Making DPI Engines Resilient to Algorithmic Complexity Attacks

Yehuda Afek, *Member, IEEE,* Anat Bremler-Barr, *Member, IEEE,*
Yotam Harchol, *Member, IEEE,* David Hay, *Member, IEEE,* Yaron Koral, *Member, IEEE*

***Abstract*—This paper starts by demonstrating the vulnerability of Deep Packet Inspection (DPI) mechanisms, which are at the core of security devices, to algorithmic complexity denial of service attacks, thus exposing a weakness in the first line of defense of enterprise networks and clouds. A system and a multi-core architecture to defend from these algorithmic complexity attacks is presented in the second part of the paper. The integration of this system with two different DPI engines is demonstrated and discussed. The vulnerability is exposed by showing how a simple low bandwidth cache-miss attack takes down the Aho-Corasick (AC) pattern matching algorithm that lies at the heart of most DPI engines. As a first step in the mitigation of the attack, we have developed a compressed variant of the AC algorithm that improves the worst case performance (under an attack). Still, under normal traffic its running-time is worse than classical AC implementations. To overcome this problem, we introduce MCA^2—Multi-Core Architecture to Mitigate Complexity Attacks, which dynamically combines the classical AC algorithm with our compressed implementation, to provide a robust solution to mitigate this cache-miss attack. We demonstrate the effectiveness of our architecture by examining cache-miss algorithmic complexity attacks against DPI engines and show a goodput boost of up to 73%. Finally, we show that our architecture may be generalized to provide a principal solution to a wide variety of algorithmic complexity attacks.**

Keywords*—*Deep Packet Inspection, Multi-core, Complexity attack, DoS

I. Introduction

Security devices, such as Network Intrusion Detection or Prevention Systems (NIDS or NIPS), are the front defense line against cyber attacks over the Internet. A central component of NIDS/NIPS is a *Deep Packet Inspection* (DPI) engine, in which the payload of the messages is inspected to detect predefined signatures of malicious attacks.

Being such a central component, DPI engines may serve as a preferred target for denial-of-service attacks. In recent years, such attacks are part of a trend of a two-phase *combined attack* on security devices: the attackers first neutralize the security device (e.g., by overwhelming it with traffic), and then, when the security device has been knocked down, attack the assets it was protecting. For example, an attack on SONY in 2011 combined a DDoS attack with credit cards theft [1]. Such combined attacks usually have a different effect on NIDS and NIPS. In NIDS, where the device works in stealth-mode (namely, monitoring traffic and alerting when malicious activity is detected), these 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 inspect the packets on their critical path, these attacks might either force to drop legitimate traffic, thereby, practically causing a denial of service on the servers they protect, or simply let all traffic go through without being scanned. For example, Bro [2] and Snort [3], two popular open source examples of such systems, are both vulnerable to this kind of attack [4].

The attacks considered in this paper are *complexity attacks*, which exploit the gap between the amount of resources the system requires when processing normal packets and when processing carefully-crafted packets that consume drastically more resources (either computing, memory, cache, or other resource). These crafted packets, which we call *heavy* packets, are on the one hand easy to construct, while on the other hand, require very intensive processing from the system. This implies that with a little effort on the attacker side, the target system spends a lot of effort and is bound to lose.

We start by demonstrating the vulnerability of Snort [3] to a cache-miss complexity attack and, specifically, analyze its DPI engine vulnerability to such attacks. The default DPI implementation of Snort is derived from the full matrix encoding of the Aho-Corasick automaton [5] (described in Section II). This implementation, denoted as Full Matrix AC, is believed to have a deterministic behavior regardless of the input, as each scan makes a single memory access per (any) input byte. Our experiment show that such an attack on Snort causes a performance degradation of up to 79%.

The negative results of this experiment call for the construction of a DPI algorithm that is not susceptible to the cache-miss complexity attack. We have considered a wide range of algorithms and techniques that minimize the size of the underlying DFA and devised the Compressed-AC algorithm, which has an almost constant throughput, regardless of the input packets. Nevertheless, this comes with a price: under non-attack traffic, the throughput of the Compressed-

Partial and Preliminary versions of parts of this paper appeared in IEEE HPSR 2011 and in ACM/IEEE ANCS 2012. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259085, and by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11).

Y. Afek is with the Blavatnik School of Computer Science, Tel Aviv University.

A. Bremler-Barr is with the Efi Arazi School of Computer Science, Interdisciplinary Center Herzliya.

Y. Harchol and D. Hay are with the Rachel and Selim Benin School of Computer Science and Engineering, The Hebrew University of Jerusalem.

Y. Koral is with the Department of Computer Science, Princeton University.

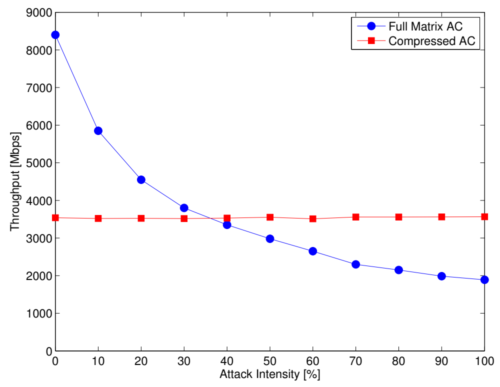


Fig. 1: Throughput of the FULL MATRIX AC and COMPRESSED-AC implementations, under traffic with different intensities of cache-miss attacks, when CPU is fully utilized.

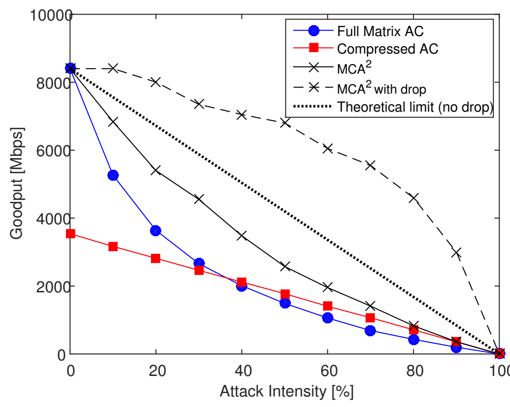


Fig. 2: The goodput of MCA^2 for different attack intensities, when CPU is fully utilized. MCA^2 with no drops maintains a balance between all cores.

AC algorithm is only 42% of the throughput of the FULL MATRIX AC (see Fig. 1). [1]

Notice that now we have two algorithms that work well under different conditions. Naturally, we would like to build a system that would switch the two algorithms in a dynamic way, using the full-matrix algorithm for normal packets and the compressed variant for heavy packets. To pursue this goal we consider multi-core architectures, which are common in every processor nowadays. Our architecture, named MCA^2—a general Multi-Core Architecture to Mitigate a variety of Complexity Attacks, essentially detects heavy packets and diverts them to a dedicated fraction of the cores. Legitimate traffic is handled by the remaining cores without much interruption and delays caused by heavy packets, and thus have higher throughput and lower latency. Specifically, to mitigate cache-miss attacks, when the fraction of heavy packets in the traffic is above a certain threshold, the system shifts gears and loads the COMPRESSED-AC algorithm to a fraction of cores that are allocated to handle heavy packets, while the rest of the cores continue to process (using FULL MATRIX AC) only normal traffic and to detect heavy packets; each subsequent *heavy packet*, and if required, the corresponding flow it belongs to, is moved to one of the dedicated cores. This process isolates the effect of heavy packets and protects the low level caches of the non-dedicated cores from being trashed.

The main performance measure we use to analyze the system is the goodput of the system at different attack intensities; namely, the maximum number of bytes per second of non malicious traffic that the system can process. Specifically, to measure the goodput at attack intensity f, we prepare a traffic trace with a fraction f of its size (bytes) consumed by bad (malicious) packets and measuring the maximum speed at which the system can process this trace. The goodput of the system is $(1-f)$ of that speed. Our experimental results are summarized in Fig. 2; for example, at 50% attack intensity, half of the bandwidth is consumed by bad packets.

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 and idealized implementation that knows in advance the attack intensity at any given time, and responses by choosing the best of the previous implementations at any given time; the goodput boost over such an implementation is up to 73% (see Figure 2 at 30% attack intensity).

Taking a step back to look at the wider picture, we observe that there is a wide range of possible complexity attacks over different DPI algorithms. Having that in mind, we generalize our MCA^2 architecture to mitigate any complexity attack that has 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 consumes very few resources.
3) Packets can be efficiently moved between the different cores.
4) (A 'nice to have', not a required property) There is a special method that handles heavy packets more efficiently than the method used for normal packets.[2]

To demonstrate the generality of the MCA^2 architecture we consider two additional examples that follow the above properties: *active states explosion attack* on regular expression detection engine and *force construction attack* on Bro IDS regular expression detection engine.

Specifically, *Active states explosion attacks* target DPI engines that match regular expressions. In such a matching, software implementations of Nondeterministic Finite Automata (NFA) are often used, with the unique property is that several states can be active simultaneously, forcing the automaton to take multiple transitions concurrently when inspecting a single input character. In this paper, we realized the *active states explosion attack* on the state-of-the-art Hybrid-FA data structure [6] , which provides efficient software implementation of

[1]See Section X for details on the experimental environment.

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

NFA. Even a light attack that takes only 12% of the bandwidth drops the goodput of the system to about 16%. We then show how the MCA^2 system in a full-drop setup can mitigate such an attack: our experiments show that under a mild *active states attack*, the system's goodput is increased by a factor of five fold.

An additional example of a complexity attack on an IDS is the *force construction attack*, which targets NIDSs that take a lazy approach and build their data structures on the fly (e.g., to cope with their enormous size). A prime example is the Bro IDS, which dynamically constructs only the parts of the DFA it actually uses, while the rest of the states are represented by an NFA. As normal traffic uses only a small part of the DFA, a simple complexity attack, which we have realized on a real-life Bro setup, forces the IDS to construct a large portion of the DFA and, by that, degrades the performance significantly. As noted above, MCA^2 is useful to mitigate also this attack.

The rest of the paper is organized as follows. In Section II we provide the necessary background on complexity attacks and DPI. Section III discusses related work. Section IV presents the cache-miss attack and its impact on Snort. In Section V we describe our COMPRESSED-AC algorithm. Section VI describes the MCA^2 architecture. In Sections VII, VIII and IX we demonstrate how MCA^2 mitigates cache-miss attacks, active-states attacks and force-construction attacks, respectively. Our experimental results appear in Section X. Finally, we conclude in Section XI.

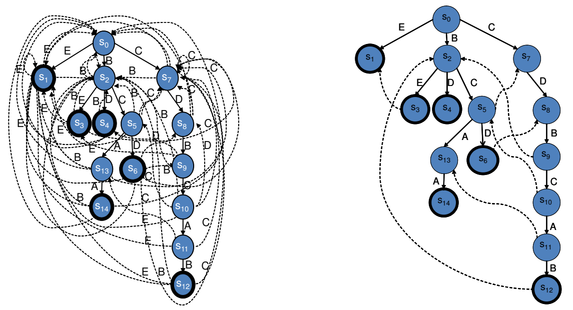
II. BACKGROUND

DPI is a major component in most security tools, which usually performs pattern matching to detect signatures of malicious traffic. Two major types of pattern matching are 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.

We mostly focus on the exact matching algorithms, which use a DFA. A DFA is a five-tuple $\langle S, \Sigma, \delta, s_0, F\rangle$, 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. 3(a) depicts the AC DFA for the pattern-set {E,BE,BD,BCD,CDBCAB,BCAA}.

In today's security tools, AC DFA's are huge—e.g., Snort's AC DFA has $77,182$ states for $6,422$ patterns—raising the question of how to store it efficiently in memory. The alternatives naturally trade memory space with execution time. In addition, most security tools (including Snort) divide their patterns to several sets, according to the type of traffic.

Snort uses a full-matrix encoding for its AC DFAs as presented in [5]. In this representation, transitions are stored in a two-dimensional array with $|S|$ rows and $|\Sigma|$ columns. An entry at position (i, j) holds the value of $\delta(s_i, j)$, implying that the number of bits in each entry is at least $\log_2 |S|$. In the typical case, when the input is inspected one byte at a time, $|\Sigma| = 256$, resulting in overall memory footprint of $256|S| \log_2 |S|$. 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 that reveals directly the next state.



(a) Full Matrix representation. (b) Failure-path based representation.

Fig. 3: Examples of Aho-Corasick automaton representations for {E, BE, BD, BCD, CDBCAB, BCAA}.

An alternative approach to represent the Aho-Corasick automaton is using a trie that contains transitions from each state in depth d to its successors in depth $d+1$, called *forward transitions*. That is, a state may not have a transition for each possible input symbol, but only for those symbols that actually transition to depth $d + 1$ in the trie. In addition, every state has a single *failure transition*, that is taken when no suitable forward transition exists: let label(s) be the word that leads from s_0 to s. The failure transition of state s is to state s' such that label(s') is the longest proper suffix of label(s) among all DFA states. Note that label$(s_0) = \varepsilon$ (that is, the empty word) is a suffix of all other labels, and therefore, the failure transitions are properly defined. The longest failure path (namely, a path that consists of failure transitions only) that starts at state s is of length at most depth(s). This, in turn, implies that the total number of transitions (both forward and failure transitions) is at most twice as the number of inspected symbols. Fig. 3(b) depicts a failure-transitions–based representation of the Aho-Corasick automaton in Fig. 3(a), where the solid edges are forward transitions and the dotted edges are failure transitions (for clarity, failure transitions to s_0 are omitted). In Section V we present our compression techniques that are based on the failure-transitions–based representation of AC. Our system later uses these techniques to mitigate complexity attacks.

III. RELATED WORK

Crosby and Wallach were among the first to demonstrate a complexity attack on the commonly-used *Open Hash* data structure [7]: an attacker designs an input that requires $O(n)$ elementary operations per insertion, instead of $O(1)$ operations that are required on the average.

More recent works show that many other systems and algorithms are vulnerable to complexity attacks including QuickSort [8], regular expression matcher [9], intrusion detection systems [10], [11], the Linux route-table cache [12], SSL authentication algorithm [13], and the retransmission algorithm in wireless networks [14]. 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 systematically alleviate the attack scenarios discussed in this paper.

Current multi-core implementations of NIDS/NIPS systems such as Snort [3] and Bro [2] split the load to many *sequential* sub-tasks in a pipeline manner. Other works, such as [15], suggest fine-grained pipelining for parallelizing network applications on multi-core architectures. This partitioning is effective if processing cost for each sub-task is similar, which is usually not the case for NIDS/NIPS.

A different line of research focuses on equally load balance the traffic flows between the different cores and performing the inspection in parallel [16]–[20]. Thus, each core has the same functionality. 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. [21] focus on DPI and present performance evaluation scheme for multiprocessor systems. The proposed design also splits the traffic between several cores with the same DPI engine on each, which supports regular expression matching. Their study identifies and evaluates algorithmic and architectural trade-offs and limitations. It also highlights how the presence of caches affects the overall performances. However, the scheme is geared at optimizing the normal case and is vulnerable to complexity attacks as we describe in this paper. Such attacks can be mitigated by incorporating MCA^2 to this scheme as well.

Another multi-core load-balancing approach is to partition the patterns among the cores (cf. [22]–[24]) and duplicate each packet to all cores. Then, different DPI algorithms, each is specialized in a different kind of pattern set, run on the different cores. 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, traffic splitting is determined a-priori and it does not take into account the incoming traffic, and therefore, is vulnerable to a complexity attack on each core separately.

In recent years there have also been some works that use graphical processing units (GPUs) for DPI. The main approaches of such works are either to execute parallel DPI engines on the hundreds of GPU cores [25] or to use the parallelism for nondeterministic traversal of a NFA [26]. These approaches provide very high throughput while being less vulnerable to the attacks presented in this paper. However, they require the existence of expensive GPU hardware. As our algorithms are designed for general purpose CPUs we do not compare them to such approaches.

With regards to DFA compression, there exists an extensive line of research (either exact pattern matching or regular expression) for hardware implementations [27]–[37], but most of these solutions are not applicable in our context, since they were tailored for specialized hardware implementation, although some works (e.g., [33], [35]) deal with impacts of the encoding of DFA transitions, as we do in Section V. In our context, the most relevant paper is of Tuck et al. [37], which focuses on improving Aho-Corasick algorithm in hardware, but also shows that such compression solutions do not drastically affect the memory performance in software. Therefore, the authors conclude that such solutions should be taken into consideration also in software. Our compression techniques reduces the space by 60% with respect to the solution proposed in [37], on real-life datasets such as Snort IDS [3] and ClamAV [38]. In addition, while Tuck et al. show only minor worst-case performance degradation, they did not take into account the system architecture (and specifically, the influence of the cache). Our worst-case scenario, on the other hand, shows a major adverse impact on the performance. Note that current state-of-the-art works [33], [35], [39]–[42] suggest a set of memory efficient schemes. This work novelty is derived from the combination of two algorithms rather the specifics of a single one. Thus, our space efficient algorithm may be replaced with any of the above schemes using our MCA^2 architecture.

Kumar et al. [43] 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 somewhat similar to MCA^2 as it handles heavy traffic with a different algorithm and applies a lightweight classification algorithm to distinguish between heavy and normal traffic. In addition, [43] suggests a protection 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 suggested protection in [43] fails under a continues DoS attack since the heavy packets that receive lower priority eventually overload the system buffer. MCA^2 is resilient also to DoS attacks with longer duration.

CompactDFA [36] introduces a compressed DFA design for string matching for either a TCAM implementation or a software implementation. While for TCAMs, CompactDFA achieves a notably improved performance, its software implementation performance are inferior to both the classical Aho-

Corasick algorithm and our compressed automata encoding (presented at Section V). In fact, CompactDFA is equivalent to our compressed automaton using only leaves compression.

IV. Snort Cache-Miss Complexity Attack

Common practice shows that when scanning normal traffic, only a small number of states within the Aho-Corasick DFA is used. Our experiments show that the scan of 90% of typical traffic visits less than 7% of DFA's states (see Fig. 8 for a deeper analysis). Therefore, most memory accesses in DPI are cache hits, as the code of the DFA keeps reading the pointers of the same small subset of DFA states. 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. These cache-misses have two negative effects. First, an access to main memory is at least 10–20 times slower than a cache access, implying it takes significantly longer to deal with this malicious input traffic. Second, and even more importantly, dealing with the malicious traffic causes significant cache pollution, slowing down also the processing of normal traffic. In the stand-alone setting where we only run the DPI engine itself, separately from the NIDS implementation, the Cache-Miss Attack degrades the performances of the DPI by a factor of between 1.9 in a mild attack to *over 4.7* in a severe attack (see Fig. 1). Thus, this is considered an efficient algorithmic complexity attack. The circles-curve in Fig. 2 shows the goodput reduction for different values of attack intensity.

While the stand-alone setup demonstrates the attack on a stand-alone AC DFA, we show that the attack works on Snort, a complete NIDS that is used in practice. Recall that Snort divides the pattern sets into classes according to the types of the traffic. The largest of the resulting AC DFAs is the one representing HTTP traffic, with a memory footprint of about 32 MB. We craft a cache-miss attack in two steps. First, we collected from Snort's publicly available signatures set, the patterns that are represented in this automaton. To prevent this attack from being detected by Snort built-in mechanisms, we omit the last character of each collected pattern. The payload of the attack packets consists of collections of different patterns from this truncated collection, which does not generate any alert. Then, we constructed a set of HTTP traffic traces that mix attack packets with normal HTTP traffic to the most-visited web sites [44]. We created eight different traces that differ in the proportion of attack traffic in them. The intensity of attack goes from 0% (only normal traffic) to 100% (only attack traffic). We inject these traces as network traffic to Snort 2.9.4 with the default Aho-Corasick settings, and measure its throughput.

Fig. 4 shows the overall maximum goodput of Snort under these traces. The throughput of Snort drops by a factor of 1.2 when attack intensity is 16%, and up to a factor of 1.5 as the cache-miss attack becomes more intense. Namely, even if one over-provisions Snort NIDS that it should handle only a traffic of bandwidth 70% of its maximum capability, an attack that consumes only one sixth of the bandwidth degrades Snort to a point in which it cannot handle the entire well-behaved traffic, and therefore either choke or let packets go by uninspected. 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 sub-strings and invokes a regular expression engine (for a single expression) only if all sub-strings were matched by the exact matching engine.

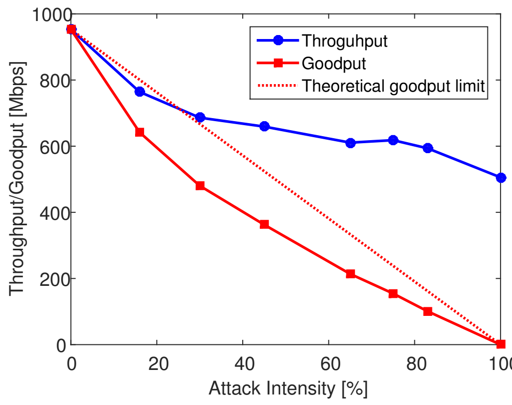


Fig. 4: The effects of a cache-miss attack on the throughput and goodput of Snort, facing attacks of different intensities, when Snort's CPU core is fully utilized. All attacks do not cause any alert from Snort NIDS.

V. Automaton Scheme Fitted for Cache-Miss Algorithmic Complexity Attacks

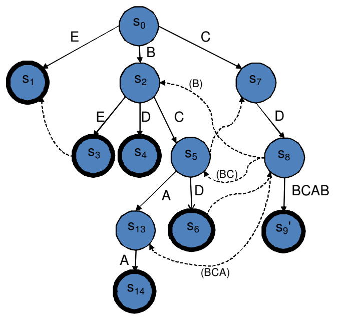
As noted above, the main problem with the naïve AC DFA is its size, which does not fit into the cache. We now propose an alternative implementation, denoted Compressed-AC, that is memory-conserving, with which the automaton may fit in a CPU cache. The rest of this section presents our compression techniques that are based on the failure-transitions–based representation of AC, which was introduced in Section II.

A. Branching States Representation

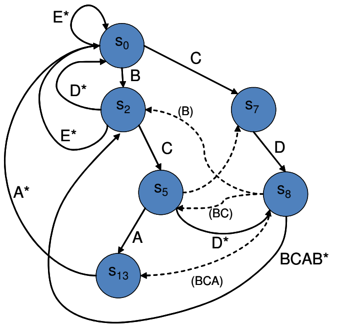
Whenever a state has more than one forward transition, denoted as a branching state, we represent it using a *lookup table*: state holds an array of $|\Sigma|$ entries, such that the i^{th} entry holds the next state to transit to, had the symbol been i. If the corresponding transition from state s is a failure transition, the next state that is encoded is s' such that label(s') is the longest prefix of label(s), as explained above. Notice that if all the states are encoded with a lookup table, there is no advantage over the naïve matrix encoding. As such, most implementations use lookup tables only to encode states with high out-degree.

B. Path Compression

A common approach to improve performance of trie traversal is an efficient representation of *one-way branches*; namely, a sequence of consecutive states in the trie, such that each



(a) The automaton of Fig. 3(b) after path compression.



(b) The automaton of Fig. 5(a) after leaves compression.

Fig. 5: Illustration of the compression process of the AC automaton of Fig. 3(b). 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.

(except, maybe, the last one) has only one forward transition. Such approach lies at the core of the seminal PATRICIA trie algorithm [45]. However, this approach is not directly applicable in our case, since one may traverse the trie over failure transitions, and these transitions should also be taken into account when compressing the branches. We overcome this problem by compressing one-way branches of *any length*, as long as they have *no incoming failure transitions*.

Tuck et al. [37] were the first to consider path compression for AC-like automaton. Our approach reduces the memory footprint of the Snort IDS AC automaton by approximately 25% over the compression achieved by their method, and reduces the number of trie nodes by about 85%. Fig. 5(a) depicts our toy example automaton from Fig. 3(b) after path compression.

C. Leaves Compression

By definition, trie leaves do not have any forward transitions, implying they consist only of a single failure transition, which is taken every time the corresponding state is reached. In addition, by the AC's DFA construction, these leaves correspond to accepting states of the automaton. Thus, *the whole purpose of these states is to indicate that a match was found*. A simple way to reduce the number of states in the trie is therefore to push this indication to the penultimate node by adding one bit for each forward transition in a node, indicating whether it leads to an accepting state. This process can be repeated recursively, until there is no transition to a leaf.

To apply both path compression and leaves compression we add one bit for every symbol of the corresponding compressed path. The bit of the i^{th} symbol of the path is set to 1 in two cases: (i) when a transition with the first i symbols of the path is to an accepting state (in the pre-compressed automaton), or (ii) if the failure transition of the pre-compressed state reached after the first i symbols of the path, is to a leaf. This way, any pattern that should be matched during the traversal of the compressed path is found. This method is illustrated in Fig. 5(b); we denote with an asterisk transitions that also indicate a match.

D. Pointer Compression

A key observation that is common to AC-like DFAs is that there are many transitions that go to states whose depth is small. Specifically, in the AC DFA used by Snort NIDS, 13% of the failure transitions go to the root, 31% of the failure transitions go to states whose depth is 1, while additional 35% of the failure transitions go to states whose depth is 2. Therefore, by representing these states in a compact manner, we can significantly reduce the memory footprint.

We use *variable-size pointers* of two lengths: 2 and $2 + \lceil \log_2 |S| \rceil$. The first two bits of the pointer indicate the action that should be taken as either one of the following four options: go to s_0, go to a state at depth 1 using a given lookup table, go to a state at depth 2 using a given hash table or to a state encoded by the next $\lceil \log_2 |S| \rceil$ bits. Pointer compression achieves an improvement of additional 43% over the path-compressed data structure described in Section V-C.

E. Compression Effectiveness

The compressed encoding of Snort patterns requires only about 1.5 MB, compared to 75 MB of non-compressed encoding. COMPRESSED-AC implementation has almost-constant throughput, independent of traffic it handles (the squares-curve in Fig. 1 presents the throughput of this implementation). However, it is *slower* by a factor of 2.37 than the FULL MATRIX AC implementation under normal traffic, as it performs between 8 to 15 times more memory accesses per input symbol than the non-compressed implementation. Thus, our compression techniques cut the throughput by about half in order to overcome cache-miss attacks.

VI. THE MCA^2 SYSTEM DESCRIPTION

A. MCA^2 Design overview

MCA^2 operates over a multi-core platform as described in Fig. 6, where each core runs one or more hardware threads (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 packets, places them in main memory, and enqueues a pointer to the packet to the cores' incoming-packets queues. We follow recent works [17], [46] to load balance the incoming traffic

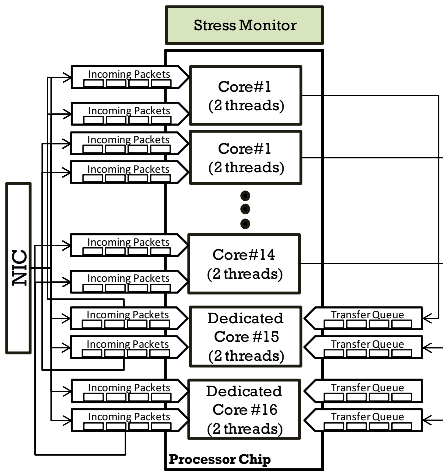


Fig. 6: Illustration of MCA^2. Cores 1 through 14 are general, while Core 15 and 16 are dedicated. Each core has two threads. Threads of general cores transfer their heavy packet to a specific thread in a dedicated core through a transfer queue. Only the logical structure of the queues is presented. We have used a machine with more cores than the one we used in our setup as it demonstrates better the suggested architecture.

between the different hardware threads in the NIC. Note that each packet has a single copy in main memory (created by the NIC).

Moving a packet between threads is considered a very light operation in terms of performance. The packets are not copied rather than only the pointer to the packet location in memory and a field that indicates the index when the root state was visited for the last time [3]. This way the scan can continue from this position correctly even though the compressed automaton does not use exactly the same set of states.

The system works either in *routine-mode* or in *alert-mode*.

In routine-mode, all threads are 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 type of attack). From that point, the dedicated threads receive messages from other threads with references to heavy packets. Thus, the dedicated thread handles packets from both its transfer queue and 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 packet 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.

[3]The reason for storing the last index where the DFA was in the root state is that if dedicated cores use a different algorithm we would not miss any potential match.

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

B. Cross-Thread Communication Mechanism

Concurrency in multi-core systems usually suffers from cross-thread communication overhead, which might become significant in some cases. 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 several general threads may 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.

Such queue is equivalent to the single producer single consumer problem for which there are known non-blocking algorithms that use only atomic read/write operations (e.g., [47] [48, Chapter 8.1]). Notice that these algorithms can be even further simplified in our case, since if the queue becomes full the producer is allowed to either process the packet by itself, stall shortly and retry, or try to enqueue it to a different dedicated thread (assuming such exists). When drop is allowed, the producer may simply drop the packet, which is equivalent to discarding a heavy packet, when we have too many of them. While dropping introduces some false positives, this behavior may still be a better alternative to a complete service failure.

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 VI-A). Our simulations show that even under worst-case traffic, the overhead of this communication mechanism does not exceed 0.98% degradation in system throughput.[4]

C. Thread Allocation Scheme

In this section, we focus on *no drop setup* in which no packets may be dropped by the NIDS. We note that in the

[4]To experiment this worst case, we modified our code to transfer *every* packet received on a regular input queue to another thread, and then have the other thread handle it. This gave a throughput of 8319 Mbps under the setup described in Section X. Without transferring packets the throughput was 8402 Mbps.

alternative *full drop setup*, the NIPS does not inspect any heavy packets, and therefore no dedicated threads should be allocated. The goal of our thread allocation scheme is to maximize *goodput* assuming that the system is balanced.

Naturally, the number of dedicated threads should grow as the fraction of the heavy packets in traffic grows. In addition, we take into account the performance of the two algorithms and consider how these two algorithms perform while handling either only heavy packets or only normal packets. It is important to notice that the throughput of the algorithm usually depend on the fraction of heavy packets they handle. For brevity, our model does not consider these exact numbers and uses only the two extreme points.

Let r be the fraction of heavy packets out of all traffic, let T be the throughput of the system when there are no heavy packets, and let β be the ratio between the throughput of the general threads' algorithm and the throughput of the dedicated threads' algorithms, running solely on heavy packets and all threads act as general threads. 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

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

where N is the number of available threads. Notice that D_f is not an integer, and therefore should be rounded to give the required number of threads. According to the type of attack 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 only a simplified model on deciding how many dedicated threads to allocate. A more accurate model may support additional aspects such as using the algorithm of the general threads by a dedicated thread for lower rates of r, taking into account the detection overhead of packets, and optimizing the number of packets that are exchanged between dedicated and general threads, taking into account the load balancing among them, as demonstrated in the experimental results in Section X. Finally, we note that a useful practice is to limit the maximal value of D_f to preserve a share of general threads under any attack intensity.

D. 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. [17], [46]). MCA^2 can be combined with any method that provides flow affinity, yet to divert heavy flows to dedicated cores and thus reduce their effect on legitimate flows.

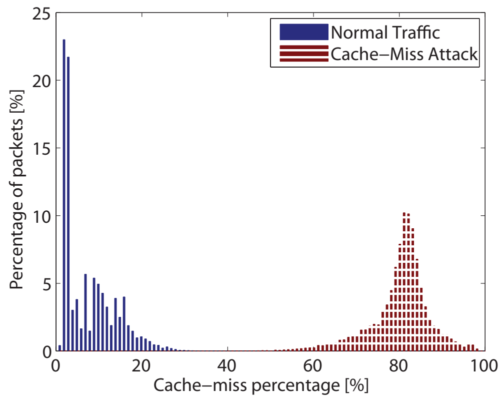


Fig. 7: Distribution of cache-misses under normal traffic and under attack.

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 [49] or a hash table with timestamps, so that outdated records can be easily removed.

VII. MCA^2 for Cache-Miss Attacks

In this section we present an algorithm for detecting heavy packets in AC DFA algorithmic complexity attack. Cache-miss attacks are characterized by *a large number of different state machine traversals that cause cache-miss* (compared with the normal operations of the system), as clearly illustrated in Fig. 7: 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 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, which we use in 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 frequently visited sates of the DFA when processing the normal packets. MCA^2 establishes the *non-common states* ratio after scanning the first 20 bytes of a packet and then it keeps updating this ratio while scanning

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

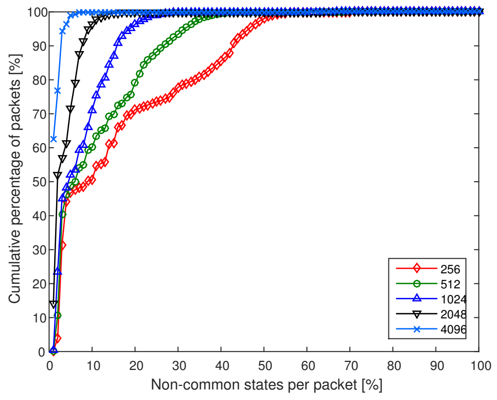


Fig. 8: CDF of the percentage of normal traffic packets by their non-common states ratio for different numbers of common states, 256, 512,…, 4096.

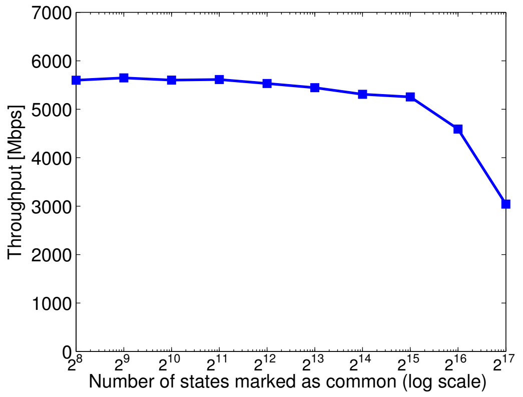


Fig. 9: The total system throughput for a different number of common states, under an attack of intensity 33%.

the packet data[6]. When this ratio crosses a certain threshold (25% in our experiments), the packet is marked heavy.

An important parameter that should be chosen is the number of common states (that is, in the list of states, ordered by the frequency of visits, 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, in which each state is represented 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.

Before determining the number of common states, we first 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 length of the packet (which is the total number of state visits, common and non-common). 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.

False Positive Rate	Number of common states				
	256	512	1024	2048	4096
1.0%	**53%**	**38%**	**25%**	**15%**	**6%**
2.5%	50%	35 %	22%	11%	5%
5.0%	47%	32%	19%	10%	4%

TABLE I: The minimal thresholds for non-common states ratio given different desired false-positive rates and different numbers of common states.

[6]We tested the effect of scanning a different minimal number of bytes before deciding whether a packet is heavy and found that 20 bytes give the best tradeoff between accuracy and throughput.

Fig. 8 considers a normal traffic and depicts a CDF showing the percentage of packets by their non-common states ratio. This percentage grows quickly as the number of common states increases. In addition to the number of common states we also need to set the threshold for non-common states ratio, as the two values are dependent on each other. We would like to minimize the number of packets falsely identified as heavy. We conducted an experiment that given the number of common states and the real-life web traffic trace (see Section X), provides the minimal threshold of non-common states ratio that produces a certain false-positive rate on that trace. The results of this experiment are summarized in Table I.

Using the above thresholds and the thread-allocation scheme (see Section VI-C), we measured the system throughput for different numbers of common states. Fig. 9 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 8K states. We have repeated these experiments for various attack 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% of 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. 10.

VIII. 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, but rather in a non-deterministic finite automaton (NFA) [50]. [7] When using an NFA, the matching algorithm keeps a vector of active states and for each input

[7]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 VII.

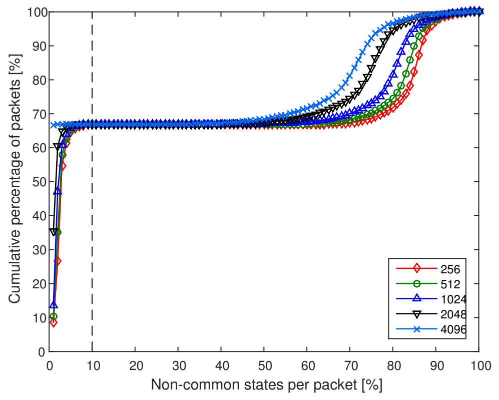


Fig. 10: CDF of the percentage of mild attack (33%) traffic packets by their non-common states ratio.

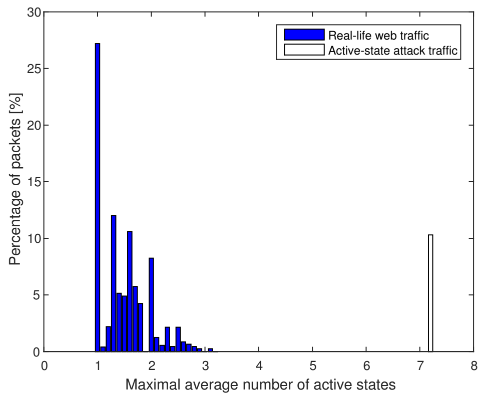


Fig. 11: Distribution of maximal average number of active states per 40 bytes window, per packet, under real-life web traffic and under attack.

symbol, 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, the NFA performs k times slower than a DFA).

Becchi et al. [6] 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 which can cause a space blow-up can be easily determined. They interrupt the transformation of these specific states, keeping them non-deterministic, such that they connect two deterministic automatons. This process produces a hybrid finite automaton (*Hybrid-FA*) which 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 which is called a *tail DFA*. As border states are non-deterministic, arriving at such a state during the 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 [6], on certain inputs, the average number of active states can be potentially larger by a factor of 30 than on an average case input. This gap reveals a potential algorithmic complexity DoS attack on a system that uses the algorithm. To illustrate the attack we used the Hybrid-FA code [51] (provided by the authors of [6]), along with a set of regular expressions taken from the Bro NIDS (which was also provided in [51]). We carefully crafted one malicious packet that causes activation of at most eight active states simultaneously.[8] 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.

[8]To create the malicious packet we selected prefixes of regular expressions which 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.

We have replaced the pattern matching module, described in Section VII, with the Hybrid-FA pattern matching code [51] 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 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. 11 shows the distribution of the highest average number of active states per 40 bytes window (denoted as 'maximal average'), per packet, in 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 X-C we show the results of our experiments with Hybrid-FA and MCA^2.

IX. MCA^2 for Force-Construction Attacks

The previous section discussed an algorithmic complexity attack over an NFA design that maintains a set of active states. A different approach to implement an NFA, which avoids maintaining a set of active states was presented by Heering et.al. in [52]. This approach starts with an NFA and transforms only the parts which are being used to a DFA on-the-fly. Thus, this method also avoids constructing the entire DFA while maintaining the DFA efficiency for the parts that are actually used. This method is sometimes called a Lazy-FA and it is used by the Bro IDS [2]. While efficient under normal traffic, this method is vulnerable to certain inputs that force it to construct the entire DFA.

The Bro implementation does not clear an already-built DFA state. Thus, with a moderate size set of regular expressions, such as the one in Snort, not only Bro suffers from a DFA construction penalty, but eventually the system memory depletes and Bro crashes. As opposed to Snort, Bro comes with a small basic pattern set, which focuses mainly on application classification. It leaves the task of writing rules to the IDS users. We used only this basic pattern set of Bro to illustrate a force construction algorithmic complexity attack. The simulated input traffic contained normal packets along with packets that were constructed to traverse the entire DFA, thus causing its complete construction. This very modest attack caused a throughput degradation of 24%.

Mitigating this kind of attack using MCA^2 is done by marking heavy packets as those who cause the construction of more than a certain threshold of states, depending on the potential size of the complete DFA. Dedicated cores should run an exact string matching DFA, such as Aho-Corasick, which is constructed from the exact strings part of the regular expressions. This DFA is used as a prefilter. In case all substrings of a regular expression were matched, this specific regular expression's NFA is invoked and the data is scanned again using this NFA. Each NFA is dedicated for a single rule and has a low memory footprint, therefore all required NFAs may be precompiled in advance. Thus the input traffic for the force-construction attack will not blow up the memory for this DFA scheme. This two stage scan is the method used by Snort. Note that Snort's prefilter introduces some false positives and the second stage eliminates them. Thus, there are no false negatives at all.

Still, the Snort implementation is exposed to back-tracking attack, which may lead to recursive execution and exponential running times or multiple NFA invocations. Both of these cases can be discovered. In our case, since the Snort-like algorithm acts only as the fall-back, we may decide that traffic that causes both Force Construction Attack and backtracking is not benign, and drop this traffic.

X. Experimental Results

A. Experimental Environment

We use a system with Intel Sandybridge Core i7 2600 CPU, quad-core, where 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 used the granularity of a core rather than a thread when allocating dedicated threads, thus a dual-thread core runs two dedicated threads. Thread affinity was used to associate threads to cores. Our experimental code is available online at `https://github.com/DeepnessLab/mca2`.

The pattern set consists of 6,422 strings that were extracted from the "content" property of the Snort rules. Duplicate strings were eliminated. 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 [44]. We used a crawler to surf through the selected URLs and captured the received responses to build the trace files. One of these traces is our *real-life web traffic* trace and the other trace is a training set to determine the common states set for cache-miss attack, as described in Section VII. 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 with specific malicious packets that were constructed as described in Section IV. These traces contain different volumes 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, with the same packet size distribution). Note that these traces were also used for Fig. 2. 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 VIII. The intensity of attack also varies from 0% to 100%.

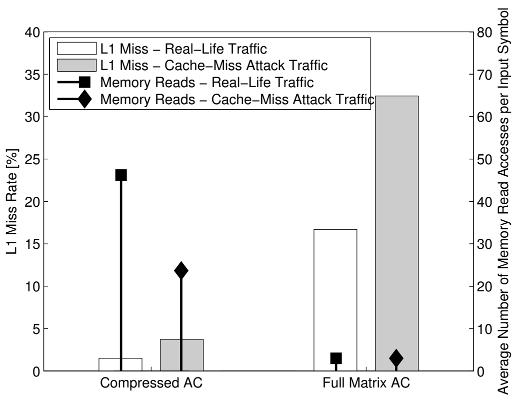


Fig. 12: Comparison of the memory behavior of the two AC implementations, in terms of memory access and L1 cache-miss rate, under real-life traffic and under cache-miss attack traffic.

B. Cache-Miss Attack Simulation Results

1) DPI Algorithms Analysis: We compare the two implementations for the AC automaton: Full Matrix AC, and Compressed-AC as discussed in Section V.

In terms of memory requirement, Compressed-AC requires significantly less memory space than Full Matrix AC, and in fact it may fit into any modern CPU cache. However, this compression comes with a price of additional computation. While Full Matrix AC requires a small, constant number of memory accesses per input symbol inspected, Compressed-AC may require many more memory accesses in order to find the next state, as it might have to traverse failure-paths and decode compressed pointers. The stems in Fig. 12 (and the right vertical axis) show these differences in memory access behavior between the two implementations.

The harmful potential for cache-miss attack on the Full Matrix AC implementation is illustrated by the bars in Fig. 12 (and the left vertical axis). These bars show the L1 miss rate of the two implementations under real-life traffic and under cache-miss attack. While both implementations show higher miss rate under attack, the miss rate of COMPRESSED-AC is still less than 4%, while the miss rate of FULL MATRIX AC is beyond 32%. The L2 miss rate (not presented in the figure) of FULL MATRIX AC is only 0.7% on real-life traffic, but under a cache-miss attack it goes above 23%. COMPRESSED-AC has L2 miss rate of at most 0.06%, under both real-life traffic and attack traffic.

Eventually, as shown in Fig. 1, these cache behaviors lead to a situation where on real-life traffic, FULL MATRIX AC performs better than COMPRESSED-AC since it stays in L2 cache and performs much less memory read operations than COMPRESSED-AC. However, under a cache-miss attack, the throughput of FULL MATRIX AC drops by a factor of 4. COMPRESSED-AC, on the other hand, provides a low throughput on real-life traffic as compared to the FULL MATRIX AC. However, this implementation is not vulnerable to cache-miss attacks, and when operating on attack traffic, it actually provides better performance.

2) Goodput: Fig. 2 depicts the *goodput* of MCA^2 when processing the different traffic traces. Note that as the attack intensity increases, the goodput decreases as the the non-malicious traffic occupies a 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. Clearly, upon an attack we gain a goodput improvement of 67%–102%, as compared to FULL MATRIX AC, which does not take cache-miss attack into consideration. Fig. 2 also depicts the *full-drop setup*, as described in Section VI-C, with a significant goodput improvement.

We also 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) FULL MATRIX AC implementation with no MCA^2 at all.

We have also evaluated the above tests with a larger pattern set, extracted from ClamAV [38], an open source network anti-virus (see Fig. 13). We have randomly picked a subset of patterns that resulted in a much larger DFA (159,790 states as compared to 77,182 in the case of Snort)[9]. The results of running MCA^2 are shown in Fig. 13. It can be observed that MCA^2 provides a higher throughput than either the FULL MATRIX AC or the COMPRESSED-AC algorithms, under cache-miss attacks.

3) Accuracy: To analyze the *heavy packet identification*, we first examine the identification results of our mechanism on the *real-life web traffic* and on the *cache-miss attack* traces. In the *real-life web traffic* trace, only 0.4% of the packets are falsely identified as heavy. When using flow affinity, and marking an entire flow as heavy if a single packet in it is identified as heavy, 4% of the flows are falsely identified as heavy. These flows will get transferred to a dedicated thread, although being legitimate, and suffer some slowdown. Nevertheless, without MCA^2, these flows would suffer even lower throughput under such an attack. In the *cache-miss attack* traces we know exactly which packet is heavy and can measure precise values for false identification rate. Under our attack scenario, less than 0.001% of the malicious packets were falsely classified as normal by either one of the configurations from Section VII. We see that our detection mechanism provides an accurate identification.

4) Vulnerability of the Detection Mechanism: An attacker can target directly the MCA^2 architecture by trying to either exploit the detection mechanism or to bypass it by keeping the rate of non-common states below the threshold for declaring a packet as *heavy*. Since the cross-thread communication mechanism has a negligible overhead, exploiting the mechanism itself by triggering packet transfers has almost no performance effect. On the second approach, in order to bypass the detection mechanism, the attacker must send 'lighter' packets that cause fewer cache-misses. Fig. 14 shows the throughput of MCA^2 under a set of attacks with a different ratio of non-common states: The effect caused when using 'very heavy' packets while holding 40% of the bandwidth, may be reproduced with 'light' packets only by acquiring

[9]The memory footprint of the ClamAV automaton is 722MB for the FULL MATRIX AC and 5MB for the COMPRESSED-AC. For Snort, the FULL MATRIX AC takes 75MB while the COMPRESSED-AC takes 1.5MB.

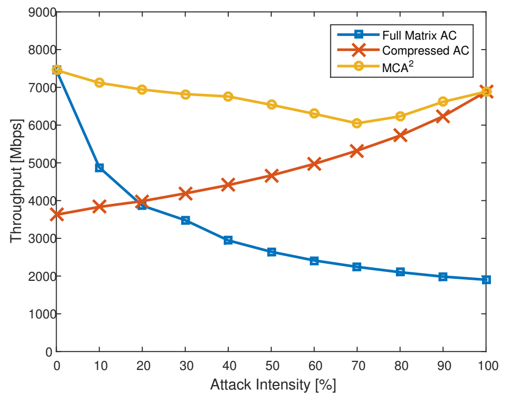


Fig. 13: Throughput of the various pattern matching algorithms on a larger pattern set extracted from ClamAV network anti-virus.

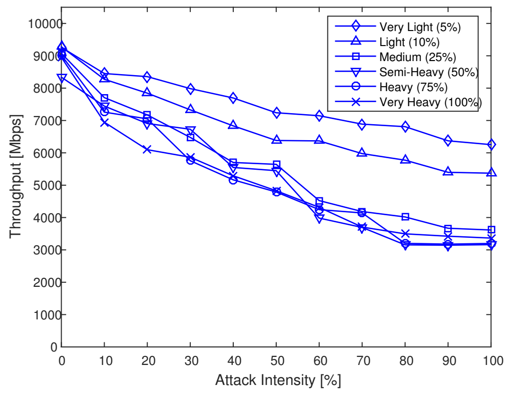


Fig. 14: Throughput of MCA^2 under attacks of different rates of non-common states and different intensities.

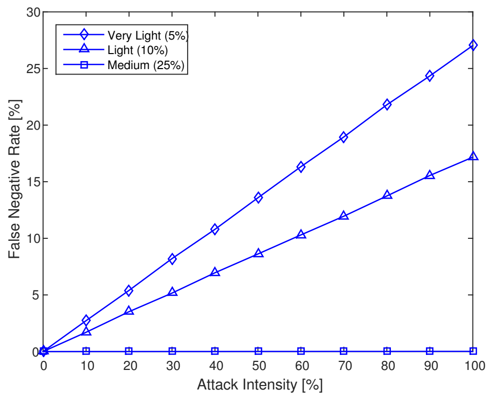


Fig. 15: False-negative rate of the detection mechanism for attacks of different rates of non-common states and different intensities. False-negative rates for heavier attacks are always lower than those for 'medium'.

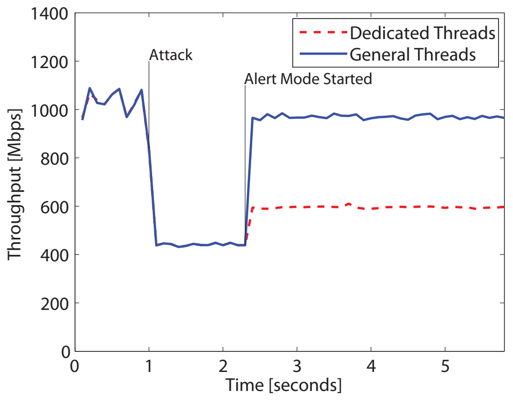


Fig. 16: Average throughput per thread over time, when a sudden *cache-miss attack* happens. The system uses eight threads, and when alert mode starts, two of them become dedicated threads.

100% of the bandwidth. As soon as the attacker tries to use even slightly heavier packets, our detection mechanism identifies the heavy packets, as illustrated in Fig. 15.

5) Identifying Cache-Miss Attacks: In order to evaluate the system behavior under 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* 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 for all dedicated threads. Fig. 16 depicts the result of this experiment. The system starts with all its eight threads being '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'

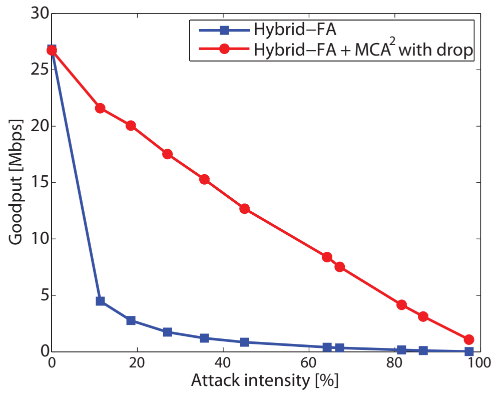


Fig. 17: Goodput of Hybrid-FA and of Hybrid-FA with MCA^2 full-drop setup, facing different intensity of active-state attack, when CPU is fully utilized

.

with the COMPRESSED-AC algorithm, optimized for handling heavy packets. General threads now transfer heavy packets to the dedicated threads, preserving high *goodput* (General threads still have to scan 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).

C. Active-State Attack Simulation Results

For the *Active-State Attack* simulations we have used the same traffic along with regular expressions that were extracted from the *Bro217* set located at [51]. Fig. 17 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.

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 design 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, an attacker can create lighter packets that might go under the radar, however such packets are bound to induce much smaller slowdown, 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.

XI. CONCLUSION

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

A simple method to mitigate an algorithmic 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 in 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 heavy traffic, 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. In our experimental setup we demonstrate a performance boost of up to 73% in the case of a *Cache-Miss Attack* and increase of the system goodput by a factor of 4.8 in the case of an *Active-State Attack*.

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

References

[1] “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.

[2] “The Bro Network Security Monitor.” http://bro-ids.org.

[3] “Snort: The Open Source Network Intrusion Detection System.” http://www.snort.org(accessedonMay2010).

[4] 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*, pp. 252–273, 2002.

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

[6] M. Becchi and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” in *ACM CoNEXT*, pp. 1:1–1:12, 2007.

[7] S. A. Crosby and D. S. Wallach, “Denial of service via algorithmic complexity attacks,” in *USENIX Security Symposium*, pp. 3–3, 2003.

[8] M. D. McIlroy, “A Killer Adversary for Quicksort,” *Software–Practice and Experience*, pp. 341–344, 1999.

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

[10] M. Fisk and G. Varghese, “Fast Content-Based Packet Handling for Intrusion Detection,” 2001. UCSD Tech. Report CS2001-0670.

[11] R. Smith, C. Estan, and S. Jha, “Backtracking Algorithmic Complexity Attacks Against a NIDS,” in *ACM ACSAC*, Dec. 2006.

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

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

[14] 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*, pp. 1431–1439, Apr. 2011.

[15] J. Wang, H. Cheng, B. Hua, and X. Tang, “Practice of parallelizing network applications on multi-core architectures,” in *ICS*, pp. 204–213, 2009.

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

[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*, pp. 60–68, 2008.

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

[19] W. Cong, J. Morris, and W. Xiaojun, “High performance deep packet inspection on multi-core platform,” in *IEEE BNMT*, pp. 619 –622, 2009.

[20] T. Nelms and M. Ahamad, “Packet scheduling for deep packet inspection on multi-core architectures,” in *ACM/IEEE ANCS*, 2010.

[21] M. Becchi, C. Wiseman, and P. Crowley, “Evaluating regular expression matching engines on network and general purpose processors,” in *ACM/IEEE ANCS*, pp. 30–39, 2009.

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

[23] O. Villa, D. P. Scarpazza, and F. Petrini, “Accelerating real-time string searching with multicore processors,” *Computer*, vol. 41, pp. 42–50, April 2008.

[24] B. Xu, K. Zheng, Y. Xue, and J. Li, “Scalable string matching framework enhanced by pattern clustering,” *Ubiquitous Computing and Communication*, vol. 5, June 2010.

[25] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, “Kargus: A highly-scalable software-based intrusion detection system,” in *CCS*, pp. 317–328, 2012.

[26] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, “infant: Nfa pattern matching on gpgpu devices,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 20–26, Oct. 2010.

[27] J. van Lunteren, “High-performance pattern-matching for intrusion detection,” in *INFOCOM*, pp. 1–13, April 2006.

[28] W. Lin and B. Liu, “Pipelined parallel ac-based approach for multi-string matching,” in *ICPADS*, 2008.

[29] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker, “High performance string matching algorithm for a network intrusion prevention system (NIPS),” in *IEEE HPSR*, 2006.

[30] D. Pao, W. Lin, and B. Liu, “Pipelined architecture for multi-string matching,” in *Computer Architecture Letters*, 2008.

[31] L. Tan and T. Sherwood, “Architectures for bit-split string scanning in intrusion detection,” *Micro, IEEE*, pp. 110–117, 2006.

[32] F. Yu, R. Katz, and T. Lakshman, “Gigabit rate packet pattern-matching using tcam,” in *Proc. ICNP*, 2004.

[33] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, “HEXA: Compact data structures for faster packet processing,” in *IEEE ICNP*, 2007.

[34] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner, “Algorithms to accelerate multiple regular expression matching for deep packet inspection,” in *ACM SIGCOMM*, 2006.

[35] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in *ACM/IEEE ANCS*, pp. 145–154, 2007.

[36] A. Bremler-Barr, D. Hay, and Y. Koral, “CompactDFA: Scalable pattern matching using longest prefix match solutions,” *IEEE/ACM Trans. Netw.*, vol. 22, pp. 415–428, April 2014.

[37] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic

memory efficient string matching algorithms for intrusion detection," in *IEEE INFOCOM*, 2004.

[38] "Clam AntiVirus." http://www.clamav.net (version 0.82).

[39] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *INFOCOM*, pp. 166 – 170, 2008.

[40] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 29–40, September 2008.

[41] M. Becchi and P. Crowley, "A-DFA: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 4:1–4:26, Apr. 2013.

[42] R. Smith, C. Estan, and S. Jha, "Xfa: Faster signature matching with extended automata," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, (Washington, DC, USA), pp. 187–201, IEEE Computer Society, 2008.

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

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

[45] D. E. Knuth, *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1973.

[46] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *ACM IMC*, pp. 218–224, 2010.

[47] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 190–222, Apr. 1983.

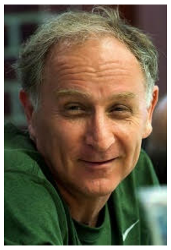
[48] G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming*. Pearson Prentice Hall, 2006.

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

[50] "Pcre - perl compatible regular expressions." http://www.pcre.org/.

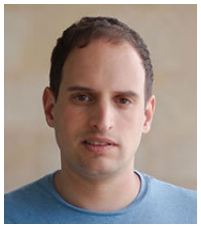
[51] "Regular expression processor." http://regex.wustl.edu.

[52] J. Heering, P. Klint, and J. Rekers, "Incremental generation of lexical scanners," *ACM Trans. Program. Lang. Syst.*, vol. 14, pp. 490–520, Oct. 1992.



Yehuda Afek B.A.in EE, Technion, M.Sc and Ph.D. in CS, UCLA 1983 and 1985, is the head of the Blavatnik School of Computer Science Tel-Aviv University. 1985-88 he was a Member of Technical Staff at AT&T Bell-Labs and joined the faculty of CS in Tel-Aviv University in 1988. In 2001 he co-founded Riverhead Networks, developing the DDoS Guard enabling clean pipes mitigation of Distributed Denial of Service attacks on the Internet. Riverhead was acquired by Cisco in 2004, and he was a director of Technology in Cisco 2004-2009.

Anat Bremler-Barr is an associate Professor at the School of Computer Science, Interdisciplinary Center, Herzliya, Israel. Prof. Bremler-Barr holds a Ph.D. (with distinction) in computer science from Tel Aviv University. In 2001, she co-founded and was the chief scientist of Riverhead Networks Inc., which provided systems to protect from Denial of Service attacks. The company was acquired by Cisco Systems in 2004. Prof. Bremler-Barr then joined the Interdisciplinary Center, Herzliya in 2004, where she co-founded with Prof. David Hay the DEEPNESS lab (funded by an ERC starting grant) that focuses on designing deep packet inspection for next generation network devices. Her research interests are in computer networks and network security.



Yotam Harchol is a Ph.D. student at the Hebrew University of Jerusalem, Israel, supervised by Prof. David Hay and Prof. Anat Bremler-Barr. Yotam holds a B.A. in Computer Science (magna cum laude) from IDC Herzliya in 2008 and a M.Sc. in Computer Science from the Hebrew University of Jerusalem in 2012. His research is focused on high-performance algorithms for network middleboxes and deep packet inspection. Yotam is the recipient of the Intel Award for Graduate Students (2010), Hammer Fellowship for Master Students (2009), and the Chais Scholarship for Social Leadership (2007).

David Hay is an associate professor at the Rachel and Selim Benin School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel. He received his BA (summa cum laude) and PhD degree in computer science from the Technion - Israel Institute of Technology in 2001 and 2007, respectively. In addition, Prof. Hay was with IBM Haifa Research Labs, Cisco Systems (San Jose, CA), the Electronic Department at Politecnico di Torino, and the Electrical Engineering Department at Columbia University, New York, NY. In 2010, Prof. Hay co-founded (with Prof. Brembler-Barr) the DEEPNESS lab, focusing on deep packet inspection in next generation network devices. His research interests are in computer networks - in particular, network algorithmics, packet classification, deep packet inspection, network survivability, and software-defined networking.

Yaron Koral is currently a postdoctoral researcher working with Professor Jen Rexford at the Department of Computer Science, Princeton University since October 2014. Prior to starting at Princeton he was a postdoctoral researcher at the Hebrew University working with Prof. Anat Bremler-Barr and Prof. David Hay. Yaron received his B.A. degree in computer science and economics in 1996, and M.B.A. degree in business administration in 2001 (cum laude), all from Tel Aviv University. His M.Sc degree in computer science was received in 2009 (summa cum laude) from the Interdisciplinary Center, Herzlia. Yaron received his Ph.D. degree in computer science in 2013 from Tel Aviv University under the supervision of Prof. Yehuda Afek and Prof. Anat Bremler-Barr. Yaron's main research interests are in computer networks systems and cyber security.