MCA^2: Multi-Core Architecture for Mitigating Complexity Attacks

Yehuda Afek[§], Anat Bremler-Barr[†], Yotam Harchol[‡], David Hay[‡], Yaron Koral[§]

[§]Tel Aviv University
Tel Aviv, Israel

[†]The Interdisciplinary Center
Hertzelia, Israel

[‡]The Hebrew University
Jerusalem, Israel

{afek, yaronkor}@post.tau.ac.il, bremler@idc.ac.il, {yotamhc,dhay}@cs.huji.ac.il

ABSTRACT

This paper takes advantage of the emerging multi-core computer architecture to design a general framework for mitigating network-based complexity attacks. In complexity attacks, an attacker carefully crafts "*heavy*" messages (or packets) such that each heavy message consumes substantially more resources than a normal message. Then, it sends a sufficient number of heavy messages to bring the system to a crawl at best. In our architecture, called MCA^2—Multi-Core Architecture for Mitigating Complexity Attacks—cores quickly identify such suspicious messages and divert them to a fraction of the cores that are dedicated to handle all the heavy messages. This keeps the rest of the cores relatively unaffected and free to provide the legitimate traffic the same quality of service as if no attack takes place.

We demonstrate the effectiveness of our architecture by examining cache-miss complexity attacks against Deep Packet Inspection (DPI) engines. For example, for Snort DPI engine, an attack in which 30% of the packets are malicious degrades the system throughput by over 50%, while with MCA^2 the throughput drops by either 20% when no packets are dropped or by 10% in case dropping of heavy packets is allowed. At 60% malicious packets, the corresponding numbers are 70%, 40% and 23%.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management, Network monitoring*

General Terms

Design, Reliability, Performance, Security

Keywords

Intrusion Detection, Multi-core, Complexity Attack, DDoS

1. INTRODUCTION

Security devices, such as Network Intrusion Detection/Prevention Systems (NIDS/NIPS), are the front defense line against cyber attacks over the Internet. Open source examples of such devices include Snort [27] and Bro [10]. In recent years, a trend of two-phase *combined attack* on security devices is becoming common: the attackers first neutralize the security device, for example, by overwhelming it with traffic, and then, when the security device has been knocked down, attack the assets it was protecting. For example, a recent attack on SONY, combined a DDoS attack with credit cards theft [29]. The combined attacks usually have different effect on NIDS and NIPS. In NIDS, where the stealth-mode device only monitors the traffic and issues alerts when it detects malicious activity, these DDoS attacks may force the device to stop inspecting part, or all, of the traffic and thereby allowing another attack to pass unnoticed. On the other hand, in-line NIPS, which inspects the packets on their critical path, might be forced to drop legitimate traffic and therefore practically causing a denial of service on the servers it protects. For example, Bro and Snort are both vulnerable to this kind of attacks [20].

This paper deals with *complexity attacks*, which are used for the first phase. These attacks exploit the gap between the amount of resources the system requires in processing normal packets and carefully crafted packets that consume drastically more resources (computing, memory, cache, or other). These crafted packets, which we call *heavy* packets, are, on one hand, easy to construct, while, on the other hand, they require very intensive processing from the system. This implies that a small effort on the attacker's side leads the target system to spend great effort, and therefore, it is bound to lose.

We present MCA^2—a Multi-Core Architecture for Mitigating Complexity Attacks. MCA^2 essentially isolates the malicious traffic to a fraction of the cores and deals with legitimate traffic on the remaining cores, which are therefore not affected by the attack.

Our MCA^2 system can be configured to mitigate any complexity attack with the following properties:

1. There are heavy and normal packets, where heavy packets consume considerably more resources from the security device when being processed.

2. There is a method to identify heavy packets. This method requires very few resources.

3. Packets can be moved efficiently between system cores.

4. There is a special method that handles heavy packets more efficiently than the method used for normal packets.[1]

It turns out that there are quite a few complexity attacks that meet these criteria. However, we restrict our discussion to a central component of NIDS/NIPS, namely the Deep Packet Inspection (DPI) engine. DPI is the process in which the payload of the messages is inspected to detect predefined signatures of malicious activities. We consider three examples that have the above properties: *cache-miss attack* on Snort's signature detection engine; *active states explosion attack* on the Hybrid-FA [5] regular expression detection engine; and *force construction attack* on the Bro IDS regular expression detection engine.[2]

We focus on the first example and use it to explain our method and the above-mentioned list of properties. We then show that the active states explosion complexity attack fits our requirements as well. The third example is omitted due to space consideration. We back up all our findings with experimental results, showing the benefits of using MCA^2 in conjunction with the NIDS.

In general, any complexity attack that satisfies these four properties can be mitigated, given a proper heavy packet identification method. We discuss in detail two examples of such methods in this paper. Although each attack requires a different identification method, all methods share a common general technique of scanning the first few bytes and detecting malicious behavior as early as possible.

Specifically, considering cache-miss attacks, we target Snort's DPI engine, which uses some variant of the Aho-Corasick (AC) [1] algorithm for performing pattern matching. A complexity attack on the AC algorithm (in a stand-alone environment) is shown in [8]: AC uses a large deterministic finite automaton (DFA) that cannot fit entirely in the cache. The common traffic, however, uses only a very small part of it, resulting in fast memory references and few cache misses. An attacker can easily craft malicious packets that cause an exhaustive traversal over the DFA that pollutes the cache. In this paper, we show for the first time that Snort is indeed vulnerable to this attack: an attack on its DPI component degrades its *overall* performance by a factor of 4.2.

After establishing that the threat of this attack is real, we turn to investigate how MCA^2 mitigates such an attack. The key challenge is how to detect and isolate malicious traffic. This is done in two steps. First, training data is used to identify and mark the common states of the DFA. These are the states frequently visited while processing normal common traffic. Then, for each packet, we count the fraction of non-common states visited (out of the total number of states traversed by the packet). As soon as this fraction exceeds a certain threshold, the packet is marked *heavy*. When the fraction of heavy packets is above a certain threshold, we allocate one or more cores to deal with them exclusively, while the rest of the cores continue to process only normal traffic

[1]This special method usually handles normal packets poorly, otherwise it would have been used by the system in the first place.

[2]Bro takes a lazy approach to cope with the large DFA size. Namely, it constructs only the DFA parts it actually uses. Normal traffic uses only a small part of the DFA. Hence, a simple complexity attack forces Bro to construct a large portion of the DFA and, by that, degrades the performance significantly.

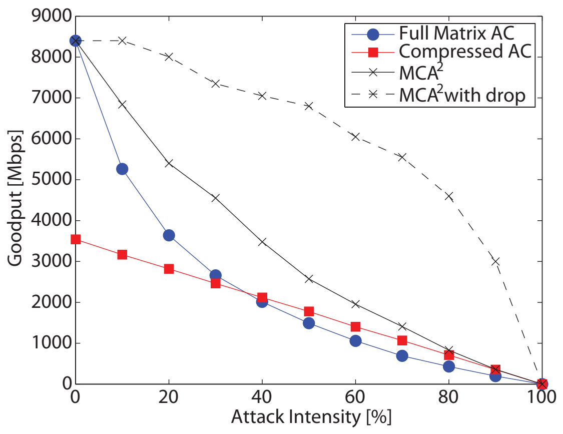


Figure 1: The goodput of MCA^2 for different attack intensities. MCA^2 with no drops maintains a balance between all cores.

(and to detect heavy packets); each subsequent *heavy packet* is moved to one of the dedicated cores. This process isolates the effect of heavy packets and protects the private caches of the non-dedicated cores from being polluted. MCA^2 can be further optimized by running on the dedicated cores an implementation that is optimized for heavy packets (albeit with penalty in the normal case).

The main performance measure we use is the *goodput* of the system, namely the volume of non-malicious packets that were processed. Our experimental results are summarized in Fig. 1, which shows the system's goodput under different attack intensities (namely, in 50% attack intensity, half of the incoming traffic is malicious). We compare the performance of MCA^2 with two implementations of the AC algorithm: the first, denoted "Full Matrix AC", is optimized for well-behaved normal traffic, and the second, denoted "Compressed AC" , is optimized to work under cache-miss attacks (as described in Section 2.2).

When the system is not allowed to drop packets, MCA^2 uses the "Full Matrix AC" on the cores that process normal traffic and the "Compressed AC" on the dedicated cores. The number of cores of each type is dynamically determined as a function of the attack intensity. When there is no attack, MCA^2 is reduced to "Full Matrix AC".

We also consider the case when the NIDS/NIPS is allowed to drop packets. Dropping all heavy packets implies that no dedicated threads are required, freeing up all processing resources for the detection of heavy packets and processing of non-heavy (mostly legitimate) packets, thus increasing the goodput.

Our experiments show a significant goodput improvement: MCA^2 achieves up to twice the goodput of both implementations, even without dropping packets. Furthermore, it *always* outperforms a hybrid implementation that chooses the best of the previous implementations at any given time, with a goodput boost of up to 73%.

As for the second example, we use the regular expressions Hybrid-FA data structure to illustrate an *active states explosion attack*. Hybrid-FA uses a single "head-DFA" for commonly-used states while other parts of the automaton are kept as separate DFAs, which are activated simultaneously when required. Usually, only the "head-DFA" is activated. Our complexity attack causes the Hybrid-FA to activate many states in parallel, thus forcing the system to traverse several states per input byte; this degrades system throughput significantly. We show that MCA^2 in full-drop setup can mitigate such an attack: our experiments show that under a mild *active states explosion attack*, the goodput of the system is increased by a factor of 4.8.

This paper is organized as follows. In Section 2 we provide the necessary background on complexity attacks and DPI. Section 3 discusses related work. Section 4 presents the cache-miss attack and its impact on Snort. Section 5 describes the MCA^2 architecture. In Sections 6 and 7 we demonstrate how MCA^2 mitigates cache-miss attacks and active-states explosion attacks, respectively. Our experimental results appear in Section 8. Finally, we conclude in Section 9.

2. BACKGROUND

2.1 Complexity attack

In a complexity attack, the attacker exploits the system's worst-case performance, which differs from the average case that the system was designed for. Crosby and Wallach were among the first to demonstrate the phenomenon on the commonly-used *Open Hash* data structure [13]: an attacker designs an input that requires $O(n)$ elementary operations per insertion, instead of $O(1)$ operations that are required on the average.

Recent works show that many other systems and algorithms are vulnerable to complexity attack, including QuickSort [22], regular expression matcher [25], intrusion detection systems [8,15,26], the Linux route-table cache [33], SSL authentication algorithm [11], and the retransmission algorithm in wireless networks [7]. Complexity attacks on different components of NIDS/NIPS were suggested in the past. For example, Bro maintains a hash table with the IP header fields of packets as keys; thus, by tailoring the traffic with specific headers, one can cause the hash insert-operation to last significantly longer, resulting in Bro failure. While in some cases modifying the algorithm suffices to mitigate the problem (e.g., Crosby and Wallach's attack can be solved by using hash functions that are not known to the attacker), this does not hold in general. We believe that only a system approach like MCA^2, can alleviate the attack scenarios discussed in this paper.

2.2 Deep Packet Inspection (DPI) and Snort

DPI is a crucial component in contemporary security tools, which heavily relies on pattern matching to detect signatures of malicious traffic. We consider the following two classes of pattern matching: exact matching and regular expression matching. The former usually uses a Deterministic Finite Automaton (DFA), while the latter uses either a DFA or a Nondeterministic Finite Automaton (NFA) for the ongoing inspection of the input data [18].

In our main example, we focus mostly on the exact matching algorithms, which use DFA. A DFA is a five-tuple $\langle S, \Sigma, \delta, q_0, F\rangle$, where S is a finite set of states, Σ is a finite set of input symbols, $\delta : S \times \Sigma \to S$ is a transition function, returning the next state, given the current state and any symbol from the input, $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states. Aho-Corasick algorithm provides a method to build such an automaton (a.k.a. AC DFA) from a set of patterns. Given the DFA, a packet is inspected by traversing the automaton symbol by symbol from s_0; a pattern is detected if a state in F is reached in this traversal. Fig. 2(a) depicts the AC DFA for the pattern set {E,BE,BD,BCD,CDBCAB,BCAA}.

In today's security tools, AC DFAs are huge—e.g., Snort's AC DFA has $77,182$ states for $31,094$ patterns—raising the question of how to store it efficiently in memory. The alternatives naturally trade memory space with execution time. Additionally, most security tools (including Snort) divide their patterns to several sets, according to the traffic type.

Snort uses a full-matrix encoding for its AC DFAs as presented in [1]. In this representation (see Fig. 2(b)), transitions are stored in a two-dimensional array with $|S|$ rows and $|\Sigma|$ columns. An entry at position (i, j) stores the value of $\delta(s_i, j)$, implying that the number of bits in each entry is at least $\log_2 |S|$. Typically, input inspection of one byte at a time results in an overall memory footprint of $256|S| \log_2 |S|$ ($|\Sigma| = 256$). For Snort's AC DFAs, this translates to a combined footprint of 75.15 MB. On the other hand, the main advantage of this encoding is that a transition consists of a *single* memory load operation, which reveals directly the next state.

Alternative encodings require more than one memory access, but offer significant memory reduction. Such encodings exist in the literature [4, 8, 30]. Fig. 2(d) depicts such encoding, as proposed in [8]; this encoding is based on a compressed automaton as depicted in Fig. 2(c).

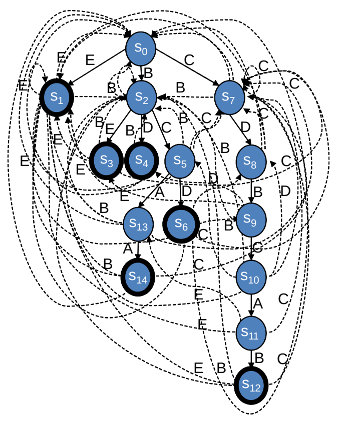
3. RELATED WORK

The recent proliferation of multi-core general purpose processors motivated many researchers to reinvestigate well known problems in this new domain. Among these, there are several works that proposed multi-core solution for DPI processing. These papers' main focus is on different ways to load balance the system tasks between the available cores.

Current NIDS/NIPS systems such as Snort [27] and Bro [10] split the load to many *sequential* sub-tasks in a pipeline manner. Other works, such as [32], suggest fine-grained pipelining for parallelizing network applications on multicore architectures. This partitioning is effective if the processing cost for each sub-task is similar, which is usually not the case for NIDS/NIPS.

A different line of research focuses on load balance the traffic flows equally between the different cores and performing the inspection in parallel [12,17,21,23,28]. The load balancing is based on both the packet header parameters and some layer-7 parameters. We note that such architectures are orthogonal to MCA^2 and can be applied to load balance the work between general threads that process the normal traffic. If MCA^2 is not used in conjunction with these architectures, they are all vulnerable to complexity attacks.

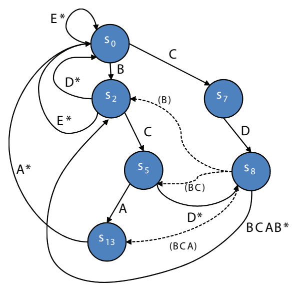
Becchi et al. [6] focus on DPI and present a performance evaluation scheme for multiprocessor systems. The proposed design also splits the traffic between several cores with the same DPI engine that supports regular expression matching. Their study identifies and evaluates algorithmic and architectural trade-offs and limitations, and highlights how the presence of caches affects the overall performance. However, it is geared at optimizing the normal case and is vulnerable to similar complexity attacks as we describe in the paper. Such attacks can be mitigated by incorporating MCA^2 to this scheme as well.



(a) The AC DFA for pattern-set {E, BE, BD, BCD, CDBCAB, BCAA}

	A	B	C	D	E
S_0	0	2	7	0	1
S_1	0	2	7	0	1
S_2	0	2	5	4	3
S_3	0	2	7	0	1
S_4	0	2	7	0	1
S_5	13	2	7	6	1
S_6	0	9	7	0	1
S_7	0	2	7	8	1
S_8	0	9	7	0	1
:					

(b) Full-matrix Encoding



(c) Compressed Automaton

S_0	B: 2, C: 7, E: 0*, *fail: 0*
S_2	C: 5, D: 0*, E: 0*, *fail: 0*
S_5	A: 13, D: 8*, *fail: 0*
S_7	D: 8, *fail: 0*
S_8	BCAB: 2*, BCA: 13, BC: 5, B:2, *fail:0*
S_{13}	A: 0*, *fail: 0*

(d) Compressed Encoding

Figure 2: Example of an AC DFA and two methods to store it in memory: non-compressed (full-matrix) encoding, and compressed encoding. The compressed encoding is derived from a compressed automaton, in which *fail* transitions are taken without consuming input symbols, and transitions marked with '*' indicate that a match was found.

Another multi-core load-balancing approach is to partition the patterns among the cores (cf. [31, 34, 35]). Then different DPI algorithms, each specializing in different kinds of pattern sets, is run on each core. In some cases, the partitioning itself is done so as to balance the load between the algorithms. It is important to note that, unlike MCA^2, in this kind of architectures, each packet is examined by several cores (each performs only part of the inspection). In addition, it does not take into account the incoming traffic, and is vulnerable to an attack on each core separately.

Kumar et al. [19] present several methods to reduce regular-expressions-based DFA size. One of the mechanisms used in that paper is based on the assumption that normal flows rarely match more than the first few symbols of any signature. Thus, the most frequently visited portions of the automaton are used to build a *fast path* DFA, and the rest of the automaton is represented by a separated NFA, which is the *slow path*. The authors suggest a solution, which is similar to MCA^2 in that it handles heavy traffic with a different algorithm and applies a lightweight classification algorithm to distinguish between heavy and normal traffic. In addition, [19] proposed to protect against DoS attacks by attaching lower priority to flows with higher probability of being malicious. Nevertheless, that work analyzes the case of a single core, and therefore could not benefit from the multi-core properties as MCA^2 does. Furthermore, the proposed protection in [19] fails under a continuous DoS attack because the heavy packets that receive lower priority eventually overload the system buffer. MCA^2 is also resilient to DoS attack of longer duration.

4. SNORT CACHE-MISS COMPLEXITY ATTACK

It has been shown that only a small number of states within the AC DFA is used, when scanning normal traffic [8]. Therefore, a very large fraction of the DPI memory accesses result in a cache hit. With this information, an adversary can launch a *Cache-Miss Attack*, consisting of input traffic that causes the DFA to traverse a large number of states, and therefore, having many cache misses. Such traffic can be constructed easily, since the signatures (and hence the AC DFA) are known publicly. These cache misses have two negative effects. First, a main-memory access is at least 10–20 times slower than a cache access, implying that it takes significantly more time to deal with this malicious input traffic. Second, and even more importantly, dealing with the malicious traffic causes significant cache pollution, which in turn slows down also the processing of well-behaved traffic. In the stand-alone setting considered in [8], the Cache-Miss Attack degrades the performance of the DPI routine by a factor of *four* and is considered an

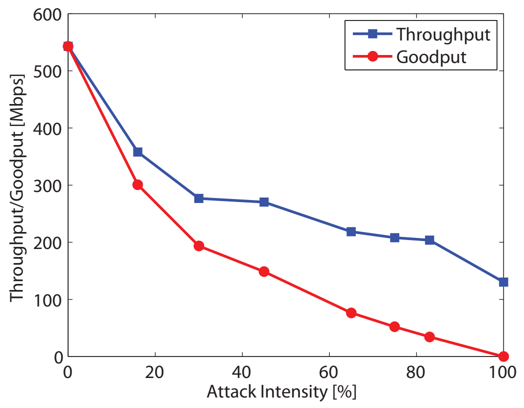


Figure 3: The effects of a cache-miss attack on the throughput and goodput of Snort, facing attacks of different intensities. All attacks do not cause any alert from Snort NIDS.

effective algorithmic complexity attack. The circles-curve in Fig. 1 shows the goodput reduction for different values of attack bandwidth.

While [8] demonstrates the attack on a stand-alone AC DFA, we show that the attack works on Snort, an entire NIDS, which is used in practice. Recall that Snort divides the pattern sets into classes according to traffic types. Among these, the largest DFA is the one that represents the HTTP traffic, with a memory footprint of about 32 MB. We devise a cache-miss attack in two steps. First, we collected the patterns for the automaton from Snort's publicly available signatures set. To prevent this attack from getting detected by Snort built-in mechanisms, we omit the last character of each pattern; this means that our attack packets, which contain these truncated patterns, go under the radar and do not activate any rule that alert the system. Then, we constructed a set of HTTP traffic traces that mix attack packets with normal HTTP traffic to the most-visited web sites [2]. We created eight different traces that differ in the proportion of attack traffic in them. The intensity of attack varies from 0% (only normal traffic) to 100% (only attack traffic).

Fig. 3 shows the overall goodput of Snort under these traces. The throughput of Snort drops by a factor of 1.5 when attack intensity is 16%, and *up to a factor of 4.2* as the cache miss attack becomes more intense. Namely, a Snort IDS with traffic bandwidth of 70% of its maximum capability would be knocked down or let packets go by uninspected under an attack that consumes only one sixth of this bandwidth. This proves the claim that the exact string matching engine is a bottleneck in Snort and shows the great impact that a cache-miss attack may have on such systems. We note that the exact matching in Snort is also an important building block for regular expression matchings: Snort breaks each regular expression into several exact patterns, and invokes a regular expression engine (for a single expression) upon matching all its exact patterns.

Next, we turn to discuss the solution for the complexity attacks that were presented in [8]. The gist of the solution is to use a compressed data-structure that fits mostly in cache (see Fig. 2(c)), and therefore is not prone to this kind of attack. Recall, however, that this data structure requires more than a single memory access per input byte.

The compressed encoding of Snort patterns requires only 1.5 MB. Compressed AC implementation has almost the same throughput, regardless of the kind of input traffic (the squares-curve in Fig. 1 presents the goodput of this implementation; the linear goodput decrease is due to the increased bandwidth of the attack and not due to an overall throughput degradation). However, it is *two times slower* than that of the full-matrix encoding under normal traffic. This implies that the solution in [8] recommends to always cut the throughput by half in order to overcome cache-miss attacks.

In this paper, we show how a multi-core architecture can be used to break the barrier and enjoy both worlds: high throughput on normal traffic and resiliency to cache-miss attacks. It uses the two encodings as building blocks and provides an efficient way to use them simultaneously, such that each handles the kind of traffic it is best designed for.

5. THE MCA^2 SYSTEM DESCRIPTION

5.1 MCA^2 Design overview

MCA^2 operates over a multi-core platform as described in Fig. 4, where each core runs one or more hardware threads (typically two in the Intel machines). Each hardware thread receives references to packets for inspection via its incoming-packets queue. The Network Interface Card (NIC) receives incoming traffic-packets and places them in main memory. It also writes packet references to the cores' incoming-packets queues. We follow recent works [16, 17] to load balance the incoming traffic between the different hardware threads in the NIC. Note that each packet has a single copy in main memory (created by the NIC). Sending a packet into a queue (or moving it from one queue to another) is performed efficiently by passing a pointer between the cores' queues without a message copy.

The system works either in *routine-mode* or in *alert-mode*. In routine-mode, all threads operate the same: they receive packets from the NIC and process them with the same monitoring algorithm. However, upon switching to alert-mode, the dedicated threads' primary role is to handle heavy packets. Therefore, they might switch to an algorithm that is optimized for such traffic pattern (depending on the kind of the attack). From that point, the dedicated threads receive messages from other threads with references to heavy packets. Thus, the dedicated thread handles the packets references in the messages of its transfer queue, as well as the packets in its incoming packets queue. Furthermore, the following *stealing* policy is incorporated to prevent load imbalance and increased latency for non-heavy packets that were sent by the NIC to the dedicated threads: when a general thread sends a heavy packet to a dedicated thread, it "steals" one or more not-yet-processed packets from that dedicated thread's incoming packets queue and places them at the head of its own incoming packets queue. Our experiments show that the system becomes balanced when the number of packets traded for a single heavy packet is between two and four, depending on the algorithms in use.

The last component of MCA^2 is its *stress monitor*, whose role is to monitor the percentage of heavy packets in the system and to switch between system modes. Namely, when the percentage of heavy packets crosses a specific threshold, the

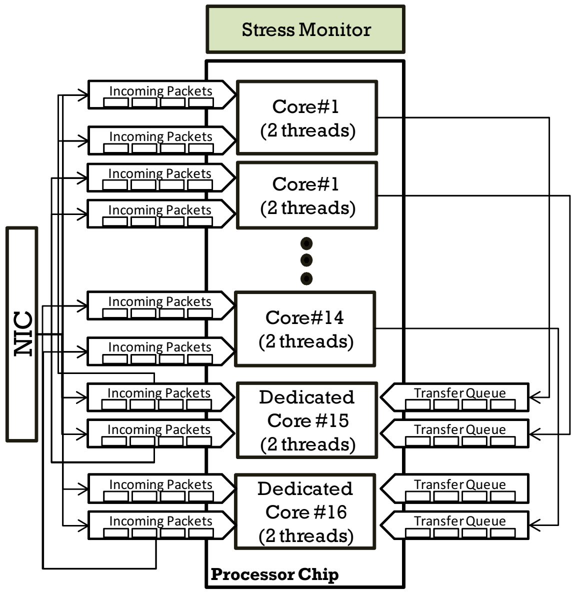


Figure 4: Illustration of MCA^2. Cores 1 through 14 are general, while Core 15 and 16 are dedicated. Each core has two threads. Threads of general cores transfer their heavy packet to a specific thread in a dedicated core through a transfer queue. Only the logical structure of the queues is presented.

system switches into alert-mode; conversely another threshold is used to determine when the system switches back into routine-mode. The thresholds are determined to maximize the system goodput.

We note that in some multi-core environments, the load balancing is done on the flow level (that is, all packets of the same flow are sent to the same core by the NIC) [23]. In such cases, MCA^2 should preserve this property; namely, after classifying a packet of some flow as heavy, all the consecutive packets of the same flow are treated as heavy packets.

5.2 Cross-Thread Communication Mechanism

Concurrency in multi-core systems usually suffers from cross-thread communication overhead, which might become significant in some constellations. The common cross-thread communication techniques require synchronization mechanisms that use expensive system calls and may cause blocking situations. In MCA^2, we use a non-blocking (that is, without any synchronization) mechanism with minimal overhead.

Notice that the most challenging stage of the cross-thread communication in MCA^2 is when writing references of heavy packets to the transfer queues: synchronization might be required since many general threads can transfer heavy packets to the same dedicated thread, resulting in simultaneous access to that queue. Therefore, we implement the transfer queue for each dedicated thread as a collection of queues, one for each general thread that transfers heavy packets to the corresponding dedicated thread. The dedicated thread, in turn, reads from these queues in a round-robin manner. Notice that each such queue is a single-writer single-reader queue.

In order to keep track over the state of records in the queue, the reader and writer threads use *phase* bits that alternate every time a round of read/write from/to the queue is completed. Specifically, each queue is implemented as an array of records, where each record has a left phase bit, a right phase bit, and a content field. The content field contains a pointer to the location of the packet in memory, along with the *offset* in the payload, which indicates the last byte in which the AC scan was in the root state of the AC DFA (see Fig. 5). Moreover, each queue has two global bit-fields that track the phases of the threads. These fields are `writer_phase`, which keeps the phase of the (single) enqueuing thread, and `reader_phase`, which stores the phase of the (single) dequeuing thread. Finally, each queue has two global pointers: `head`, which points to the next array entry to write to, and `tail`, which points to the next array entry to read from. All these fields are accessible to both threads. However, `head` and `writer_phase` are only written by the enqueuing thread, while `tail` and `reader_phase` are only written by the dequeuing thread.

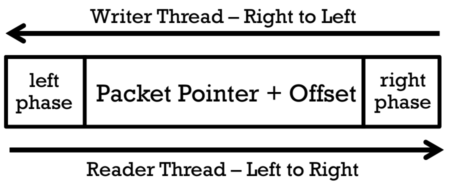


Figure 5: Sketch of a record in the bad packet queue

To write a packet, the enqueuing (general) thread first checks if it does not overwrite a packet that was not dequeued. This is done by checking whether `reader_phase` $=$ `writer_phase` or `tail` $>$ `head`. It can be easily proved that both cases of this condition imply that the dequeuing thread is at most Q packets away from the enqueuing thread, where Q is the length of the array.[3]

If the queue is full, then a packet is not enqueued; the general thread can either process this packet or stall and retry later. Otherwise, the enqueuing thread writes into the queue record *from right to left*. It first writes `writer_phase` in the `right_phase` field, then it writes the pointer and offset, and finally it writes `writer_phase` in the `left_phase` field. If the written entry is the last entry of the array, the thread flips the `writer_phase` bit and writes the next entry to the array beginning.

The dequeuing thread reads from the queue record *in the opposite direction*. It first reads the `left_phase` field, then it reads the pointer and offset, and finally the `right_phase` field. We distinguish between three possible cases when reading a record from the queue:

1. `left_phase` $\neq$ `right_phase`. This implies that the record is now being written. The dequeuing thread should stall shortly and retry reading the entry.

2. `left_phase` $=$ `right_phase` $\neq$ `reader_phase`. This implies that the queue is empty (the dequeuing thread

[3]Proof outline: Assume that the condition does not hold. The absolute index of the first packet to overwrite another packet differs by exactly Q from the index of the next packet to read. This implies that `reader_phase` $\neq$ `writer_phase` and `tail` $=$ `head` (namely, `tail` $\not>$ `head`), and hence a contradiction.

should do nothing, and try to dequeue a packet from the next queue).

3. `left_phase` = `right_phase` = `reader_phase`. This implies that the record is valid for reading and processing. The dequeuing thread starts processing the payload of the packet, after skipping its first *offset* bytes. [4]

When a record is read successfully from the last entry of the array, the dequeuing thread flips `reader_phase` and continues dequeuing from the first entry.

Similar mechanism is applied to all other queues in the system (except for the input packet queues of the dedicated threads, which use `test&set` locks to allow packet *stealing*, as discussed in Section 5.1). Our simulations show that even under worst-case traffic, the overhead of this communication mechanism does not exceed 0.98% degradation in system throughput.

5.3 Thread Allocation Scheme

The number of threads allocated to handle heavy packets depends on the exact setup in which the NIDS/NIPS system works. Specifically, we differentiate between two extreme cases: a *no-drop setup* in which no packets are dropped by the NIDS, and a *full drop setup* in which all heavy packets are considered malicious and are dropped immediately. In between, we also consider a *limited drop setup* that allows dropping heavy packets when their percentage exceeds a certain threshold. It is important to notice that in all setups non-heavy packets are not dropped.

The no-drop setup is adequate for an NIDS that only alerts upon an attack. On the other hand, the limited-drop and full-drop setups are used in NIPS; limited-drop is suitable when the security administrator wishes to invest only limited resources in the process, to monitor sporadic attacks over the network and to deal with false malicious traffic.

When deciding how many threads to allocate in each setup, our goal is to maximize *goodput* assuming that the system is balanced. Notice that this goal coincides with maximizing the overall *throughput* of the system. Determining the number of dedicated threads to allocate in the full-drop setup is trivial: no dedicated threads should be allocated and heavy packets are dropped immediately upon their identification. As for limited-drop setup, we need the minimal number of threads to handle only a small fraction of the heavy packets. This can be done either by using only a single thread or all hardware threads in a single core, depending on the attack's characteristic and the multi-core architecture.

A more challenging task is to determine the number of dedicated threads in the no-drop setup. This number depends on the parameters summarized in Table 1.

Naturally, the number of dedicated threads grows along with the fraction of the heavy packets. In addition, we take into account the performance of the two algorithms and consider how they perform while handling either only heavy packets or only normal packets. It is important to notice that the throughput of the algorithm usually depends on r—the fraction of heavy packets it handles. For brevity, our model does not consider these exact numbers and uses only the two extreme points.

[4]Since the AC DFA was in its root state, when scanning the byte in the offset position, it implies that patterns cannot begin before that byte and end afterwards. Hence, it is safe to skip the scanning up until this byte [9]

Parameter	Description
r	The fraction of heavy packets out of all traffic
$AlgG_h$	The throughput of the general threads' algorithm running solely on heavy packets
$AlgD_h$	The throughput of the dedicated threads' algorithm running solely on heavy packets
$AlgG_n$	The throughput of the general threads' algorithm running solely on normal packets
$AlgD_n$	The throughput of the dedicated threads' algorithm running solely on normal packets
N	The number of available threads

Table 1: The parameters used to determine the number of dedicated threads (no-drop setting). All throughput values are given for a single thread.

Let β be the ratio between $AlgG_n$ and $AlgD_h$ (see Table 1), and let T be the system's throughput when the entire traffic is normal (that is, no heavy packets) and all threads are general. Thus, when the traffic has a fraction r of heavy packets, the best allocation scheme can achieve a throughput of

$$T\left((1-r)+\frac{r}{\beta}\right).$$

This throughput is achieved when the number of dedicated threads is

$$D_f = N\frac{r/\beta}{1-r+r/\beta}.$$

Notice that D_f is not an integer, and therefore, it should be rounded to provide the required number of threads. According to the attack type and the multi-core architecture on which MCA^2 is running, one can choose to round D_f so that all hardware threads of the same core would be either dedicated or general. We denote this rounded number by D and it is the output of our model.

Note that we have presented a simplified model for deciding how many dedicated threads to allocate. A more accurate model supports additional aspects. For example, one can use the algorithm of the general threads by a dedicated thread for lower rates of r. This is beneficial when $AlgG_n$ is significantly larger than $AlgD_n$, and since r is too small, most of the packets handled by the dedicated threads are not heavy; second, one might consider the packets' detection overhead. If it is too large as compared to the gain in throughput, the system should not allocate any dedicated thread. Moreover, one can optimize the number of packets that are exchanged between dedicated and general threads, by taking into account the load balancing among them. Our experimental results with the thread allocation scheme are shown in Section 8.2.4. Finally, a useful practice is to limit the maximal value of D_f to preserve a share of general threads under any attack intensity.

5.4 Flow Affinity

NIDS/NIPS systems are required sometimes to preserve flow affinity; namely, all packets from the same flow should be processed in the same core (e.g., to communicate the results of different modules of the system, and to keep inter-packet context). In that case, MCA^2 marks *heavy flows* instead of heavy packets. We note that significant research effort has been devoted to flow affinity in multi-core environment (cf. [16, 17]). MCA^2 can be combined with any

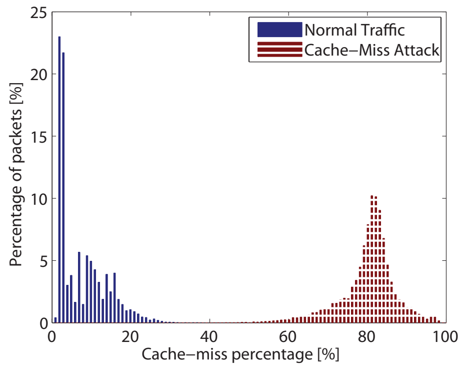


Figure 6: Distribution of cache-misses under normal traffic and under attack.

method that provides flow affinity, yet to divert heavy flows to dedicated cores and thus reduce their effect on legitimate flows.

More specifically, given a system with a packet dispatcher that sets flow affinity, we add a *preliminary* data structure for fast determination of whether a flow was marked heavy or not. This data structure supports insertion of flows and deletion of flows when they either become inactive or when they recover (namely, when they stop behaving maliciously for a certain amount of time/packets). Due to their compact memory footprint and fast lookup time, we suggest using either a counting bloom filter [14] or a hash table with timestamps, so that outdated records can be easily removed.

6. MCA^2 FOR CACHE-MISS ATTACKS

In this section we present an algorithm for detecting heavy packets in AC DFA complexity attack. Cache-miss attacks are characterized by *a large number of different state machine traversals that cause cache-misses* (as compared with the normal system operations), as clearly illustrated in Fig. 6: On normal traffic,[5] the system has a very low average cache-miss per packet ratio of around 10%, where under a cache-miss attack it is around 80%, leaving an evident margin with a factor of more than 8.

A direct way to measure the value of this parameter is to actually monitor the system cache-misses through the hardware counters. However, this approach is processor-dependent and may not be applicable in our case (either due to lack of appropriate interface or due to the overhead that such monitoring introduces). A more efficient way that was used when implementing MCA^2 for these attacks, is to approximate the cache-miss upon each input symbol based on the underlying AC DFA itself. This is done by studying the set of states with training traces, ordering the states by the number of visits, and marking as *common states* the most visited states of the DFA when processing the normal packets as discussed below.

An important parameter that should be chosen is the number of common states (that is, in the ordered list of states, what is the rank above which a state is marked as common). Recall that upon normal traffic, DPI is performed with a full-matrix encoding, where each state is represented

[5]Namely, real-life web traffic, see Section 8 for discussion on this trace.

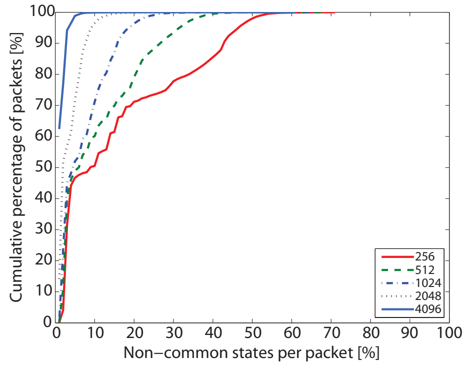


Figure 7: CDF of the percentage of normal traffic packets by their non-common states ratio for different numbers of common states, 256, 512,. . . , 4096.

Non-Heavy packets	Number of common states				
	256	512	1024	2048	4096
99.0%	**53**	**38**	**25**	**15**	**6**
97.5%	50	35	22	11	5
95.0%	47	32	19	10	4

Table 2: The non-common states ratio for different percentage of non-heavy packets and different numbers of common states.

by a row in a matrix of size $256 \log |S| \approx 1KB$ (for Snort's AC DFA). One may suggest to keep the number of common states such that they all fit in the available cache bank. We state that $1KB$ is an overestimate, and in fact, many more states may fit in the cache without causing performance degradation. The reason is that only few outgoing transitions for each state are actually accessed, implying that only a small part of the state's row is actually loaded into the cache.

Prior to determining the number of common states, we explain the interplay between this number and the fraction of the packets that are eventually considered heavy. For each packet, let the *non-common states ratio* be the ratio between the number of non-common states visits and the overall number of DFA traversals per packet. A packet is marked heavy if its non-common states ratio exceeds a certain threshold. Our goal is that under well-behaved traffic the number of packets marked heavy would be very small, as it corresponds to false identifications. Naturally, as the number of common states increases, the number of potentially heavy packets decreases, and therefore the threshold may be increased.

Fig. 7 considers a normal traffic and depicts a CDF showing the percentage of packets by their non-common states ratio. As one can see, this percentage grows quickly as the number of common states increases. Table 2 shows the correlation between the non-common states ratio and the percentage of non-heavy packets. Since the normal traffic contains almost no heavy packets, we set the threshold so that only 1% of the normal packets are marked heavy. These thresholds, for each number of common states, are marked in bold in Table 2.

Using the above thresholds and the thread-allocation scheme (see Section 5.3), we ran experiments in which we mea-

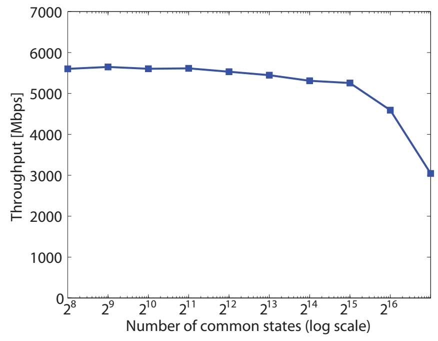


Figure 8: The total system throughput for a different number of common states, under an attack of intensity 33%.

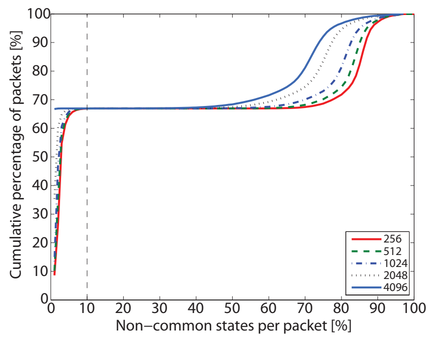


Figure 9: CDF of the percentage of mild attack (33%) traffic packets by their non-common states ratio.

sured the system throughput for different numbers of common states. Fig. 8 depicts these measurements under mild attack, in which 33% of the packets are malicious. Note that there is no significant difference between set sizes below 8 KB. We have repeated these experiments for various attacks scenarios and determined that the highest throughput is achieved when the number of common states is 1,024. This translates to a threshold of 25% traversals to non-common states to mark a packet as heavy. Finally, we note that under mild attack of 33% of the traffic, any threshold above 10% suffices; this is clearly evident by the CDF in Fig. 9.

7. MCA^2 FOR ACTIVE-STATES ATTACKS

In addition to exact-string matching, contemporary DPI engines usually support regular expression matching. However, unlike exact-string matching, a set of regular expressions is not represented in a DFA, due to the infamous *state blow-up* phenomenon, which implies that these DFAs have prohibitively large memory footprint. A common approach is to replace the DFA with a non-deterministic finite automaton (NFA) [24].[6] When using an NFA, the matching algorithm keeps a vector of active states and for each input symbol, it computes the next state according to all active states. Naturally, this makes NFA significantly less efficient (namely, when k states are active at the same time on average, an NFA performs k times slower than a DFA).

Becchi et al. [5] proposed a hybrid approach that combines NFA with DFA. Therefore, they have noticed that in the process of transforming an NFA to a corresponding DFA, the states that cause a space blow-up can be easily determined. They interrupt the transformation of these specific states by keeping them non-deterministic, such that they connect two deterministic automatons. This process produces a hybrid finite automaton (*Hybrid-FA*) that consists of a *head DFA*, which is a regular DFA, though some of its leaves are "border states"—states that are non-deterministic and lead to another DFA, named *tail DFA*. As border states are non-deterministic, reaching such a state during traversal requires keeping more than one active state at a time. Thus, this data structure trades space for time by letting more than one active state at a time, but doing so only when space blow-up is actually prevented.

As discussed also in [5], on certain inputs, the average number of active states may be potentially higher by a factor of 30 than on an average case input. This gap reveals a potential complexity DoS attack on a system that uses the algorithm. To illustrate the attack we used the Hybrid-FA code [3] (provided by the authors of [5]), along with a set of regular expressions taken from the Bro NIDS (which was also provided in [3]). We carefully crafted one malicious packet that causes activation of at most eight active states simultaneously.[7] To simulate an attack, we used a trace with 90% legitimate web traffic and only 10% malicious traffic. Our experiment on this trace shows a slowdown of 83% in goodput, implying that the system is very vulnerable even under very mild attack.

We have replaced the pattern matching module, described in Section 6, with the Hybrid-FA pattern matching code [3] to combine MCA^2 with Hybrid-FA. To identify heavy packets in this case we used a window of 40 bytes in which we examined the average number of active states (Hybrid-FA code keeps a vector of active states, therefore it is simple to poll its size at any time). If during packet processing, the average number of active states in a 40 bytes window exceeds a certain threshold, then the packet is marked as heavy. If the MCA^2 system is in alert mode it can either drop the packet or send it to a dedicated core, according to the selected configuration.

Fig. 10 shows the distribution of the highest average number of active states per 40 bytes window, per packet, under traffic that contains 90% real-life web packets and 10% attack packets: while normal packets do not exceed 3.1 active states per window on average, our attack packets have maximal average of 7.1. Thus, one can easily differentiate between legitimate and malicious traffic. In Section 8.3 we show the results of our experiments with Hybrid-FA and MCA^2.

[6]Another common practice, used by Snort, is to extract exact-string anchors from the regular expressions and use a DFA to match these anchors. If an anchor is matched, the regular expression engine is applied on the packet for matching only the relevant expression. This reduces the problem to the exact-string matching problem, which was discussed in Section 6.

[7]To create the malicious packet we selected prefixes of regular expressions that contain a "dot-star" in them. We only got eight active states as this is the limit of the specific pattern-set we used, and also since it is enough to illustrate the DoS attack. Many different such packets can be crafted, for convenience we use one example.

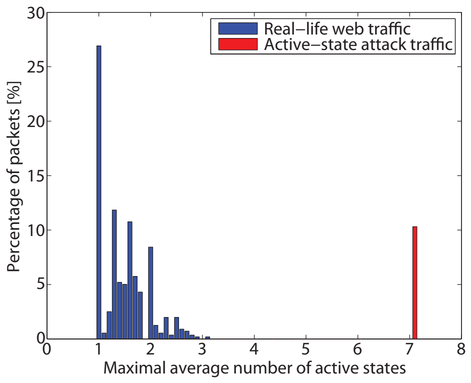


Figure 10: Distribution of maximal average number of active states per 40 bytes window, per packet, under real-life web traffic and under attack.

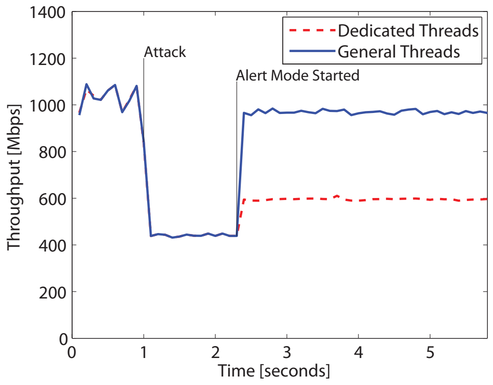


Figure 11: Average throughput per thread over time, when a sudden *cache-miss attack* happens. The system uses eight threads, and when alert mode starts, two of them become dedicated threads.

8. EXPERIMENTAL RESULTS

8.1 Experimental Environment

We use a system with Intel Sandybridge Core i7 2600 CPU, quad-core, each core has two hardware threads, 32 KB L1 data cache (per core), 256 KB L2 cache (per core), and 8 MB L3 cache (shared among cores). The system runs Linux Ubuntu 11.10. Since hardware threads of the same core share the L1 and L2 caches, we have treated them together, as illustrated in Fig. 4. Thread affinity was used to associate threads to cores. In such a way, dedicated threads share the same core, and do not mix with the general threads.

Two web traffic traces are used with size of 145 MB each. These traces contain traffic from randomly selected URLs taken from Alexa top web-sites list [2]. One of these traces is used as our *real-life web traffic* trace and the other trace is used as a training set for determining the common states set for cache-miss attack, as described in Section 6. To simulate a cache-miss attack, we created several *cache-miss attack traffic* traces. These traces contain both normal packets, which cause few cache misses, and flows of specific malicious packets. The latter contains a concatenated list of all patterns from the pattern-set, in order to make as many cache misses as possible. These traces contain different volume of such malicious packets, corresponding to the intensity of the attack. We use cache-miss attack traffic traces with a growing rate of malicious packets, from 0% to 100% (that is, the 20% attack intensity trace contains 20% malicious packets and 80% normal packets). Note that these traces were also used for Fig. 1. To simulate an active-state attack we created another set of traces. These traces also mix normal packets with malicious traffic, with a growing rate of malicious traffic. Adversarial packets in these traces are clones of the malicious packet described in Section 7. The intensity of attack also varies from 0% to 100%.

8.2 Cache-Miss Attack Simulation Results

8.2.1 Goodput

Fig. 1 depicts the *goodput* of MCA^2 when processing the different traffic traces. Note that, as the attack intensity increases, the goodput decreases, since the the non-malicious traffic occupies smaller portion of the entire traffic. In addition, the penalty of identifying heavy packets (including initial inspection, packet loading, counters initialization, etc.) becomes more significant. Upon an attack we gain a goodput improvement of 67%–102%, as compared to the full-matrix implementation, which do not take cache-miss attack into consideration.

Fig. 1 also depicts the *full-drop setup*, as described in Section 5.3, and also shows a significant improvements in goodput.

Finally, we ran MCA^2 on our web traffic traces. As expected, alert-mode was never activated, and a throughput of 8219 Mbps was obtained on average. This is statistically the same as a light (yet vulnerable) implementation with no MCA^2 at all.

8.2.2 Accuracy

To analyze the *heavy packet isolation*, we first examine the isolation results of our mechanism on the *real-life web traffic* and on the *cache-miss attack* traces. According to the analysis in Section 6, the system would ideally identify 99% of the packets in the first trace as non-heavy. However, as the mechanism estimates of the non-common states rate based on the packet's prefix, it is accurate. Our tests show that the actual rate of packets in the *real-life web traffic* trace that are identified as non-heavy is 96%. These 4% packets would be transferred to a dedicated thread, although being legitimate, and suffer some slowdown. Nevertheless, without MCA^2, these packets would suffer even lower throughput under such attacks. In the *cache-miss attack* traces we know exactly which packet is heavy and can measure precise values for false identification rate. Neither one of the configurations from Section 6 falsely classified malicious packet as normal in more than 0.001% of the trace. We see that our detection mechanism provides an accurate isolation.

8.2.3 Identifying Cache-Miss Attacks

In order to experiment system behavior upon a sudden *cache-miss attack*, we have created a trace that consists of the web traffic trace in which, at some point of time, 33% of the traffic is a cache-miss attack traffic. We set the time interval for checking the rate of heavy packets to *one second*

r	D_F	Optimal Thread Allocation
0	0	0
0.2	0.75	2
0.33	1.26	2
0.5	2.35	4
1	8	8

Table 3: Validation of the thread allocation model of Section 5.3.

for this experiment. We measure the approximate throughput of each thread per intervals of 100ms each. Then, we average the timing per interval for all general threads and dedicated threads. Fig. 11 depicts the result of this experiment. The system starts when all its eight threads are 'general threads'. At the beginning, from time 0 to time 1, input traffic is regular web traffic. Then, at time 1, attack packets begin to arrive, lowering threads throughput by a ratio of about 68%. After a second (time 2), the system identifies the attack and switches to *alert mode*. It sets a pair of threads that belong to a single core as 'dedicated threads' with the compressed matching algorithm, optimized for handling heavy packets. General threads now transfer heavy packets to dedicated threads and therefore are much less affected by them, preserving high *goodput* (general threads still have to scan the first few bytes of heavy packets in order to classify them as heavy. This causes the slight relative slowdown in their throughput as compared to their performance before the attack has started).

8.2.4 Thread Allocation

We validate our thread allocation model as described in Section 5.3. First, we determine the value of β based on our experiments, where all the threads are running either the full-matrix or the compressed implementation. Specifically, we got that $AlgG_n = 1040.6$ Mbps (per thread) and $AlgD_h = 444.3$ Mbps (per thread), therefore $\beta = 2.34$. Table 3 presents the fractional number D_f obtained by our model, and compares it with the allocation that achieves the highest throughput in our experiments (that is, for each value of r we tried all possible thread allocations and picked the optimal ones). Since, in our system, threads are allocated in pairs, all the optimal experimental results coincide with the model's calculations.

8.3 Active-State Attack Simulation Results

Fig. 12 depicts the goodput of Hybrid-FA when processing the different traffic traces, and the goodput of Hybrid-FA when combined with MCA^2 full-drop setup. The MCA^2 full-drop setup provides significant improvements in goodput (as in Section 8.2.1, goodput decreases as attack intensity increases as the the non-malicious traffic occupies smaller portion of the entire traffic).

Considering the other possible configurations of MCA^2, unlike for cache-miss attack, we do not have an off-the-shelf algorithm that can be used on the dedicated cores to boost performance on heavy packets in an active-state attack. The designing of such an algorithm is left for future research.

In terms of accuracy of isolation, MCA^2 isolates our attack traffic from the legitimate traffic. Nevertheless, attacker can create lighter packets that might go under the radar, however such packets are bound to induce much smaller slow-

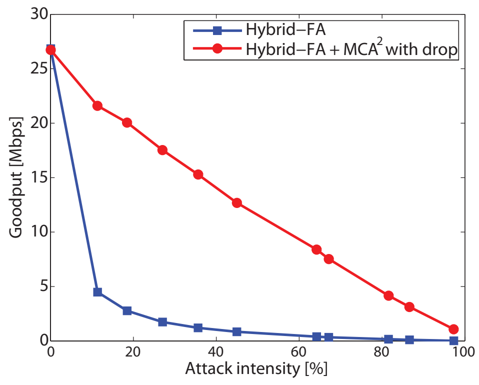


Figure 12: Goodput of Hybrid-FA and of Hybrid-FA with MCA^2 full-drop setup, facing different intensity of active-state attack.

down, if any. We also note that in different legitimate traffic, some packets may be identified as heavy (if they use more active states), but we did not find such traffic in our traces.

9. CONCLUSION

In this paper, we expose a known security hole, the complexity attack, demonstrate its effectiveness, and provide a system solution to mitigate the attack. In the demonstrated complexity attack, negligible effort on the attacker side results in a substantial effort (namely, resource consumption) on the target system. This is a security hole calling for a DDoS attack.

A simple method to mitigate a complexity attack is to throw more computing resources into the system. Obviously, often this is a prohibitively expensive and wasteful approach. An alternative approach is to design algorithms that are efficient in processing malicious packets (e.g., compressing states in the Aho Corasick DPI algorithm). Unfortunately, in many cases an algorithm that works well on malicious packets performs worse on normal packets, thus again requiring more computing resources at normal times. Our MCA^2 architecture provides a method to enjoy from both, special treatment is given to suspicious (a.k.a. heavy) packets in dedicated cores with an optimized algorithm designed for the heavy, while treating the rest of the traffic in the other cores with the best algorithm for the average traffic. This architecture provides several advantages, first the overall system throughput is increased; second, treating heavy packets on the side with dedicated cores isolates the normal traffic from the suspicious traffic; third, we can choose different treatments for heavy packets, without affecting the normal packets; and finally the system may shift gears and decide how many resources to allocate for the processing of heavy packets.

MCA^2 architecture is a general framework to deal with different kinds of complexity attacks. While in this paper we have demonstrated it on one domain—Deep Packet Inspection in NIDS—we are looking to apply the framework for the mitigation of other attacks.

Acknowledgments

This work was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n° 259085.

10. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[2] Alexa: The web information company, Dec 2011. `http://www.alexa.com/topsites`.

[3] M. Becchi. Regular expression processor. `http://regex.wustl.edu`.

[4] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ACM/IEEE ANCS*, pages 145–154, 2007.

[5] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *ACM CoNEXT*, pages 1:1–1:12, December 2007.

[6] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *ACM/IEEE ANCS*, pages 30–39, 2009.

[7] U. Ben-Porat, A. Bremler-Barr, H. Levy, and B. Plattner. On the Vulnerability of the Proportional Fairness Scheduler to Retransmission Attacks. In *IEEE INFOCOM*, pages 1431–1439, Apr. 2011.

[8] A. Bremler-Barr, Y. Harchol, and D. Hay. Space-time tradeoffs in software-based deep packet inspection. In *IEEE HPSR*, 2011.

[9] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic state machine compression for scalable pattern matching. In *INFOCOM*, pages 659–667, 2010.

[10] The Bro Network Security Monitor. `http://bro-ids.org`.

[11] C. Castelluccia, E. Mykletun, and G. Tsudik. Improving Secure Server Performance by Re-balancing SSL/TLS Handshakes. In *USENIX Security Symposium*, Apr. 2005.

[12] W. Cong, J. Morris, and W. Xiaojun. High performance deep packet inspection on multi-core platform. In *IEEE BNMT*, pages 619 –622, Oct. 2009.

[13] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, pages 29–44, 2003.

[14] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[15] M. Fisk and G. Varghese. Fast Content-Based Packet Handling for Intrusion Detection. Technical report, University of California at San Diego, CA, USA, 2001.

[16] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *ACM IMC*, pages 218–224, 2010.

[17] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding. A scalable multithreaded l7-filter design for multi-core servers. In *ACM/IEEE ANCS*, pages 60–68, 2008.

[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 3rd edition, 2007.

[19] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ACM/IEEE ANCS*, pages 155–164, 2007.

[20] W. Lee, J. a. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *RAID*, pages 252–273, Octover 2002.

[21] T. Liu, Y. Sun, and L. Guo. Fast and memory-efficient traffic classification with deep packet inspection in CMP architecture. In *IEEE NAS*, pages 208–217, 2010.

[22] M. D. McIlroy. A Killer Adversary for Quicksort. *Software–Practice and Experience*, pages 341–344, 1999.

[23] T. Nelms and M. Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *ACM/IEEE ANCS*, pages 21:1–21:11, 2010.

[24] PCRE - perl compatible regular expressions. `http://www.pcre.org`.

[25] T. Peters. Algorithmic Complexity Attack on Python, May 2003. `http://mail.python.org/pipermail/python-dev/2003-May/035916.html`.

[26] R. Smith, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *ACM ACSAC*, Dec. 2006.

[27] Snort: The Open Source Network Intrusion Detection System. `http://www.snort.org`.

[28] R. Sommer, V. Paxson, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurr. Comput. : Pract. Exper.*, 21:1255–1279, July 2009.

[29] Sony Ericsson Latest Victim of SQL Injection Attack, 2011. `http://www.eweek.com/c/a/Security/Sony-Data-Breach-Was-Camouflaged-by-Anonymous-DDoS-Attack-807651`.

[30] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM*, 2004.

[31] O. Villa, D. P. Scarpazza, and F. Petrini. Accelerating real-time string searching with multicore processors. *IEEE Computer*, 41(4):42–50, April 2008.

[32] J. Wang, H. Cheng, B. Hua, and X. Tang. Practice of parallelizing network applications on multi-core architectures. In *ICS*, pages 204–213, 2009.

[33] F. Weimer. Algorithmic Complexity Attacks and the Linux Networking Code. `http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html`.

[34] B. Xu, K. Zheng, Y. Xue, and J. Li. Scalable string matching framework enhanced by pattern clustering. *Ubiquitous Computing and Communication Journal*, 5(2):16–26, June 2010.

[35] K. Zheng, H. Lu, and E. Nahum. Scalable pattern matching on multicore platform via dynamic differentiated distributed detection. *IEEE Trans. on Comput.*, 60:346–359, 2011.