Tolerating Overload Attacks Against Packet Capturing Systems

Antonis Papadogiannakis,* Michalis Polychronakis,† Evangelos P. Markatos*
*FORTH-ICS, †Columbia University
{papadog,markatos}@ics.forth.gr, mikepo@cs.columbia.edu

Abstract

Passive network monitoring applications such as intrusion detection systems are susceptible to overloads, which can be induced by traffic spikes or algorithmic singularities triggered by carefully crafted malicious packets. Under overload conditions, the system may consume all the available resources, dropping most of the monitored traffic until the overload condition is resolved. Unfortunately, such an awkward response to overloads may be easily capitalized by attackers who can intentionally overload the system to evade detection.

In this paper we propose Selective Packet Paging (SPP), a two-layer memory management design that gracefully responds to overload conditions by storing selected packets in secondary storage for later processing, while using randomization to avoid predictable evasion by sophisticated attackers. We describe the design and implementation of SPP within the widely used Libpcap packet capture library. Our evaluation shows that the detection accuracy of Snort on top of Libpcap is significantly reduced under algorithmic complexity and traffic overload attacks, while SPP makes it resistant to both algorithmic overloads and traffic bursts.

1 Introduction

Passive network monitoring systems have been increasingly used to improve the performance and security of our networks. These systems operate in unpredictable and sometimes hostile environments where transient traffic and malicious attackers may easily overload them up to the point where they cease to function correctly. Unfortunately, traditional packet capturing systems have not been designed for such hostile environments, and do not gracefully handle overload conditions. For example, when faced with overload conditions and full packet queues, most packet capturing systems start to discard all incoming packets for as long as the overload persists.

This naive approach to packet discarding has three major disadvantages: (i) it may drop packets that contain *important information*, such as an attack or a particular pattern; (ii) it can be exploited by attackers to *hide their attack* by flooding the system with bogus packets up to the point where the system overloads and starts discarding (i.e., not inspecting) most of the incoming packets [2, 15, 17]; (iii) it robs monitoring applications from the opportunity to *selectively discard the unimportant packets* in the traffic [9, 10, 12], and forward for processing and further inspection the *important* ones.

To cope with high traffic volumes, several techniques have been proposed for improving the performance of Network Intrusion Detection Systems (NIDSs) by accelerating the packet processing throughput [7, 14], or by balancing detection accuracy and resource requirements [3, 8]. However, even after carefully tuning a NIDS according to the monitored environment, it will still have to cope with inevitable traffic bursts or unpredictable algorithmic attacks.

To address these problems we propose *Selective Packet Paging (SPP)*, a novel approach for mitigating both traffic overloads and algorithmic attacks by exploiting the following two dimensions: (i) we introduce a *new level in the memory hierarchy* of packet capturing systems, a level which is able to store all packets during overload periods; and (ii) we propose a *randomized timeout algorithm* which is able to detect and isolate malicious packets that trigger algorithmic overload attacks.

The main contributions of this paper are: (i) we demonstrate that the root of packet discarding under overload in the current packet capturing system in Linux [11] is the poor design choices in memory management. (ii) we propose Selective Packet Paging, a novel two-layer memory management system that can store practically all network packets during overloads, and resolve algorithmic complexity attacks by removing from the critical path any malicious packets that slowdown a monitoring system; (iii) we implement Selective Packet Paging within the Libpcap packet capture library; (iv) we experimentally evaluate our approach using the Snort NIDS [16], and we show that it can sustain algorithmic attacks and traffic overloads without discarding any packets, while the traditional approach is forced to discard the largest percentage of the incoming packets and miss 100% of the attacks, and (v) we analytically evaluate the randomized timeout selection approach of SPP and show that the probability of detecting an algorithmic attack reaches certainty exponentially fast.

2 Selective Packet Paging

The main cause of packet loss during overloads is usually the limited number of packets that the Operating System's packet capturing subsystem can store in main memory. Thus, in case of traffic overloads or algorithmic attacks, the main memory fills up quickly and the rest of the incoming packets are just dropped. One obvious solution would be to increase the main memory available to the packet capturing subsystem. Unfortunately, main memory typically can not store more than a few seconds of network traffic for a high-speed link. Thus, an algorithmic attack or a network overload that lasts for more than a few seconds will eventually lead to packet drops.

In modern systems, the available disk storage is up to three orders of magnitude larger than the available storage in main memory. Thus, captured packets can be buffered on disk for several hours under overload conditions, instead of just a few seconds in main memory.

2.1 Multi-level Memory Management

In this paper we propose to break away from the single-level memory hierarchy traditionally used by packet capturing subsystems and employ a multi-level memory hierarchy consisting of at least two levels: a main memory and a secondary storage. Under normal circumstances captured packets are written in main memory. Under traffic overload or algorithmic attacks, when the main memory fills up, extra packets are written to secondary storage. Figure 1 presents the two-level memory hierarchy of our approach. As long as it is not full, newly arriving packets are written in the memory buffer. Upon filling up, newly arriving packets are stored in the second layer of the memory hierarchy, i.e., the disk buffer.

Note that while newly arriving packets are being written to disk, memory space is being freed up as monitoring applications continue to consume existing packets. In this case, we would like to be able to write newly arriving packets in main memory and thus avoid the disk access overheads. However, this choice implies that sequentially arriving packets may be written to different levels of the memory hierarchy, oscillating between main memory and disk. For this reason we use a Packet Receive Index which keeps the incoming packets strictly in FIFO order. To deliver packets in the correct order, we use one bit for each incoming packet in the Packet Receive Index, as shown in Figure 1. This bit indicates whether the packet was stored in main memory or on disk.

2.2 Randomized Timeout Intervals

Although multi-level memory management makes sure that no packets are lost during an overload, algorithmic attacks may force the CPU to spend most of its time on processing bogus attack packets that trigger an algorithmic overload—benign network packets will just keep accumulating on disk. Selective Packet Paging advocates that instead of blindly sending subsequent packets to secondary storage when the main memory is full, we should develop mechanisms to detect packets that trigger an algorithmic overload, weed them out, and send *them* to secondary storage for processing at a later point in time.

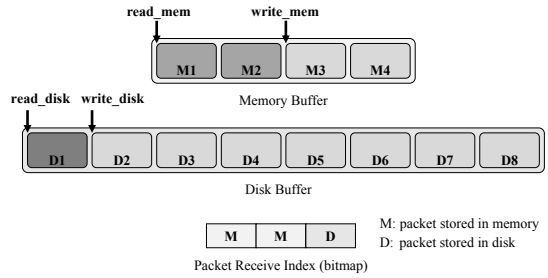


Figure 1: **A snapshot of Packet Paging for buffering packets to memory and disk.** The Packet Receive Index indicates that the first two packets are stored in the Memory Buffer, while the third packet is in the Disk Buffer.

To detect packets that trigger algorithmic overload attacks we could use a timeout counter: when a new packet arrives, the counter is initialized to a timeout value larger than the processing time of benign packets. If the timeout expires while the application is still processing the same packet, then this packet is considered suspicious. Unfortunately, setting timeouts on each and every packet imposes a prohibitively large processing overhead.

To reduce this overhead, the counter can be set at *periodic* intervals: if the packet being processed during the interval expiration is the same packet that was being processed at the time the counter was set, then SPP considers this packet as suspicious. As a result, the packet, along with all subsequent packets from the same network flow, will be buffered to disk. Although setting the timeout at periodic intervals has the potential to reduce the processing overhead, choosing an appropriate timeout can be challenging: a very large value may miss a lot of attack packets, while a very small value may impose a large processing overhead. To make matters worse, a single predefined timeout value (or a deterministic sequence of timeout values) could theoretically be evaded by a sophisticated attacker who manages to send all attack packets between successive timeouts.

To solve this problem, SPP uses a *randomized* timeout interval. Instead of choosing a predefined constant timeout, SPP selects a random timeout uniformly distributed in the interval $[low, high]$. Choosing a large value for $high$ reduces the average timeout overhead, while choosing a small value for low makes detection of algorithmic attacks easier. Indeed, to make sure they avoid detection, attackers should only send attack packets that impose a processing delay of no more than low seconds. Therefore, a small low value forces a (what used to be) sophisticated algorithmic attack to degenerate into a brute force Denial of Service attack consisting of a torrent of attack packets, which can be easily detected and filtered out.

3 Implementation

We have implemented Selective Packet Paging within the popular packet capturing library Libpcap [11], so that existing network monitoring applications can benefit from SPP without any code modifications. In our prototype implementation we use three separate threads: (i) the *packet capturing and storing thread*, which receives packets from the NIC and stores them to memory or disk; (ii) the *packet processing* thread, which finds the next packet through the Packet Receive Index, and calls the callback function for processing each packet; and (iii) the *disk I/O thread*, which handles all communication with the secondary storage. We give higher priority to the packet capturing thread over the packet processing thread to ensure that all packets will be stored during overloads. To optimize disk throughput, the disk I/O thread transfers packets between main memory and disk in batches. Moreover, to avoid delays from blocking read operations, the disk I/O thread prefetches the next batch of packets from disk to a memory cache.

The processing thread keeps a counter of the processed packets. When the timer expires, it checks how many packets have been processed from the previous timer expiration. If the number of processed packets remains the same, then the current packet delays the system for an unreasonably long time. Thus, the packet is evicted and buffered to disk, while its flow and source IP address are marked as suspicious. Packets belonging to suspicious flows are written to disk as low priority packets. If there are no normal priority packets in the queues, then a low priority packet is processed. The next timer interval is scheduled to a random time between the low and $high$ limits. The timer expires based on the time passed while only the current process is executing, so SPP is not affected by external background activities. To avoid false positives, a proper value for the low limit should be used. Then, only packets with significant processing delays will be detected as suspicious. But even in case of false positives, packets will not be dropped. They will be processed when the system has the available resources.

4 Analytical Evaluation

Using a random timeout uniformly distributed in the range $[low, high]$, SPP makes it difficult for attackers to evade detection, while keeping the timeout overhead reasonably low. Since, however, the timeout is a random variable, it is theoretically possible even for an attack packet that triggers a long algorithmic attack to evade detection. This is especially true if the timeout interval chosen while the attack packet is being processed is relatively large. In this section we show that although it is theoretically possible for one attack packet to evade detection, it is very unlikely that several attack packets will go undetected. An attacker who wants to sustain an algorithmic attack has to send several attack packets, and it is improbable that none of them will be detected.

To simplify our analysis, we initially assume that there are only attack packets, that each attack packet is being analyzed for a constant interval of d µs, and $low < d$. Selective Packet Paging can detect an attack if two successive timeouts expire within the same interval for the same attack packet. The first timeout expires at time t_1, which will fall within an interval i of an attack packet. Thus, $i \times d < t_1 < (i+1) \times d$. The probability that the second timeout t_2 will also fall within the interval i is:

$$P(t_2 < (i+1) \times d) = \frac{d - t_1 - low}{high - low} \quad (1)$$

since there are $high - low$ possible choices for a timeout but only $d - t_1 - low$ accepted choices so that the second timeout expires within the interval i. In the unfortunate for the attacker case that t_1 falls in the beginning of the interval i, there are $d - low$ accepted choices for t_2. In case t_1 falls in the position $(i+1) \times d - low - 1$ of the interval i, there is only one accepted choice for t_2: the low timeout value. On average, there are $(d - low)/2$ accepted choices for t_2 in case t_1 falls within the first $(d - low)$ values of the interval i. If t_1 falls in the last low values of the interval i, there is no accepted choice for t_2. Overall, the probability for detection with two timeouts in the same interval is:

$$P(det) = \frac{(d - low)^2}{2 \times d \times (high - low)} \quad (2)$$

since the possible choices for two timeouts are $(high - low) \times (high - low)$, the accepted choices for the first timeout are $(high - low)$, and the accepted choices for the second timeout are $(d - low)/2 \times (d - low)/d$.

The probability of not detecting an attack after N timeouts have expired is $(1 - P(det))^N$, and thus the probability of detecting the attack after N timeouts is $1 - (1 - P(det))^N$: we see that the detection probability approaches 1 very fast as N increases. Also, the detection probability from Equation 2 implies that, on average, SPP will need $T = 1/P(det) + 1$ timeouts to detect the attack. This number corresponds on average to $T \times (high - low)/(2 \times d)$ attack packets and $T \times (high - low)/2$ µs.

The outcomes of our analysis are also valid in case that the attack packets induce variable delays with an average delay of d µs. In a more realistic scenario there will be both benign and attack packets, so that attack packets will be a percentage a of the total packets, with $0 < a < 1$. The average processing time for a benign packet is t µs, and we expect that $t < d$. In this case the detection probability from Equation 2 is:

$$P(det) = \frac{a \times d}{d + t} \times \frac{(d - low)^2}{2 \times d \times (high - low)} \quad (3)$$

since the probability of the first timeout to expire within an interval of an attack packet is $a \times d/(d + t)$.

To validate our analysis we compare its results with a simulation-based evaluation. Figure 2 presents the detection time as a function of the processing time for each

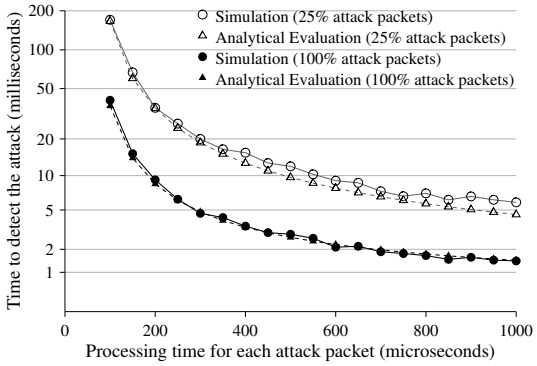


Figure 2: **Detection time as a function of the processing time of attack packets.**

attack packet for two attack scenarios: i) when all packets are attack packets, and ii) when the percentage of attack packets is 25%. The processing time t of each benign packet is uniformly distributed between 1 and 30 µs, while the processing time d of each attack packet is constant for each simulation. We vary d from 100 to 1000 µs to examine how the detection time is affected. The randomized timeout for SPP is randomly chosen between low=50 and $high$=1000 µs. When two successive timeouts expire during the processing interval of the same attack packet, the experiment ends and the detection time is recorded. Each experiment was repeated a million times.

For the analytical evaluation we used the probability from Equation 3 to compute the number of timeouts T needed for the detection:

$$T = 1/P(det) + 1 = \frac{2 \times d \times (high - low) \times (d + t)}{(d - low)^2 \times a \times d} + 1 \quad (4)$$

The average detection time is $T \times (high - low)/2$ µs.

In Figure 2 we can see that simulation results are very close to the expected results based on our analysis. We observe that SPP with the randomized timeout can detect even attacks with very small delays within just a few milliseconds. For instance, when the processing time of an attack packet is 200 µs, SPP detects the attack within the first 10 ms in case all packets belong to this attack. In a more conservative attack, where only 25% of the total packets impose 100 µs processing time, SPP needs about 170 ms to detect it. However, such a conservative attack for a period of a few milliseconds will not affect significantly the system. More aggressive attacks are detected by SPP within less than 2 ms.

5 Experimental Evaluation

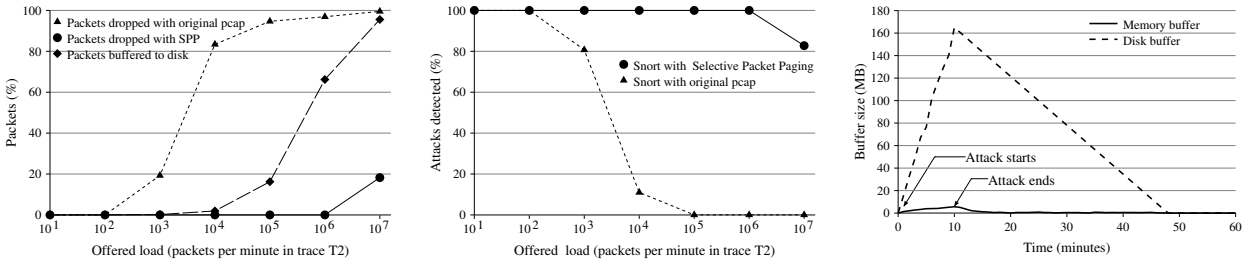
Our testbed consists of two PCs interconnected through a 10GbE switch. The first is used for traffic generation, which is achieved by replaying real network traffic at different rates. The second (NIDS PC) is equipped with two quad-core Intel Xeon 2.00 GHz CPU with 6 MB L2 cache, 4 GB RAM, and a 10GbE network interface. Beyond the system disk, the NIDS PC has four 750 GB 7200 RPM SATA disks organized in RAID 0 (totaling 3 TB of secondary storage for SPP), which can sustain a 3 Gbit/s read and 1.8 Gbit/s write throughput. The size of the memory buffer for storing packets is set to 1 GB in all cases. We use Snort v2.8.3.2 [16] with the latest official Sourcefire VRT rule set, containing 9276 rules.

For our evaluation we use three traces. As real background traffic, we replay a one-hour full payload trace (named `T1`) captured at the access link that connects a large university campus to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 flows, with an average traffic rate of 110 Mbit/s. The second trace (`T2`) is used to trigger an algorithmic overload in Snort using crafted packets that exploit the backtracking vulnerability of a regular expression used in a particular rule. The third trace (`T3`) contains 120 real attacks that are detected by Snort using the default rule set, resulting to 276 alerts from 14 different rules. We replay this trace continuously and measure the alerts that Snort was able to detect with the original Libpcap and SPP.

5.1 Algorithmic Complexity Attack

In this experiment we perform an algorithmic complexity attack against Snort, which uses the PCRE library [5] for regular expression matching, as described by Smith et al. [17]. For a given input string, PCRE iteratively explores paths in its internal tree-like structure until it finds a matching state. If it fails to find a match, it backtracks and tries another path until all paths have been explored. As the number of backtracks increases, more time is spent on matching, and overall performance decreases. The attack we use targets the Snort rule 2682, which detects exploitation attempts of a known vulnerability that allows e-mail attachment execution. We created 1500-byte packets belonging to an established connection destined to port 25 (trace `T2`). When processed, each crafted packet results to a processing time about 1360 times slower compared to the average time that Snort spends for processing SMTP packets in trace `T1`.

We set out to explore what is the packet loss of the Libpcap and SPP during this algorithmic overload attack. While we replay the background traffic and the actual attacks (traces `T1` and `T3`) at low rates, we also replay the `T2` trace at a variable rate, from 10 crafted packets up to 10^6 crafted packets/min. Figure 3(a) shows the percentage of dropped packets when Snort runs on top of the original Libpcap and on top of SPP. We observe that when the traffic load reaches a mere 10^3 packets/min, Libpcap starts losing packets, and when the load exceeds 10^4 it loses more than 80% of the packets. On the contrary, at these loads, SPP loses no packets and manages to store them to disk. Figure 3(a) also shows that the packets buffered to disk by SPP are fewer than those dropped by Libpcap at similar rates. This is because by identifying and weeding out algorithmic attack packets, SPP frees more CPU cycles for processing ordinary packets.

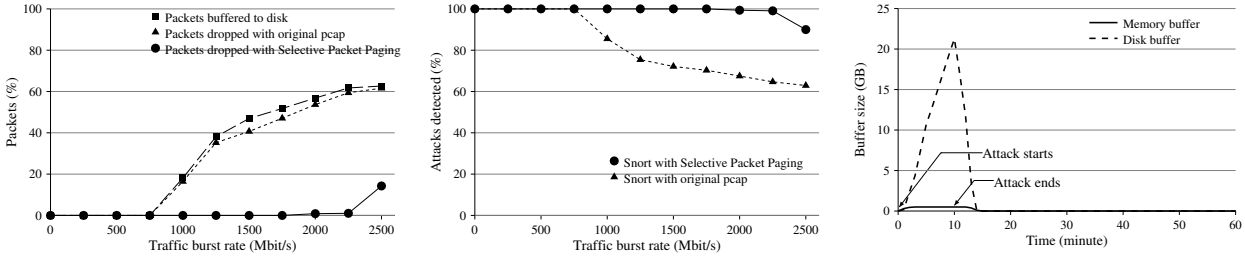


(a) Percentage of dropped packets and packets buffered to disk as a function of the offered load.

(b) Percentage of detected attacks as a function of the offered load.

(c) Size of memory and disk buffers over a 60-minute time period when sending 10,000 crafted packets/minute for the first 10 minutes.

Figure 3: Performance of SPP and original Libpcap under an algorithmic complexity attack.



(a) Dropped packets and packets buffered to disk as a function of the traffic burst rate.

(b) Detected attacks as a function of the traffic burst rate for 30-second bursts.

(c) Size of memory and disk buffers over a 60-minute time period when sending 30-second traffic bursts of 1.5 Gbit/s for the first 10 minutes.

Figure 4: Performance of SPP and original Libpcap in case of 30-second traffic bursts.

Packet loss is directly translated to undetected attacks. Figure 3(b) shows the percentage of attacks detected by the two systems. We see that Snort on top of Libpcap starts missing attacks when the traffic load exceeds 10^2 packets/min, and misses all attacks as the load reaches 10^5 packets/min. At the same rates, Snort on top of SPP detects *all* attacks, as all packets are stored to secondary storage and are eventually inspected, and does not miss any attack for rates up to 10^6 packets/min. That is, an attacker needs to send about 10^7 packets/min to reduce the probability of being detected just by 17%. In this extreme case, SPP was not able to store all incoming packets to disk due to the high traffic rate. Compared to the original Libpcap, SPP can handle 10,000 times more crafted packets, offering significant tolerance against highly efficient algorithmic complexity attacks.

To measure the time that the system needs to recover from overload, we replayed traces `T1` and `T3` at low rates for 60 minutes, and replayed the `T2` trace at a rate of 10^4 packets/min for the first 10 minutes of the experiment. Figure 3(c) presents the size of memory buffer and disk buffer over time. We observe that with SPP the size of the memory buffer was always less than 6 MB, for the whole 60-minute period. The attack packets (and their associated flows) identified by SPP were sent to disk. Indeed, to accommodate the attack packets, the disk buffer size increased from 16.23 MB (at minute 1) to 165 MB (at minute 10, which was the highest point of the attack), and then slowly decreased back to zero at minute 48.

5.2 Traffic Overload

In this experiment we explore how Snort on top of SPP and the original Libpcap responds to traffic bursts. We replay trace `T1` at its original rate as background traffic and trace `T3`, containing the 120 real attacks, at 1 Mbit/s for the entire duration of the experiment. At each minute we send a traffic burst that lasts for 30 seconds using traffic from `T1`. The peak rate of the burst is varied from 1 Gbit/s up to 2.5 Gbit/s, to evaluate how burst intensity may influence SPP. Each experiment lasts 10 minutes.

Figures 4(a) and 4(b) present the percentage of dropped packets and detected attacks as a function of the rate of the 30-second traffic bursts. We observe that Libpcap starts dropping packets when the traffic bursts are around 1 Gbit/s, resulting in about 17% undetected attacks. When the bursts reach a rate of 2 Gbit/s, 53% of the packets are dropped and 32.5% of the attacks are missed. On the other hand, Snort with SPP drops no packets and misses no attacks even at rates as high as 2 Gbit/s. Although our disk system writes packets with a throughput of 1.8 Gbit/s, the two-level memory hierarchy allows processing of 2 Gbit/s traffic without packet loss. Only when the burst rates exceed 2.25 Gbit/s the secondary storage is not able to keep up with network traffic and SPP starts losing packets.

Figure 4(c) shows the size of memory and disk buffers when sending 30-second traffic bursts with a rate of 1.5 Gbit/s for 10 minutes, and continue sending only background traffic for another 50 minutes. The memory

buffer remains full at 500 MB for the first 12 minutes, while the disk buffer fills up continuously during the first 10 minutes, all the way up to 21.3 GB. From minute 11 to minute 13, the disk buffer size is reduced from 21.3 to 3.5 GB, as the system's resources are sufficient to process the excessive packets buffered during the traffic bursts. Thus, in the 14th minute, both memory and disk buffers are empty, so the system has fully recovered from the traffic overload attack. Compared with the algorithmic complexity attack, the system recovers faster from this traffic overload, (within just four minutes) because packets are not maliciously crafted to further slowdown Snort.

6 Related Work

To cope with high traffic volumes, several research approaches propose to distribute the load across multiple computers instead of using a single sensor [7], or utilize multi-core processors for parallel inspection [4, 14]. Other approaches propose to dynamically reconfigure a NIDS based on the run-time conditions [3, 8], or use load shedding techniques to defend against overloads [1, 13]. Recent works deal with high traffic volumes by applying a per-flow cutoff to selectively discard most of the traffic and focus on the beginning of each connection when the system is under load [9, 10, 12]. Unfortunately, overloads are still possible in all these systems, especially in case of algorithmic attacks [15, 17].

Smith et al. [17] propose memoization as an algorithmic solution to prevent overload attacks targeting backtracking-based algorithms. Crosby and Wallach [2] present an algorithmic complexity attack that exploits deficiencies of common data structures, and propose new hashing techniques which sacrifice average case performance for worst case performance. Khan and Traore [6] propose a model to detect algorithmic complexity attacks based on historical information of execution time and input characteristics, using regression analysis.

7 Conclusion

We presented Selective Packet Paging, a two-level memory management approach that buffers (otherwise dropped) packets to tolerate algorithmic complexity attacks and traffic overloads for network monitoring and security applications. Empowered with a randomized timeout, SPP can detect and isolate algorithmic attack packets, enabling the CPU to be used for more useful purposes. We have implemented SPP within the popular Libpcap packet capture library, so that existing applications can use it without any code modifications. Our experimental evaluation shows that NIDS, such as Snort, are vulnerable to both algorithmic complexity and traffic overload evasion attacks. Using SPP, a NIDS can handle both algorithmic and traffic overload conditions.

We believe that as network monitoring applications get more complicated, they will be increasingly vulnerable to algorithmic and traffic overload attacks. SPP offers a memory management approach and a dynamic overload detection technique that provide a seamless solution to this problem without requiring any changes to the monitoring applications themselves.

Acknowledgments

We would like to thank our shepherd Samuel T. King and the anonymous reviewers for their valuable feedback. This work was supported in part by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007.

References

[1] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load shedding in network monitoring applications. In *Proc. of the USENIX Annual Technical Conf. (ATC)*, 2007.

[2] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proc. of the 12th Conf. on USENIX Security Symp.*, pages 3–3, 2003.

[3] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. of the 11th ACM Conf. on Computer and communications security (CCS)*, pages 2–11, 2004.

[4] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. of the 10th annual Conf. on Internet measurement (IMC)*, pages 218–224, 2010.

[5] P. Hazel. Pcre: Perl compatible regular expressions. http://www.pcre.org.

[6] S. Khan and I. Traore. A prevention model for algorithmic complexity attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment, Second Intern. Conf., (DIMVA)*, pages 160–173, 2005.

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

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

[9] T. Limmer and F. Dressler. Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation. In *14th IEEE Global Internet Symp. (GI)*, pages 833–838, 2011.

[10] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *Proc. of the 2008 Conf. on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 183–194, 2008.

[11] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Lab., Berkeley, CA. (http://www.tcpdump.org/).

[12] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *Proc. of the Third European Workshop on System Security (EUROSEC)*, pages 15–21, 2010.

[13] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

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

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

[16] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 1999 USENIX LISA Systems Administration Conf.*, 1999.

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