A Hybrid Finite Automaton for Practical Deep Packet Inspection

Michela Becchi
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-4306
mbecchi@cse.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

ABSTRACT
Deterministic finite automata (DFAs) are widely used to perform regular expression matching in linear time. Several techniques have been proposed to compress DFAs in order to reduce memory requirements. Unfortunately, many real-world IDS regular expressions include complex terms that result in an exponential increase in number of DFA states. Since all recent proposals use an initial DFA as a starting-point, they cannot be used as comprehensive regular expression representations in an IDS.

In this work we propose a hybrid automaton which addresses this issue by combining the benefits of deterministic and non-deterministic finite automata. We test our proposal on Snort rule-sets and we validate it on real traffic traces. Finally, we address and analyze the worst case behavior of our scheme and compare it to traditional ones.

Categories and Subject Descriptors
C.2.0 [**Computer Communication Networks**]: General – Security and protection (e.g., firewalls)

General Terms
Algorithms, Performance, Design, Security.

Keywords
Deep packet inspection, DFA, NFA, regular expressions.

1. INTRODUCTION
Increasingly, network packets are classified not only by the fields of their headers, but also by the content of their payloads. In particular, signature-based deep packet inspection has taken root as a dominant security mechanism in networking devices and computer systems. Most popular software tools—including Snort [6][7] and Bro [10]—and devices—including the Cisco family of Security Appliances [8] and the Citrix Application Firewall [9]—use regular expressions to describe payload patterns. While more expressive than simple patterns of exact-match strings, and therefore able to describe a wider variety of payload signatures [12], regular expression implementations demand far greater memory space and bandwidth. As a result of these trends, there has been a considerable amount of recent work on implementing regular expressions for use in high-speed networking applications, particularly with representations based on discrete finite automata (DFAs).

DFAs have attractive properties that explain the attention they have received. Foremost, they have a predictable memory bandwidth requirement. In fact, processing an input string involves one DFA state traversal per character, which translates into a deterministic number of memory accesses. Moreover, it has long been established that, for any given regular expression, a DFA with a minimum number of states can be determined [4][5]. Even so, DFAs corresponding to large sets of regular expressions, each one representing a different rule, can be prohibitively large.

Recent work has tackled this problem in two ways. First, since an explosion in states can occur when many rules are grouped together into a single DFA, Yu et al. [15] have proposed segregating rules into multiple groups and evaluating the corresponding DFAs concurrently. This solution decreases memory space requirements, sometimes dramatically, but increases memory bandwidth linearly with the number of active DFAs. The second approach, proposed by Kumar et al. [15], aims at reducing the memory space requirement of any given DFA and is based on two observations. First, the memory space required to store a DFA strictly depends on the number of transitions between states. Second, many states in DFAs have identical sets of outgoing transitions. Substantial space savings in excess of 90% are achievable in current rule-sets when this redundancy is exploited. The compression technique proposed trades off memory storage requirements with processing time.

Unfortunately, *DFAs are infeasible for regular expressions found in the most frequently used rule-sets*. Specifically, when repeated wildcards are present in a regular expression, it may be impossible to build a DFA with a reasonable number of states. For example, the regular expression "*prefix.{100}suffix*", which matches if and only if "*prefix*" is separated from "*suffix*" by 100 characters, would require well over 1 million states to be represented in a DFA. Since, as we will see, such constructs occur frequently within popular security rule-sets, DFA-based approaches, including the recent work described above, are infeasible as comprehensive solutions.

As an alternative, one could consider using a solution based on non-deterministic finite automata (NFAs) [22]. The number of NFA states required to represent a regular expression is on the order of the number of characters present in the regular expression itself. As an example, the regular expression above would require just 101+ (# chars in *prefix*) + (# chars in *suffix*) NFA states. Therefore, an NFA-based representation would alleviate the memory storage problem. However, an NFA may lead to a variable, and potentially large, memory bandwidth requirement. In fact, multiple NFA states can be active in parallel and each input character can trigger multiple state transitions, and therefore require multiple parallel memory operations. In the worst-case, all NFA states can be active concurrently, requiring a prohibitive amount of memory bandwidth.

In this paper we propose a hybrid DFA-NFA finite automaton (*Hybrid-FA*), a solution bringing together the strengths of both DFAs and NFAs. When constructing a hybrid-FA, any nodes that would contribute to state explosion retain an NFA encoding, while the rest are transformed into DFA nodes. The result is a data structure with size nearly that of an NFA, but with the predictable and small memory bandwidth requirements of a DFA.

We evaluate the hybrid-FA structure by comparing it to both DFA and NFA representations on rule-sets from the popular security package Snort. The primary contribution of the hybrid-FA is that entirely new classes of regular expressions can be implemented in fast networking contexts.

The remainder of this paper is organized as follows. Additional background and motivation are presented in Section 2. Section 3 describes the conditions of DFA state explosion in greater detail. The hybrid-FA structure is introduced in Section 4. Extensions to establish correctness and worst-case bounds are presented in Section 5. Section 6 provides a brief discussion on implementation issues and alternatives. Experimental results are found in Section 7. Further discussion of related work is found in Section 8. The paper concludes with discussion in Section 9.

2. BACKGROUND AND MOTIVATION

Several techniques for minimizing the memory requirements of DFAs representing sets of regular expressions have been recently proposed [15][17][18]. These proposals have some common properties; specifically:

a) They are based on the assumption that a *DFA can be computed* and is given as input.

b) They exploit the observation that DFAs corresponding to rule-sets derived from commonly used Network Intrusion Detection Systems (NIDS) have *significant state transition redundancy*.

The second aspect - i.e., the presence of significant transition redundancy - can be easily explained as follows. Regular expressions used within NIDS typically consist of sets of patterns containing: simple strings, character ranges, wildcards, indefinitely repeating sub-patterns, and sub-patterns repeated a discrete number of times. Notably, nested repetitions—causing loop-backs in the corresponding NFAs and DFAs—are not found in practice. One can think about compiling together the set of regular expressions corresponding to several rules by first building an NFA which represents the disjunction of the NFAs of the single regular expressions, and then converting it to a DFA through the well-known subset construction procedure [4]. Such NFAs will typically have a tree-like structure (with the exception, as we will see, of few loops and backward transitions), where the root corresponds to the starting state and the leaves to the accepting states. If common prefixes are collapsed, the root and the nodes at the first levels will have several outgoing transitions, whereas, moving towards the leaves, the tree will tend to become "skinny", i.e., to consist of long chains of nodes. Moreover, except for the first levels of the NFA and for the transitions representing wildcards and large character ranges, most nodes will have outgoing transitions defined only for a few characters. When building the corresponding DFA, missing transitions on the NFA typically translate into backward transitions to the nodes at the first levels of the hierarchy (or intermediate nodes when the represented regular expression contains dot-star conditions or repetitions of wide character ranges). Thus, a restricted number of nodes in the final DFA tend to be the target of most transitions.

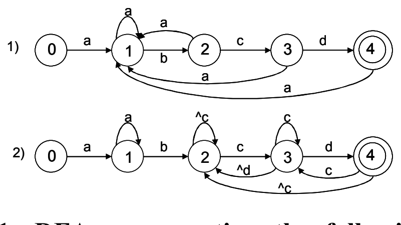

Figure 1: DFAs representing the following RegEx: *abcd* (1) and *ab.*cd* (2). Transitions to state 0 are omitted.

Taking advantage of this redundancy enables very effective memory compression techniques *for a given DFA*, but does not address a major problem. Namely that, due to state explosion during NFA-to-DFA transformation, *DFAs cannot be built for many individual regular expressions and sets of expressions in NIDS rule-sets*. Theoretically speaking, during subset construction, an exponential growth in the number of states can take place. In the case of large NFAs, this can make DFA construction infeasible. In practice, if subset construction is performed "lazily" (i.e., new DFA states are created only when they happen to be targets of any other state), there are only few recognizable cases where this can happen. To this end, we are interested in two distinct situations:

a) State blow-up happens when compiling a single regular expression in isolation.
b) Given two regular expressions RE_1 and RE_2 the corresponding DFA_1 and DFA_2 can be built without incurring state explosion. However, when compiling RE_1 and RE_2 into a unique DFA_{12}, either the number of states in DFA_{12} is significantly greater than the sum of DFA_1 and DFA_2, or DFA_{12} cannot be built at all, due to exponential state explosion.

Clearly, in the first case, DFAs are not a feasible representation of the given regular expression. The second case can be treated by keeping the two DFAs separated and operating them in parallel (i.e., trading memory space for bandwidth). However, given a set of regular expressions, it would be beneficial to be able to predict this situation without testing all possible combinations.

The goals of this work are the following:

- Explore two distinct conditions which lead to the state blow up during subset construction.
- Propose a hybrid automaton which deals with the above problems in a unified way.
- Refine the proposal in order to provide an acceptable worse case bound on the memory bandwidth requirement.

As previously mentioned, the proposal and the results focus on practical data-sets from Snort NIDS. However, these dot-star and counting constraint terms are not unique to the Snort rule-sets. Via personal communication with colleagues at Cisco Systems Inc., we have learned of proprietary IDS regular expression rule-sets for in 14% and 1% of the rules include dot-star and counting terms, respectively.

3. STATE BLOW-UP

Given an NFA with N states, the corresponding DFA can consist of potentially 2^N states [4]. In practice, this upper bound is never reached and, in most cases, the number of states in the DFA is comparable to that of the corresponding NFA. If the NFA represents simple patterns and common prefixes and suffixes have not been collapsed, a state-minimized DFA can actually have slightly fewer states. However, there are common conditions which can bring the number of DFA states close to the theoretical upper bound. An analysis of those conditions within DFAs representing single regular expressions is presented in [15]. Here we want to focus on two patterns which occur frequently in practical data-sets, namely "dot-star" conditions and "counting constraints."

3.1 "Dot-star" conditions

A dot-star condition is a sub-pattern of the type ".*", meaning "a wildcard repeated any number of times." As an extension, we include in this category sub-patterns of the form "*[^$c_1c_2...c_k$]**", where the repetition involves a large range of characters (namely, all characters but c_1, $c_{2,...,}$ c_k). While excluding characters from the repetition introduces some additional issues that we will discuss later, this feature exhibits the same characteristics as a pure ".*" condition in terms of state blow-up.

Dot-star conditions are common in practical data-sets. Their primary use is to detect occurrences of sub-patterns separated by an arbitrary number of characters. In the case of Snort rules, many regular expressions use a "*[^\n\r]**" term to search for an occurrence of the prefix sub-pattern *in the same line* of text as the suffix sub-pattern. Multiple dot-star terms can appear within the same expression.

For example, the Snort spyware rule "*User-Agent\x3A[^\r\n]*ZC-Bridge*", looks for an occurrence of the sub-pattern "*ZC-Bridge*" only if "*User-Agent\x3A*" has been previously detected and no carriage return or new line character occurred in between. That means, the two sub-patterns must occur in the given order and on the same line, and may be separated by an arbitrary number of characters.

In practical rule-sets, dot-star conditions do not cause state blow-up when individual regular expressions are compiled in isolation. In fact, those patterns affect the transitions in the DFA but not the number of states. Figure 1, which compares DFAs accepting regular expressions

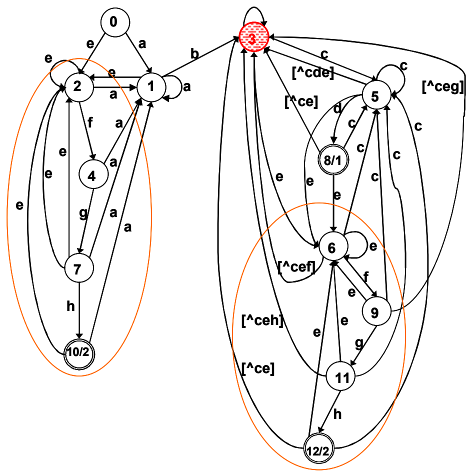

Figure 2: DFA representing (1) ab.*cd and (2) efgh. In the accepting states, the number following the "/" represents the accepted regular expression.

abcd and *ab.*cd*, illustrates this fact. It can be observed that the number of states is the same in the two cases; the transitions are "moved toward" the tail of the DFA in the second one.

However, dot-star conditions add complexity when distinct regular expressions are compiled together (note that the same condition would arise in case of single regular expressions consisting of disjunctions of complex sub-expressions). To see why, assume that we compile together expressions RE_1 and RE_2 (that is, build a DFA for the expression ".*(RE_1|RE_2)") and that RE_1 contains a ".*" term and RE_2 does not. Since the dot-star term in RE_1 can match any string, including all those strings matching RE_2, a properly formed combined DFA will have additional states to determine a match of RE_2 within the ".*" pattern belonging to RE_1. This condition effectively duplicates the *sub-DFA* representing RE_2 within the sub-DFA for RE_1.

Figure 1 illustrates this situation in the composite DFA for regular expressions "*ab.*cd*" and "*efgh*". Notice that the sub-DFA matching "*efgh*" is replicated: first in states 2, 4, 7 and 10, and second in states 6, 9, 11 and 12. The second replica originates from state 3, which derives from expanding the dot-star condition.

If the regular expressions compiled together contain common sub-patterns, the replication may involve only sub-expressions. However, in general, a sub-DFA will be replicated once for every occurrence of a dot-star term in other regular expressions. Thus, dot-star terms create linear increases in the number of DFA states.

3.2 Counting constraints

A counting constraint corresponds to the repetition of a sub-pattern for a given number of times, and is expressed in the form *sub-pattern{n,m}* where *n* and *m* are the minimum and the maximum cardinality of the repetition. If *n* and *m* are equal, the counting constraint is expressed in the form *sub-pattern{n}.*

The most frequent situation occurring in practical data-sets corresponds to the repetition of only one character, which can be a specific symbol, a wildcard or a character within a range. Since we are interested only in situations causing state blow-up, we will restrict ourselves to repetitions of wildcards and of large character ranges.

In Snort, regular expressions with counting constraints are commonly used to detect buffer overflow situations. Again, "*[\n\r]{n}*"-like sub-expressions are utilized in order to split the text string on a line basis. As an example, the Snort rule "*AUTH\s[^\n]{100}*" would detect an IMAP authentication overflow attempt, where the buffer is 100 characters long and a new line terminates the authentication string.

In contrast to dot-star terms, counting constraints on wildcards and large character ranges cause *exponential* state blow-up when creating DFAs even *for single regular expressions*. This can be explained as follows: when expanding the counting constraint, all possible occurrences of the regular expression prefix must be considered at each instance of the wildcard or character range. Figure 3 illustrates this fact on the simple regular expression "*ab.{3}cd*". Clearly, the size of the DFA grows rapidly with the cardinality of the counting constraints.

The situation gets dramatically worse when multiple regular expressions are compiled together in a combined DFA. In this situation, one has to also consider all possible occurrences of the other regular expressions into the one having the counting constraint. We note that counting constraints in typical data-sets consist of at least 100 repetitions; it is therefore impossible to build reasonable DFAs for such rules, much less for groups of them.

4. HYBRID-FA

One obvious way to keep the size of the automaton contained when transforming a NFA into DFA is to interrupt the subset construction operation at those NFA states whose expansion would cause state explosion to happen. In the two specific cases described above, the critical states can be easily determined. In fact, they correspond to the first state of the dot-star constraint (that

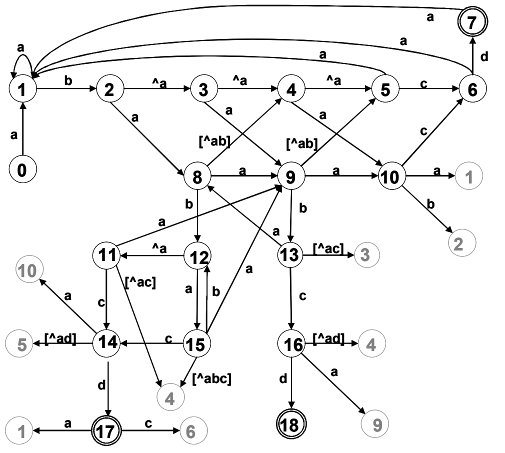

Figure 3: DFA representing regular expression ab.{3}cd.

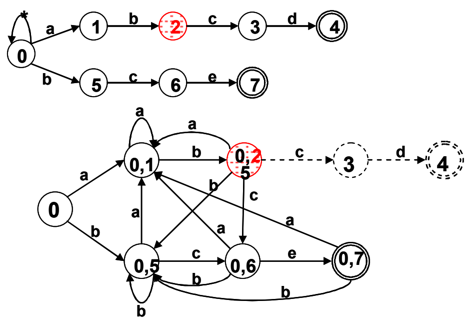

Figure 4: NFA and hybrid-FA for regular expressions abcd and bce. Subset construction is interrupted at NFA-state 2. Within the hybrid-FA, the DFA part is solid whereas the NFA part is dashed.

is, the one with the auto-loop) and the initial state of the repetition sub-expression.

The outcome of interrupting subset construction at an intermediate state will be a hybrid automaton (which we will call hybrid-*FA*), consisting of not expanded NFA-like states, DFA-like states and "*border*" states. The latter can be considered as being part of both a DFA and of an NFA.

Figure 4 shows a small example where subset construction is interrupted at NFA state 2. State numbering in the hybrid-FA reflects the subset construction operation. Since, for instance, processing symbol *a* in NFA state 0 leads to NFA states 0 and 1, processing the same character in (DFA) state 0 of the hybrid automaton leads to a (DFA) state tagged 0-1. It can be noted that the border state 0-2-5 has two distinct outgoing transitions on character *c*: one falling into the NFA-part and one into the DFA-part. Moreover, its sub-state 2 is ignored when computing the transition targets to the DFA-part.

If we restrict ourselves to regular expressions consisting of sequences of sub-patterns possibly separated by dot-star conditions and counting constraints, we start subset construction at the NFA initial state and interrupt it as just described, then the resulting hybrid-FA will exhibit some useful properties. Specifically: i) the starting state will be a DFA-state; ii) the NFA part of the automaton will remain inactive till a border state is reached; and iii) there will be no backwards activation of the DFA coming from the NFA.

In order to better illustrate these concepts, let us consider two examples: the first one containing a ".*" sub-expression and the second one a counting constraint. The goal of the discussion will be two-fold. First, we want to show the characteristics of a hybrid-FA compared to the corresponding DFA and NFA. Second, we want to give an intuition about how the traversal of hybrid-FA works.

4.1 "Dot-star" regular expressions

In Figure 5 the NFA representing regular expressions: *ab.*cd*, *cefc, cad* and *efb* is shown. The double-circled states are accepting states; within them, the number following the slash indicates the accepted regular expression. State 0 is the initial state. The NFA is reduced by merging common prefixes.

The corresponding DFA (which we don't show for

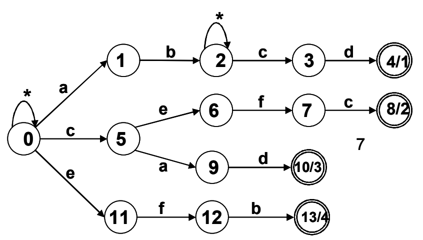

Figure 5: NFA for RegEx: (1) ab.*cd, (2) cefc, (3) cad, (4) efb.

Table 1: NFA traversal example. Stable states are represented in bold; accepting states are underlined.

b	*a*	*a*	*c*	*a*	*b*	*c*	*a*	*c*	*e*	*f*	*c*	*d*	*e*
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	1	1	5	1	**2**	5	1	5	11	12	5	**2**	11
				9		**2**	9	**2**	6	7	8	4	**2**
						3	**2**	3	**2**	**2**	**2**		
											3		

readability) has 21 states. For the reasons explained above, the dot-star term in the first regular expression leads to a replication of the portion of DFA devoted to the second, third and fourth regular expressions. Clearly, such state replication would increase with the number of regular expressions contained in the data set. Moreover, the situation would worsen if the number of regular expressions containing dot-star conditions also increased.

We note that no state explosion would occur if the regular expressions containing dot-star conditions were compiled into separate DFAs; while this would avoid state explosion, it would trade space for bandwidth. In fact, memory bandwidth requirements increase linearly with the number of concurrent DFAs (each DFA makes one state transition for each character).

While reducing the number of states, a NFA representation can increase memory bandwidth requirements. Specifically, the non-determinism inherent in an NFA implies that many states may be active at once. Unlike a DFA, an NFA can make multiple state transitions when consuming a single input character.

The dynamic memory bandwidth needed by an NFA representation depends on the size of the *active state set*, that is, the set of states active in parallel. In fact, the number of active states implies the number of memory accesses required to make state transitions for each input character processed. In theory, processing a character in an NFA requires O(N_{NFA}) memory operations, where N_{NFA} is the total number of states. In practical cases, however, the active set size is much lower than N_{NFA}. Operationally, the number of active states tends to increase if any current state

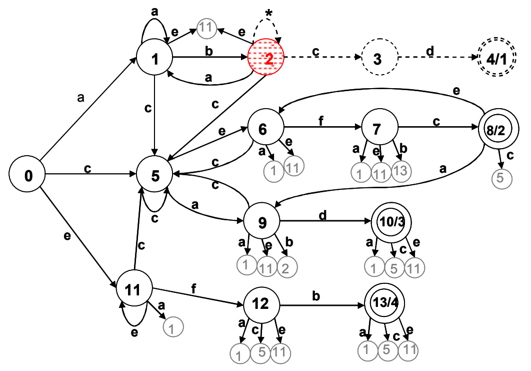

Figure 6: Hybrid-FA for (1) ab.*cd, (2) cefc, (3) cad, (4) efb. Transitions to state 0 in the DFA part are omitted for readability. The DFA part is solid, the NFA part is dashed and the boundary state is red.

Table 2: Hybrid-FA traversal example. Stable states represented in bold; accepting states are underlined.

b	*a*	*a*	*c*	*a*	*b*	*c*	*a*	*c*	*e*	*f*	*c*	*d*	*e*
0	1	1	5	9	**2**	5 **2** 3	9 **2**	5 **2** 3	6 **2**	7 **2**	$\underline{8}$ **2** 3	0 **2** $\underline{4}$	11 **2**

has several outgoing transitions on the given input character. Conversely, it tends to decrease if an active state has no transitions defined on the current input character. An important special case is represented by states having wildcard transitions back to themselves (e.g., states 0 and 2 in Figure 5); these states are *stable:* once they are visited, they will never leave the active set.

An example of traversal of the NFA in Figure 5 with input string "*baacabcacefcde*" is shown in Table 1. In this example, the states in the active set are never more than 5 out of 14.

Let us now consider the hybrid-FA for the given regular expressions (Figure 6). Subset construction is interrupted at state 2, which would cause state explosion to happen. As can be seen, the second, third and fourth regular expressions are completely matched within the DFA part. On the other hand, the first regular expression is matched within the DFA part only up to the second character. Beyond that, the matching operation is performed in an NFA. Note that the number of states in the hybrid FA does not exceed that of the NFA. Finally, the matching operation involves one state traversal per character as long as the border state 2 is not traversed. In other words, as long as the prefix of the first regular expression "*ab*" is not matched, processing is restricted to the DFA portion.

An example of hybrid-FA traversal with text string "*baacabcacefcde*" is shown in Table 2. State 0 is no longer a "stable" state, but state 2 is. As can be seen, the active set will contain only one state until the border state 2 is traversed. One and only one activation of the DFA is possible; conversely, the NFA can have several parallel activations. Note that the size of the active set is in general lower than what we have with the pure-NFA counterpart (with at most 3 versus 5 states).

4.2 Regular expressions with counting constraints on wildcards

Figure 7 represents the hybrid-FA for a small dataset containing a regular expression with a wildcard repeated exactly 3 times. The reader can easily draw the corresponding NFA, also consisting of 19 states. The state-minimized DFA (not shown for readability) has 46 states. In this case subset construction is interrupted at the state

Table 3: Hybrid-FA traversal examples. Stable states represented in bold.

b	*a*	*a*	*c*	*a*	*b*	*a*	*b*	*c*	*e*	*f*	*c*	*d*	*e*
0	1	1	5	9	2	1 14	2 15	5 14 16	6 15	7 16	$\underline{8}$ 3	0 $\underline{4}$	11

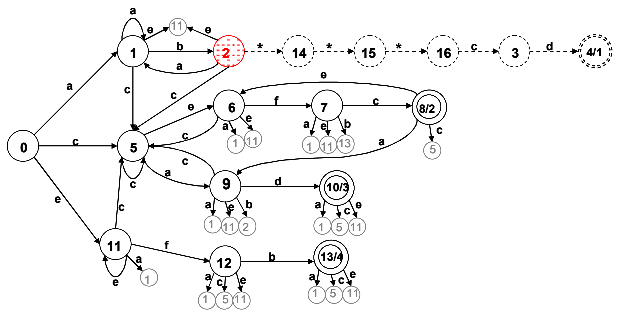

Figure 7: Hybrid-FA NFA for RegEx: (1) ab.{3}cd, (2) cefc, (3) cad, (4) efb

which immediately precedes the counting constraint.

An example of traversal of the hybrid-FA with text string "*baacababcefcde*" is shown in Table 3. Notice that the hybrid-FA does not have any stable states. Again, the DFA is always active and there is a single activation of it during the whole matching operation. On the contrary, the NFA part can have several parallel activations, one for each border state traversal. Note that, if this was not the case, the match reported on state 4 would have not been detected. Finally, the active set size is less than that of the NFA counterpart (whose maximum value is 6).

5. IMPROVING THE WORST CASE

As mentioned, the hybrid-FA consisting of a head-DFA and many tail-NFAs represents a compromise between a mere DFA and a mere NFA solution, and allows dealing with situations where a DFA is unfeasible. In particular, this solution trades memory occupancy (number of states) with processing time/memory bandwidth requirements (size of the active set).

While the described automaton can provide satisfactory average case performance and improves the worst case as compared to a pure NFA, the worst case bound can still result unacceptable. In fact, as it has been pointed out:

- The head-DFA is always active in one and only one state;
- Each tail-NFA is activated each time the border state is reached. Moreover, every activation may involve several states.

Therefore, the theoretical worst case is represented by the number of NFA states present in the hybrid automaton plus one (the DFA active state).

In this section we explore two techniques to further reduce the worst case bound: one suitable to dot-star conditions and the other applicable to counting constraints.

5.1 Tail-DFAs

The first obvious way to limit the worst case active set size is to transform the tail-NFAs into tail-DFAs, as exemplified in Figure 8. In fact, this will ensure that, *for every activation*, each tail-automaton will be active only in one state.

While this technique can be applied to any hybrid-FA, it is effective only in case of dot-star conditions. In the general case, the number of *parallel* activations of a tail-DFA depends on the number of times the border state is traversed. If, for any given DFA, it is possible to compute the minimum number of characters to be processed between two consecutive border-state traversals, this measure is DFA dependent and not likely to provide satisfactory bounds.

In the context of NIDS we are interested in determining the set of rules to be fired on a packet. Thus, it is enough to detect *only one* (the first) possible match of each regular expression. This will allow us to show that, in the case of the most dot-star conditions, *a single tail-DFA activation is sufficient* to have correct traversal and detect all possible matches. This allows us to limit the worst case bound on memory bandwidth/processing time to the *number of sub-DFAs* the hybrid-FA is decomposed into.

In the remainder of this section we provide evidence of this consideration. The reader interested only in the main results can skip to section 5.2.

To demonstrate the above property, we distinguish pure wildcard repetitions from *[^x]*-like* conditions. Note that the following discussion can be directly extended to the more general case $[\hat{}c_1c_2...c_k]*$.

Wildcard-repetitions (.*) Let us assume to have a regular expression of the form *sub-pattern*$_1$*.*sub_pattern*$_2$. This means, "try to match *sub_pattern*$_2$ if and only if *sub-pattern*$_1$ did previously occur in the text string". Operationally, the head-DFA will recognize .**sub-pattern*$_1$ and the tail-DFA will match .**sub_pattern*$_2$. The activation of the tail-DFA will occur upon border-state traversal. This, in turn, will happen once *sub-pattern*$_1$ is matched. Since, upon matching of *sub-pattern*$_1$, we are interested only in the first occurrence of *sub-pattern*$_2$, we may ignore any subsequent activation of the tail-DFA. Also note that, since tail-DFA represents a regular expression starting with ".*", it won't contain any "dead-states" (that is, any stable state which, once reached, will prevent any progress).

[^x]*-like conditions Let us assume to have a regular expression of the form *sub-pattern*$_1$*[^x]*sub_pattern*$_2$. This means, "try to match *sub_pattern*$_2$ if and only if *sub-pattern*$_1$ did previously occur in the text string and the two sub-patterns are not separated by character *x*". Again, the head-DFA will recognize .**sub-pattern*$_1$ whereas the tail-DFA will match *[^x]*sub_pattern*$_2$.

In this case, the tail-DFA will have a *dead-state* which can be reached on character *x* for *some* tail-DFA states. We can safely assume that reaching the dead-state is equivalent

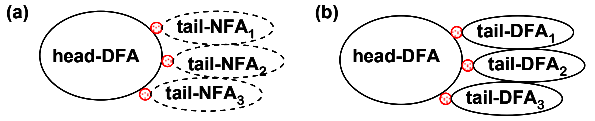

Figure 8: Hybrid-FA exemplification.

to deactivating the tail-DFA.

There are two possibilities: *x* may or may not appear in *sub_pattern*$_2$. Let us consider those two cases separately.

- ***x*** $\notin$ ***sub_pattern***$_2$: All the states in tail-DFA will have a transition to the dead-state on character *x*. This situation is exemplified in Figure 9, where the NFA and the DFA corresponding to *[^x]*abc* are represented.
  Let us assume to reach the border state when the tail-DFA is active. There are two sub-cases:
  - *x* is the last character processed (e.g.: *ax[^x]*abc*). In this case the former activation of tail-DFA will die, and the new activation will be the only one in place.
  - *x* is not the last character processed (e.g.: *ad [^x]*abc*). Since we are interested in the first match of *sub_pattern*$_2$, we can safely ignore the second activation. Notice that, doing that, we don't risk missing matches. In fact, let us assume that an occurrence of *x* followed, which would inactivate tail-DFA. Since such occurrence would follow also the potential second activation, it would invalidate it as well. Therefore, ignoring the second activation is, in this case, safe.
- ***x*** $\in$ ***sub_pattern***$_2$: In this case some tail-DFA states will have a transition to the dead state on character *x*, but some won't. Therefore, depending on the current state, an occurrence of character *x* can cause either a deactivation of the tail-DFA or a progress in the match of *sub_pattern*$_2$. This fact is exemplified in Figure 10, where the NFA and the DFA corresponding to *[^x*]axb* are represented. Note that all states starting from 3 have mismatching transitions leading to state Ø. In this situation, it is in general not true that a single activation of the tail-DFA is always sufficient to preserve correct operation. If the border state is traversed when the tail-DFA is active, discarding one of the two activations is unsafe. In fact, the next transition could invalidate the first one while keeping the second alive. One simple example is given by regular expression *ax[^x]*axb* and string *axaxaxb.*

From the above discussion it should be clear how, in the case of [^x]*-like conditions, we can ensure that keeping only one activation of the tail-DFA preserves correctness only if the sub-expression following the repetition does not contain the characters excluded from the repetition itself. However, there are a few exceptions to this general rules which represent common cases in Snort rule-set.

.* masking: Let us consider rules where the part of regular expression following the first [^x]* condition is a complex sub-pattern containing a ".*" repetition. In other words, let us consider regular expressions of the type: *sub_pattern*$_1$ *[^x]*sub_pattern*$_2$*.*sub_pattern*$_3$. The ".*" condition will "mask" all occurrences of *x* in *sub_pattern*$_3$. Therefore, if *x* does not occur in *sub_pattern*$_2$, then keeping

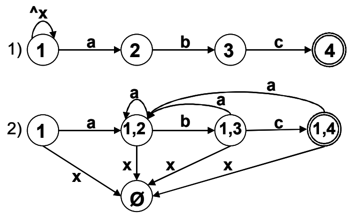

Figure 9: NFA (1) and DFA (2) for regular expression [^x]*abc. The state numeration in the DFA reflects subset construction. State Ø is the dead state. Missing transitions in DFA lead to state 1.

at most a single activation of the tail-DFA will preserve correct operation.

Overlapping tail-DFA activations: A second case which occurs frequently in Snort rule-sets can be described as follows: *sub_pattern*$_1$ is a simple string and tracing it from any state of the tail-DFA always bring to its entry state. In this case, one can ensure that a new activation of the tail-DFA will take place either if such DFA is inactive, or if it finds itself in the entry state. Therefore, two consecutive activations will always overlap.

The argument above, which refers to regular expressions in isolation, can be easily extended to groups of regular expressions sharing a common prefix (at least up to the dot-star repetition included).

5.2 Counter mechanism

Even if applicable, tail-DFAs would not be effective in addressing counting constraints. In fact, for correct operation, a new activation of the tail-DFA is required each time the border state is traversed. To have an intuition about this fact one can consider the simple regular expression *ab.{3}cd*, whose head- and tail- DFAs are represented in Figure 12, and the text string *ababxyzcd*. Ignoring the second tail-DFA activation would in this example lead to missing the match on the last character.

Since, in the worst case, the tail-automaton can be activated every clock cycle, the bound does not improve with a DFA solution. We will therefore think of a mechanism to limit the number of state traversals starting from a tail-NFA. For an exhaustive discussion on a general methodology to handle this case we address the reader to our technical report [20].

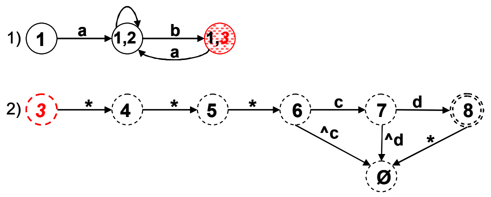

Figure 11: head-DFA (1) and tail-DFA(2) for regular expression ab.{3}cd. The missing transitions in the head-DFA are to state 1. The state numbering is according to subset construction.

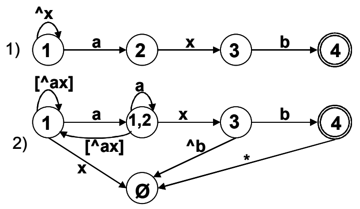

Figure 10: NFA (1) and DFA (2) for regular expression [^x]*axb. The state numeration in the DFA reflects subset construction. State Ø is the dead state. All the transitions are represented.

Let us first consider counting constraints of the form "*.{n}*", where the wildcard is repeated exactly *n* times. Figure 12 shows the NFA for the generic *.{n}suffix* regular expression. As can be seen, the NFA consists of *n-1* similar states (from *b+1* to *b+n-1*), each having all outgoing transitions directed towards the next state of the chain. Those states simply operate as a counter. The last state of the sequence *b+n* is the first one whose outgoing transitions represent progress information within the suffix.

The same information could be simply stored through an auto-decrementing counter and a pointer to state *b+n*. The counter can be activated and set to *n* when the border state is reached. At each character processed, the counter gets auto-decremented. Only when the counter is nullified the state associated to the corresponding pointer is accessed.

The worst-case is characterized by *n* active counter instances plus the size of the suffix-NFA. However, it can be noticed that the counters can be kept in on-chip memory, and do not involve real state traversals. Moreover, as we point out in [20], a proper representation allows the update and query of *at most two counter instances* to suffice for correct operation.

The *[^*$c_1c_2...c_k$*]{n}* condition can be treated in a similar way; in this case the counter should be associated the set of characters $c_1c_2...c_k$ which would cause its de-allocation.

A special case which is very common in practice is the one where the counting constraint is located at the end of the regular expression. In this situation, *a single counter instance* always suffices independent of the number of times the border state is traversed. In fact, in case of wildcard repetitions, the occurrence of the first *n* wildcards will determine a match. In case of *[^*$c_1c_2...c_k$*]{n}*-like counting constraints, an occurrence of an invalidating c_i character within *n* characters from the oldest tail-NFA activation would be also within *n* characters from any newer parallel one. Therefore, it is in this case safe to

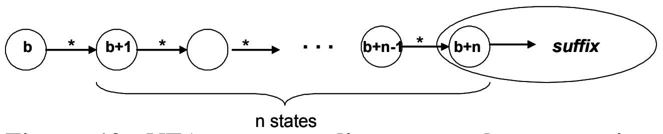

Figure 12: NFA corresponding to regular expression *.{n}suffix*

Table 4: Summary of Snort rule-sets

Rule-set	Nr. of rules	Header					Characteristics	
		Protocol	*Source IP*	*Src. Port*	*Destination IP*	*Destination Port*	*.* and [^x]**	*.{n,m}*
Group1	329	Tcp	$HOME_NET	any	$EXTERNAL_NET	$HTTP_PORTS/any	283	-
Group2	40	Tcp	$HOME_NET	any	$EXTERNAL_NET	25/any	24	-
Group3	18	Tcp	$EXTERNAL_NET	any	$HOME_NET	7777:7778/any	5	10
Group4	45	Tcp	$EXTERNAL_NET	any	$HOME_NET	143/any	24	19
Group5	20	Tcp	$EXTERNAL_NET	any	$HOME_NET	119/any	6	11
Group6	24	Tcp	$EXTERNAL_NET	any	$HOME_NET	110/any	7	12

ignore subsequent activations of the tail-NFA thus keeping at most one active counter.

Counting constraints of the form *.{n,}* and *[^$c_1c_2...c_k$]{n,},* where *at least n* occurrences of the wildcard/character range are of interest, can be treated as a direct generalization of the above. In the NFA of Figure 12, this would correspond to adding an auto-loop to state *b+n* on the same character range in the repetition. Thus: i) the counter mechanism can also be applied to states from *b+1* to *b+n-1.* ii) Additionally, the suffix (of which state *b+n* is the entry state) can be converted to DFA. Again, in the case of wildcard repetitions or if the invalidating characters $c_1c_2...c_k$, do not appear in the suffix, a single activation of the suffix-DFA does always guarantee proper operation.

Finally, cases *.{n,m}* and *[^$c_1c_2...c_k$]{n,m}* can be treated as follow. Upon traversal of the border state *b*, the auto-decrementing counter is set to *m*, and a fixed value *m-n* is associated to it. Once again, the counter will be dropped once nullified (or upon occurrence of any invalidating character c_i). However, state *b+m* is accessed for every value of the counter less than or equal to *m-n*. The considerations above about the worst case apply to this situation as well.

In conclusion, if the data-set contains N_T counting constraints located at the end of the corresponding regular expression and N_{NT} counting constraints in intermediate positions, then the worst case bound on memory bandwidth is reduced to $N_T + 2N_{NT} + 1$ ("one" representing the head-DFA activation) memory accesses per character processed.

6. MEMORY LAYOUT

One important point to address to implement the proposed scheme is how to layout the data structure representing the above automaton so to limit memory requirement and allow an efficient state traversal.

As far as the DFA part (head-DFA and possible tail-DFAs) is concerned, any compression technique proposed in literature [15][17][18][19] can be reused.

Let us now address the encoding of the NFA portion of the automaton. Content addressing, a technique proposed in [16] in the context of DFAs, can be adapted to NFAs. The goal is to limit the number of memory accesses when processing a state without any transitions defined on the current input character. Specifically, one can observe that the most part of NFA states have either transitions defined on a very small set of characters, or on all but one or two characters. Thus, by encoding in the state identifier the information about the set of symbols a transition is (or is not) defined on, it is possible to limit the number of memory accesses below the active set size. For details the interested reader can refer to [15].

Finally, border and counter states should be treated in a special way: the former imply the need for pointers from the DFA to the corresponding NFA entries, whereas for the latter the information listed in Section 5.2 must be stored.

7. EXPERIMENTAL RESULTS

In this section we validate the proposal on rule-sets from the Snort IDS.

7.1 Rule-sets

The rule-sets considered have been taken from the Snort IDS [7]. Specifically, since some Snort rules only use exact-match strings, in this paper we only consider those having a Perl Compatible Regular Expression (PCRE) in their firing condition.

As mentioned before, the rules under consideration do not exhibit the whole expressive power of regular expressions. Rather, they can normally be decomposed into sequences of simple sub-patterns separated by dot-star conditions (either in the pure *.** or in the *[^$c_1c_2...c_k$]** form) and counting constraints on wildcards and character ranges.

Character ranges (and their repetition) are very common within sub-patterns. Specifically, they appear either in the form *[c_1-c_k],* or as special escape sequences: *\s* (all space characters), *\S* (all but space characters), *\d* (digits), *\D* (all but digits), *\w* (alphanumeric characters) and *\W* (all but alphanumeric characters).

Nested repetitions and disjunctions of complex sub-patterns (e.g.: patterns containing dot-star conditions or wildcard repetitions) have not been observed in the rule-sets. We expect that these more general types of patterns will be the subject of important future work, but we do not consider them here since they are not found our rule-sets.

Of 982 distinct regular expressions: 25% contain long counting constraints, generally located at the end of the regular expressions, 11.4% contain .* conditions and 54.89% *[^$c_1c_2...c_k$]** conditions.

A large part of Snort rules start with character “^”, which normally forces the match operation only at the beginning of the text string (i.e., of the packet payload). This could theoretically decrease the complexity of the corresponding DFA, and avoid state explosion when regular expressions with counting constraints are compiled in isolation. Unfortunately, nearly all Snort PCREs use the “*m*” modifier. Combined with symbol “^” at the beginning of the regular expression, this forces the match operation not only at the beginning of the text string, but also at the beginning of each line. In other words, the *m* modifiers acts on regular expression *^pattern* transforming it into *^pattern | ([\n\r]pattern)*. This, in turn, keeps the complexity of the resulting DFA high.

The Snort IDS performs packet payload inspection only after header filtering (i.e. packet classification). Therefore, we clustered rules with common header and performed experiments on some of the largest groups. A summary of the derived rule-sets is presented in Table 4.

7.2 Memory storage requirement

In this section, we study the memory storage requirement of the different rule-sets by generating the corresponding automata. As can be observed in the second and in the last two columns of Table 4, the rule-sets differ in the number of regular expressions, dot-star conditions and counting constraints they include.

Rule-sets *group1* and *group2* do not contain counting constraints, whereas *group3*-*group6* do. The counting constraints encountered consist of 20 to 1024 repetitions of large character ranges; however, they are always located at the end of the corresponding regular expressions (and can therefore benefit in the best way of the counter mechanism). Moreover, the *[^x]**-like conditions present in the rule-set make it possible to build tail-DFAs which can be safely traversed with at most one activation.

In Table 5 a summary of the characteristics of the size of the NFA, DFA and hybrid-FA corresponding to the given rule-sets are reported. Consider the following observations.

First, a DFA solution is never feasible in the case of rule-sets containing counting constraints. In fact, because of the high number of repetitions, exponential state explosion is observed also if DFAs are generated in isolation for each of the regular expressions. Therefore, in those cases, N-A (not applicable) is indicated in the table (experimentally, subset construction was aborted after generation of 10 million states).

For rule-sets *group1* and *group2*, a DFA solution is possible but the regular expressions must be distributed across multiple DFAs in order to avoid state-blow-up. Rule partitioning is performed according to heuristics, as follows. First, rules containing a dot-star condition and sharing the prefix to it are compiled together. Second, rules containing multiple dot-star conditions are compiled in isolation or in combination with just a few other rules. In effect, as explained in Section 3, each dot-star condition tends to generate a replication of the DFA being merged with the current regular expression.

When creating the hybrid-FA, tail-DFAs and the counter mechanism have been used in order to limit the worst case bound. Because of the varying complexity of the rule-sets, dot-star conditions have been treated in different ways: some of them have been expanded through subset construction, and some have been made border-states. Specifically, the goal was the one of keeping the head-DFA below 50,000 states. This threshold was selected as a good head-DFA target size because proposed DFA compression techniques [15][17][19] can encode those states in around 2MB, a size that can be realized in on-chip memory in an ASIC or microprocessor. We can create a head-DFA of any specific number of states by expanding the head DFA in a greedy fashion until the target size has been reached; thereafter, all dot-star conditions become border-states and lead to tail-FAs. As a result, all dot-star conditions in *group1* have been moved to tail-DFAs, the ones in *groups 3-6* have been expanded in the head-DFA, and a mixed solution has been adopted for *group2.*

In the case of rule-set *group2,* two different DFA groupings have been tested: the first consisting of the same number of DFAs as the hybrid-FA, and the second consisting of one less DFA. As one could expect, in the first scenario the overall number of states in the two automata is similar, whereas in the second one the pure DFA solution pays for the better worst case performance bound with a higher memory occupancy (specifically, it requires 50% more states).

In the case of rule-set *group1,* the pure-DFA and the hybrid-FA solution have comparable size. But, as will be pointed out in the next section, the hybrid automaton is preferable in terms of average case memory bandwidth.

For rule-sets *group3-6,* the DFA cannot be constructed at all due to exponential state blow-up, while the hybrid-FA solution has an easily realizable size. Also, it is worth noticing the reduction in the number of states when moving from a NFA to a hybrid-FA, which is due to removing the long chains of counting states.

Table 5: Automata sizes for corresponding rule-sets.

Rule-set	**NFA**	**DFA**		**Hybrid-FA**		
	# states	**# DFAs**	**Total states**	**# tail-FA**	**head-DFA states**	**Total tail-states**
Group1	15679	31	71234	30	40461	30321
Group2	1036	3 2	22651 31521	2	20724	1905
Group3	8871	N-A	N-A	10	514	-
Group4	3119	N-A	N-A	19	2560	-
Group5	5205	N-A	N-A	11	2485	-
Group6	1952	N-A	N-A	12	4878	-

Table 6: Active vector sizes for Snort rule-sets

	NFA		Hybrid-FA		
	Avg	Max	Avg	Max	Worst case
Group1	1.15	34	1.009	5	32
Group2	1.06	13	1.001	2	3
Group3	1.04	4	1.002	2	11
Group4	2.45	12	1.001	2	20
Group5	1.04	5	1.001	2	12
Group6	2.99	6	1.088	2	13

In terms of absolute memory occupancy, the use of default transitions [15][19] and of content addressing [17] to encode the hybrid-FA lead to storage requirements varying from 21KB (*group3*), up to 3MB (*group1*). In fact, the former technique allows eliminating around 98-99% of the DFA transitions, while the latter imply the use of 64 bit wide state identifiers. Notice that this range makes it possible to accommodate the automaton data structures in on-chip memory [26].

7.3 Memory bandwidth requirement

The memory bandwidth requirement can be expressed in terms of the number of memory operations to be performed for each input character processed. In this section, we want to compare the different automata in both worst case and average case behaviors.

7.3.1 Worst case behavior

The worst-case memory bandwidth requirement can be seen in Table 5. In the case of NFAs, the worst-case bandwidth corresponds to the number of states, which is reported in the second column of the table. For example, 1036 states corresponds to 1036 concurrent memory operations to implement state transitions for each input character processed. This bound, even if rarely achieved, is clearly unacceptable.

DFA solutions have a worst case bound corresponding to the number of DFAs needed to represent the regular expressions (i.e.: the number of groups the rule-set is decomposed into). This value (column 3) is attractive when those solutions are feasible, that is when the regular expressions do not contain large counting constraints.

In the case of Hybrid-FAs, the worst case bound is equal to 1 plus the number of tail-DFAs (each tail-DFA being a simple counter for rule-sets *group3-6*). For data-sets *group1-2*, this coincides with the worst case bound of the DFA counter-part. In case of counting constraints, this value is far less than that of the NFA solution, and depends only *on the number of regular expressions* (as opposed as to the number of states).

7.3.2 Average case behavior

To evaluate the average case memory bandwidth, we compare the behavior of the different solutions on real traffic. To this end, we perform simulations using twelve packet traces downloaded from [25] of size varying from about 17MB to about 264MB.

Table 6 reports statistics about the size of the active vector across the different data-sets. The average values have been derived by first computing, for each trace, the weighted average of the active vector size across the simulation interval. Then, the values obtained for different traces on the same rule-set have been again averaged. For a given rule-set, we have not observed a substantial variance across the traces. The maximum value displayed is, in all cases, the maximum active vector size achieved for a particular rule-set across all traces and all simulations.

As can be seen, the average behavior of the NFA solution is far better than what the worst case would indicate. This is due to the fact that only a few rules are matched, and dead branches in the NFA are often taken.

The hybrid-FA outperforms the NFA both in terms of average behavior and maximum active vector size. In fact, the automaton traversal remains for the most part within the head-DFA. Note that a value of 1 could be achieved only if it was possible to compile all the regular expressions in a single DFA.

Finally, since the average case behavior of a DFA solution is the same as its worst case, the hybrid-FA outperforms also the DFA solution with regular expression grouping adopted for the *group1* and *group2* data-sets.

8. RELATED WORK

Regular expression matching at line rate has been recognized as an important problem, and has been considered in related work. The prior work in this area focuses on two distinct directions: FPGA based implementations [22][23][24] and general-purpose or software oriented approaches [6][15][15][17][18][19]. Our work falls into the second category, although one could offload the tail-automaton operation to an FPGA.

As mentioned, memory compression techniques allowing an efficient representation of generic DFAs have been presented in [15][17][18][19]. However, such proposals assume that the DFA is given a priori. On the opposite, in this work we address the case where a DFA is either practically unfeasible or not a suitable representation of the regular expressions of interest.

Our work has a practical character in that it does not address generic regular expressions, but particular subclasses which are common in broadly used NIDS [6] [7] [8]. To this end, our work has commonalities with the one presented in [15], which proposes rewriting rules to simplify DFA in the case of common patterns. However, our focus is different in that we concentrate on the automaton rather than on modifying the input patterns.

It is worthwhile to compare and contrast the hybrid-FAs presented in this work and lazy-DFAs [12]. The two proposals share the common idea of partially performing subset construction on a NFA. However, lazy-DFAs assume that subset construction is done dynamically depending on the input string (that is, on the incoming packets' payload.)

Specifically, the NFA paths covered by the input string are dynamically converted to DFA. While this may be helpful in the average case, it does not address the worst case, which is of first interest in the context of NIDS. Moreover, this solution is suitable only for a software implementation. Therefore, we assume that the partial subset construction be done statically a priori so to prevent state explosion from happening. Moreover, we introduce refinements to further bound the worst case.

9. CONCLUDING REMARKS

Regular expression matching is an important task in modern NIDS. Recent proposals have in their experimental evaluations drawn selectively from regular expression rule-sets to avoid troublesome rules. For example, fully 25% of the regular expressions in the current Snort rule-set include counting constraints for which no DFA can be constructed using a reasonable amount of memory, such as that normally found in a workstation or PC. In all prior work we have seen, these rules have been excluded from discussion or evaluation, presumably for this reason.

The primary contribution of this work is the hybrid-FA, which is, to our knowledge, the first automaton that is capable of evaluating all the regular-expression types found in common NIDS systems such as Snort and can be implemented efficiently in practical high-speed systems.

The key characteristics of a hybrid-FA are: a modest memory storage requirement comparable to those of an NFA solution, an average case memory bandwidth requirement similar to that of a single DFA solution (although the DFA would be unfeasibly large), a worst case memory bandwidth linear in the number of regular expressions containing counting constraints and dot-star conditions (and, notably, independent of the number of states in the automaton).

10. ACKNOWLEDGEMENTS

This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794, and by gifts from Intel and Cisco Systems.

REFERENCES

[1] A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search,” Communications of the ACM, pp. 333–340, 1975.

[2] B. Commentz-Walter, “A string matching algorithm fast on the average,” in ICALP, July 1979.

[3] S. Wu, U. Manber, “A fast algorithm for multi-pattern searching,” Tech. Report TR-94-17, Univ of Arizona, 1994.

[4] J. E. Hopcroft and J. D. Ullman, “Introduction to Automata Theory, Languages, and Computation,” Addison Wesley, 1979.

[5] J. Hopcroft, “An nlogn algorithm for minimizing states in a finite automaton,” in Theory of Machines and Computation, J. Kohavi, Ed. New York: Academic, 1971, pp. 189--196.

[6] M. Roesch, “Snort: Lightweight Intrusion Detection for Networks,” in System Administration Conf., 1999

[7] Snort: http://www.Snort.org/

[8] Cisco Systems. Cisco ASA 5505 Adaptive Security Appliance. http://www.cisco.com. 2007.

[9] Citrix Systems. Citrix Application Firewall. http://www.citrix.com. 2007.

[10] Bro: http://bro-ids.org/

[11] Vern Paxson et al., “Flex: A fast scanner generator,” http://www.gnu.org/software/flex/

[12] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context.”, in CCS 2003.

[13] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” in Infocom 2004.

[14] L. Tan, and T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” ISCA 2005.

[15] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection”, in ANCS 2006

[16] S. Kumar et alt., “Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection,” in ACM SIGCOMM, Sept 2006.

[17] S. Kumar, et alt, “Advanced Algorithms for Fast and Scalable Deep Packet Inspection”, in ANCS 2006

[18] M. Becchi and S. Cadambi, “Memory-Efficient Regular Expression Search Using State Merging”, in INFOCOM 2007

[19] M. Becchi and P. Crowley, “An Improved Algorithm to Accelerate Regular Expression Evaluation”, in ANCS 2007

[20] M. Becchi and P. Crowley, “Addressing complex regular expressions through counting automata”, Washington University Tech. Report, July 2007.

[21] R. W. Floyd, and J. D. Ullman, “The Compilation of Regular Expressions into Integrated Circuits”, Journal of ACM, vol. 29, no. 3, pp 603-622, July 1982.

[22] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs", in FCCM 2001

[23] C. R. Clark and D. E. Schimmel, “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” in FLP 2003.

[24] J. Moscola et alt., “Implementation of a content-scanning module for an internet firewall,” in FCCM, USA, April 2003.

[25] Internet traffic traces: http://cctf.shmoo.com/

[26] Cu-11 standard cell/gate array ASIC, IBM. www.ibm.com