Operational Experiences with High-Volume Network Intrusion Detection

Holger Dreger
TU München
Germany
dreger@in.tum.de

Anja Feldmann
TU München
Germany
anja@in.tum.de

Vern Paxson
ICSI / LBNL
Berkeley, CA, USA
vern@icir.org

Robin Sommer
TU München
Germany
sommer@in.tum.de

ABSTRACT

In large-scale environments, network intrusion detection systems (NIDSs) face extreme challenges with respect to traffic volume, traffic diversity, and resource management. While crucial for acceptance and operational deployment, the research literature mainly omits such practical difficulties. In this paper, we offer an evaluation based on extensive operational experience. More specifically, we identify and explore key factors with respect to resource management and efficient packet processing and highlight their impact using a set of real-world traces. On the one hand, these insights help us gauge the trade-offs of tuning a NIDS. On the other hand, they motivate us to explore several novel ways of reducing resource requirements. These enable us to improve the state management considerably as well as balance the processing load dynamically. Overall this enables us to operate a NIDS successfully in our high-volume network environments.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks]: Network Operations - *Network monitoring*.

General Terms: Measurement, Performance, Security.

Keywords: Bro, Evaluation, Network Intrusion Detection, Security

1. INTRODUCTION

The practical experience of running a network intrusion detection system (NIDS) operationally is that with increasing volume the challenges grow faster than linear. Three major difficulties arise. First, the sheer packets-per-second (pps) rates can reach levels at which the load due to interrupts and filtering push the system into thrashing. Second, as volume rises—particularly if it rises due to greater numbers of hosts—so does the traffic's *diversity*, which can stress the NIDS's fidelity by generating both more false alarms and a wider range of types of false alarms. Third, as the number of hosts increases, so does the burden of managing *state* and other resources.

These practical difficulties with high-volume network intrusion detection rarely see investigation in the research literature: NIDS vendors often have a commercial interest in downplaying the difficulties and keeping private their techniques for addressing them, and researchers seldom have opportunities to evaluate high-volume, operational environments.

In this paper, we offer such an evaluation. Our study is in the context of using commodity PC hardware running open-source software for operational security monitoring of quite high-volume environments (Gbps, 10s of thousands of hosts transferring 2-3 TB/day). We found that in such environments, if we simply install and run an untuned/uncustomized NIDS such as the open-source Snort [19] or Bro [16] systems, they are unable to effectively cope with the amount of traffic. Snort immediately consumes the entire CPU, leading to excessive packets losses, while Bro, in addition, quickly exhausts all available memory.

Obviously, the volume is too great a burden for the NIDS. But what are the key factors that lead to such severe difficulties? In this study we look at a number of issues that arise due to the problems of resource management and efficient packet capture and filtering. We aim to analyze the main contributors to CPU load and memory consumption and look for means to ameliorate their impact, which sometimes requires developing new mechanisms if the available tuning parameters do not suffice.

For a *stateless* NIDS, the load imposed on the CPU is the main limiting factor. This load is correlated with the types of analysis as well as the traffic's volume and makeup. A *stateful* NIDS, additionally, maintains an in-memory representation of the current state of the network, which must be meticulously maintained at all times. This state provides the context necessary to evaluate the network events. Like CPU load, the volume of the state is also correlated with the traffic volume as well as the types of analysis, and is constrained by the system's available memory. Since maintaining state requires state management, the NIDS requires some significant CPU time just for updating data structures.

Common approaches for limiting NIDS resource usage include different kinds of state management (e.g., via timeouts and/or fixed size buffers); checkpointing [16] (i.e., regularly restarting the system to flush old state); limiting the traffic by analyzing only certain protocols or subsets of the address space; and distributing the work to multiple machines. To understand the efficacy of these approaches, we examine the resource requirements of a NIDS and the associated trade-offs *in operational use*.

Our study is in the context of the Bro NIDS, which we have deployed operationally in a couple of high-performance environments. Bro is a highly stateful NIDS. Its basic model has three layers: packet filtering, event generation, and policy script execution. Packet filtering is done using a static BPF expression [10]. Events are generated by an *event engine* which performs policy-neutral analysis of network traffic at different semantic levels. For example, there are events for attempted/established/terminated/rejected connections, the requests and replies for a number of applications, and successful and unsuccessful user authentication. Finally, the user writes *policy scripts* using a specialized, richly-typed high-level language. These scripts execute on the events generated by the event engine and codify the actions the NIDS should take: updating data structures describing the activity seen on the network, sending

CCS'04, October 25-29, 2004, Washington, DC, USA.

out real-time alerts, recording activity transcripts to files, and executing programs as a means of reactive response. Thus, both the event engine layer and the policy script layer generate and manage a great deal of state.

We find that three factors dominate overall resource consumption: *(i)* the total amount of state kept by the system, *(ii)* the traffic volume, and *(iii)* the (fluctuating) per-packet processing time. While these factors certainly are not surprising by themselves, the key is understanding the trade-offs between them with respect to tuning a NIDS and adapting it to the environment. In addition, we found several new ways to reduce resource requirements, considerably improving state management and dynamically balancing the processing load. While the concrete realization of these is tied to the particular system we examine, the underlying concepts are applicable to other NIDS as well.

Overall this work provides us with a NIDS much more suitable for use in high-volume networks, both in terms of raw capabilities and greater ease of tuning. In addition, our study illuminates complexities inherent in analyzing, tuning and extending systems that must process tens-to-hundreds of thousands of packets per second in real-time. We find that to understand memory usage and CPU load spikes, particular care is needed to soundly instrument the system; the somewhat atypical trade-off between CPU and memory vs. detection rate, and the sensitivity of such a system to quite small programming errors is rather illuminating; furthermore, a high-volume monitoring environment can exhibit artifacts that significantly affect any analysis.

In §2 we summarize related work. After describing the main high-volume environment that we use for our study (§3), we discuss our operational experiences and demonstrate certain effects using a set of traces (§4). In §5 we present several enhancements to Bro which together enable us to now operate Bro successfully in high-volume environments.

2. RELATED WORK

Reports in the literature of operational experiences with high-volume network intrusion detection are quite rare. More generally, a major question for evaluation studies is what sort of traffic to use. [7] proposes a methodology to craft traffic with different characteristics. But in high-volume environments, such characteristics are often unpredictable. Traffic patterns vary widely between different environments [5, 15], and Internet traffic includes significant short-term fluctuations [4]. Moreover, attack traffic can change the picture considerably: denial-of-service floods using spoofed source addresses can generate many thousands of new (apparent) flows per second [13], greatly altering the total traffic pattern, as can worm propagation [11, 24]. In addition, attackers can target the NIDS itself to try to evade [17] or overload the system [2, 16]. Tools like Snot [21] or Stick [25] craft packets to match known attack signatures, thereby stressing the NIDS's logging system.

To avoid overload, some systems distribute the analysis across multiple machines (e.g., [8, 23]). This certainly can help, but the individual machines still face the fundamental problem of limiting and managing their resource usage. Along these lines, [9] presents an approach for adapting the configuration of a NIDS to the current load. By quantifying benefits and costs of analysis tasks, they dynamically determine the best configuration under given resource constraints. While our concept of load-levels (see §5.3) is similar in spirit, we find it quite hard to crisply define such cost metrics for high-volume traffic analysis (see §4.3 and §4.4). Thus, we statically define a set of configurations appropriate for medium-term traffic changes (which may not suffice under overload attack situations).

Most evaluations consider detection rate as their major performance criteria, gauging trade-offs between false positives and false negatives. But from our experience we argue one must not lose sight of the fundamental trade-off between detection rate and resource usage. It is rare that studies explore this consideration, and in fact often the particular configurations of evaluated systems are unclear. For example, for signature matching [22] shows that alerts often depend on the underlying implementation and its concrete parameterization. For a general discussion of these difficulties and pitfalls, see [18].

Using commodity hardware, high-speed packet capture is quite challenging [1]. A key factor is the architecture of the operating system's packet filter [3]. On our monitors, we use FreeBSD, which provides an in-kernel implementation of the *Berkeley Packet Filter* [10], giving us quite efficient, stateless packet filtering.

3. ENVIRONMENTS

The basis for our study is our operational experiences monitoring a heavily-loaded Gb/s environment, Münchener Wissenschaftsnetz (*MWN*), Germany. The MWN provides Internet connectivity to 2 major universities and a number of research institutes. Overall, the network contains about 50,000 individual hosts and 65,000 registered users. On a typical day, 1–2 TB of data is transferred, which averages to 44,000 packets/sec. Usually, most of the connections are HTTP (65–70%), while the biggest contributor to traffic volume is FTP (about half of the total volume).[1] The primary NIDS monitor is a Dual Athlon MP 1800+ with 2 GB Memory, currently running FreeBSD 5.2.1. It is connected via a Gigabit Ethernet link to a port of the MWN's upstream router (a Cisco 6509), allowing it to monitor all traffic between MWN and the Internet.

In addition, we gathered operational experiences from a second high-volume environment, University of California, Berkeley (*UCB*), USA. The traffic volume at UCB is even higher (2–3 TB). Since the challenges already manifest in the MWN environment, and due to easier access to experimental platforms, in this study we concentrate on MWN.

While we gained our experiences and insights from deploying the NIDS operationally, live traffic poses limitations for any systematic performance evaluation study, e.g., in terms of repeatability of experiments. Therefore we draw upon a set of traces captured using tcpdump [26] at the MWN monitor to demonstrate the challenges that a NIDS is facing:

The trace `mwn-week-hdr` contains all TCP control packets (SYN, FIN, RST) for a 6 day period. The compressed trace totals 73 GB, contains 365M connections, and 1.2G packets. 71% of the packets in the trace use port 80 (HTTP), with no other port comprising more than 3% of the traffic. `mwn-all-hdr` is a 2-hour trace containing all packet headers, captured during the daily "rush-hour" between 2PM and 4PM. (Basic statistics: 13 GB compressed, 471M packets, 11M TCP connections, 96.7% of the packets are TCP (57.8% HTTP, 3.7% FTP data transfer on port 20), 2.9% UDP). `mwn-cs-full` is a 2-hour trace including the full payload of all packets to/from one of the CS departments in MWN, with some high-volume servers excluded. This trace was captured at the same time as `mwn-all-hdr`. (11 GB compressed, 19M packets, 404K connections, 88% of the packets are TCP (41% HTTP, 11% NNTP), 10% UDP). `mwn-www-full` is a 2-hour trace including full payload from `dict.leo.org`, a popular Web server. (2.8 GB compressed, 38M packets, 1M connections of which nearly all are HTTP). `mwn-irc-ddos` is a 2.5-day trace including full payload from `irc.leo.org`. During the moni-

[1]The network is configured to *block* well-known peer-to-peer ports, along with certain other services—primarily SNMP, NetBIOS/SMB, and Microsoft SQL.

toring period this IRC server was subjected to a large distributed denial-of-service attack which used random source addresses. The trace contains three major attack bursts, with peaks of 4,800, 5,300, and 35,000 packets per second, respectively (2.8 GB compressed, 76M packets, 96% TCP / ports 6660–6668, 2% UDP).

Tcpdump reported 0.01% or fewer lost packets for each of these traces. For the evaluation itself we had exclusive access to three systems. One is the monitor itself, which we mainly used for any analysis involving the large `mwn-week-hdr` trace. The others are separate Athlon XP 2600+ based systems with 1 GB of RAM running Linux 2.4. To keep the analysis comparable in terms of memory use, we imposed a memory limit of 1 GB on all experiments independent of the system. Furthermore, results used for comparisons are derived using the same experimental system.

The performance evaluations presented in the following sections use the measurement methodologies for system memory usage and run-time measurements presented in Appendix A and B.

4. OPERATIONAL EXPERIENCES

Deploying NIDSs operationally in our high-volume environments presents several different challenges. Most problems announce themselves either by exhausting the system's memory or by consuming all available CPU time—or both. But while the symptoms often are similar in appearance, they have a number of different causes. Often, detecting and fixing a particular problem leads to the immediate appearance of another one. Overall, each choice, e.g., of analysis depth or of parameter values, faces a trade-off between quality (i.e., detection rate) and quantity (i.e., required resources).

In this section we discuss major issues that had to be addressed: state management that is either too liberal, or not existent at all; data and processing peaks causing missed packets; and small programming deficiencies causing major problems. Next we recapitulate a recurring experience: in network intrusion detection, one faces a rather unusual trade-off between resource requirements and detection rate. Finally we conclude the section outlining some problems due to the monitoring environments rather than the NIDS itself. We discuss various mechanisms that allow us to overcome these difficulties in Section 5.

4.1 Connection State Management

For a stateful NIDS, it is vital to limit the overall memory requirements for state management to a tractable amount. In a high-volume environment, this is particularly difficult if the NIDS keeps per-connection state. In MWN, on a typical day we see up to 4,000 new TCP connections per second, and a total of about 75M TCP connections per day.

The amount of memory required for connection state is determined by two factors: *(i)* the size of a state entry, and *(ii)* the maximum number of concurrent, still active connections. In Bro, the size of state entries differs due to factors such as IP defragmentation, TCP stream reassembly, and application-layer analysis, which determine the amount of associated state. To limit the number of concurrent connections, NIDSs employ timeouts to expire connection state, i.e., connections are removed from memory when some expected event (e.g., normal termination) has not happened for a (configurable) amount of time. In addition, some NIDS either limit the total number of concurrent connections or the amount of memory available for connection state. In either case, they flush connections aggressively once the limit is reached. Snort, for example, simply deletes five random connections once it reaches a configurable memory limit. While such a limit makes the memory requirements more predictable, it leads to ill-defined connection lifetimes.

For TCP connections, Bro's state entries consist of at least 240 bytes. If Bro activates an application analyzer for a connection, this can grow significantly. Running Bro in its default configuration[2] on the traces `mwn-week-hdr` and `mwn-www-full`, we observe average connection entry sizes of about 1 KB. When performing HTTP decoding on `mwn-www-full`, Bro needs 6–8 KB per connection (excluding data buffered in the stream reassembler, whose peak usage for this trace is in fact less than 5 KB in total).

When we store a significant number of bytes per connection, it is important to limit the number of concurrent connections. Yet, Figure 1(a, bottom), shows that on `mwn-week-hdr`, with Bro's default configuration, the number of connections increases over time. Consequently, the system crashes after 2.5 days due to reaching the 1 GB memory limit.

The arrival of new connections (Figure 1(a, top)) does not exhibit a similar increasing trend. This implies that the problem is not a surge in connection arrivals but rather Bro's state management. It does not limit the number of current connections, and at the same time it apparently fails to remove a sufficient number of connections from memory.

Since the number of connections in the "SYN seen" state does not increase dramatically, we conclude that the problem is not that Bro fails to time out unsuccessful connection attempts. Indeed, Bro provides an explicit timeout mechanism for dealing with such connections. Decreasing these timeouts to the more aggressive thresholds used by Bro's `reduce-memory` configuration enables Bro to process an additional 110 minutes of the trace, only a minor gain.

Figure 1(a, bottom) indicates that the number of connections in the established or half-closed state increases to the same degree as the total number of connections. After further analysis of Bro's state management, we find that many connections are not removed *at all*. This behavior is consistent with Bro's original design goal: to not impose limits that an attacker might exploit to evade detection [16]. The problem faced by a NIDS is that there is no point at which it can be *sure* that a TCP connection in the "established" state can be safely removed. Instead, Bro's original, very coarse-grained state management approach is to periodically terminate the monitor, flushing all state, and then restart the analysis from scratch. Clearly, this approach degrades the quality of the analysis and provides easy evasion to attackers who split their attacks across restarts.

In accordance with the above design goals, Bro does not remove connections unless it sees an indication that they are properly closed. There are at least three reasons for why one may not see the end of a connection: *(i)* hosts which, for whatever reason, do not close connections, *(ii)* packets missed by the NIDS itself (see Sections §4.3 and §4.4), and *(iii)* artifacts caused by the monitoring environments (see Section §4.7).

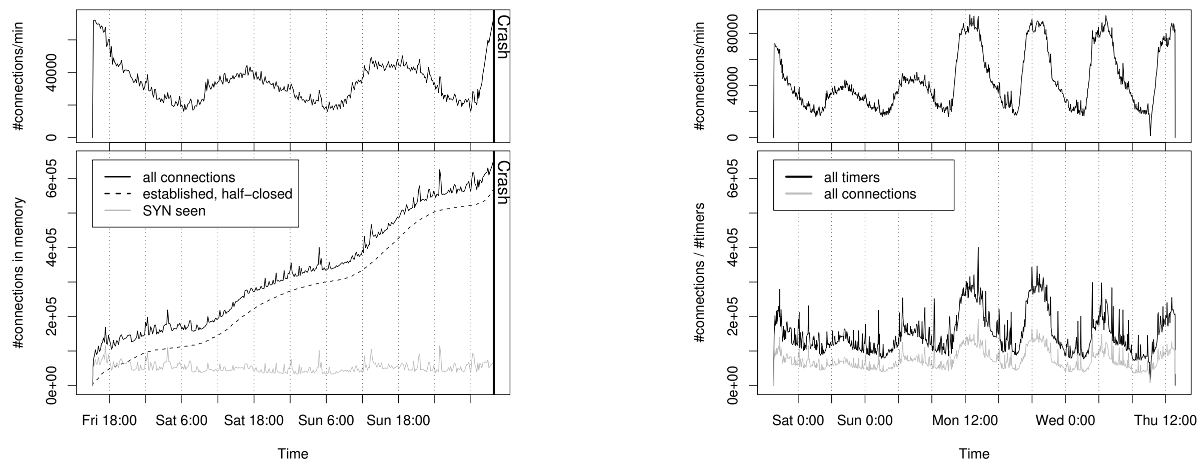
Whatever the cause, however, accumulating connection state indefinitely over time is clearly not feasible. There are similar problems with UDP and ICMP "connections".[3] Most of these are also never removed from memory. While there already is a way to mitigate these problems,[4] this still does not suffice. In Section §5.1.1 we develop an approach to mitigate this problem using inactivity timers. Figure 1(b) shows the success of including this extension.

[2]If not stated otherwise, we deactivate most of Bro's analyzers for the measurements presented in this paper. The only (major) user-level script we include is `conn.bro`, which outputs one-line summaries of all connections [16]. In this configuration, Bro performs stateful analysis of all TCP control packets, but no application-layer analysis.

[3]While UDP and ICMP are not connection-oriented, Bro uses a flow-like definition to fit them into its connection-oriented framework.

[4]There is a (by default deactivated) timeout to expire all not-further analyzed connections after a fixed amount of time.

Figure 1: Connection state from `mwn-week-hdr`.



(a) Default configuration

(b) With inactivity timeouts

4.2 User-Level State Management

A NIDS may provide the users with the capability to dynamically create state themselves. While not all NIDSs provide such *user-level state*—Snort, for example, does not—other systems, like Bro, provide powerful scripting languages. But similar to the system's connection state, user-level state must be managed to avoid memory exhaustion. There are two approaches for doing so: *(i) implicit* state management, where the system automatically expires old state, perhaps using hints provided by the user; and *(ii) explicit* state management, where the user is responsible to flush the state at the right time.

Bro provides only explicit mechanisms, and the default policy scripts supplied with the public distribution make little use of these, motivated by Bro's original philosophy of retaining state as long as possible to resist evasion. Consequently, user-level state accumulates over time, generally causing the system to crash eventually. Two examples are the scan detector and the FTP analyzer. The former stores a table of host pairs for which communication was observed. Figure 2(a, bottom) shows the memory allocation for the user-level state of the scan analyzer versus total memory allocation running on `mwn-week-hdr`. (The chosen configuration avoids the growth of connection state by using inactivity timeouts as described in Section 5.1.1.) Figure 2(a, top) again shows the number of connections seen per minute. We recognize from Figure 2(a, bottom) that the table mentioned above grows rapidly, since its entries are never removed. While this maximizes the scan detection rate (we will not miss any scans, thus thwarting evasion), it is infeasible in environments with large numbers of connections. For example on `mwn-week-hdr` the memory limit is reached after a bit more than 4 days. Here the main question is not whether to remove the table entries but *when*.

The FTP analyzer remembers which data-transfer connections have been negotiated via an FTP session's control channel, and removes this information only when the connection is indeed seen. While there is a point when this information can be safely removed—when the control connection terminates—this point is difficult to robustly detect from a user-level script, because Bro provides a multitude of event handlers for numerous kinds of connection termination. Even worse, there are (rare) cases when none of these events are generated.[5]

In Section 5.2 we develop another extension to Bro, user-level timeouts, for expiring table entries which results in a significantly reduced memory footprint as shown in Figure 2(b).

4.3 Packet Drops due to Network Load

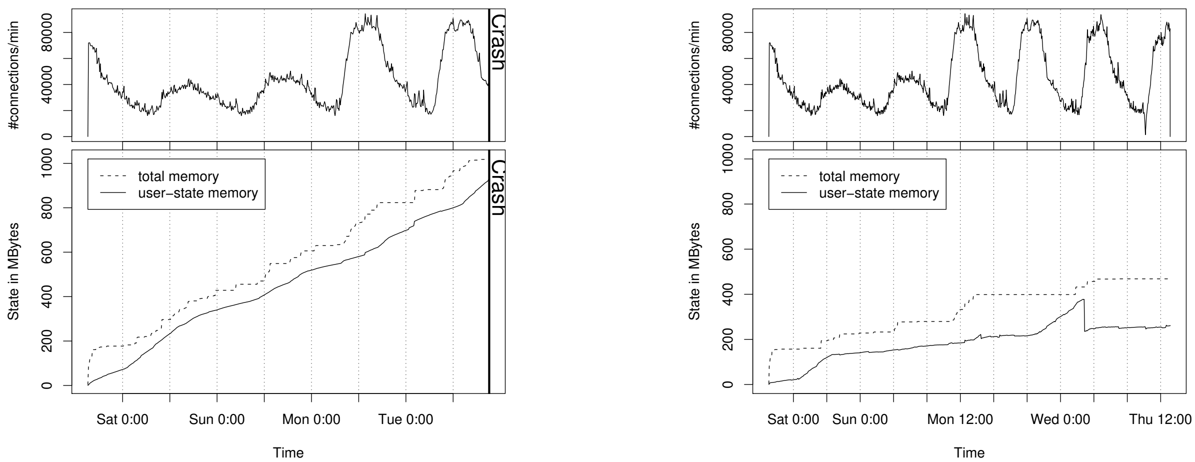
In high-bandwidth environments, even after carefully tuning the NIDS to the traffic one still has to deal with inevitable system overloads. Given a heavily-loaded Gbps network, current PC hardware is not able to analyze every packet to the desired degree. For example, within MWN it is usually possible to generate Bro's connection summaries for all traffic, yet decoding and analyzing all traffic up to the HTTP protocol level is infeasible. To demonstrate how expensive detailed protocol analysis can be, we ran the HTTP analyzer on `mwn-www-full` and `mwn-cs-full`. Compared to generating connection summaries only, the total run-times increase by factors of 6.2 and 5.6, respectively.

Therefore, we need to find a subset of the traffic and types of analysis that the NIDS can handle *given its limited CPU resources*. Doing so for real-world traffic is especially challenging, as the traffic exhibits strong time-of-day and day-of-week effects. (In the MWN, the traffic in the early afternoon is usually about 4 times larger than during the night; in fact, during night we *are* able to analyze all HTTP traffic.) If we configure the analyzers for the most demanding times, we waste significant resources during low-volume intervals. Thus, we miss the opportunity to perform more detailed analysis during the off-hours. Alternatively, we could configure for the off-hours, but then we may suffer massive packet drops during the peaks.

In Section §5.3, we develop a mechanism to mitigate this problem by dynamically adjusting the NIDS to the current load. While this is extremely useful, we note that it remains an imperfect solution. We can still expect to encounter occasional peaks either due to the widespread prevalence of strong correlations and "heavy tailed" data transfers in Internet traffic [4, 28], or due to unusual situations such as flooding attacks, worm propagation, or massively misbehaving software (once we observed one of our local hosts

[5] The "crud" [16] seen in real-world networks sometimes misleads Bro's internal connection management. This also leads to some connections missing in Bro's connection summaries while others appear twice. We fixed this using the mechanism described in Section §5.2.

Figure 2: Memory required by scan detector on `mwn-week-hdr` using inactivity timeouts for connections.



(a) Default configuration

(b) With user-level timeouts

generating 100s of thousands of connection requests; a user was testing a new P2P client). Such situations can invalidate the assumptions underlying either the configuration of the NIDS or the processing of the NIDS itself. For example, the floods contained in `mwn-irc-ddos` contain millions of packets with essentially random TCP headers, which highly stress Bro's TCP state machine.

Thus, in practice, finding a configuration that never exceeds the resource constraints is next-to-impossible unless one keeps extremely large capacity margins. As perfect tuning is out of range within the trade-off of analysis depth vs. limited resources. We aim instead at a good balance: accepting some packet loss due to occasional overload situations while maintaining a reasonable analysis depth. For the MWN, we found a configuration which is able to run continuously (i.e., without the need to regularly checkpoint [16] the system) even in such demanding situations as caused by floods or large-scale scans. The occurrences of packet drops is within acceptable limits (e.g., 2–3 times an hour).

4.4 Packet Drops due to Processing Spikes

A NIDS processing traffic in real-time has a limited per-packet processing budget. If this NIDS spends too much time on a single packet (or on a small bunch), it may miss subsequent ones. It turns out that the per-packet processing time fluctuates quite a bit. If these fluctuations together add up to a significant amount of CPU time, the system will inevitably drop packets.

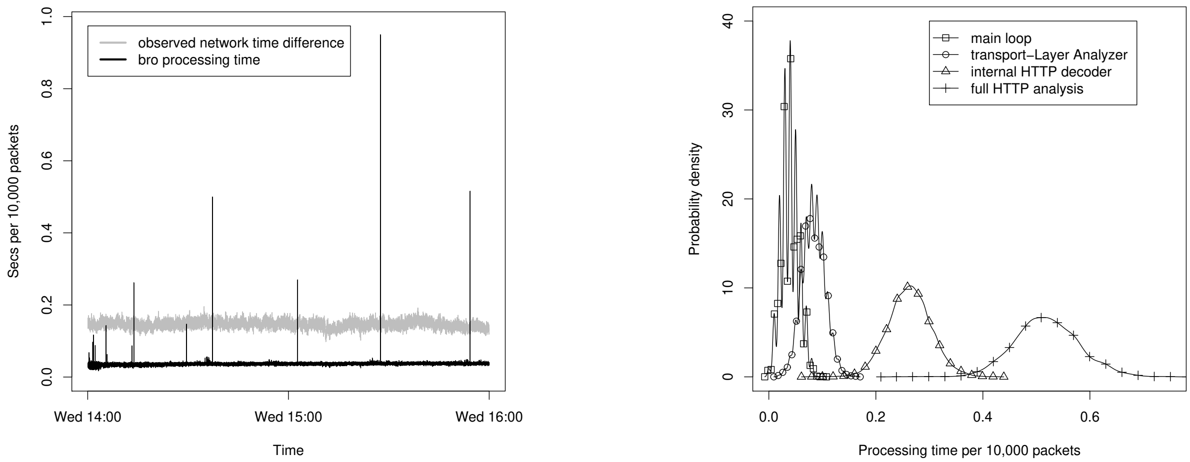
We find there are two major reasons for fluctuating packet processing times:

First, a single packet can trigger a special, expensive type of processing. For example, Bro dynamically resizes its internal hash tables when their hash bucket chains exceed a certain average length, in order to ensure that lookups do not take too long. Figure 3(a) shows the processing time for each group of 10,000 packets and the timespan in which this group of 10,000 packets was observed on the network. The plot allows us to compare the time needed to process 10,000 packets vs. the time needed to transmit them across the monitored network link. If the processing is faster, indicating that Bro's processing is staying ahead, the corresponding black sample point is below the gray sample point. Otherwise, if the gray sample is below the black sample, Bro is unable to keep up with the incoming packet rate (see Appendix B for details about this measurement model).

Note the spikes in Bro's processing time. These are caused by hash table resizing. Each resize requires Bro to copy all pointers from the old table to a new position within the resized table. For large tables—such as those tracking 100s of thousands of connections—such a copy takes hundreds of msec. This time is alloted to a single packet and therefore to a single group of 10,000 packets, causing the spike in the processing time. Note that the spike exceeds the network time, indicating the danger of packet drops. We have verified that this phenomenon indeed leads to packet drops in our high-volume environments.

To address this problem, we modified the hash table resizing to operate *incrementally*, i.e., per packet only a few entries are copied from the old table to the new table. Doing so distributes the resizing across multiple packets. While the amortized run-time of insert and remove operations on the table does not change, the worst-case run-time is decreased, which avoids excessive per-packet delays. We have confirmed that this change significantly reduces packet drops. Second, different types of packets require different analysis and therefore different processing times. For example, analyzing TCP control packets requires less time than the analysis of HTTP data packets. Yet since the content of packets differs even at the same processing levels, the times can vary significantly. Figure 3(b) shows the probability density functions of the processing time for groups of 10,000 packets for four different configurations. Each configuration adds an additional degree of analysis. The simplest configuration, "Main-loop," consists of Bro's main loop, which implements the full TCP state machine but does not generate any output. The second configuration, "Transport-Layer Analyzer," generates one-line summaries for every connection. The next configuration, "Internal HTTP decoder," does HTTP decoding without script-level analysis, while the last and most complex one, "Full HTTP analysis," adds script-level analysis. Note that the per-packet processing times vary significantly for each configuration. The amplitude of the fluctuations increases with the complexity of the configuration. This is due to the influence of the individual characteristics of each single packet, which gain more prominence as the depth of analysis increases. For the most complex configuration (full HTTP analysis), the standard deviation is 0.060 sec, whereas for the simplest configuration (only Bro's connection tracking and internal state management), the standard deviation is only 0.016 sec. In general, we observe that more detailed analysis increases the average processing time *and* increases its variability.

Figure 3: Processing time on `mwn-all-hdr` (left) and `mwn-www-full` (right).



(a) Old hash table resizing spikes

(b) Fluctuations in per-packets time

This increasing variability implies that interpreting such general statements as "decoding HTTP increases the run-time by x%" (cf. Section §4.3) need to be interpreted with caution. The actual change in run-time depends significantly on the particular input, and the additional processing delays may have even larger impact on real-time performance, by exceeding buffer capacities, than one might initially expect. More generally, this implies that judging NIDSs in simple terms such as maximum throughput (see, e.g., [7]) is questionable.

4.5 Sensitivity to Programming Errors

A surprising consequence of operating a NIDS in a high-volume environment is the degree to which the environment exacerbates the effects of programming errors. We have repeatedly encountered two kinds of mistakes that inevitably lead to significant problems no matter how minor they may first appear: *(i)* memory leaks, and *(ii)* invalid assumptions about network data.

Even the smallest memory leak can drive the system to memory exhaustion. Simply put, we require that every function that is part of the system's main loop must not leak even a single byte. For example, we once introduced a small leak in Bro's code for determining whether a certain address is part of the local IP space. This bug caused an operational system with 1 GB of memory to crash after two hours. Unfortunately, these kinds of errors are particularly hard to find. With live traffic, the main indicator is that the system's memory consumption slowly increases over time. Yet this does not yield any hints about the culprit. Furthermore, memory leaks are often hard to reproduce on small captured traces. Yet on large traces, conventional memory checkers like mpatrol [14] and valgrind [27] are terribly slow. In fact, such difficulties motivated us to instrument the NIDS to account for its memory consumption, as discussed in Appendix A. (The common riposte that one should use a language with garbage collection, rather than C++, is not as simple as some might think, as garbage collection processing can lead to processing spikes similar to those discussed in Section 4.4.)

The second problem concerns invalid assumptions about the system's input. If a protocol decoder assumes network data to be in some particular format, it will eventually encounter some non-conforming input. This problem is exacerbated in high-volume environments, due to the traffic's *diversity* as well as high rate, as discussed earlier. Indeed, we have several times encountered a protocol decoder running fine even on large traces, but crashing within seconds when deployed in one of our environments. Along these lines, not only does this observation mean that expecting strict conformance to an RFC will surely fail; but that expecting *any* sort of "reasonable" behavior risks failing. This problem is closely related to Paxson's observations of "crud" in network traffic [16] as well as "crash" attacks: not only must a NIDS be coded defensively to deal with bizarre-but-benign occurrences such as receivers acknowledging data that was never sent to them; but they must also be coded against the possibility of attackers maliciously sending ill-formed input in order to crash the NIDS, or, even worse, compromise it, as happened with the recent "Witty" worm [12].

4.6 Trade-off: Resources vs. Detection Rate

So far, we have seen several indications of a rather unusual trade-off in network intrusion detection: memory/CPU-time on one side against detection rate in the other. This is in contrast to computer science's more traditional trade-off between memory and CPU-time.

If we decrease the amount of state stored by the system, we automatically decrease the size of the internal data structures. Thus, we reduce both memory usage and processing time (even with efficient data structures like hash tables, more state requires more operations to maintain it). But, at the same time, we lose the ability to recognize attacks whose detection relies upon this state. Consider an interactive session in which the attacker first sends half of his attack, then waits some time before sending the remaining part. If the NIDS happens to remove the connection state before it has seen sufficient information to recognize the attack, it will fail to detect it. Similarly, if we decrease the CPU usage of the NIDS by avoiding certain kinds of analysis, we usually also reduce the amount of stored state. But again, we will now miss certain attacks.

Bro's original design emphasized detection. Many design decisions were taken to avoid false negatives, at the cost of large resource requirements. Unfortunately, as documented above, this approach can be fatal when monitoring high-volume networks. For example, recall that by default Bro does not expire any UDP state. In terms of detection, this is correct: being a stateless protocol, there is no explicit time at which the state can be removed safely. On the other hand, keeping UDP state forever quickly exhausts all available memory on a high-volume link.

Trading resource usage against detection rate is an environment-specific policy decision. By leaving the final decision (e.g., choosing the concrete timeouts) to the user, one avoids predictability. This is a variant of Kerkhoff's principle: while the detection mechanisms are public (the software is open source), their parameterizations are not. We note that choosing appropriate timeouts is not easy, and are investigating developing a tool for suggesting reasonable values for a particular environment based on traffic samples.

4.7 Artifacts of the Monitoring Environment

So far we have examined problems originating in the NIDS itself. We find that, in addition, high-volume environments also stress the monitor environment (i.e., the monitoring router and the NIDS's network interface card).

First of all, there are some general capacity limitations. At UCB, we monitor the traffic of several routers simultaneously by merging their Gbps streams using an RSPAN-VLAN [20]. This can exceed the monitor's Gbps capacity. Indeed, we see both missing and duplicated packets in the NIDS's input stream. We believe that this is due to the RSPAN setup. While only a single router is monitored at MWN, both directions of the network's Gbps upstream link are merged into one uni-directional monitor link using a SPAN port. While the available capacity is usually sufficient, the router does report occasional buffer-overruns (i.e., causing the monitor to miss packets). To overcome these limitations we intend to switch from a SPAN port to optical taps. Yet this introduces the problem of merging two traffic streams into one within the NIDS's system. This requires tight synchronizing between the two streams to maintain causality (e.g., SYN ACKs must be processed after the corresponding SYN).

To address this problem, we have developed both a kernel mechanism ("BPF bonding") and user-level support in Bro for merging multiple packet streams. The latter is useful for systems for which the kernel modification is not available. However, even this does not fully address the problem. *Interrupt coalescence* [1] provides a way to minimize the interrupt load incurred when capturing high-volume packet streams. When coupled with merging multiple streams, however, this can result in the NIDS receiving packets with non-monotone timestamps. Processing them out of order can then lead to incorrect state tracking. To overcome this problem, we implemented a "packet sorting" buffer in which Bro keeps recently received packets for some (user-configurable) time. Now packets with earlier timestamps can then be processed prior to those with later timestamps yet received earlier.

The Gbps NIC in the MWN monitoring system is a Intel Pro/1000 MF-LX. To avoid packet losses, we patched the FreeBSD kernel to increase the NIC driver's internal receive buffers and the packet capture library to increase its buffer by three orders of magnitude (after configuring the kernel to allow this).

5. HIGH-VOLUME IDS EXTENSIONS

Based on our experiences discussed in Section §4, there are two major areas where improvements of the NIDS show promise to improve high-volume intrusion detection: *(i)* state management and *(ii)* control of input volume. We devised new mechanisms for both of these. While their current implementation is naturally tied to Bro, the underlying ideas apply to other systems as well.

In the following, we discuss each improvement individually to gain an understanding of its impact. In practice, we use all of them. Together they are able to cope with the network load.

5.1 Connection state management

One major contributor to the NIDS's memory requirements is the connection state (see §4). To reduce its volume, we use two complementary approaches: *(i)* introducing new timeouts to improve state expiration, and *(ii)* avoiding state creation whenever possible.

5.1.1 Inactivity timeouts

In Section §4.1 we show that connections for which Bro does not see a regular termination accumulate. These amount to a significant share of the total connection state unless they are removed in some way. For expiring such connections, most NIDSs rely on an "inactivity timeout," i.e., they flush a connection's state if for some time no new activity is observed. There is one caveat: such a timeout relies on seeing all relevant packets. If a packet is missed, it might incorrectly assume that a connection is inactive. Missed packets can be related to drops due to monitoring issues (see §4.3, §4.4 and §4.7). But more importantly packets are also missed when the specified packet filter does not capture all relevant traffic. For example, if one only analyzes TCP SYN/FIN/RST control packets, then an inactivity timeout degrades to a static maximum connection lifetime.

We added three inactivity timeouts to Bro, for TCP, UDP, and ICMP respectively. We also added the capability that the user's policy scripts can define individual timeouts on a per-connection basis. The timeouts can be adjusted separately based on the service/port number of the connection using a default policy script. This enables us to, for example, select shorter values for HTTP traffic than for SSH connections. Figure 1(b, bottom) shows Bro's resource consumptions on `mwn-week-hdr` with an overall TCP inactivity timeout of 30 minutes.[6] In contrast to Figure 1(a, bottom), we see that the number of concurrent connections in memory no longer exhibits the increasing trend. It instead follows the number of processed connections per time-interval closely (see Figure 1(b, top)).

Naturally, inactivity timeouts should be as large as possible. But using timeouts on the order of tens of minutes or even hours revealed a significant problem with Bro's timer implementation: for processing efficiency, when a connection's state is removed, associated timers are only disabled, not removed. These timers are deleted once their original expiration time is reached. Using large timeout values, this results in more than 90% of the timers in memory being disabled. To reduce the memory requirements, we changed the code to explicitly remove old timers, expecting to accept a minor loss in performance. However, we found the run-time on `mwn-week-hdr` actually *decreased* by more than 20%. Figure 1(b, bottom) shows the number of timers in memory after this change.

5.1.2 Connection compressor

Examining the TCP connections monitored in our operational environments showed that a significant fraction corresponds to connection attempts without a reply. For example, for `mwn-all-hdr` 21% of all TCP connections are of this kind. For `mwn-week-hdr`, they account for 26% (recall that this trace contains only TCP control packets). Many of these connections are due to scans. In addition, we find that energetic flooding attacks—and also large worm events—vastly increase the number of connections attempts. Nearly all of these attempts are sure to fail.

As already discussed in Section §4.1 the minimum size of a connection state entry is 240 bytes. To reduce the memory requirements for such connections, we implemented a *connection compressor* to compress their state, leveraging the prevalence of unanswered connection attempts. The idea behind the connection compressor is simple: defer the instantiation of full connection state un-

[6]The inactivity timer in this example degrades to a static maximum connection lifetime since `mwn-week-hdr` only contains TCP control packets.

til we see packets from both endpoints of a connection. As long as we only encounter packets from one endpoint, the compressor only keeps a *minimal state record*: fixed-size blocks of 36 bytes which contain just enough information to later instantiate the full state *if required*. Most notably, this minimal state contains the involved endpoints and the information from the initiating SYN packet (e.g., options, window size, and initial sequence number). If we do not see a reply after a configurable amount of time, the connection attempt is deemed unsuccessful, and its (minimal) state record is removed.

Using fixed size records allows for very efficient memory management: we simply allocate large memory chunks for storing the records and organize them in a FIFO. Since the FIFO ensures that connection attempts are ordered monotonically increasing in time, connection timeouts are extremely simple. We just check if the first entry in the FIFO has expired. If so, we pop the record and continue until we reach a not yet expired entry.

Using the connection compressor when generating connection summaries[7] for `mwn-all-hdr` (`mwn-week-hdr`), the total connection state (including the minimal state records buffered inside the compressor) decreases by 51% (36%). In addition, we observed run-time benefits which at times can be rather significant. These benefits appear to depend on the traffic characteristics as well as the system's memory management, and merit further analysis to understand the detailed effects.

During our experiments, we encountered one problem when using the compressor: for some connections, Bro's interpretation of the connection's TCP state changes when activating the compressor. The compressor alters some corner-case facets of TCP state handling. While we attempt to model the original behavior as closely as possible, this is not always possible. In particular, we may see multiple packets from an originator before the responder answers. Sometimes originators send multiple *different* SYNs without waiting for a reply. In other cases we miss the start of a connection, stepping right into the data stream. While the change is definitely noticeable—affecting 2% of the connection summaries for `mwn-all-hdr`—nearly all of the disagreements are for connections which failed in some way. Since the semantics of not-well-formed connections are often ambiguous, these discrepancies are a minor cost if compared to the benefits of using the compressor.

Two additional optimizations are possible. First, if the responder answers with a RST to a connection request, the connection could be deleted immediately, rather than instantiated as is currently done. In this case, the compressor could avoid instantiating full connection state by directly reporting the rejected connection and flushing the minimal state record. This should be particularly helpful during floods. Second, we can choose to either not report non-established sessions at all, or only generate summaries such as "42 attempts from host a.b.c.d". For Bro, this would avoid creating user-level state for such attempts, potentially a significant savings.

5.2 User-level state management

As discussed in Section §4.2, a NIDS may provide the user with the capability to dynamically create their own custom state. In this regard, to cope with the requirements of our high-volume environments, we extended Bro's explicit state management and introduced an additional, implicit mechanism.

For Bro's existing, explicit state management mechanism, the fundamental (and only) question is *when* to decide to flush state. We inspected the state stored by its scripts and determined that a large fraction of the state is per-connection and stored in tables. Often, this can and should be removed when the connection terminates. To facilitate doing so, we added a new event which is reliably generated whenever a connection is removed from the system's state for whatever reason. We then modified the scripts to base their state management on the generation of this event (for example, we modified the FTP analyzer to remove state for tracking expected data-transfer connections whenever the corresponding control session terminates).

Often, however, we would rather have implicit—i.e., automatic—state management, relieving us of the responsibility to explicitly remove the state. Carefully-designed timeouts provide a general means for doing so. While Bro supports user-configurable timers, using them for state management requires the user to manually install the timers and also specify handlers to invoke when the timers expire.

To support implicit state management, we extended Bro's table and set data structures to support per-element timeouts of three different flavors: creation, write, and read. That is, for a given table (or set), the user can specify in the policy script a timeout associated with each element which expires T seconds after any of: the element's creation; the last time the element was updated; or the last time the element was accessed.

One benefit of this approach is that these timeouts provide a simple but effective way to add state management to already-existing scripts. Consider for example the scan analyzer, which, as mentioned above, can consume a great deal of state. Figure 2(b, bottom), shows the quite significant effects of running Bro on `mwn-week-hdr` using 15-minute read timeouts for the scan detection tables. (Note, the large spike on Wednesday seems to stem from a single host in the scan-detection data structures that performs large vertical scans. Eventually, all of its state is expired at once.)

Adding timeouts to the scan detector also revealed a problem, though: sometimes state does not exist in isolation, but in context with other entities. In this case, when we implicitly remove it, we can introduce inconsistencies. For example, the scan detector contains one table tracking pairs of hosts and another counting for each host the number of such distinct pairs involving the host. Automatically removing an entry from the first table invalidates the counter in the second.

To accommodate such relationships, we added an additional attribute to Bro's table type which specifies a script function to call whenever one of the table's entries expires and is removed. In the scan detector, this functions simply adjusts the counter and thus maintains consistency between the two tables.

5.3 Dynamically controlling packet load

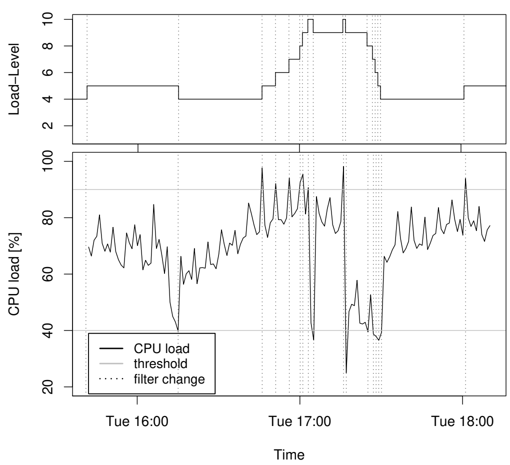
As discussed in Section §4.3, to avoid CPU exhaustion we need to find ways to control the packet load. Doing so statically—i.e., by controlling the BPF filter Bro uses for its packet capture—lacks the flexibility necessary to adapt to the wide range of conditions we can encounter over a relatively short period of time. Thus, we devised two new dynamic mechanisms: *(i) load-levels*, which allow us to adapt to the system's current load, and *(ii)* a *flood detector*.

We define *load-levels* as a set of packet filters for which we maintain an *ordering*. Each filter that is "larger" in the ordering than another imposes a greater load on the NIDS than its predecessor. (Note that the extra load is not due to the burden of the packet filtering per se, but rather the associated application analyzers that become active due to the packets captured by the filter, and the processing of the events that these analyzers then generate.)

At any time, the kernel has exactly one of the filters installed. By continuously monitoring its own performance, the NIDS tries to detect overloads (ideally, incipient ones) and idle times. During overloads, it backs off to a filter earlier in the ordering (i.e., one

[7]For these measurements, we used inactivity timeouts of 30 minutes. We only analyzed TCP control packets.

Figure 4: Load-levels



requiring less processing); during idle times, it ramps up to a filter that reflects more processing, because during these times the NIDS has sufficient CPU resources available and can afford to do so.

The filters are defined by means of Bro's scripting language. For example, the following code makes the activation of the DNS, SMTP and FTP decoders dynamic rather than static. A decoder is enabled if the system's level is less than or equal to the specified load level:

```
redef capture_load_levels += {
        ["dns"]   = LoadLevel1,
        ["smtp"]  = LoadLevel2,
        ["ftp"]   = LoadLevel3,
};
```

For such an adaptive scheme to work—particularly given its feedback nature—it is important to estimate load correctly to avoid rapid oscillations. Two of the possible metrics are CPU utilization and the presence of packet drops. (For either, we would generally average the corresponding values over say a couple of minutes, to avoid overresponding to short-term fluctuations.)

We have experimented with both of these metrics. We found that while the latter (packet drops) is indeed prone to oscillations, the former (CPU utilization) proves to work well in practice. The particular algorithm we settled on is to adjust the current packet filter if the CPU utilization averaged over two minutes is either *(i)* above 90% or *(ii)* below 40%, respectively.

To make this work, we cannot afford to compile new BPF filters whenever we adapt. Accordingly, we precompile the entire set of filters to keep switching inexpensive in terms of CPU. (As FreeBSD's packet filter flushes its captured-traffic buffers when installing a new filter, we also had to devise a patch for the kernel-level driver to avoid losing packets.)

Figure 4 shows an example of load-levels used operationally at MWN. When the CPU load crosses the upper threshold (Figure 4, bottom), the current load-level increases, i.e. Bro shifts to a more restrictive filter (Figure 4, top). Accordingly, if the load falls below the lower threshold, a more permissive filter is installed.

The MWN environment includes an IRC server which, unfortunately, is a regular victim of denial-of-service floods. It sometimes suffers such attacks several times a week, being at times targeted by more than 35,000 packets per second. Such a flood puts an immense load on a stateful NIDS, although ideally the NIDS should just ignore the attack traffic (of course after logging the fact), since there's generally no deeper semantic meaning to it other than clogging a resource by sheer brute force.

None of the mechanisms discussed above can accommodate in a "reasonable" fashion the flood present in `mwn-irc-ddos`. Thus, we devised a *flood detector* that is able to recognize floods and dynamically installs a new filter until the attack passes.

Detecting floods is straightforward: count the number of new connections per local host and assume a flood to be in progress if the count surpasses a (customizable) threshold. Doing so requires keeping an additional counter per internal host, which can be quite expensive. Therefore, we instead sample connection attempts. Similarly, instead of ignoring all packets to the victim after detecting a flood, we sample them at a low rate to quickly detect the end of the flood.

For Bro, we added such a flood detector by means of a new script. It samples connection attempts at a (customizable) rate of 1 out of 100, reporting a flood if the estimated number of new connections per minute exceeds a threshold (default, 30,000). When this occurs, the script installs a host filter, sampling packets also at 1:100.[8]

On `mwn-irc-ddos`, this mechanism detects all contained attack bursts. The total memory allocation stays below 122M (whereas all other considered configurations exhaust the memory limit of 1 GB during the last attack burst, at the latest).

6. CONCLUSION

In large-scale environments, network intrusion detection systems face extreme challenges with respect to traffic volume, traffic diversity, and resource management. In this study, we discuss our operational experiences with a NIDS in a Gbps network transferring multiple TBs each day. We identified the main contributors to CPU load and memory usage, understood the trade-offs involved when tuning the system to alleviate their impact, and devised new mechanisms when existing tuning parameters did not suffice.

Our study is in the context of the Bro NIDS. We are deploying it operationally in a couple of high-performance environments and were faced with several difficulties in terms of memory and CPU exhaustion. While the symptoms often appeared similar, these problems were due to a number of different reasons. First, the system's state management was designed to resist evasion, and thus traded detection-rate in favor of resource consumption. Second, the dynamic nature of the traffic makes it hard to find a stable point of operation without wasting resources during idle times, affecting both long-term traffic variations (e.g., due to strong time-of-day effects) and short-term fluctuations (e.g., due to "heavy-tailed" traffic and varying packet processing times). Third, even small programming errors (e.g, tiny memory leaks or not fully validated input) will almost certainly bother us eventually. Fourth, independent of the NIDS itself, high-volume traffic also demands a great deal of the rest of the monitoring environment (e.g., the monitoring router and the OS's packet capture subsystem).

For problems that could not be solved with the available tuning parameters, we developed new mechanisms. We improved state management by introducing new timeouts, deferring instantiation of connection state by means of a *connection compressor*, and adding new means to dynamically control the packet load (*load-levels* to automatically adapt the NIDS to the current network load; a *flood detector* to revert to sampling of high-volume denial-of-service attack flows).

[8]Unfortunately, the standard BPF packet filter does not support sampling. Thus, we augmented Bro's packet capture with a new, user-level packet filter that can directly support sampling. While this does not relieve the main process from receiving the flood's packets, they do not reach the system's main loop. Note, we are presently working with colleagues on extending BPF to support random and deterministic sampling.

In summary, our work provides us with *(i)* a thorough understanding of the trade-offs involved when tuning a NIDS for use in a high-volume network, and *(ii)* the tuning mechanisms necessary to successfully operate these systems in such challenging environments.

7. ACKNOWLEDGMENTS

Craig Leres implemented Bro's incremental resizing for hash tables and also the "BPF bonding" mechanism. Ruoming Pang implemented packet sorting. This work was supported in part by the U.S. National Science Foundation grant STI-0334088.

8. REFERENCES

[1] D. Agarwal, J. M. Gonzalez, G. Jin, and B. Tierney. An infrastructure for passive network monitoring of application data streams. In *Proc. Passive and Active Measurement Workshop*, 2003.

[2] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proc. 12th USENIX Security Symposium*, 2003.

[3] L. Deri. Improving passive packet capture: Beyond device polling. Technical report, University of Pisa, 2003.

[4] A. Feldmann, A. C. Gilbert, and W. Willinger. Data networks as cascades: Investigating the multifractal nature of Internet WAN traffic. In *Proc. of ACM SIGCOMM*, 1998.

[5] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4), 2001.

[6] GNU Binutils. `http://www.gnu.org/software/binutils`.

[7] M. Hall and K. Wiley. Capacity verification for high speed network intrusion detection systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[8] C. Krügel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[9] W. Lee, J. B. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[10] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter 1993 USENIX Conference*, 1993.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Magazine of Security and Privacy*, 2003.

[12] D. Moore and C. Shannon. The spread of the Witty. `http://www.caida.org/analysis/security/witty`, 2004.

[13] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proc. 10th USENIX Security Symposium*, 2001.

[14] `mpatrol`. `http://www.cbmamiga.demon.co.uk/mpatrol`.

[15] V. Paxson. Empirically-derived analytic models of wide-area tcp connections. *IEEE/ACM Transactions on Networking*, 2(4), 1994.

[16] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24), 1999.

[17] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[18] M. J. Ranum. Experiences benchmarking intrusion detection systems. Technical report, NFR Security, Inc., http://www.itsecurity.com/papers/nfr2.htm, 2001.

[19] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. 13th Systems Administration Conference (LISA)*, pages 229–238, 1999.

[20] Configuring SPAN and RSPAN (Cisco Catalyst 6500 Series). `http://www.cisco.com/univercd/cc/td/doc/product/lan/cat6000/sw_7_5/conf%g_gd/span.pdf`.

[21] Snot. `http://www.stolenshoes.net/sniph/index.html`.

[22] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[23] R. Sommer and V. Paxson. Exploiting independent state for network intrusion detection. Technical report, TU München, 2004.

[24] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proc. 11th USENIX Security Symposium*, 2002.

[25] Stick. `http://packetstormsecurity.nl/distributed/stick.htm`.

[26] `tcpdump`. `http://www.tcpdump.org`.

[27] `Valgrind`. `http://developer.kde.org/~sewardj/`.

[28] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1), 1997.

APPENDIX

A. MEASURING MEMORY USAGE

If we want to reduce the memory usage of a stateful NIDS, we need to understand where exactly it stores the state. To analyze the memory layout during run-time we can either use external tools, or add internal measurement code.

External Tools: There are several tools available for memory debugging, of which we have used two that are freely available: mpatrol [14] and valgrind [27]. The former comes as a library which is linked into the system and allows very fine-grained analysis by taking memory snapshots at user-controlled points of time. Unfortunately, mpatrol turned out to decrease the system's performance by multiple orders of magnitude, making it unusable on all but tiny traces (let alone real-time use). Valgrind takes another approach: it simulates the underlying processor on the instruction-level. While its performance is much better, it is still not sufficient for more than medium-size traces. Both programs proved to be most useful for finding illegal memory accesses.

Internal measurement: For internal measurements, we instrument the system using additional code to measure its current memory consumption. We identified Bro's main data structures and added methods to track their current size. During run-time, we regularly log these values. Additionally, we print the maximum *heap size* as reported by the system, and the *effective* memory allocation, i.e. the amount of memory currently handed out to the application by the C library's memory management functions. On Linux using glibc, the heap size is monotonically increasing and always provides us with an upper bound for the application's peak allocation.[9] We note that there is a gap between the peak heap size and the peak memory allocation: glibc keeps 8 bytes of hidden information in every allocated memory block, which is not counted against the current allocation. There is another pitfall when measuring memory. If we ask the C library for n bytes of memory, we may actually get $n + p$, of which $p \geq 0$ are padding bytes. For example, on Intel-Linux, glibc's malloc() always aligns block sizes to multiples of eight and does not return less than 16 bytes. The memory allocation *includes* these padding bytes.

We note that in practice it is very hard to instrument a complex system accurately. Therefore, values delivered by internal instrumentation are often only estimates of lower bounds. Bro, e.g., creates data structures at many different locations and often recursively combines them into more complex structures. Often it is not determinable what part of the code should be held accountable for particular chunk of memory. Consequently, we did not try to classify every single byte of allocated memory. Rather we identified the main contributors. By comparing their total to the current memory allocation, we ensure that we indeed correctly instrumented the code (on average, we are able to classify about 90% of the memory allocation; the rest is allocated at locations that we did not instrument).

In the main text, *total* memory allocation refers to the heap size. When we discuss the size of a particular data structure we refer to the values reported by our instrumentation, and thus to lower bounds. These values *include* malloc()'s padding and assume a glibc based system. When we give the memory allocation for a particular trace, we always refer to the *maximum* for this trace.

B. MEASURING CPU USAGE

To measure the CPU usage of a NIDS, we have similar options as for quantifying memory usage: external tools and internal instrumentation.

External Tools: An obvious tool to measure CPU usage is the Unix `time` tool. It reports overall real-, user-, and system-time and does not impose any overhead on the observed process. It does not provide any hints about the system's real-time behavior, though (while the CPU load may be sufficiently low on average, processing spikes can lead to packet drops). Performance profilers, like `gprof` [6], provide more fine-grained insight, but their overhead is much too large to infer real-time behavior.

Internal measurement: When examining CPU load, our main concern are packet drops. If we would know the exact time required to process each packet, we could say when drops occur: assuming BPF's double buffering-scheme [1], we lose packets when the total time required to process the first buffer's packets exceeds the time which can be stored in the second buffer.

Unfortunately, we cannot accurately measure the CPU time per packet. The overhead would be too large and the system's time granularity too coarse.[10] Thus, we use another model. We measure the time t required for a group of n packets and chose n so that t lies in the order of the timing resolution. When t exceeds the interval s in which the same n packets appeared on the network, we assume the system would drop packets. Additionally, assuming a packet buffer of size n, there will not be any packet drops when t does *not* exceed s. We note that by averaging over n packets, we cannot blame a single packet or a small group of packets as being responsible for a sudden increase of CPU usage. Also, we cannot quantify how many packets would have been lost.

For our experiments, we used $n = 10,000$, giving us times t within 30–50ms with Bro's minimal configuration (on an Athlon XP 2600+). Figure 3 shows that this method is indeed able to identify processing spikes. Also, we see that fluctuations in per-packet processing times are easily observable.

We note that this is an idealized model. The system's time measurements are not accurate. Also, on a real system, there are other factors that influence the packet drop

[9] Different systems behave differently. On FreeBSD, we can allow the C library to return unused memory to the system, thereby decreasing the total heap-size. On the other hand, FreeBSD's C library does not provide an easy way to access the current allocation.

[10] Linux/FreeBSD's `getrusage` system calls provide a default resolution of approximately 10ms on Intel hardware.

rate (such as load imposed by interrupts and other processes, or the OS itself). Finally, BPF's buffer implementation differs from our model. Thus, we do not claim to get perfect values of real-time CPU usage. But our measurements give us some very valuable intuition on the system's behavior. Part of our ongoing work is to flesh out the model further.