Space-time Tradeoffs in Software-based Deep Packet Inspection

Anat Bremler-Barr*, Yotam Harchol†, and David Hay†

*Computer Science Department, Interdisciplinary Center, Herzliya, Israel. bremler@idc.ac.il
†School of Engineering and Computer Science, Hebrew University, Israel. {yotamhc,dhay}@cs.huji.ac.il

***Abstract*—Deep Packet Inspection (DPI) lies at the core of contemporary Network Intrusion Detection/Prevention Systems and Web Application Firewalls. DPI aims to identify various malware (including spam and viruses) by inspecting both the header and the payload of each packet and comparing it to a known set of patterns. DPI is often performed on the critical path of the packet processing, thus the overall performance of the security tools is dominated by the speed of DPI.**

The seminal algorithm of Aho-Corasick (AC) [1] is the de-facto standard for pattern matching in *network intrusion detection systems* (NIDS). Basically, the AC algorithm constructs a *Deterministic Finite Automaton* (DFA) for detecting all occurrences of a given set of patterns by processing the input in a single pass. The input is inspected symbol by symbol (usually each symbol is a byte), such that each symbol results in a state transition. Thus, in principle, the AC algorithm has deterministic performance, which does not depend on specific input and therefore is not vulnerable to algorithmic complexity attacks, making it very attractive to NIDS systems.

In this paper we show that, when implementing the AC algorithm in software, this property does not hold, due to the fact that contemporary pattern sets induce very large DFAs that cannot be stored entirely in cache. We then propose a novel technique to compress the representation of the Aho-Corasick automaton, so it can fit in modern cache. We compare both the performance and the memory footprint of our technique to previously-proposed implementation, under various settings and pattern sets. Our results reveal the space-time tradeoffs of DPI. Specifically, we show that our compression technique reduces the memory footprint of the best prior-art algorithm by approximately 60%, while achieving comparable throughput.

I. Introduction

One of the fundamental techniques which is used today by security tools such as Network Intrusion Detection/Prevention System (NIDS/IPS) or Web Application Firewall (WAF) to detect malicious activities is Deep Packet Inspection (DPI). DPI consists of inspecting both the packet header and payload and alerting when signatures of malicious software appear in the traffic. These signatures are identified through *pattern matching* algorithms, where the patterns are either exact strings or regular expressions. Today, the performance of the security tools is dominated by the speed of the underlying pattern matching algorithms [10]. Moreover, DPI and its corresponding pattern matching algorithms are also crucial building blocks for other networking applications such as monitoring and HTTP load balancing.

This paper focuses on *exact string matching*, in which the content of the packet is compared against a predetermined set of strings (signatures). Specifically, this paper deals with the Aho-Corasick (AC) [1] algorithm, which is the most common algorithm used today. The AC algorithm uses a Deterministic Finite Automaton (DFA) to represent the pattern set. Then, the input is inspected symbol by symbol by traversing the DFA. Given the current state and the symbol from the input, the DFA should determine which state to transit to. A naïve implementation stores (in a two-dimensional table) a rule for each possibility; namely, one rule per each state-symbol pair. This resolves in prohibitively large memory requirement (e.g., 75 MB for Snort IDS [22] that has approximately 31,000 patterns); yet, a single memory access suffices to resolve what the next state is.

An alternative approach is to implement AC automata using the concept of *failure transitions*. In such implementations, only part of the outgoing transitions from each state are stored explicitly. While traversing the DFA, if the transition from state s with symbol x is not stored explicitly, one will take the failure transition from s to another state s' and look for an explicit transition from s' with x. This process is repeated until an explicit transition with x is found, resulting in *failure paths*. A classical result states that the longest failure path is at most the size of the longest pattern, and that, *regardless of the traffic pattern,* the total number of transitions (failure and explicit) is at most twice the number of symbols. Naturally, since only part of the transitions are stored explicitly, these implementations are more compact.

In this paper, we first compare several implementations that use failure transitions and differ in the way each state of the automaton is represented. For example, each node can store a lookup table so that when encountering some symbol x, one can determine the next state (or, in case of a failure transition, an intermediate state) in one memory access. On the other hand, a more compact representation but more time consuming is to store all explicit transitions in a linked list; if the transition is not found within the list, one will take the failure transition.

We then propose an implementation that outperforms the best prior-art ones. Our implementation is based on two observations: (i) most of the failure transitions go to a small subset of states, and therefore a short representation of such

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 n^{o} 259085.

transitions (even on the expense of a longer representations of other transitions) reduces space considerably; and (ii) there are very long one-way branches (that is, sequence of consecutive states in which there is only a single explicit transition from each state); these one-way branches can be compressed in a similar manner as in PATRICIA tries; a similar compression, which was tailored for hardware implementation, was mentioned also in [24]; Specifically, we calculated the memory footprint of the best prior-art representation of Tuck et al. [24] and show that our representation reduces the memory footprint of the automaton by 60%, under the pattern sets considered (namely, [7], [22]). A common metric to evaluate the efficiency of a representation is its *bytes per symbol* (namely, the ratio between the memory footprint and the total number of symbols in the pattern sets): our implementation requires as low as 3.23 bytes per symbol (see Table I), while the result reported in [24] is 60.31 bytes per symbol[1].

Finally, we implemented all the proposed representations in software and evaluate the throughput achieved by each of them on real-life traffic pattern as well as adversarial traffic. Our results show that, *unlike common belief*, the naïve implantation of the AC algorithm has non-constant throughput due to its huge memory footprint: a relatively simple traffic pattern can cause the algorithm to have many cache misses and degrades its performance by as much as 88%. This, in turn, make such an implementation very vulnerable to *Algorithmic Complexity Distributed Denial-Of-Service (DDOS) attacks* [8] that exploit gaps between worst-case and average-case performance to launch sophisticated attacks, forcing the device to operate always in the worst-case scenario. On the other hand, since our implementation has small memory footprint, it fits almost entirely in L2 cache and therefore is not sensitive to the locality of the traffic pattern. Algorithmic complexity DDOS attacks aiming at traversing failure paths reduce its performance only by approximately 30%. Moreover, our results show that both the naïve and the failure-transitions–based implementations have *comparable worst-case throughput*, while the latter have two order of magnitude smaller footprint. Finally, Fig. 1 presents the space-time tradeoff of different implementations of Snort IDS automaton, under worst-case and real-life traffic pattern. Clearly, under real-life traffic, there is a significant throughput gain as the memory footprint increases; however, this gain is almost lost when considering worst-case traffic. In addition, there is no significant throughput change between our compressed form of the failure-transitions–based automaton in its non-compressed form; this implies that our compression techniques come *almost for free*.

Paper Organization: The paper is organized as follows: an overview on the related work is in Section II. In Section III, we first describe the different implementations of automa based on failure transitions. Then, we present our new implementation and discuss its memory footprint. In Section IV we show experimental results using the two databases of Snort [22] and ClamAV [7], under two testing environments: one with relatively small cache and another with very large cache. Concluding remarks and future work appear in Section V.

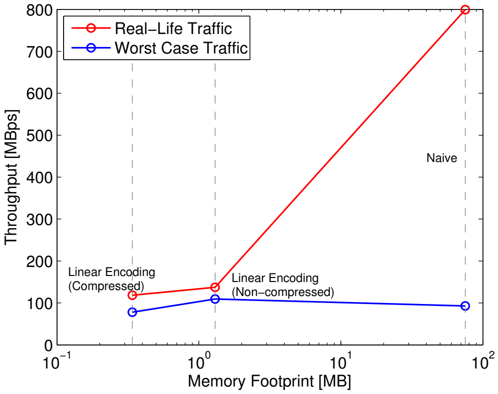


Fig. 1. Space-time tradeoff of representative three different implementations of Snort IDS [22] automaton, under real-life and adversarial traffic patterns. See Section IV for the exact experiment settings.

II. Related Work

Distributed Denial of Service (DDOS) attacks against network devices, and in particular NIDS and WAF, are considered one of the major threats with which these devices face. In general, in DDOS, attackers try to consume the resources of the device by sending huge amount of traffic *that is difficult for the device to analyze*. More specifically, *complexity DDOS attacks* [8] exploit gaps between the worst case and average (or common) case performance to launch attacks which result in either quality reduction or a complete denial of service. It is known that such attacks are very effective against regular expression matching algorithms, which are closely related to the algorithms considered in this paper, including the algorithm used by Snort NIDS [10], [20], [21]. The common characteristic of these attacks, as well as other complexity attacks (e.g., [3], [5], [8], [11], [16], [18], [21]), is that they leverage the performance gaps within the *algorithm itself* (e.g., the algorithm has sub-linear running time on some inputs and exponential time on the other). Our paper takes a *system point of view* and consider the *interaction* between the system and the algorithm, and in particular the effect of the cache on the performance of the algorithm. It is important to note that the only work that deals with caches and complexity attacks was on the linux route-table cache [26]; however, in that work the cache was implemented as part of the algorithm and not as part of the system architecture.

Our paper focuses on DPI solution *in software* for exact string matching, and particularly on the influence of the data structure memory footprint on the performance. Other common DPI solutions use dedicated hardware as FPGA, ASIC, or TCAM. There is an extensive line of research on compressing DPI solution (either exact pattern matching or regular expression) for hardware implementation [2], [4], [14],

[1] We note that the comparison is on different pattern sets, and therefore, might be misleading. On the other hand, the reduction of the memory footprint by 60% was calculated on the same pattern sets.

[15], [17], [19], [23]–[25], [27], [28], but these solutions are not applicable in our context (since they were tailored for hardware implementation), although some (e.g., [2], [14]) deal with impact of the transitions encoding of DFA. In our context, the most relevant paper is of Tuck et al. [24], which focuses on improving the hardware solution of Aho-Corasick, but also shows that such compression solutions do not drastically effect the memory performance in software. Therefore, the authors conclude that such solutions should be take into consideration also in software. Our compression techniques reduces the space by 60% than the solution proposed in [24]. In addition, they also analyze the worst-case performance degradation, however its worst case was not tailored to the system architecture, and especially to the influence of the cache, and hence the paper shows only minor degradation, while our worst-case shows a major adverse impact on the performance.

III. Aho-Corasick's Automata Representations

In this section, we describe how to represent Aho-Corasick automata. We first give a general background in Section III-A, and proceed with describing representations that use failure transitions in Section III-B. We then show how to improve the representation, in particular, using path compression (Section III-C), leaves compression (Section III-D), pointer compression (Section III-E) and Huffman coding compression (Section III-F).

A. Background

The Aho-Corasick algorithm works by traversing a DFA whose construction is done in two phases. First, the algorithm builds a *trie* of the pattern set: All the patterns are added from the root as chains, where each state corresponds to a single symbol. When patterns share a common prefix, they also share the corresponding set of states in the trie. In the second phase, additional edges are added to the trie. These edges deal with situations where the input does not follow the current chain in the trie (that is, the next symbol is not an edge of the trie), and therefore, we need to transit to a different chain. In such a case, the edge leads to a state corresponding to a prefix of another pattern, which is equal to the longest suffix of the previously matched symbols.

Formally, a DFA is a 5-tuple structure $\langle S, \Sigma, s_0, F, \delta \rangle$, where S is the set of states, Σ is the alphabet, $s_0 \in S$ is the initial state, $F \subseteq S$ is the set of accepting states, and $\delta : S \times \Sigma \mapsto S$ is the transition function. It is sometimes useful to look at the DFA as a directed graph whose vertex set is S and there is an edge between s_1 and s_2 with label x if and only if $\delta(s_1, x) = s_2$. The input is inspected one symbol at a time: Given that the algorithm is in some state $s \in S$ and the next symbol of the input is $x \in \Sigma$, the algorithm applies $\delta(s, x)$ to get the next state s'. If s' is in F (that is, an accepting state) the algorithm indicates that a pattern was found. In any case, it then transits to the new state s'. Upon beginning of the input, the algorithm is in state s_0.

We use the following simple definitions to capture the meaning of a state $s \in S$: The *depth* of a state s, denoted depth(s), is the length (in edges) of the shortest path between s and s_0. The *label* of a state s, denoted label(s), is the concatenation of symbols of the edge of the shortest path between s_0 to s.

The DFA is encoded and stored in memory, which is accessed by the AC algorithm when inspecting the input. A naïve approach is to store the set of transitions in an $|S| \times |\Sigma|$ matrix, where the cell at position (i, j) holds the value of $\delta(i, j)$. Hence, the width of each such cell is $\lceil \log_2 |S| \rceil$. In the typical case, when the input is inspected one byte at a time, the number of edges, and thus the number of entries is $256|S|$. For example, Snort patterns require 75.15 MB for 31,094 patterns that translate into 77,182 states (see Section IV).

B. Failure-transitions–based Implementations

An alternative approach is to store only the original trie which was used in the first phase of the DFA construction. The transitions which are the edges of the trie are called *forward transitions*, and each one of them link a state of some depth d to a state of depth $d + 1$. In addition, one should add a *failure transition* to each node of that trie, in order to capture implicitly the DFA transitions that are not trie edges: the failure transition of a state s is to state s' such that label(s') is the longest 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 edges are properly defined. The longest failure path (namely, a path that consists of failure transitions only) that starts at state s is of length of at most depth(s). This, in turn, implies that the total number of transitions (both forward and failure transitions) is at most as twice as the number of inspected symbols. Fig. 2 depicts an AC automaton used for patterns {E,BE,BD,BCD,CDBCAB, BCAA} over alphabet Σ = {A,B,C,D,E}, where the solid edges are forward transitions and the dotted edges are failure transitions (for clarity, failure transitions to s_0 are omitted).

For each symbol x and current state s, one should first determine whether the transition (s, x) is a forward transition (and therefore encoded explicitly) or not. This operation depends on the specific implementation of a state. In this paper we consider three such implementations, that trade memory footprint size with processing time (see [2] for more information)

Lookup Table Encoding: Each state holds an array of $|\Sigma|$ entries, such that the i^{th} entry holds the next state to transit to, had the symbol was i. If the corresponding transition is a failure transition, the next state that is encoded is an intermidate state, as explained above. Notice that if all the states are encoded as lookup table, there is no advantage over the naïve matrix encoding. However, most implementations use lookup tables to encode states with high out-degree (namely, states that many forward transitions originate from them). In the rest of the paper, we set the threshold for encoding states as lookup tables to 64 outgoing forward transitions.

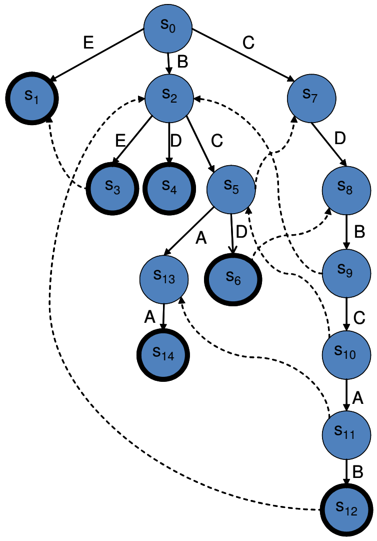


Fig. 2. An example of the AC automaton for pattern-set {E,BE,BD,BCD,CDBCAB,BCAA}.

Linear Encoding: Each state holds an array of symbol-state pairs. The number of pairs is as the number of forward transitions from the state. To find the next state, one should iterate over the array of pairs and find the one that corresponds to the current symbol. If no such pair is found, the failure transition, which is stored separately, should be taken.

Bitmap Encoding [24]: Each state holds a bitmap of length $|\Sigma|$ bits, such that the i^{th} bit is set if and only if there is a forward transition from the state with symbol i. In addition, the state holds an array of states, such that one entry corresponds to each forward transition, sorted by the values of the corresponding symbols. When encountering a symbol x, one first checks the corresponding bit (in the bitmap). If it is zero, then the failure transition (stored separately) is taken. Otherwise, let j be the number of 1-bits prior to index x. The next state is the $(j+1)^{\text{th}}$ entry in the array of states. Notice that unlike linear encoding, in bitmap encoding we store only the states and not symbol-state pairs. In addition, counting the number of 1-bits in a bitmap can be done relatively cheaply with the `popcnt` assembly command, implemented in SSE 4.2 instruction set (available, for example, in Intel's core i7).

Unlike the naïve implementation, in failure-transition–based implementations nodes have different sizes. Thus, one needs to store a pointer (of width 32 or 64 bits, depending on the environment) to the node's memory location rather than its index (e.g., of width 17 bits for Snort). A simple way around this problem is to add a *global conversion table* with $|S|$ entries, such that entry i holds the memory location corresponding to state s_i. This conversion table reduces the memory footprint significantly albeit with an additional memory access per (explicit or failure) transition.

C. 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 state in that sequence (except, maybe, the last one) has only one forward transition. Such approach lies at the core of the seminal PATRICIA trie algorithm [13]. 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.

Tuck et al. [24] were the first to consider path compression for AC-like automaton. Their solution, which was intended for hardware implementation, suggested to compress each one-way branch of a fixed length (e.g., 4) to a single transition. In order to deal with failure transitions that go to the middle of a compressed branch, the authors suggested to add a 2-bit `skip` counter to each failure transition, indicating how many input symbols should be consumed when taking this failure transition.

While this approach is correct and captures the essence of one-way branches compression, one can do better when implementing the pattern matching algorithm in software. We suggest to compress one-way branches of *any length*. On the other hand, since such compression implies an unbounded `skip` counter width, we compress only branches whose states have a single outgoing forward transition and *no incoming failure transitions*. Thus, failure transitions lead only to the beginning of branches and the `skip` counter is redundant. In addition, path-compressed nodes have several outgoing failure transitions: one for each original (i.e., pre-compression) state.

Our results shows that this approach reduces the memory footprint of real-life data-sets by approximately 25% over the compression achieved by Tuck et al. (the specific numbers depend on the state implementation as discussed in Section III-B). Moreover, path compressions reduces the number of trie nodes by about 85%: for the Snort IDS, the number of states is reduced from $77,182$ to $11,927$. Fig. 3 depicts our toy example automaton after path compression. In the branch corresponding to pattern CDBCAB (states $s_0, s_7 - s_{12}$), the first two states (s_7 and s_8) are not compressed since they have incoming failure transitions. States $s_9 - s_{12}$ are compressed (to state s'_9) and the transition from s_8 to s'_9 corresponds to the pattern BCAB. In case there is a partial match for the string BCAB in s_8, the outgoing failure transition that corresponds to the longest prefix of BCAB is taken. The strings correspond to these different failure transitions appear in Fig. 3 in parenthesis.

D. 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 (i.e., states in F). Thus, *the whole purpose of these state 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, just before the leaf node is reached. This is done by adding one bit for each forward transition in a node,

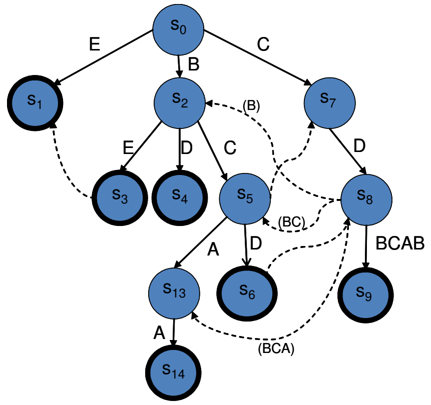


Fig. 3. The automaton of Fig. 2 after path compression.

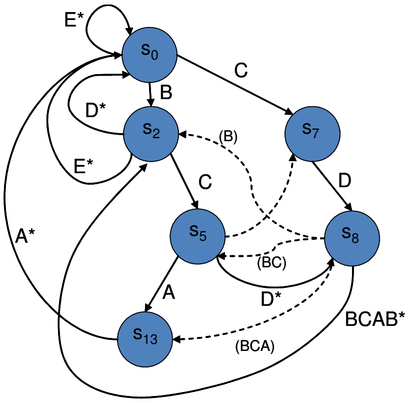


Fig. 4. The automaton of Fig. 2 after path compression and leaves compression.

indicating whether it leads to an accepting state. Then, all leaf nodes can be eliminated while their failure transitions are copied to the corresponding penultimate nodes as their new forward transitions. Furthermore, 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.

Leaves compression has relatively small influence on the number of nodes, since many of them were already compressed in the path compression phase. For Snort IDS, this compression reduces the number of nodes by 503 and the memory footprint by additional 3%. However, this compression has another positive effect, since it reduces the number of transitions taken when traversing the automaton.

In Fig. 4, we denote with an asterisk transitions that also indicate a match. The following states are eliminated in this stage: $s_2, s_3, s_4, s_6, s_9',$ and s_{14}. Their failure transitions are copied (as forward transitions in the predecessor state) to $s_0, s_0, s_0, s_8, s_2,$ and s_0, respectively. Notice that when compressing the state s_3 there is a recursive compression: the original failure transition of s_3 was to s_1, however since s_1 is also eliminated, the corresponding forward transitions (from s_2) is to s_0.

E. 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 Snort DFA, 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.

More specifically, we suggest to use *variable-size pointers* of two lengths: 2 and $2 + \lceil \log_2 |S| \rceil$. The first two bits of the pointer indicates the action that should be taken:

00: Go to state s_0.

01: Go to the state whose label is the current symbol. A table (with $|\Sigma|$ entries) translating one-symbol labels to their corresponding states is stored globally.

10: Go to the state whose label is the concatenation of the last symbol and current symbol. Since a direct-lookup table should have $|\Sigma|^2$ entries, we store the translation information in a global hash-table whose keys are two-symbol pairs and values are the corresponding states (of depth 2). Our results shows that, for Snort's DFA, there are only 1524 valid key-value pairs in the hash table (out of over 65K possible pairs).

11: Go to the state encoded in the next $\lceil \log_2 |S| \rceil$ bits. This prefix indicates a regular pointer.

Pointer compressions achieve a significant reduction in the memory footprint. Our results shows an improvement of additional 43% over the path-compressed data structure described in Section III-D. For example, in the automaton depicted in Fig. 4, states s_2 and s_7 appears in the table corresponds to depth 1 (in the second and third entries, all the other entries are `null`); states s_5 and s_8 are stored in the hash-table with keys BC and CD, respectively. State s_{13} is encoded directly.

F. Compression Using Huffman Coding

Huffman coding [12] may improve the compression of linear-encoded and path-compressed states by reducing the memory required to explicitly store *symbols* within such states. In general, Huffman coding allocates short code for frequent symbols and long code for infrequent ones. Thus, reducing the average per-symbol memory requirement. In our case we first compute offline the frequency of each symbol (as it does not depend on the input traffic). Then, a lookup table is used to provide symbol-to-Huffman-code conversion.

Our computations show that this technique can save up to 10% more space when using linear encoding and path compression. However, since this is a very limited additional compression, while incurring extra processing, we did not implement this technique while performing our experiments.

IV. Experimental Results

We have implemented a prototype of the system written in C, running on Unix systems. The prototype software creates the data structures from a given pattern set and then runs the matching algorithm on a given packet trace. In this section, we present the execution results under different configurations and types of traffic, and compare them to the throughput achieved by a naïve implementation of a deterministic AC automaton.

A. Test Setup and Traffic Types

We use two test environments: a Macbook Pro with Core 2 Duo 2.53GHz (dual core), 32KB L1 data cache, and 3MB L2 cache; and an iMac with Core i7 2.93GHz (quad core), 32KB L1 data cache, 256KB L2 cache (per core), and 8MB L3 cache (shared). Both systems run OS X Snow Leopard 10.6.

To test different sizes of patterns sets we use two common pattern sets: Snort [22] and a partial[2] ClamAV [7]. We compare the throughput of each configuration when used on each patterns set. The prototype software loads packets payload from the disk to the main memory in advance and then uses one or more threads to scan these packets with the pattern matching algorithm. We show results for several number of scanning threads for each configuration.

We test the different configurations of the algorithm using a real-life traffic trace from DARPA [9]. In addition, we define two types of adversarial traffic patterns with which we test each configuration. First, we look at a traffic pattern that intends to make the most *exhaustive traversal* of the automaton. To create such traffic, we use the pattern-set itself and randomly concatenate patterns to each other with some delimiter character in between them. This kind of traffic will make the automaton visit a large number of states and thus reduce locality and may increase the number of cache misses. The second type of adversarial traffic pattern aims to *traverse as much failure-paths as possible*. To do this, we find all the states whose longest failure path is of length is equal to their depth. There are 4,188 and 14,522 such states in Snort and partial ClamAV, respectively. Then, we concatenate the states' labels in a random order, adding a delimiter character (which does not cause any forward transition) in between.

B. Results

1) Space Requirement: Table I shows the space requirement of each data structure for both Snort and ClamAV. In all *compressed* implementation we performed path, leaves, and pointer compressions, and therefore, the difference is solely in the implementation of the non–path-compressed states, as presented in Section III-B. In addition, as discussed before, states with more than 64 forward transitions are always encoded using a lookup table. In our experiments, most of the nodes of the automaton are path-compressed. More specifically 79.9% of the nodes of Snort automaton and 92% of the nodes of ClamAV automaton have exactly the same implementation. This fact also explains the relatively small change between the different implementations in terms of time and speed. The *naïve* implementation stands for the two-dimensional table representation, as discussed in Section III-A; while the *non-compressed linear encoding implementation* is the failure-transitions–based implementation whose nodes are linear encoded, without compression (see Section III-B).

TABLE I
The memory footprint of the data structure required for each configuration. Bytes per Symbol is the ratio between the size of the data structure and the total number of symbols in the pattern-set

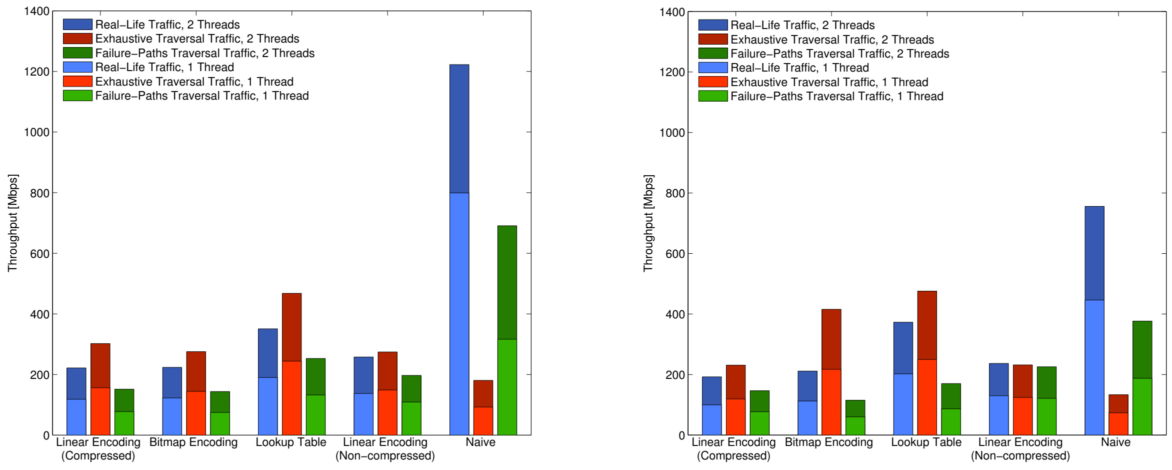
Pattern Set	Implementation		Size (MB)	Bytes/ Symbol
Snort	Compressed	Linear Encoding	0.34	3.23
		Bitmap Encoding	0.41	3.89
		Lookup Table	1.52	14.44
	Non-Compressed	Linear Encoding	1.30	12.36
		Naïve	75.15	714.04
Partial ClamAV	Compressed	Linear Encoding	2.46	3.09
		Bitmap Encoding	2.59	3.25
		Lookup Table	5.02	6.31
	Non-Compressed	Linear Encoding	11.90	149.50
		Naïve	722.14	907.24

2) Throughput: Fig. 5 shows the throughput achieved by each configuration, with Snort and ClamAV pattern-sets, on the dual core system. On real-life traffic, the advantage of the naïve AC algorithm is clear. However, its performance is very unstable. It is very sensitive to both pattern-set size and structure, and to the type of input traffic. Changing the pattern-set from Snort to ClamAV reduces its performance by 38%-44%. *Exhaustive traversal adversarial traffic drops its performance by 73%-88%* (e.g., in case of Snort with a single thread the throughput drops from 799 Mbps to 93 Mbps; see Fig. 5 for other configurations). Our technique shows much greater stability, and it is most sensitive to the failure-paths traversal traffic which induces a performance degradation of about 30% for Snort and between 20% to 50% for ClamAV.

3) Cache: In order to measure memory accesses and cache misses we used the *Cachegrind* [6] tool. We focus on the data memory read accesses and misses and ignore write accesses and misses and instruction cache misses as these are negligible. Fig. 6 shows the average number of data memory read accesses that the prototype implementation performs per input symbol, for each algorithm and traffic pattern, when using the Snort pattern-set. This figure motivates the great performance advantage of the naïve AC algorithm over our technique when scanning real-life traffic; this advantage stems from the fact that in real-life situations the automaton is, most of the time, in low-depth states, implying that only very small part of the automaton (that fits in L2) is used frequently.

Fig. 7 shows the L1 and L2 data cache miss rates for Snort (ClamAV has a similar behavior which is omitted for brevity). While the naïve AC implementation on real-life traffic has

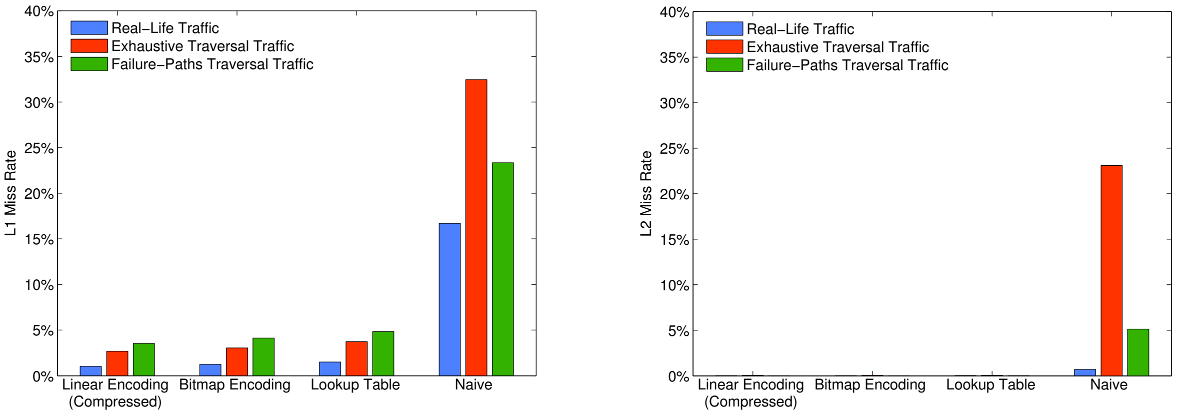
[2] Due to a limitation in the prototype implementation and as we mainly focused in smaller pattern sets, we used only about 50% of the ClamAV signatures. The result AC automaton has 745,303 states in it.



(a) Throughput with Snort pattern-set

(b) Throughput with ClamAV pattern-set

Fig. 5. Throughput of each configuration with different pattern-set and different traffic type



(a) L1 data cache miss rate

(b) L2 data cache miss rate

Fig. 7. Data cache miss rate of each configuration with Snort pattern-set and different traffic type, using a single thread

a noticeable L1 miss rate, its L2 miss rate is relatively low. However, when operating on adversarial traffic, L2 miss rate gets much higher and thus can explain the major slowdown.

As our technique produces much smaller data structures, they tend to remain in L2 most of the time, especially when the pattern-set size is moderate (for Snort, maximal L2 miss rate is 0.06%, when using lookup table encoding and scanning exhaustive traversal traffic).

4) Larger Cache: To verify that the cache is the cause for the severe slowdown of the naïve implementation on adversary traffic, we have used a stronger system with Core i7 processor, which has much larger cache. In addition to a larger cache, it also has four cores (instead of two), where each core is slightly faster. Fig. 8 shows the throughput on this system with Snort pattern-set. Clearly, the naïve implementation achieves significantly higher throughput. However, even on this system, *adversarial traffic reduces its performance by 56%* (from 2.064 Gbps to 904 Mbps). This is still a high rate comparing to our technique. Nevertheless, had the pattern matching process not been the only process to use the CPU and its cache, this advantage would have lost due to more cache misses.

V. Conclusions

DPI is a critical component in next generation network applications, such as security, content filtering, traffic monitoring, load balancing, etc. This paper sheds light on the performance of DPI using software implementation of the seminal AC pattern matching algorithm. The paper shows that implementation details of the algorithm and the system environment have high impact on the performance and space requirement of the algorithm. We demonstrate the role of the system cache in the performance of AC algorithm and show that, if implemented

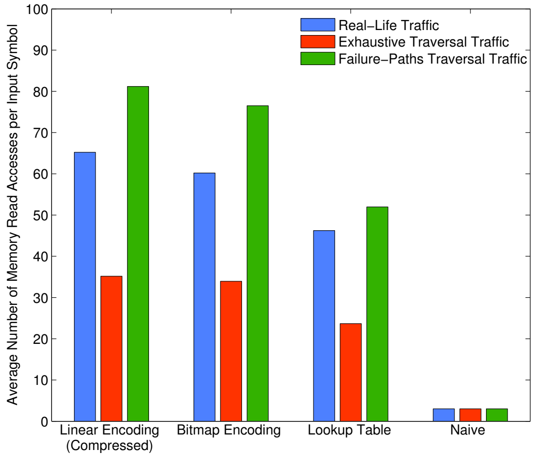


Fig. 6. Average number of memory read accesses per input symbol, for the different configurations and traffic data with Snort pattern-set, using a single thread.

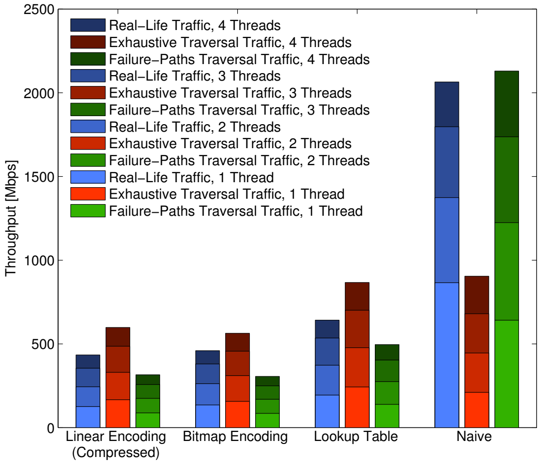


Fig. 8. Throughput of each configuration with different traffic type on the Core i7 processor, with Snort pattern-set. In L2, all implementations of our approach has a miss rate of less than 1%.

naïvely, the real-time throughput is ten times larger than the worst-case throughput. Hence, although AC algorithm has a deterministic one memory reference per char cost, in real-life an attacker can exploit the interaction between the algorithm to the system and launch complexity algorithmic attack against the DPI software. As far as we know, we are the first to show that complexity attack, which takes into consideration the system architecture, (specifically, the cache) can have such a high impact. Using compression techniques, we demonstrate that we can ensure that the whole data structure can fit into the cache, requiring less than a 3.25 bytes per symbol. While such a compression imposes some penalty in performance, the compressed solution is more stable, implying that the worst-case and common case are similar. Thus, compressed implementation should be considered as an important option when aiming at a stable and secure system with guaranteed worst-case performance and moderate memory requirement.

References

[1] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.

[2] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *ACM/IEEE ANCS*, 2007.

[3] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "Evaluating the vulnerability of network mechanisms to sophisticated DDoS attacks," in *IEEE INFOCOM*, 2008.

[4] A. Bremler-Barr, D. Hay, and Y. Koral, "CompactDFA: Generic state machine compression for scalable pattern matching," in *IEEE INFOCOM*, 2009.

[5] A. Bremler-Barr, H. Levy, and N. Halachmi, "Aggressiveness protective fair queueing for bursty applications," in *IEEE IWQoS*, 2006.

[6] Cachegrind (a Valgrind tool). [Online]. Available: http://valgrind.org/info/tools.html

[7] ClamAV, 2010. [Online]. Available: http://www.calmav.net

[8] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Usenix Security*, 2003.

[9] DARPA Intrusion Detection Data Sets, 1998. [Online]. Available: http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html

[10] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," *UCSD Technical Report CS2001-0670*, 2002.

[11] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang, "Reduction of quality (RoQ) attacks on internet end-systems," in *IEEE INFOCOM*, 2005.

[12] D. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the I.R.E.*, 1952, pp. 1098–1102.

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

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

[15] 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.

[16] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants)," in *ACM SIGCOMM*, 2003.

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

[18] M. D. McIlroy, "A killer adversary for quicksort," *Software–Practice and Experience*, pp. 341–344, 1999.

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

[20] T. Peters, 2003. [Online]. Available: http://mail.python.org/pipermail/python-dev/2003-May/035916.html

[21] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a NIDS," in *Annual Computer Security Applications Conference December*, 2006.

[22] SNORT, 2010. [Online]. Available: http://www.snort.org

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

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

[25] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *IEEE INFOCOM*, 2006.

[26] F. Weimer, "Algorithmic complexity attacks and the linux networking code," http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html.

[27] 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.

[28] F. Yu, R. Katz, and T. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *IEEE ICNP*, 2004.