High Performance Intrusion Detection Using HTTP-Based Payload Aggregation

Felix Erlacher and Falko Dressler
Heinz Nixdorf Institute and Department of Computer Science
Paderborn University, Germany
{erlacher,dressler}@ccs-labs.org

***Abstract*—Signature-based Network Intrusion Detection Systems (NIDS) are an integral part of modern network security solutions. They help to detect and prevent network attacks and intrusions. However, they show critical performance problems in today's high speed networks. Filters have been proposed to reduce the amount of traffic to be analyzed by a NIDS, yet, such filters need to be very carefully designed in order not to miss relevant data. We address this problem by proposing a novel concept for filtering taking into account the pipelining architecture of modern web traffic. Our concept, which we named HTTP-based Payload Aggregation (HPA), is able to retain the first N bytes of the basic Protocol Data Unit (PDU) of an application layer protocol and discard the rest, arguing that the retained payload portion contains almost all relevant data for intrusion detection. We demonstrate the feasibility of our approach focusing on HTTP traffic as the most prominent protocol in many Internet applications. The idea is, thus, to capture the first N bytes of every pipelined session and forward this data to a NIDS. In our evaluation, we show that for the used traces we still detect more than 97% of the events with only 2.5% of the network traffic to be analyzed. We achieve an increase in packet throughput of up to 44 in our experiments.**

I. Introduction

Network Intrusion Detection Systems (NIDS) are an important tool for network operators to detect and defend against attacks but also to enforce usage policies to avoid internal misuse [1]. We can distinguish between anomaly-based systems that build a traffic model of 'normal' network traffic and then detect deviations of this model [2], and signature-based or rule-based systems that operate on a pre-defined rule set of known attacks and incidents [3], [4]. We concentrate on signature-based systems such as Snort [3] or Bro [4], which allow a very detailed description of attacks using also Deep Packet Inspection (DPI) methods and, therefore, offer a very high detection rate at the cost of comparably low performance and, thus, are limited to lower packet throughput.

The main steps of a signature-based NIDS are the following: First, the incoming traffic goes through a predefined set of preprocessors. Here, if necessary, packets are reassembled (fragmentation, TCP reassembly) and checked for validity (e.g., TCP checksum). Afterwards, a detection engine applies a ruleset to the received data. The majority of state-of-the-art Snort rules contain patterns that are matched against the payload. These patterns range from selected bytes to a concatenation of Regular Expressions (RegExes). To narrow down the number of packets that a pattern has to be matched to, a rule contains also source, target, direction, or protocol filters that are applied first. The performance of a NIDS mainly depends on the number of rules [5]. Thus, in practical applications, the ruleset is customized for the domain-specific use case. Nevertheless, a high number of rules remain, and further reducing the number of rules would elevate the risk of not detecting a possible intrusion. Various approaches tried to speed up NIDS, e.g., by parallelizing the operation [6]–[8] or simply by reducing the input data to be analyzed [9]–[11].

We focus on the latter approach to reduce the amount of data by filtering the network traffic that the NIDS has to analyze. In particular, we aim at filtering the payload of the HTTP protocol, which became the basis for the majority of current Internet applications. Recent studies show, that the amount of Hypertext Transfer Protocol (HTTP) in Internet traffic has increased significantly, accounting for more than $50\,\%$ of the overall traffic volume [12], [13]. This is also reflected in the high number of HTTP related rules for the popular NIDS Snort. To speed up web applications and minimize delay, HTTP has evolved from a simple request/response protocol to a pipelined operation. This is especially relevant as the new HTTP/2 protocol shows that future application protocols will be much more interleaved [14], [15]. Filtering solutions such as Time Machine [9] or Dialog-based Payload Aggregation (DPA) [11] cannot cope with this change as they are based on a per flow operation, a flow being defined as a single TCP connection. Pipelined HTTP, however, sends multiple request/response messages over the same TCP connection.

In this paper, we propose HTTP-based Payload Aggregation (HPA), a filtering technique that is able to deeply investigate the full HTTP protocol including pipelining in order to correctly identify HTTP request-response pairs, which are then fed to a NIDS. In order to substantially reduce the network traffic, HPA follows the Time Machine / DPA concept to only forward the first N bytes of every HTTP message. Empirical evidence shows that this part of the payload is most relevant for attacks while the remainder of the payload is mostly irrelevant. In order to also support encrypted HTTP traffic, TLS interception proxies can be used [16], which is standard for most commercial firewall systems.

Our main contributions can be summarized as follows:

- We present HPA, a novel filtering technique for speeding up signature-based NIDS;

© 2017, Felix Erlacher. Under license to IEEE.
DOI 10.1109/LCN.2017.18

IEEE computer society

- our concept fully integrates with HTTP pipelining as proposed for HTTP/1.1 and HTTP/2;
- we fully evaluate our approach using a typical ruleset for Snort and show that it clearly outperforms both Snort as well as Snort in combination with previous filter techniques.

II. Related Work

Several efforts have been made to facilitate and speed up signature-based NIDS. The biggest performance bottleneck in these systems is the pattern matching [5]. As RegExes are an integral part of such systems, there have been approaches to speed up RegEx matching with specialized hardware [17]. Yet, the majority of NIDS are deployed using off-the-shelve hardware and the possible speed-up is rather limited.

Other approaches focus on using multiple instances of NIDS in parallel. For example, the possibility of modern switches to distribute incoming traffic according to Quality of Service (QoS) criteria on multiple outbound interfaces has been explored [18]. Multiple NIDS instances listening to these interfaces work on a fraction of the incoming traffic. The main problem remaining is the interdependency of network flows, which requires massive maintenance between these instances.

Most promising are results from solutions that reduce the traffic data that has to be processed by a NIDS. Random sampling has been explored in the context of the packet sampling IETF group [19]. It helps reducing the load but, unfortunately, also significantly reduces the detection quality. As a result, methods for intelligent filtering have been proposed, which try to exploit the heavy-tailed nature of Internet traffic [9]–[11], [20], [21]. They all have in common that they use the first N bytes of a TCP connection, arguing that it contains the majority of attack signatures. After all, such filtering approaches can also be used in combination with hardware based RegEx matching or parallel NIDS operation.

A very generic approach is to use the first 500 kB of every flow [21]. It has been shown that 99 % of the threats are located in this portion of the network traffic. This is very similar to early filtering algorithms such as Time Machine [9] or Front Payload Aggregation (FPA) [10]. TCP sequence numbers were used to aggregate the first N bytes of payload in both TCP flow directions. While this works well for rather simple protocols like SMTP, it fails for protocols that use one connection in an interleaved way for control commands and transfer of data.

A first step towards handling more complex application protocol behavior focusing on HTTP/1.0 has been addressed in [11]. The presented DPA approach collects the first N bytes after every TCP direction change, i.e., keeping track of multiple HTTP request / response pairs in the same transport layer connection. This helps increasing the detection rate: results show that DPA can filter about 96 % of the data while retaining 89 % of the events reported by Snort. The main weakness of this approach is that DPA cannot deal with modern pipelining features in HTTP/1.1 or HTTP/2 [14], [15].

We advance the concept of filtering in this paper by also considering pipelining as a core feature of modern Internet protocols. Because HTTP is the most widespread application layer protocol, we apply our concept first to HTTP, calling it HPA. Throughout this paper we apply the taxonomy and methods suggested in [22].

III. HPA: HTTP-Based Payload Aggregation

In this section, we introduce and explain our new filtering approach HPA. The goal is to greatly reduce the data that has to be analyzed by a NIDS while retaining all security relevant events. As motivated in Section II, we focus on HTTP as a means of transport for modern Internet applications. Following the findings in [9]–[11], the underlying assumption is that the most relevant data for successful intrusion detection is stored close to the start of an application layer Protocol Data Unit (PDU). This is mainly because important information is stored in the (application layer) protocol header, but also the protocol payload contains important information in its initial portion.

A. Rule Analysis

An analysis of the current rulesets for the NIDS Snort underlines that most relevant information can be found at the beginning of an HTTP message. We downloaded the most current rules from three sources (versions of 2017-01-15):

- All Snort rules (snapshot 2990) as provided to Snort.org subscribers;
- the community ruleset from Snort.org; and
- all rules from the Emerging Threats ruleset[1].

We then filtered all rules so that only active (uncommented) rules remained that were related to HTTP:

- 67 % (18363) of all active rules (27375) are related to HTTP (using http_* content restrictors[2], using the "service http" metadata tag or using the $HTTP_SERVER or $HTTP_PORTS variable);
- 66 % (12141) of these rules use one of the http_* content restrictors; among these
- 94 % (11468) of the rules apply the pattern to a field in the HTTP header and thus the beginning of the HTTP message.

Overall, about 62 % of the HTTP rules (42 % of all rules) apply a content search to the beginning of the HTTP message.

B. Concept

Because we are interested in the protocol details and, thus, the structure and content of the PDUs of the application protocol, simply following the transport protocol and using its sequence numbers is not longer sufficient. In the case of HTTP, for example, pipelined messages can not be detected by simply looking at sequence numbers or TCP direction changes. Thus, a stateful protocol parser is necessary that can keep track of all incoming flows and export flows when they are in an exportable state. This parser needs further to be robust to cope with packet loss and, in the case of HTTP, to be able to

[1] http://doc.emergingthreats.net/bin/view/Main/AllRulesets

[2] Content restrictors in Snort rules reduce the pattern search in the content (payload) to only the portion defined in the restrictor

> GET / HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..
> GET /js/script HTTP/1.1 Host: ccs-labs.org
> GET /img/nice.jpg HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 304 Not Modified Server: nginx Date: Fri, 13 May 2016 ..
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..

(a) Original HTTP connection

N
> GET / HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..
> GET /js/script HTTP/1.1 Host: ccs-labs.org
> GET /img/nice.jpg HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 304 Not Modified Server: nginx Date: Fri, 13 May 2016 ..
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..

(b) Filtering using FPA

N
> GET / HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..
> GET /js/script HTTP/1.1 Host: ccs-labs.org
> GET /img/nice.jpg HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 304 Not Modified Server: nginx Date: Fri, 13 May 2016 ..
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..

(c) Filtering using DPA

N
> GET / HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..
> GET /js/script HTTP/1.1 Host: ccs-labs.org
> GET /img/nice.jpg HTTP/1.1 Host: ccs-labs.org
< HTTP/1.1 304 Not Modified Server: nginx Date: Fri, 13 May 2016 ..
< HTTP/1.1 200 OK Server: nginx Content-Encoding: gzip Content-L ..

(d) Filtering using HPA

Fig. 1. Example HTTP connection annotated with the different filtering techniques FPA, DPA, and HPA

handle non-standard implementations by current web servers and browsers. After collecting all HTTP messages, this data needs to be exported in the form of packets for a seamless integration with NIDS such as Snort. Snort is assuming it is receiving the packets from the wire, so some steps like TCP reassembly and HTTP parsing will be done twice, once in the HPA framework and once in Snort. This can, of course, be improved for practical use but is sufficient to evaluate the conceptual approach of HPA.

Figures 1a to 1d show a comparison of the ability to select data from an original HTTP connection using Time Machine / FPA, DPA, and our new approach HPA. The baseline HTTP connection contains two pipelined GET requests and the cor-

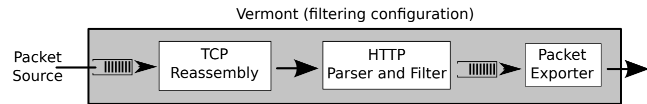


Fig. 2. Architecture of the Vermont monitoring toolkit with HPA integration

responding responses following each other (cf. Figure 1a). The Time Machine / FPA solution simply retains the first N bytes of every direction of the underlying TCP connection, thus, it misses all but the first two HTTP messages (cf. Figure 1b). DPA is able to go one step further and retains the first N bytes after every direction change (cf. Figure 1c). As can be seen, it can not deal with pipelined HTTP messages that follow each other without a direction change in between.

Figure 1d shows our new HPA filtering algorithm. Because of its HTTP capabilities it goes beyond the capabilities of DPA and can export the first N bytes of every single HTTP message. In our case, it detects the pipelined GET requests as well as the corresponding responses and exports them as single messages, thus, allowing a NIDS to analyze the first N bytes of all HTTP messages.

C. Implementation

For implementing our new algorithm, we use the network monitoring toolkit Vermont [23]. This is a rather straightforward process due to its open and modular architecture. We also make our system publicly available as Open Source.[3]

The underlying basis for HPA is the HTTP parser for Vermont to aggregate HTTP related Information Elements (IEs) into IPFIX flows [24], [25]. We use the capability of the HTTP parser to filter out only the first N bytes of every HTTP message (request and response) and then export these bytes as one packet per message to the NIDS Snort.

The relevant modules of Vermont are shown in Figure 2. First, the packets are collected from a TAP interface on the wire (or from a stored pcap trace). Next, the TCP stream is reassembled; this is necessary because the HTTP parser relies on receiving an ordered stream of TCP segments.

Now, the HTTP parser walks through the payload and 'on-the-fly' collects data related to open HTTP connections. Instead of waiting for an entire HTTP message to complete, our parser operates on the payload of multiple open HTTP messages. While analyzing the payload of new TCP segments, it undergoes a series of state changes. For example, if a packet contains only the HTTP request method and Universal Resource Identifier (URI), instead of waiting until the next portion arrives, the state for this HTTP message can be saved and parsing can be continued at some point in the future. If remaining packets got lost, the parsed message so far can be exported after a configurable timeout. This way the parser saves memory and can easily process a large number of multiple HTTP messages concurrently. Only the first N bytes of every HTTP message are aggregated; the size of N can be configured to the needs of the scenario.

[3]https://github.com/felixe/ccsVermont, branch: http-aggregation

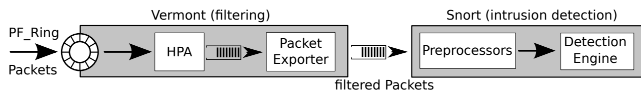


Fig. 3. Configuration of Vermont / HPA filtering network traffic for subsequent analysis using the NIDS Snort

In the last step, the HTTP messages are exported as a single packet per HTTP message to a connected NIDS.

IV. Evaluation

To evaluate the effectiveness of our HPA approach, we use real-world traffic traces and process these using the NIDS Snort. We choose Snort because it is the most widespread signature-based NIDS and has the largest source of signatures. The configuration files and other resources used for the evaluation part of this work are available online[4]. We first show that, while filtering out most of the network traffic, the attack detection accuracy of Snort is only influenced marginally by HPA; it can still detect almost all events. We further compare our approach to FPA and DPA, both of which have been integrated with Vermont as well. Finally, we evaluate the processing capacity of our approach with Snort in comparison to FPA and DPA.

A. Traces and Configuration for Functional Evaluation

Our first objective is to assess the attack detection accuracy of HPA. For this, we need to show that the events reported by Snort are not negatively influenced by our filtering method. The main configuration is depicted in Figure 3.

To represent typical HTTP traffic, we use a trace that was created by capturing the traffic going through an HTTP proxy used by a scientific work group for one week (from now on called proxy trace). We additionally inserted HTTP traffic containing five well known attacks into this trace. The related patterns of the rules triggering these attacks are rather complex. Four of these patterns are located in an HTTP response and one is in an HTTP request. All five were registered as Common Vulnerability and Exposures in 2016 and have a relatively high Common Vulnerability Scoring System score. We inserted two versions of each attack: The first version only contains one HTTP request and one response, whereas the second is a pipelined version of the attack. All five attacks were inserted 22 times (21 pipelined version, 1 with only one request and response) per attack, resulting in 110 attacks in total. For the pipelined version of the four handcrafted attacks located in a response, we requested a file with a size of $1024\,\mathrm{B}$ before the malicious response, resulting in two pipelined responses with the $1024\,\mathrm{B}$ response starting first. For the pipelined version of the handcrafted attack located in a request we pipelined an HTTP request with a header of $520\,\mathrm{B}$ before the request with the attack. In total the trace contains more than 2.2 million packets divided among 2996 unique IP Flows (IP source/destination pairs). The average packet size is $825\,\mathrm{B}$. According to the taxonomy in [22], this constitutes a typical `real-world mixed` workload trace.

For all tests, we used Snort version 2.9.9.0 in IDS mode with the default snort.conf configuration file, which is shipped with the installation archive. The only changes made to the Snort configuration are the increase of queue sizes of detection engine (max_queue_events to 1000, max_queue to 1000, log to 1000) and the max_queued_bytes for the stream5_tcp preprocessor to $1.5\,\mathrm{MB}$. If the number of events per packet/message exceeds the queue size, they are not reported anymore by Snort. As the size of the packets/messages changes between the tested filtering methods, the queue changes were necessary to be able to realistically compare the outcome of the different tests. To avoid Snort skipping packets with checksum errors, we turned this behavior off. As signature databases, we relied on the three rulesets described in Section III-A. Again, we only used the 18363 active rules related to HTTP.

B. Functional Evaluation

For a first experiment, we fed the network trace from a file instead of reading via a Network Interface Controller (NIC) in order to evaluate the functional approach. Table I shows the number of events that have been triggered by Snort. The first column shows the rule SID, which triggered the event of the corresponding line. Rule SIDs are used to uniquely identify Snort signatures. The second column shows the results if the proxy trace is processed directly using Snort without any filtering. All other columns show the number of events, if the trace is read by Vermont and filtered with FPA, DPA, or our novel HPA technique. The top row shows the numbers of bytes N that have been retained per filtering method.

Overall, HPA detects most events even when configured for a comparably small N of $500\,\mathrm{B}$. It also outperforms DPA with a larger N. This is particularly dominant when looking at rule SID 2013504. Our data streams contain up to 10 pipelined requests, so that FPA and even DPA miss some of the patterns. This is not the case for HPA. In fact, looking at the amount of data to be processed by Snort, HPA shows a better detection rate even though producing less data ($47\,\mathrm{MB}$ vs. $170\,\mathrm{MB}$ for DPA vs. $1878\,\mathrm{MB}$ for unfiltered traffic). For the entire trace, our new filtering algorithm HPA detects $97.4\,\%$ of the events when analyzing only $2.5\,\%$ of the payload. This shows that for HTTP with HPA much less payload has to be exported to detect the same number of results than with legacy filtering methods. Because the filtered traffic only contains the first N bytes of traffic this is in line with the heavy-tailed nature of Internet traffic in general and HTTP traffic in particular.

We further see that HPA triggered even more events for the first rule compared to Snort reading the unfiltered trace. Investigating the specific streams reveals that Snort does not analyze TCP streams if they are terminated with a TCP RST. When exporting data, we omit all TCP control flags and, thus, have Snort analyze the full packet stream. If the RST packets are removed from the network trace and then fed to Snort also Snort shows the same number.

[4]http://www.ccs-labs.org/~erlacher/resources/

TABLE I
NUMBER OF SNORT EVENTS WITH AND WITHOUT FILTERING USING THE PROXY TRACE

		N=2000 B			N=1000 B			N=500 B		
Rule SID	Events without filtering	HPA	DPA	FPA	HPA	DPA	FPA	HPA	DPA	FPA
2013504	2035	2106	2041	937	2106	1497	519	2106	1022	272
40360	272	272	272	272	272	272	272	272	272	272
2012810	154	154	154	154	154	154	154	154	154	154
2015561	134	1	1	1	0	0	0	0	0	0
2101201	44	44	44	44	44	44	44	44	44	44
2001595	30	30	30	30	30	30	30	30	30	30
2014170	27	27	27	27	27	27	27	27	27	27
2101350	17	17	17	17	17	17	17	17	17	17
2013031	10	10	10	10	10	10	10	10	10	10
2018959	3	3	3	3	3	3	3	0	0	0
2016846	3	3	3	3	3	3	3	3	3	3
2012708	3	3	3	3	3	3	3	3	3	3
2011037	3	3	3	3	3	3	3	2	2	2
40158	1	0	0	0	0	0	0	0	0	0
2015707	1	0	0	0	0	0	0	0	0	0
2011507	1	0	0	0	0	0	0	0	0	0
Analyzed data in kB	1878836	172601	170287	163725	89218	88027	84627	47527	46889	45084
in %	100	9.9	9.1	8.7	4.7	4.6	4.5	2.5	2.5	2.4

TABLE II
NUMBER OF SNORT EVENTS FOR HANDCRAFTED ATTACKS, WITH AND WITHOUT FILTERING USING THE PROXY TRACE

		N=2000 B			N=1000 B			N=500 B		
Rule SID	Events without filtering	HPA	DPA	FPA	HPA	DPA	FPA	HPA	DPA	FPA
40886	22	22	22	22	22	1	1	22	1	1
40778	22	22	22	22	22	1	1	22	1	1
37135	22	22	22	22	22	1	1	22	1	1
2023568	22	22	22	22	22	1	1	22	1	1
2022848	22	22	22	22	22	22	22	22	1	1

A drawback of the selection of N can be seen for rule SID 2015561. Only Snort reading unfiltered traffic detects 134, as soon as traffic is filtered only 1 event is triggered. These events are found in only 5 TCP streams, and in only one of those TCP streams the pattern is located after 1050 B and thus in the retained portion of the filtered traffic. In the other 4 streams this pattern only appears after 9000 B or more. The same drawback applies to a smaller degree also to rule SID 2018959 and the last four rule SIDs in Table I.

Table II shows the results for our handcrafted events, which were not shown in the previous table. As explained before these attacks were injected two times. First, one event is inserted as single request/response and, secondly, 21 events are inserted as pipelined HTTP messages. The event patterns in the pipelined versions of the attack can only be found if more than 1500 B (HTTP header + 1024 B message payload) are retained by FPA or DPA. HPA finds all events in the pipelined messages. As every HTTP message is presented to Snort as a single packet, all of these pipelined attacks can be found even if only the first 500 B of every HTTP message are retained. This shows that HPA is particularly suited for interleaved messages such as in HTTP pipelining.

C. *Configuration for Performance Tests*

Obviously, the additional filtering must be significantly faster than a full analysis of the network traffic by an NIDS. We assessed the overall system performance using a realistic scenario by replaying the previously used proxy trace between two workstations (Linux 3.2.0, i5-4440 CPU, 32 GB RAM), which are directly connected over a 10 Gbit/s Ethernet network (Intel 82599 NICs). The goal was to measure the workload processing capacity (in our case packet throughput) and to assess the influence of our filtering system. We adapted the proxy trace used above for the performance tests by repeating the trace 15 times. We changed the IP addresses after every repetition using the tool `tcprewrite`. Because of the deterministic way that tcprewrite changes IP addresses, TCP sessions between two hosts are maintained. The result is a trace with 33 million packets and 28 GB of data. To replay the network trace, we used the program `pfsend` from the PF_Ring program suite [26], which is more accurate than

the well-known `tcpreplay`. The main difference to the functional tests is that we are now capturing the traffic directly from the NIC (using the PF_Ring library [26]) instead of a local trace file. This means that if one of the filtering or analyzing stages cannot cope with the packet rate, it will tailback packets resulting in the previous stages to be blocked and finally packets to be dropped at the NIC.

To cope with packet bursts, we introduced buffers between the single stages. In Vermont, we defined a queue between the packet capturing engine and the TCP reassembly engine of 300 000 packets and between the HTTP parser and the packet exporter a queue of 300 000 flows. The FIFO buffer between Vermont and Snort is 128 MB, which, at a packet size of 2000 B (assuming N to be set to 2000 B) corresponds to roughly 65 000 packets.

Because Snort is a single threaded application, we set its CPU affinity to one dedicated CPU core and configured Vermont to use the other cores. This avoids context switches and, thus, improves the performance of Snort [27].

While replaying we continuously increased the speed from 0.05 Gbit/s (7500 pps) to 6.2 Gbit/s (917 000 pps) when testing Snort without filtering, and from 1.2 Gbit/s (178 000 pps) to 10.0 Gbit/s (1.5 million pps) for the combination of Vermont filtering the traffic and Snort doing intrusion detection. We repeated the experiment 10 times for every measurement point.

D. *Network Performance*

Figure 4 shows the averaged results of the 10 runs. The graphs compare the packet drop rates and detected events of Snort (reading unfiltered traffic) to the use of FPA, DPA, and our new filtering method HPA. The three subfigures show the results for three different sizes of N. The data for Snort is the same in all three graphs and thus not influenced by the size of N. The black lines show the number of dropped packets and the orange/grey lines show the number of detected events.

If packets get dropped, events that relate to rule matches in these packets cannot be reported by Snort. Snort starts to drop packets because of system overload at about 15 000 pps. At this rate, the percentage of detected events drops. Already at a rate of 230 000 pps the detection rate drops below 10 %. This throughput is higher than reported by other studies [27], [28], which could be due to a slightly more modern computer system as well as thanks to the PF_Ring NIC device driver.

When filtering the traffic and for zero packet drop, the magnitude of detected events is in line with the outcome for the functional tests shown in Table I and Table II: HPA detects most events, closely followed by DPA in case of N=2000 B. For the same N, FPA only reports about 60 % of the events. With a decreasing N, FPA and DPA show a worse performance. Instead, our novel HPA approach is able to still detect almost 100 % of the events, even for N=500 B.

When comparing the peak performance of the filtering algorithms to Snort, it can be seen that while Snort on unfiltered traffic has to drop packets at 15 000 pps, this performance increases with the use of filtering techniques to about 180 000 pps (this corresponds to about 1.2 Gbit/s) for N=2000 B and even 670 000 pps (4.4 Gbit/s) for N=500 B for our new HPA technique. This corresponds to a speedup of more than a factor of 12 for N=2000 B and 44 for N=500 B. FPA and DPA perform similarly good, being able to achieve higher data rates at the cost of more missed events. Similarly to the Snort-only experiment, we observe higher data rates compared to the data documented in the original papers for FPA and DPA [10], [11]. This is again due to a slightly more modern computer system as well as thanks to the use of the new PF_Ring packet capturing library.

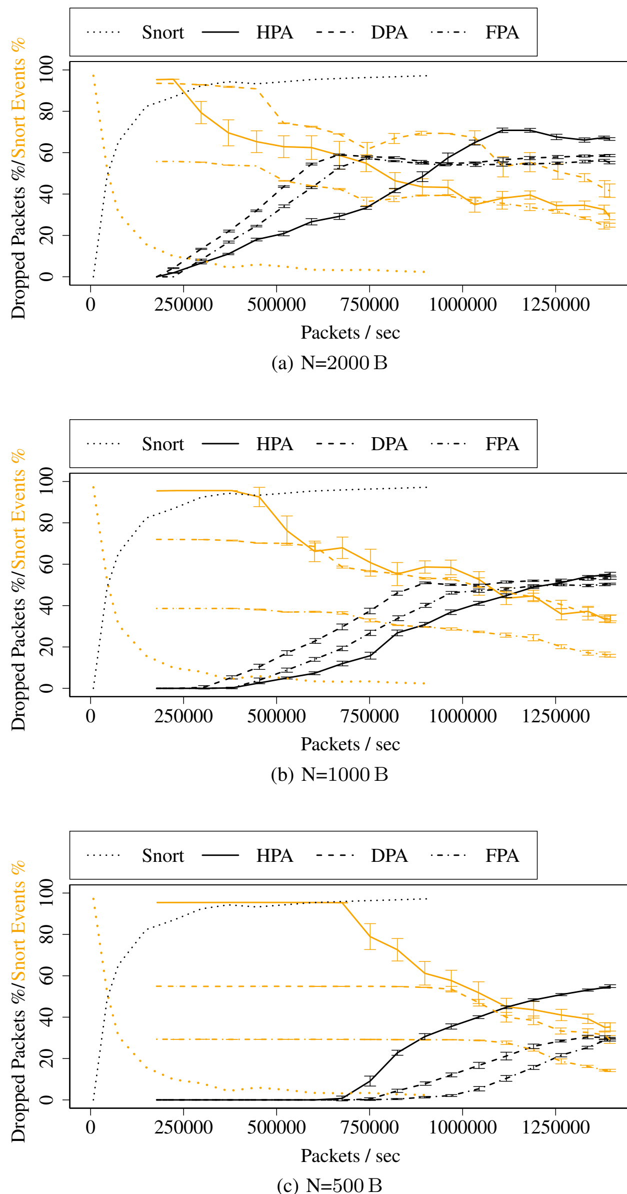
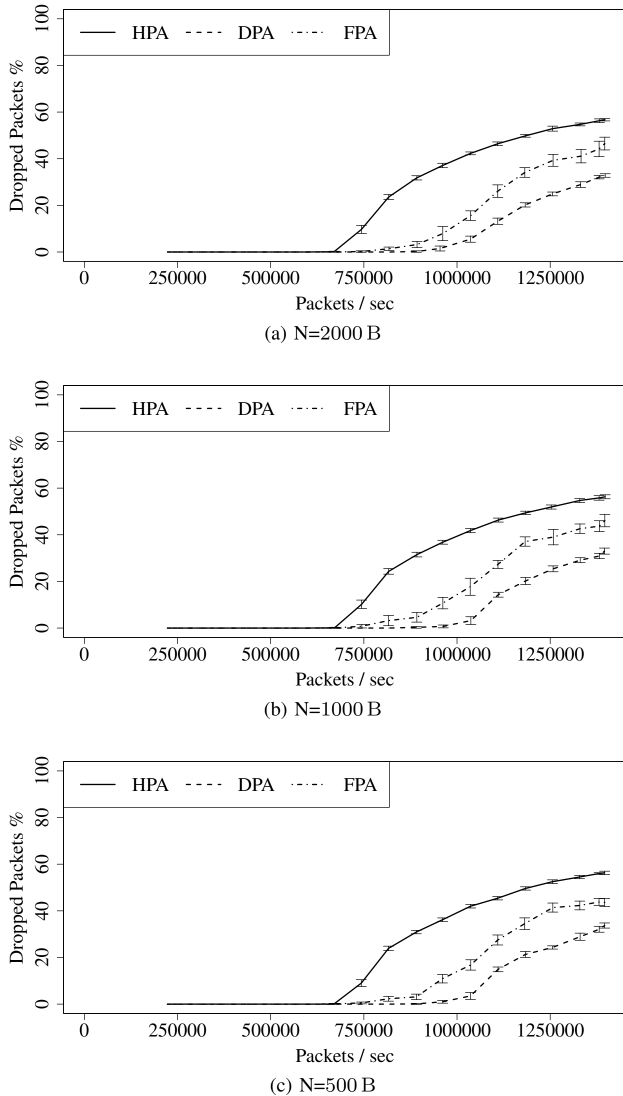


Fig. 4. Detected events and dropped packets with different replay speeds, compared to Snort reading unfiltered traffic. Mean of 10 runs with confidence intervals (95%).



(a) N=2000 B

(b) N=1000 B

(c) N=500 B

Fig. 5. Speed of Vermont filtering the traffic and exporting packets to a RAM disk instead of to a NIDS. Mean of 10 runs with confidence intervals (95%).

Generally HPA needs more CPU and memory resources compared to FPA and DPA because of its more powerful HTTP filtering engine. This does not necessarily mean that it has to drop packets earlier: An artifact can be observed for the N=2000 B and N=1000 B cases. Here, FPA and DPA start earlier to drop packets. This is due to the internal TCP reassembly strategy used by FPA and DPA. If packets are lost, these approaches fill the empty spots with binary zero and forward the result to Snort. Snort now has to analyze these zeroed out parts of the message, which costs additional CPU resources. HPA, instead, uses a stateful TCP reassembly engine and a stateful HTTP parser and, thus, does not show this behavior.

Another interesting observation from the results in Figure 4 is that while the rate of dropped packets increases pretty fast, the detected events ratio does not decrease at the same speed. The explanation is that packet drops are more or less uniformly distributed. The payload patterns that trigger events, however, can be found mostly in the beginning of HTTP messages. Thus, the probability of an event packet being dropped is lower than the overall packet drop probability.

As can be expected from the functional evaluation in Section IV-B, the performance gain of HPA compared to FPA and DPA becomes more prominent for smaller N. HPA achieves the same detection quality for small N, where FPA and DPA require large N to achieve a similar event detection rate. Thus, the performance of HPA at N=500 B should in fact be compared to FPA and DPA at N=2000 B. Hence, we can report a performance improvement of HPA of a factor of more than 3 compared to DPA.

Finally, Figure 5 shows the performance of the filtering techniques FPA, DPA, and HPA without the connected Snort NIDS. We performed the same experiments again, but did not export the data to Snort but wrote it to /dev/null. Again, the results are the average of 10 runs for every measurement point. The results show a higher throughput compared to Figure 4, which shows further evidence that packets were tailbacked by Snort and that with a faster NIDS an even higher overall packet throughput would be possible. This holds for all configured values of N. Also, it can be seen that HPA has a slightly reduced performance compared to FPA and DPA. This is due to the additional complexity for TCP and HTTP reassembly. But, as shown in Figure 4, in the case where HPA starts to drop packets earlier than FPA and DPA it retains substantially more events.

V. Conclusion

In this work, we presented a novel approach, HTTP-based Payload Aggregation (HPA), to speed-up signature-based Network Intrusion Detection Systems (NIDS). Following earlier work such as Time Machine, Front Payload Aggregation (FPA), and Dialog-based Payload Aggregation (DPA), we exploit the fact that attack patterns are usually located close to the beginning of an application layer PDU. We extend this idea by proposing a novel concept for filtering taking into account the pipelining architecture of modern web traffic. We demonstrate the capabilities of our approach focusing on HTTP traffic. HPA is able to reassemble pipelined HTTP/1.1 or HTTP/2 traffic in order to identify request/response messages. Now, the idea is to capture the first N byte of every HTTP message, which is then processed by a NIDS. We assessed the quality of our concept using real-world traffic traces analyzed using the popular NIDS Snort and compared it to state of the art solutions FPA and DPA as well as to Snort analyzing all the network traffic. We not only show that HPA can retain almost all events generated with Snort while filtering out most of the traffic, but that filtering and analyzing of the resulting traffic is substantially faster compared to Snort analyzing unfiltered traffic. Overall, we show that for the used traces we still detect more than 97 % of the events with only 2.5 % of the network traffic to be analyzed.

References

[1] R. Kemmerer and G. Vigna, "Intrusion Detection: A Brief History and Overview," *IEEE Computer, Special Issue on Security and Privacy*, pp. 27–30, Apr. 2002.

[2] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network Anomaly Detection: Methods, Systems andTools," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 303–336, 2014.

[3] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in *13th USENIX Conference on System Administration (LISA 1999)*, Seattle, WA, Nov. 1999, pp. 229–238.

[4] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Elsevier Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999.

[5] P.-C. Lin and J.-H. Lee, "Re-examining the Performance Bottleneck in a NIDS with Detailed Profiling," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 768–780, 2013.

[6] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," in *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002, pp. 285–293.

[7] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "An Active Splitter Architecture for Intrusion Detection and Prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 1, pp. 31–44, 2006.

[8] T. Limmer and F. Dressler, "Adaptive Load Balancing for Parallel IDS on Multi-Core Systems using Prioritized Flows," in *20th IEEE International Conference on Computer Communication Networks (ICCCN 2011)*. Maui, HI: IEEE, July/August 2011, pp. 1–8.

[9] S. Kornexl, V. Paxson, H. Dreger, R. Sommer, and A. Feldmann, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," in *5th ACM SIGCOMM Conference on Internet Measurement (IMC 2005)*. Berkeley, CA: ACM, Oct. 2005, pp. 267–272.

[10] T. Limmer and F. Dressler, "Flow-based Front Payload Aggregation," in *34th IEEE Conference on Local Computer Networks (LCN 2009): 4th IEEE LCN Workshop on Network Measurements (WNM 2009)*. Zurich, Switzerland: IEEE, Oct. 2009, pp. 1102–1109.

[11] ——, "Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation," in *30th IEEE Conference on Computer Communications (INFOCOM 2011), 14th IEEE Global Internet Symposium (GI 2011)*. Shanghai, China: IEEE, Apr. 2011, pp. 833–838.

[12] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger, "Anatomy of a Large European IXP," in *ACM SIGCOMM 2012*. Helsinki, Finland: ACM, Aug. 2012, pp. 163–174.

[13] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the Internet's Application Mix from Packet-Sampled Traffic," in *Passive and Active Measurement Conference (PAM 2015)*, New York City, NY, USA, Mar. 2015.

[14] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori, "DASH Fast Start Using HTTP/2," in *25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2015)*, Portland, OR, Mar. 2015, pp. 25–30.

[15] M. Belshe, M. Thomson, and R. Peon, "Hypertext Transfer Protocol Version 2 (http/2)," IETF, RFC 7540, 2015.

[16] F. Erlacher, S. Woertz, and F. Dressler, "A TLS Interception Proxy with Real-Time Libpcap Export," in *41st IEEE Conference on Local Computer Networks (LCN 2016), Demo Session*. Dubai, UAE: IEEE, Nov. 2016.

[17] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems," in *19th USENIX Conference on Security (USENIX Security 2010)*. Washington D.C.: ACM, Aug. 2010, pp. 8–23.

[18] W. Bul'ajoul, A. James, and M. Pannu, "Improving Network Intrusion Detection System Performance through Quality of Service Configuration and Parallel Technology," *Journal of Computer and System Sciences*, vol. 81, no. 6, pp. 981–999, 2015.

[19] T. Dietz, B. Claise, P. Aitken, F. Dressler, and G. Carle, "Information Model for Packet Sampling Exports," IETF, RFC 5477, Mar. 2009, draft-ietf-psamp-info.

[20] L. Braun, G. Muenz, and G. Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in *12th Network Operations and Management Symposium (NOMS 2010)*. Osaka, Japan: IEEE, Apr. 2010, pp. 264–271.

[21] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: A Low-Latency and Energy-Efficient Network-Level Intrusion Detection System," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 1, pp. 142–155, Jan. 2016.

[22] A. Milenkoski, M. Vieira, S. Kounev, A. Avritzer, and B. D. Payne, "Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 12, 2015.

[23] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont - A Versatile Monitoring Toolkit for IPFIX and PSAMP," in *IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM 2006)*. Tübingen, Germany: IEEE, Sep. 2006, pp. 62–65.

[24] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," IETF, RFC 5101, Jan. 2008.

[25] F. Erlacher, W. Estgfaeller, and F. Dressler, "Improving Network Monitoring Through Aggregation of HTTP/1.1 Dialogs in IPFIX," in *41st IEEE Conference on Local Computer Networks (LCN 2016)*. Dubai, UAE: IEEE, Nov. 2016, pp. 543–546.

[26] L. Deri, A. Cardigliano, and F. Fusco, "10 Gbit Line Rate Packet-To-Disk Using n2disk," in *32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*. Turin, Italy: IEEE, Apr. 2013, pp. 3399–3404.

[27] K. Salah and A. Kahtani, "Performance Evaluation Comparison of Snort NIDS Under Linux and Windows Server," *Journal of Network and Computer Applications*, vol. 33, no. 1, pp. 6–15, 2010.

[28] K. Thongkanchorn, S. Ngamsuriyaroj, and V. Visoottiviseth, "Evaluation Studies of Three Intrusion Detection Systems Under Various Attacks and Rule Sets," in *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, Shaanxi, China, Oct. 2013, pp. 1–4.