# Improving the Accuracy of Network Intrusion Detection Systems Under Load Using Selective Packet Discarding

Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos
Institute of Computer Science,
Foundation for Research and Technology – Hellas
Heraklion, Crete, Greece
{papadog,mikepo,markatos}@ics.forth.gr

## ABSTRACT

Under conditions of heavy traffic load or sudden traffic bursts, the peak processing throughput of network intrusion detection systems (NIDS) may not be sufficient for inspecting all monitored traffic, and the packet capturing subsystem inevitably drops excess arriving packets before delivering them to the NIDS. This impedes the detection ability of the system and leads to missed attacks. In this work we present *selective packet discarding*, a best effort approach that enables the NIDS to anticipate overload conditions and minimize their impact on attack detection. Instead of letting the packet capturing subsystem randomly drop arriving packets, the NIDS proactively discards packets that are less likely to affect its detection accuracy, and focuses on the traffic at the early stages of each network flow. We present the design of selective packet discarding and its implementation in Snort NIDS. Our experiments show that selective packet discarding significantly improves the detection accuracy of Snort under increased traffic load, allowing it to detect attacks that would have otherwise been missed.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.8 [**Operating Systems**]: Performance—*Measurements, Monitors, Operational analysis*

## General Terms

Security, Performance, Experimentation

## Keywords

intrusion detection, selective packet discarding, overload control

## 1. INTRODUCTION

Network Intrusion Detection Systems (NIDSs) are crucial for the detection of security violations and suspicious activity, enhancing the robustness and secure operation of modern networks. However, the constant increase in link speeds and number of security threats poses significant challenges to NIDSs, which need to cope with higher traffic volumes and perform increasingly complex per-packet processing. When the network traffic load becomes higher than the peak processing throughput the NIDS can sustain, the CPU becomes saturated, and the Operating System inevitably starts dropping packets before delivering them to the NIDS, impeding its detection ability. Since these packets are not inspected, if they are part of an attack or other malicious activity, then that event will be missed.

Several techniques have been proposed for improving the performance of NIDSs by accelerating the packet processing throughput and thus processing higher traffic loads [2, 4, 9, 23]. Other techniques automatically tune the NIDS configuration to balance detection accuracy and resource requirements [6, 10]. However, given a highly loaded network, intrusion detection systems based on non-specialized hardware are usually not able to analyze all traffic to the desired degree [18]. Even after carefully tuning the NIDS according to the monitored environment, it will still have to cope with inevitable traffic bursts or processing spikes [19].

In this paper, we present *selective packet discarding*, a technique that allows a NIDS to dynamically diagnose conditions of excessive traffic load and minimize their impact on its detection accuracy by choosing which packets should be dropped. Using selective packet discarding, the system selectively skips processing packets that are less likely to affect the correct operation of the detection engine as soon as possible, instead of letting the Operating System randomly drop arriving packets. This allows the NIDS to inspect a larger number of "useful" packets that are important to the detection process.

We observe that the first packets of a connection play a crucial role in the correct detection of a large class of attacks. For instance, signatures for threats like network service probes and reconnaissance attacks, brute force login attempts, protocol misbehaviour, and code-injection attacks, usually match packets that are among the first few hundred packets of a network flow. Moreover, the first control packets of a TCP connection are crucial to proper flow tracking and TCP stream reassembly, which are mandatory features of modern NIDSs [8, 16]. If any of the packets in the TCP three-way handshake is lost, the corresponding flow will not be considered established, and potential attack vectors in this flow may evade detection. On the other hand, very large flows usually correspond to file transfers, P2P traffic,


*EUROSEC* '10, Paris, France

or streaming media applications, which typically are not related to security threats. Inspecting all packets from such "heavy-hitters," which comprise a large percentage of the total traffic, usually does not contribute much to the detection accuracy of a NIDS.

We implemented selective packet discarding in the Snort NIDS [17] as a preprocessor that runs before the detection engine. We experimentally evaluated our technique using production traffic, mixed with real attacks that Snort can detect. Under overload conditions, the original Snort implementation misses a significant number of packets, resulting to a considerably lower number of alerts, with many of the labeled attacks in our trace passing undetected. In contrast, selective packet discarding significantly improves the detection accuracy of Snort under increased traffic conditions, allowing it to detect most of the attacks that would have otherwise been missed.

## 2. SELECTIVE PACKET DISCARDING

Ideally, a NIDS should be able to capture and inspect all network traffic passing through the monitored link. In highly loaded networks, this may not be possible due to the limited computational power of the monitoring sensor. For traffic speeds higher than a few hundred Mbit/s, the system cannot process all monitored traffic, which unavoidably leads to packet drops [18].

One way to offload the detection engine is to select a subset of the monitored traffic to be excluded from the NIDS processing using a capture filter during initialization [12]. However, events of excessive traffic load or bursty traffic can still occur. Given that under such conditions some packets will unavoidably get lost, we argue that it is better to proactively discard those packets that are less likely to affect the detection effectiveness of the system, instead of letting the OS drop packets at random.

In this section, we describe in detail the design and implementation of selective packet discarding. We first discuss *which* packets should be considered for discarding, and we propose a selection based on the position of packets in their flows. Then, we describe the performance measurements that the NIDS should perform periodically to monitor the system's load and decide *when* selective packet discarding should be triggered. Finally, we present an algorithm that dynamically estimates *how many* packets should be dropped according to the system performance measurements.

### 2.1 Flow-based Packet Selection

The starting point of our work is the observation that in a typical NIDS, some network packets play a more important role for the detection of a large class of threats than the rest of the traffic, i.e., without processing them, there is an increased probability to miss an attack. For example, inspecting the protocol interactions of commonly targeted services like RPC and NETBIOS seems more important than inspecting a large file transfer of a file-sharing application.

Probably the most widely used abstraction when referring to network traffic, besides the network packets themselves, is the *network flow*. A network flow comprises packets with the same protocol, source and destination IP addresses, and source and destination port numbers (same 5-tuple) and represents a connection between two hosts.

The first packets of a network connection are very important for the correct detection of a large class of attacks. Many types of threats like port scanning, service probes and OS fingerprinting, code-injection attacks, and brute force login attempts, require a new connection for each attempt, and the attack vector is usually present in the first few thousands KB of the flow. By contrast, very large streams usually correspond to file transfers, VoIP communication, or streaming media applications, which typically are not related to security threats. Very long flows usually comprise a large portion of the total traffic in an organization's network, and inspecting the packets towards the end of such flows does not contribute much to the detection accuracy of a NIDS.

Another reason for the increased importance of the first packets of a connection is the flow tracking and TCP stream reassembly functionality of modern NIDSs. The packets in the three-way TCP handshake, which are always the first packets in a TCP flow, are crucial for updating the state of a new flow as established, identifying the direction of each stream, and performing TCP stream reassembly. If a control packet is lost during the connection initialization phase, the corresponding flow will not be considered as established and possible attack vectors present in subsequent packets of this flow may evade detection.

We observe that 4627 of the 9276 rules in the default Snort rule set [1] contain the `flow:established` keyword, which defines that the detection engine should process the rest of the rule only if the packet belongs to an established TCP connection. If under high load conditions a packet of the three-way handshake does not reach Snort's flow tracking preprocessor, then the rules that rely on flow tracking will never match for that flow, and potential attacks will not be detected. Furthermore, attack vectors that span multiple packets in the beginning of the stream, such as the shellcode of a code injection attack or the URI of a malicious XSS HTTP request, are usually inspected after the original stream has been reassembled. If packets are being dropped randomly by the OS, the stream reassembly preprocessor may not receive a packet containing part of an attack, leaving the reassembled stream incomplete.

To verify our intuition based on the above observations, we analyzed traces of real attacks and extracted the actual position of the attack vector within the flow. We ran Snort using real traffic captured at the access link of an educational institution network, which triggered 1976 alerts from 78 different rules. We further augmented the trace with 120 traces containing real attacks captured in the wild [15], which Snort detects using the default rule set. We interspersed these traces in random offsets within the large trace, so that the resulting trace generates a total of 2252 alerts due to 92 attack signatures. Further details about the trace and the experimental environment are discussed in Section 3.1.

We slightly modified Snort to categorize packets into flows, and report the rank of the matching packet within its flow. Figure 1 shows the cumulative distribution of the matching packet position within the flow for the 2252 alerts in the trace. We observe that most of the alerts are triggered by the first few packets of a flow. For instance, 90% of the alerts were triggered within the first 30 packets of the flow, and only 3% of the alerts are triggered from packets coming after the 100th packet.

Flows usually follow a heavy tailed distribution on the Internet, i.e., the great majority of the flows have a quite small size, while only a very small subset has a very large size and is responsible for most of the total traffic volume [7]. Our

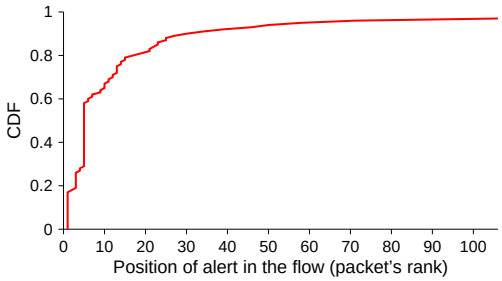


**Figure 1: Distribution of the positions of matching packets within their flows.**

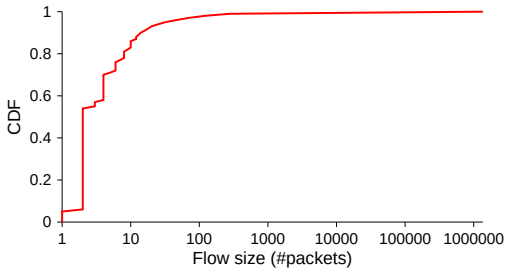


**Figure 2: Cumulative distribution of flow size**

trace also follows this property, as we can see in Figure 2, which shows the cumulative distribution of flow sizes. We can see that 86% of the flows contain up to 10 packets, while 97% of the flows have no more than 70 packets. The average flow size in the trace is 50.2 packets.

Based on these observations, we argue that NIDS under high load conditions would benefit from focusing on the processing of the first packets of each flow, and discarding the rest. Selecting the packets to be processed by setting a flow size limit seems promising, since it will affect a small percentage of flows, but will also exclude a large portion of the total traffic from processing. Dynamically setting the flow size limit according to NIDS's load is an important aspect of our approach, which we discuss in the rest of this section.

## 2.2 System Load Monitoring

Implementing selective packet discarding requires the forecasting of overload conditions that will probably lead to dropped packets before the kernel actually starts dropping them. Our system identifies overload conditions using three metrics: the occurrence of packet drops, CPU utilization approaching 100%, and a comparison of NIDS processing time and packet inter-arrival times.

Ideally, we would like to perform CPU measurements at per-packet granularity. However, this would incur a prohibitively high overhead. Instead, we measure the NIDS's CPU usage, processing time, and packet drops once every $N$ packets have been processed. We automatically choose $N$ based on the socket buffer size, to permit the timely detection of overload conditions before any packet drops are caused by the kernel, and based on the system's timing resolution in measuring accurately CPU time.

Every $N$ packets, the system examines whether any packets were dropped by the kernel in the elapsed period and measures the user, system and real time the NIDS spent while processing the group of $N$ packets. The CPU utilization is computed as $(user\,time + system\,time)/real\,time$. For the third metric, we compare the time $t$ required for processing the group of $N$ packets with the time interval $s$ during which these packets were observed on the network. If $t > s$, then this is an indication that the kernel will probably start dropping packets, if not already. Otherwise, if $t < s$, the kernel did not drop any packets in that interval.

Since we need to predict packet loss events before the CPU gets saturated, we set an upper threshold for CPU usage and processing time, above which selective packet discarding is triggered. When $t < s$ and CPU usage is relatively low, with no packet drop events during that period, the system decides whether more packets per flow should be processed or not based on a second lower threshold. These two thresholds should be close enough to allow for optimum resource usage and prevent CPU under-utilization. We also take into account the typical CPU load variation during short term intervals to avoid rapid oscillations, i.e, falling into a loop that would change very often the flow size limit up and down.

After running Snort using different traffic speeds and observing the correlation between the traffic load, CPU load, and packet drops, we set the upper threshold to 0.95 and the lower threshold to 0.8, i.e., 95% and 80% CPU utilization, respectively. Putting it all together, the NIDS triggers selective packet discarding if during the processing of the previous $N$ packets i) some packets were dropped by the kernel, or ii) $CPU\,utilization > 0.95$ and $t > 0.95s$. The condition for identifying an idle period is $CPU\,utilization < 0.8$ and $t < 0.8s$. Otherwise, the CPU utilization is within the desirable range and the flow size limits remain the same.

## 2.3 Flow Size Limit Adjustment Algorithm

Upon detection of an overload condition, the NIDS should back off and reduce the number of packets that it is going to process. First, we need to specify how many packets should be discarded, and then this number should be translated to the proper reduction of the per-flow cutoff limit. Ideally, the number of packets to be discarded should be such that it would allow the processing time for the packets in the following group to remain within the desirable range.

We need to reduce the processing time $t$ to become equal to $0.95s$, i.e., the number of packets to be discarded from the next group will correspond to processing time $t - (0.95s)$, that is $(t - 0.95s)N/t$ packets. In case the system observes packet drops in that interval, we should also consider it in our decision. Therefore, the amount of packets that will be discarded is the maximum of i) the number of packets dropped in that interval, and ii) $(t - 0.95s)N/t$ packets. If an idle period is detected, the NIDS should ramp up and process more packets. The NIDS can spend $(0.8s) - t$ more processing time in the next group of $N$ packets, which corresponds to $(0.8s - t)N/t$ number of packets.

At this point, we have the mechanisms for estimating the number of packets that the NIDS should discard in case of overload. However, our selection strategy is based on limiting the flow size, which requires setting an appropriate flow size threshold. The algorithm for deriving the new flow size limit for each interval is based on aggregated statistics the

NIDS gathers during the classification of arriving packets into flows. The flow classification engine keeps packet counters for predefined flow size ranges. Each position of the table indicates the number of packets that correspond to flows with size within the respective range.

In case of packet discarding, the algorithm descends the flow statistics table starting from the range of the current flow limit and counts the packets that will be discarded in each lower flow size range, until we reach the desirable number. The procedure for increasing the flow size limit is similar, by ascending the flow statistics table until the required number of packets is encountered. Then, the flow size limit is adapted accordingly.

## 2.4 Implementation in Snort

We have implemented our approach within the Snort [17] NIDS as a preprocessor configured to run before the detection engine and all other preprocessors. The preprocessor receives each packet immediately after Snort's Layer-4 packet decoding, looks up its corresponding flow through a hash table, and updates the size of the flow. Based on the flow size and the current cutoff limit, the preprocessor decides whether the packet should be discarded, or forwarded to the other Snort preprocessors and the core detection engine.

Furthermore, as we have discussed in Section 2.3, the preprocessor keeps the aggregate number of captured packets for predefined flow size ranges. Flows are closed either after a timeout of inactivity (set to 10 seconds in our experiments) or due to normal TCP connection termination after RST or FIN/ACK packets. It is important to precisely follow TCP connection terminations in order to prevent attackers to evade detection by closing and opening new TCP connections immediately. Thus, each new connection will be considered as new flow and its first packets will be always processed by Snort.

Finally, the flow size limit readjustment algorithm is activated every $N$ packets. The size of the interval, $N$, is automatically chosen based on the size of the socket buffer and the systems' resolution in measuring CPU time. After processing $N$ packets, the preprocessor reasons about potential overload conditions based on the performance measurements and adjusts the flow size limit accordingly.

# 3. EXPERIMENTAL EVALUATION

## 3.1 Experimental Environment

Our experimental environment consists of two PCs interconnected through a 10 Gbit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [20]. The traffic generation PC is equipped with an Intel Xeon 2.00 GHz CPU with 6 MB L2 cache, 2 GB RAM, and a 10 Gbit network interface. This setup allowed us to replay traffic traces with speeds up to 900 Mbit/s. By rewriting the source and destination MAC addresses, the traffic is sent to the second PC, which captures the traffic and inspects it using the original Snort, as well as our extended version with selective packet discarding. We modified Snort v2.8.3.2, used the latest official rule set [1] containing 9276 rules, and enabled all the default preprocessors as specified in its default configuration. The NIDS PC is equipped with an Intel Xeon 2.66 GHz CPU with 4 MB L2 cache, 2 GB RAM, and a 10 Gbit network interface. The socket buffer

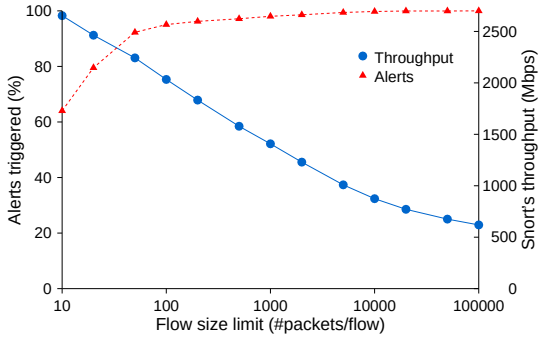


**Figure 3: Snort's throughput and detection accuracy as a function of the flow size limit.**

size was set to 6 MB in order to minimize packet drops due to short packet bursts.

For the evaluation we used a one-hour full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 flows, totalling more than 40 GB in size. Snort generates 1976 alerts from 78 different rules for this trace. Reasoning about whether these alerts are true positives or not would require manual inspection of each alert and the corresponding matching packets. Most of the matching rules are related to common threats such as probes for vulnerable web applications and database servers, old buffer overflow exploits, and protocol violations. There are also few alerts from rules that look for suspicious activity, such as `robots.txt` access or `HTTP 403 Forbidden` responses, which correspond to 49 and 263 alerts, respectively. Given the nature of the triggered alerts, we believe that most of them are true positives. However, based on our experience, and since we have not checked all alerts one by one, we speculate that some of them could be false positives. In order to strengthen our evaluation we augmented the trace with 120 short traces of real attacks, adding 276 alerts from 14 different rules which are definitively true positives.

## 3.2 Flow Cutoff Impact Analysis

In our first experiment, we explore the impact of imposing a limit in the number of packets of each flow that are going to be processed on Snort's processing throughput and detection accuracy. We modified our preprocessor to discard the packets of each flow after a certain flow size limit has been reached. We ran Snort using different flow cutoff values using the augmented network trace. Snort loads the trace for offline analysis, so there is no dynamic adaptation in the flow cutoff size—the same flow size limit is used for the whole duration of each run.

For each run, we measure Snort's processing throughput as the total trace size divided by the user plus system execution time. We repeat each measurement 10 times and report the average value of the throughput. For each run, the detection accuracy is defined as the percentage of alerts triggered for each different flow cutoff out of the total number of alerts (2252) in the trace. We are mostly interested in the detection of the 276 attacks that we have injected in the trace, since we know that the corresponding alerts are

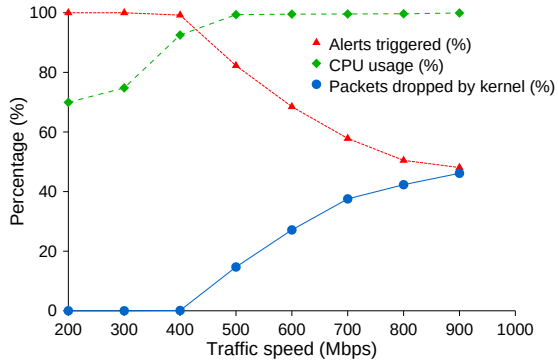


**Figure 4: Performance of unmodified Snort as a function of the monitored traffic speed.**

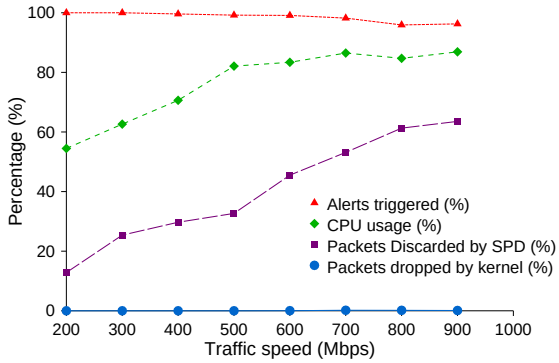


**Figure 5: Performance of Snort with selective packet discarding as a function of the monitored traffic speed.**

definitively true positives, but it is also desirable to observe as many of the rest of the alerts in the trace as possible.

Figure 3 presents Snort's throughput and detection accuracy when varying the number of processed packets per flow from 10 to 100,000. The unmodified version of Snort achieves a throughput of 560 Mbit/s. When enabling the preprocessor, as the number of inspected packets per flow decreases the throughput increases. For instance, for a flow cutoff limit of 1000 packets, Snort can process up to 1400 Mbit/s traffic, while with 100 packets per flow the processing throughput reaches 2 Gbit/s. When using larger flow cutoff sizes, the throughput approaches the unmodified Snort's throughput, e.g., with a limit of 50,000 packets the throughput drops to 675 Mbit/s, which still is a 20% improvement.

For flow limits higher than a few hundreds of packets, only a small percentage of alerts is missed. As already expected from Figure 1, alerts are triggered mostly due to packets that belong to the first few packets of a flow. For instance, processing up to 10,000 packets per flow results to 5 missed alerts out of the 2252 alerts in the trace, while at the same time the throughput increases 56%. For a cutoff size of 50,000 packets only one alert was missed. The 276 alerts due to the manually injected attacks are all triggered even for a cutoff limit as low as 20 packets per flow.

Even when inspecting just the first 100 packets of each flow, 95% of the alerts are still triggered. Considering the corresponding improvement in Snort's throughput, which is 3.62 times faster reaching up to 2033 Mbit/s, enabling selective packet discarding for traffic volumes higher that 560 Mbit/s seems promising. As we are going to see in the next section, under such conditions, the packet drops by kernel result a much higher number of missed alerts. When the monitored traffic throughput drops to normal and Snort is not high loaded, the selective packet discarding preprocessor will dynamically adapt the flow size limit as much as effectively disabling packet discarding at all.

## 3.3 Detection Accuracy under High Load

In this section, we evaluate the detection accuracy of unmodified and our extended Snort version under realistic conditions of increased load. Figure 4 shows the performance of original Snort when replaying traffic with speeds varying from 200 to 900 Mbit/s. For each traffic speed, we repeated the measurements 10 times and report the average of the percentage of triggered alerts, the CPU utilization of Snort, and the percentage of dropped packets by the OS. We see that for speeds higher than 500 Mbit/s, a significant percentage of packets is dropped by the kernel, ranging from 15% for 500 Mbit/s up to 46% for 900 Mbit/s traffic. When packets are dropped, the CPU utilization is always higher than 99%, since Snort cannot handle the high traffic volume.

The consequence of these drops is a significant reduction in the number of detected events. For a traffic speed of 500 Mbit/s, with just 15% of the packets being randomly dropped by the OS, Snort misses 18% of the alerts. When 46% of the packets are dropped, for 900 Mbit/s traffic, about half of the alerts are missed. Even for 400 Mbit/s traffic, a slight percentage of dropped packets (0.096%) causes 16 alerts to be missed (0.7%). Furthermore, among different runs for the same traffic speed, Snort generates different sets of alerts, indicative of the non deterministic results that random packet drops induce. Moreover, the 276 real alerts we injected are lost with the same probability as all other alerts in the trace. For instance, for 500 Mbit/s traffic, Snort identified 223 out of the 276 attacks. For 900 Mbit/s, just 55% of these alerts were successfully detected. These results demonstrate that Snort's detection accuracy degrades significantly under conditions of excessive traffic load.

Figure 5 shows the performance of Snort with selective packet discarding enabled. A first observation is that the number of packets dropped by the kernel is negligible. There are no packet drops for traffic speeds up to 600 Mbit/s, while there is just a 0.098% of dropped packets for 900 Mbit/s. We also notice that for high traffic speeds, the CPU utilization remains within the desirable range imposed by the 0.8 lower and 0.95 upper thresholds. Figure 5 also shows the percentage of packets that are selectively discarded. As we expected, the percentage of discarded packets increases according to the traffic speed. By discarding the desirable amount of packets according to the traffic load, Snort controls the CPU utilization and keeps it constantly within the desirable range.

The number of selectively discarded packets is larger than the number of dropped packets by the kernel in unmodified Snort for the same speeds. There are two explanations for this outcome. First, the selective packet discarding algorithm is purposely quite aggressive in discarding packets in order to proactively prevent packet drops from the ker-

nel. Thus, the preprocessor tends to discard more packets from the end of the flows and benefit from preventing uncontrolled random packet drops from the kernel. Second, in unmodified Snort, packets are dropped in kernel level, before they are copied to user level. With selective packet discarding, all packets are first delivered in user space and then are discarded by the Snort preprocessor, which results to a higher number of discarded packets. However, even with eventually less inspected packets, selective packet discarding allows Snort to achieve a much better detection accuracy, as discussed below. Moreover, the number of flows affected by selective packet discarding is just 0.42% of the total flows for the highest traffic speed of 900 Mbit/s, and even smaller for lower traffic speeds. In contrast, random packet drops by the kernel affect a significantly higher number of flows.

Finally, Figure 5 shows the significant improvement in detection accuracy by enabling selective packet discarding. For all traffic rates, even for 900 Mbit/s, Snort reports almost all of the alerts that exist in the monitored traffic. For 500 Mbit/s traffic, our modified Snort reports 2234 out of the 2252 alerts (99.2%), which is an improvement of 20% over unmodified Snort. The percentage of triggered alerts remains almost constant as the traffic speed increases, falling slightly to 96.3% for 900 Mbit/s traffic, missing just 84 events. We observe that for all traffic speeds, our modified Snort detects all the 276 real attacks that we manually inserted, suggesting that selective packet discarding indeed tends to improve the detection accuracy of real attacks. Of course, the vast majority of the alerts due to events in the original trace are still triggered.

# 4. RELATED WORK

The requirements for more complex per-packet inspection and the constant increase in network speeds have motivated numerous works for improving the performance of NIDSs. To speed-up the inspection process, many NIDS implementations use specialized hardware like content addressable memory [23], FPGAs [2], network processors [4], and graphics processing units [22]. To cope with high traffic volumes, other approaches propose to distribute the load across multiple machines instead of using a single sensor [9, 21], or to use multi-core processors for parallel inspection [14]. These solutions offer almost linear processing throughput improvement, but with the additional cost of buying specialized hardware or multi-processor machines. Overloads are still possible in such systems in case of traffic bursts that exceed the NIDS processing throughput, or if one of the individual sensors of a NIDS cluster is overloaded. Furthermore, attackers may intentionally overload a NIDS to degrade its performance and increase their chances to evade detection [16].

Lee et al. [10], propose to dynamically reconfigure the NIDS based on the current run-time conditions, in a work closely related to our approach. In contrast, our system selectively discards some packets with minimum impact to detection accuracy without changing the NIDS configuration.

Another related approach by Dreger et al. [5] deals with packet drops due to overloads using precompiled sets of filters which the NIDS enables and disables depending on the workload. The main difference from our technique is that it relies on the NIDS operator to statically define an ordering of filters. A more recent work from the same authors [6] presents a model for monitoring the resource usage of a NIDS, and then predicting its resource consumption. While this approach can help the NIDS operator to find a suitable configuration, our approach allows a NIDS to adapt its performance under high load even if no configuration can prevent overloads.

Load shedding is proposed as a defence to overload attacks in the Bro NIDS [13]. However, the discarding strategy is not discussed, so the NIDS operator is responsible to define one. In our work we propose a new subset of traffic that should be discarded, based on the position of the packet within its corresponding flow. A load shedding [3] technique has also been proposed in the CoMo passive monitoring infrastructure. Using an on-line prediction model for the query resource requirements, the monitoring system sheds load under conditions of excessive traffic using uniform packet and flow sampling.

Similarly to our work, Time Machine [11] uses a per-flow cutoff to reduce the number of packets that are stored on disk for retrospective analysis. In contrast, our work uses the per-flow cutoff limit for real time intrusion detection. Time Machine uses user-configured static cutoff values for different packet classes, while our modified NIDS adaptively selects the optimum cutoff value based on real-time measurements.

# 5. CONCLUSION

Events of excessive network traffic load are a common fact that affects the performance of NIDSs. Under conditions of heavy traffic load or sudden traffic bursts, the processing throughput of the system cannot cope with the amount of traffic that needs to be inspected, and the OS unavoidably drops excess arriving packets at random.

In this paper, we present selective packet discarding, a best effort approach that gracefully reduces the amount of traffic that reach the detection engine of the NIDS by selectively discarding packets that are less likely to affect its detection accuracy. We have implemented selective packet discarding in the Snort NIDS as a preprocessor that constantly measures performance aspects of the system in order to detect overload conditions and dynamically adjusts the number of packets that needs to be discarded. This is achieved by setting a cutoff limit to the number of packets to be inspected for each network flow.

A concern that arises when using selective packet discarding is that a sophisticated attacker could exploit the flow size limit and evade detection by filling the stream with benign requests and then send the actual attack vector after the flow cutoff limit has been reached. Although such an attack may be feasible for protocols like HTTP, which allows multiple requests to be sent through the same connection, other services terminate the connection after the end of each transaction, especially in case of protocol violations or failed requests. Furthermore, for protocols that support persistent connections, such repetitive behaviour can be detectable by following the protocol's request/response semantics. Without selective packet discarding, an attacker can evade detection from an overloaded NIDS by repeating the attack multiple times—depending on the traffic load, after a certain number of attempts the attack will go undetected. Selective packet discarding makes such overload attacks harder to achieve.

Our experimental evaluation with real-world traffic and labeled attacks demonstrates that selective packet discarding

improves significantly the detection accuracy of Snort under increased traffic load conditions, allowing it to detect most of the attacks that would have otherwise been undetected.

## 6. ACKNOWLEDGMENTS

A. Papadogiannakis, M. Polychronakis and Evangelos P. Markatos are also with the University of Crete.

## 7. REFERENCES

[1] Sourcefire vulnerability research team (vrt). http://www.snort.org/vrt/.

[2] M. Attig and J. Lockwood. A framework for rule processing in reconfigurable network systems. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pages 225–234, Washington, DC, USA, 2005.

[3] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load shedding in network monitoring applications. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*, Berkeley, CA, USA, 2007.

[4] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. SafeCard: a Gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Hamburg, Germany, September 2006.

[5] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 2–11, 2004.

[6] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 135–154, 2008.

[7] W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Global Telecommunications Conference*, 1999.

[8] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

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

[10] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 252–273, 2002.

[11] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *SIGCOMM '08: Proceedings of the 2008 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 183–194, New York, NY, USA, August 2008. ACM Press.

[12] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–270, January 1993.

[13] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1998.

[14] V. Paxson, R. Sommer, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *Proceedings of the IEEE Sarnoff Symposium*, May 2007.

[15] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

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

[17] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999.

[18] L. Schaelicke, T. Slabach, B. J. Moore, and C. Freeland. Characterizing the performance of network intrusion detection sensors. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 155–172, 2003.

[19] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[20] A. Turner. Tcpreplay. http://tcpreplay.synfin.net/trac/.

[21] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[22] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] S. Yusuf and W. Luk. Bitwise optimised CAM for network intrusion detection systems. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 444–449, 2005.