in sixth position and usually has different name per request. For example, assume that we extract information from a trace containing only two HTTP requests, one for "index.html" and and one for "docs/manual.html". The distribution corresponding to the sixth position would contain two values: "i 50%" and "d 50%". In order to generate payload for an HTTP request, for each position we probabilistically choose one character from the corresponding distribution. For first position we would choose "G", for second "E" and so on. For sixth position we would randomly choose between "i" and "d". For each cluster we only need to maintain a set of distributions with size smaller than 500 KB.

The advantage of this approach is the construction of payloads that look alike the real one. Although it is far from full protocol emulation, it preserves some protocol characteristics. While in other approaches like uniformly random payloads, the structure of a packet was lost as content was replaced by random characters, in our approach structure is partly reproduced. An example stated above is the HTTP requests. Our approach will place the string "`GET /`" in the beginning of each HTTP request. Another example is the GIF header format that puts null data in its first bytes. Furthermore, as common pairs of characters are generated (for example "`GE`") the number of rules triggered reaches the one of real case. Results for the accuracy of our method are presented in Figure 4. The difference between real and modeled payloads are smaller than the other cases and remains consistent throughout rulesets. Consistency is important as the evolution of rulesets is unpredictable. While in case of

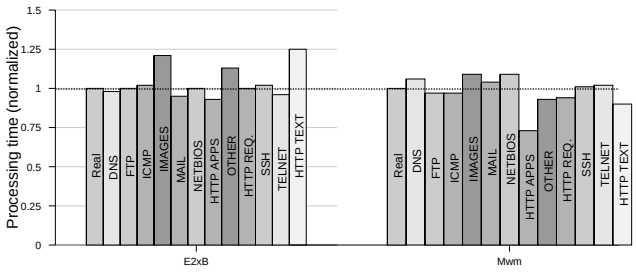

Figure 6: Error breakdown of payload model

uniformly random payloads we notice an increase of 30% for `FORTH.web` from ruleset 2.0.0 to 2.0.1 in our approach the increase was below 1%. For `IDEVAL2` and `IDEVAL3` difference between rulesets never exceeded 7% in our approach while in case of uniformly random and `ttcp`-like approaches difference varied between 6 and 32%.

Trying to locate the main reason behind differences, we measured the difference between real and synthetic payload in `IDEVAL2` trace for each cluster separately. The results are presented in Figure 6. The HTTP text cluster seems to present the largest difference at both E^2xB and MWM. This difference is expectable as the 80% of rules triggered by an HTTP packet search for a pattern consisted of exclusively printable characters. As our model implants in the payload common characters, common pairs are also created. The appearance of these common pairs slightly overloads E^2xB as it cannot decide for an exclusion in most rules (mainly short patterns). Difference is also presented in cluster *other* as it has to deal with unclassified traffic with no common characteristics or traceable properties.

4. METHODOLOGY

4.1 Architecture

In our efforts to develop a system that integrates our experimental results, we implemented a modular and extensible traffic generator called `NextGen`. Figure 7 shows the architecture of the generator. The use of modules allows us to emulation a mix of different protocols as part of a synthetic workload. The level of detail in the workload can be configured: from single flow traffic to fully specified protocols. Each module is responsible for generating a specific protocol, as for example HTTP or DNS queries. A module consists of two parts: its initialization function, where all the initialization is performed and the traffic function where the actual traffic is generated and injected into the network. The traffic function may use other modules to shape a packet, such as payload generation modules to construct payloads or modules that reproduce certain types of attacks against the nIDS.

We have implemented four modules for the following types of traffic: HTTP, DNS queries, Telnet and random background traffic. The HTTP module reproduces web traffic, the DNS module emulates queries performed by the Domain Name Service, the Telnet module emulates the telnet protocol and the random module generates traffic with random source and destination ports and addresses. The choice

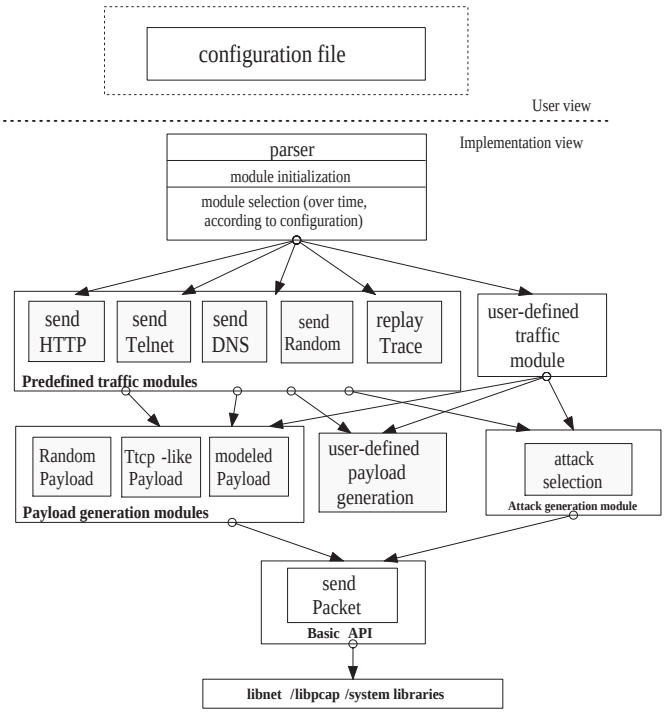

Figure 7: Generator architecture

of HTTP, DNS and Telnet protocol for building predefined modules was based on their popularity as seen in Figure 5. Except from normal traffic modules, we have developed a module for replaying traffic based on a trace. `NextGen` can be extended to reproduce more protocols as individual traffic modules can be developed and plugged in. For example, one can develop a module for sending SMTP traffic. The module itself can use its own set of options allowing customized configuration. The development of a module is simple and requires the implementation of the initialization and the traffic function. The registration of the module into the generator is committed in a way similar to preprocessors of `snort`.

4.2 nIDS workload modules

Based on the analysis performed in Section 3, emphasis has been given to the issue of payload generation. The generator comes with three predefined modules for generating payloads: uniformly random, `ttcp`-like and our model presented in Section 3. Modularity of `NextGen` modularity permits the user to write its own payload generation modules if judges that predefined modules are not accurate enough. For example, one can develop a module that generates protocol-specific payloads following a specific format and initialize modules to use that module. This characteristic can turn `NextGen` into a full protocol emulation tool. However, one must be careful in modules writing as they can become a performance bottleneck. A common technique is to prepare a set of payloads in the initialization phase and pick up a payload from the set during the traffic send.

Finally, each module has the ability to generate attacks. Attack generation is a very useful facility to nIDS evaluation as attacks cause overloading of the detection engine. Each traffic module, whenever wants to send an attack, provides the attack module its traffic characteristics. For example, the HTTP module wants to send an attack originating from a client and targeting a web server. It provides the attack module its characteristics, that is TCP protocol and destination port 80. The attack module tries to find an attack that matches these characteristics and if a match is found the attack is injected onto the network. If no match is found it searches for attacks on TCP protocol and any destination port (generic attacks). The existence of attacks that apply to any source and destination ports and addresses ensures that every module has at least one attack to generate. Attacks signatures are read from the `snort` ruleset.

4.3 Capacity estimation

Estimating the capacity of the nIDS in terms of the maximum loss-free rate (MLFR) is difficult, as it requires the setup of a network testbed with traffic generators capable of producing traffic fast enough to reach the nIDS limits. We present a shrink-wrapped capacity estimation method that can be performed locally on a single system.

The basic idea is to emulate the operation of the nIDS in virtual time, with a virtual network interface and packet queue. The system maintains a virtual clock which tracks packet arrival events (as produced by the model or trace-based workload), emulates enqueue/dequeue actions and measures the amount of time needed by the nIDS to process the packet. The measured processing time for each packet determines when processing of the next packet can begin, and therefore how many packets are waiting in the queue at each point in (virtual) time. When a packet is received, it is possible to determine whether the packet can be processed, enqueued or dropped. The measured loss rate for an experiment can then be used to tune the experiment until it reaches zero loss, which is the desired capacity estimate for the system.

This method is implemented as part of a modified `libpcap` library that communicates with the traffic generator through standard UNIX IPC mechanisms. To minimize interference between the generator and the nIDS the modified library uses a large amount of buffer space for the generator to write packets. Buffer management is properly memory-aligned to reflect the buffering strategies of modern network interfaces.

4.4 Example Applications

4.4.1 Measuring nIDS sensor capacity

We measured the capacity of a `snort`-based sensor under various workloads. Firstly, we generated a workload similar to `FORTH.web`. `snort` was running with all 1923 available rules activated. Capacity was calculated for both MWM and E^2xB on a gigabit testbed. Our measurements showed that for both E^2xB and MWM capacity reached approximately 210 Mbit/s. These values tend to agree with measured running time (was 36.47 for E^2xB and 37.12 for MWM, giving 206 and 205 Mbit/s respectively as workload generated one gigabyte of traffic). As workload did not contain bursty traffic (as originally happened in case of `FORTH.web`) the little difference between running time and capacity is expectable.

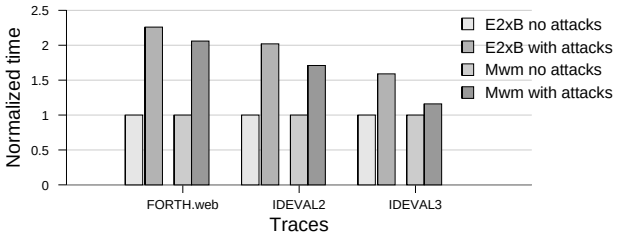

Figure 8: Normalized running time for traces with and without attacks

.

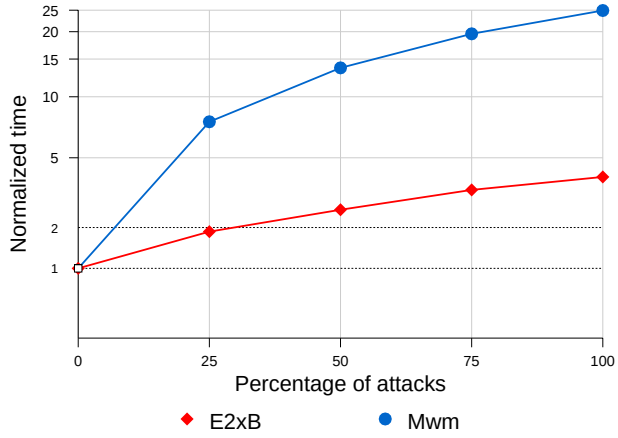

Figure 9: Normalized running time for traces with and without algorithmic attacks

.

4.4.2 *Measuring nIDS robustness against overload attacks*

To illustrate the value of this attack module we perform a simple experiment involving regular (attack-free) packet traces and feeding them through the attack module. We use a trace containing HTTP packets from a portal mirror (`FORTH.web`) and two traces with WAN traffic (`IDEVAL2` and `IDEVAL3`). Figure 8 shows the difference in the load of a nIDS between the original trace and the attack trace. We observe that this type of attack increases the load by more than 100% in the case of Web-dominated traffic (`FORTH.web`) and between 30% and 100% for the other two traces. Although an attacker would not be able to similarly modify regular traffic, this experiment illustrates how an attacker could overload a nIDS by exploiting the sensitivity of nIDS performance to packet content. Except from common attacks, that is attacks that match to a nIDS rule like the ones produced by snot, one can evade a nIDS by injecting attacks against nIDS pattern matching algorithm. Such attacks include normal packets with special payload that overload pattern matching engine. For example, MWM's implementation on `snort2` finds all the occurrences of a pattern in the payloads. If we build a packet containing only zeros in its payload then performance of `snort` is amazingly degraded. This happens because some rules search for adjacent zeros in the payload so MWM finds 1500 occurrences for each rule, assuming maximum payload size (approximately 1500 bytes for a typical Ethernet network). For E^2xB, an algorithmic attack can be a payload containing all unique pairs of characters met in the patterns we search. As the number of unique pairs exceeds the maximum payload size, we can only build a partial attack. The effect of algorithmic attacks on nIDS performance is presented in Figure 9. We instrumented the attack modules to replace the payloads of a trace containing normal web traffic with payloads that contained algorithmic attacks. We evaluated the performance for both MWM and E^2xB against these attacks by changing the percentage of packets containing them. The increase in processing time is linear to the percentage of attacks for both algorithms. We notice that for MWM, performance is degraded about 25 times while in case of E^2xB about 7 times.

5. SUMMARY

We have studied the problem of modeling nIDS workloads and developing a methodology for evaluating the performance of a content matching nIDS.

There are three main results from this study. First, we have shown that nIDS performance is highly sensitive to the underlying traffic and that simple stream-based analysis (*e.g.*, performed with tools like `ttcp`) as well as trace-based analysis can easily be misleading. Considering this observation, we argue that a nIDS workload model needs to accurately reproduce application-level characteristics of the input traffic.

Second, we have examined how to accurately reproduce the interaction between packet *payloads* and the nIDS content matching component. We have presented a payload model that is based on empirical distributions of single-byte patterns at a given distance from the beginning of the packet, with different sets of distributions for each protocol and protocol message type. We have compared model-based payloads to real traffic traces, showing that the model error is in most cases insignificant. This is essential, as it enables the creation of the necessary statistical workload models while also allowing the use of widely-available header-only traffic traces as part of a comprehensive, multi-trace nIDS benchmark.

Third, we have proposed a method for measuring nIDS capacity that accurately emulates the operation of the nIDS without requiring the use of a fully-fledged network testbed or high-performance traffic generators. The basic idea is to use a virtual clock to emulate packet arrivals and queuing behavior based on real measurements of the required processing time for each packet.

Our results have been implemented as part of a traffic generator tool that has been designed to be modular and extensible so users can adapt it to the needs of different environments. Although much work remains to be done, we believe that these results presented here offer important insights on the problem of nIDS performance evaluation.

6. AVAILABILITY

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

7. REFERENCES

[1] http://www.eurocompton.net/stick.

[2] http://www.stolenshoes.net/sniph/index.html.

[3] Nlanr/dast : Iperf - the tcp/udp bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/.

[4] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. E^2xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, May 2003.

[5] S. Antonatos, K. G. Anagnostakis, M. Polychronakis, and E. P. Markatos. Benchmarking and design of string matching intrusion detection systems. Technical Report 315, ICS-FORTH, December 2002.

[6] N. Athanasiades, R. Abler, J. Levine, H. Owen, and G. Riley. Intrusion detection testing and benchmarking methodologies. In *Proceedings of the IEEE Information Assurance Workshop*, pages 63–72, March 2003.

[7] R. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[8] 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.

[9] 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.

[10] 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.

[11] M. Hall and K. Wiley. Capacity verification for high speed network intrusion detection systems. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2002.

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

[13] Lawrence Berkeley National Laboratory. The internet traffic archive. *http://ita.ee.lbl.gov/*.

[14] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000.

[15] C. Liu, S. V. Wiel, and J. Yang. A nonstationary traffic train model for fine scale inference from coarse scale counts. *IEEE Journal on Selected Areas in Communications: Internet and WWW Measurement, Mapping and Modeling*, 21:895–907, August 2003.

[16] M. T. Lucas, D. E. Wrege, B. J. Dempsey, and A. C. Weaver. Statistical characterization of wide-area IP traffic. In *Proceedings of Sixth International Conference on Computer Communi cations and Networks (IC3N'97)*, 1997.

[17] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis. ExB: Exclusion-based signature matching for intrusion detection. In *Proceedings of CCN'02*, November 2002.

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

[19] J. McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security*, 4(3):262–294, November 2000.

[20] P. Mell, V. Hu, and R. Lippmann. An overview of issues in testing intrusion detection systems. *http://csrc.nist.gov/publications/nistir/nistir-7007.pdf*.

[21] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. In *Proceedings of the Passive and Active Measurement Workshop (PAM)*, 2003.

[22] P. Mueller and G. Shipley. Dragon claws its way to the top. *Network Computing*, pages 45–67, July 2001.

[23] M. Muuss and T. Slattery. http://ftp.arl.mil/ftp/pub/ttcp.

[24] NLANR Measurement and Operations Analysis Team. NLANR network traffic packet header traces. *http://pma.nlanr.net/Traces/*.

[25] D. Robert, C. Terrence, W. Brian, M. Eric, and S. Luigi. Testing and evaluating computer intrusion detection systems. *Communications of the ACM*, 42(7):53–61, September 1999.

[26] S. Roux, D. Veitch, P. Abry, L. Huang, J. Micheel, and P. Flandrin. Statistical scaling analysis of TCP/IP data using cascades.

[27] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the performance of network intrusion detection sensors. In *Proceedings of Recent Advances in Intrusion Detection (RAID 2003)*, September 2003.

[28] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2003.

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

[30] The NSS Group. Intrusion detection systems group test, December 2001. http://www.nss.co.uk/ids.

[31] The Shmoo Group. Capture the flag contest (defcon). Available at http://www.shmoo.com/cctf/.

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