Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions

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

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

ABSTRACT

Regular expression matching is a crucial task in several networking applications. Current implementations are based on one of two types of finite state machines. Non-deterministic finite automata (NFAs) have minimal storage demand but have high memory bandwidth requirements. Deterministic finite automata (DFAs) exhibit low and deterministic memory bandwidth requirements at the cost of increased memory space. It has already been shown how the presence of wildcards and repetitions of large character classes can render DFAs and NFAs impractical. Additionally, recent security-oriented rule-sets include patterns with advanced features, namely back-references, which add to the expressive power of traditional regular expressions and cannot therefore be supported through classical finite automata.

In this work, we propose and evaluate an extended finite automaton designed to address these shortcomings. First, the automaton provides an alternative approach to handle character repetitions that limits memory space and bandwidth requirements. Second, it supports back-references without the need for back-tracking in the input string. In our discussion of this proposal, we address practical implementation issues and evaluate the automaton on real-world rule-sets. To our knowledge, this is the first high-speed automaton that can accommodate all the Perl-compatible regular expressions present in the Snort network intrusion and detection system.

1. INTRODUCTION

Finding patterns of interest within large datasets is a central task in many applications and has been a well-studied area of research for many years. However, there exist contexts where the design of high-performance pattern matching sub-systems is still challenging. In particular, this is the case with the networking domain, which includes several applications where packet payloads must be inspected at line rates (up to several gigabits per second) against large data-sets, sometimes consisting of thousands of patterns. Examples include network intrusion detection and prevention systems (e.g., Snort [6][7], Bro [8], Cisco Security Appliance [10], Citrix Application Firewall [11]), email scanning systems (ClamAV [9]), application-level filtering and content-based routing [12].

While a substantial amount of work has focused on exact-match string search, research interest has recently moved toward designing data structures, algorithms and architectures to support regular expressions, which are more expressive than exact-match strings and therefore able to describe a wider variety of pattern signatures [13][14]. The basic challenge with high-speed regular expression evaluation is to minimize both memory space and memory bandwidth.

Finite automata (FA) are typically used to represent regular expressions [2]. Two classic automata are used for this purpose, and each has its strengths and weaknesses. Non-deterministic finite automata (NFAs) have the benefit of a limited memory space requirement, which is dependent only on the number of characters present in the set of regular expressions. However, in the worst case, the matching operation requires up to N_{NFA} state traversals per input character processed, where N_{NFA} is the number of states in the NFA. Assuming at least one memory access per state traversal, this may require an unacceptable amount of memory bandwidth in high-speed contexts. On the other hand, deterministic finite automata (DFAs) offer the advantage of a limited memory bandwidth requirement. In particular, they require only a single state traversal for each input character processed, independent of the number of regular expressions in the data-set. However, the memory space required to encode a DFA representing a set of regular expressions can increase exponentially as compared with an NFA representation, a fact that often renders DFAs infeasible for practical rule-sets [16][20].

Practical rule-sets often include three categories of patterns that make FA implementations problematic. The first are unbounded repetitions of sub-patterns, particularly those involving wildcards and character ranges. These “dot-star terms” have a less dramatic effect and can be handled through proper regular expression clustering [16] (an approach that has severe scalability limits). The second are bounded repetitions, or “counting constraints,” in which a pattern is repeated a specific number of times; these are more troublesome and can render DFA and NFA solutions impractical, as we will see, due to their unsustainable memory storage and bandwidth requirements [20].

CoNEXT 2008, December 9-12, 2008, Madrid, Spain
Copyright 2008 ACM 978-1-60558-210-8/08/0012 ...$5.00.

The final problematic type of pattern found in practical rule-sets is the back-reference, in which a matching substring in the prefix is matched again later in the input string. Back-references add to the expressive power of regular expressions, because regular languages cannot evoke this capability and cannot therefore be handled through the classic finite state machines. Therefore, implementations based on pure NFAs or DFAs fail to support back-references altogether.

To be concrete, let us consider the rule-set characteristics from the popular Snort network intrusion and detection system (NIDS) [7]. As of November 2007, 5,549 of the 8,536 Snort rules contain at least one Perl-Compatible Regular Expression (PCRE). Among these, 905 (16.3%) and 2,445 (44%) contain unbounded and bounded repetitions of large character classes, respectively, and 338 (6%) include back-references.

The practical implications of these observations are dramatic. First, none of the existing DFA-based proposals for high-speed network regular expression evaluation can handle the 44% of the Snort PCREs containing large counting constraints (even a single rule with a large counting constraint renders these infeasible). Second, DFA-based designs need to partition the 16.3% of regular expressions containing dot-star terms in order to compile them into feasible data structures. Note that this is independent of the use of efficient DFA compression techniques [17][19]. Rule partitioning implies that several DFA instances must be created and operated in parallel, which requires an increase in memory bandwidth linear in the number of DFAs. Therefore, 60.3% of Snort regular expressions are not handled in a practical manner with DFA-based solutions. As we will see, this has led various groups to propose NFA-based architectures. Finally, none of the existing solutions based on finite automata, either DFAs or NFAs, can handle the 6% of Snort rules containing back-references.

The problem of unconstrained repetitions of large character classes has recently been addressed in [20] and [21]. However, as we discuss in Section 2, neither of these proposals treat counting constraints in an exhaustive way. Finally, back-references have been fully omitted from previous work.

In this paper, we propose an extended automaton to efficiently handle counting constraints and back-references. This allows us to cover all the patterns in the Snort rule-set, which is to our knowledge the most popular and expressive publicly available NIDS. We focus on a solution that can be implemented using a restricted amount of storage and that requires as little memory bandwidth as possible. Specifically, our contributions are both theoretical and practical, and can be summarized as follows:

- We design a *counting-FA* that is functionally equivalent to a pure finite automaton, requiring a limited amount of storage and a finite and deterministic number of memory accesses per character processed. The automaton is proposed in both deterministic and non-deterministic form.
- We design an extended-automaton that handles back-references with an NFA-like operation.
- We describe a practical state representation that is compatible with the compression and encoding techniques used on standard finite automata.
- We propose suitable system architecture.

For large rule-sets, we show how the proposed extended automaton can be integrated with the hybrid-FA described in [20] and, in fact, how our proposal can be viewed as a natural extension of it. Also, our extended deterministic finite automata can be used to generalize existing techniques based on multiple DFAs [25] in order to handle regular expressions with counting constraints.

The remainder of this paper is organized as follows: In Section 2, we provide additional background and describe our contributions in the context of previous work. In Section 3, we present the counting automaton with a motivating example. In Section 4, we discuss the back-reference problem and describe our solution to it. In Section 5, we discuss a specific and practical encoding and compression technique that can be used to store the automaton. In Section 6, we extend the scheme to encode multiple regular expressions. In Section 7, we propose a system architecture that can use the automaton to implement high-speed regular expression evaluation. In Section 8, we provide experimental evaluations on real data-sets. We conclude in Section 9.

2. BACKGROUND

The prior work in the area of regular expression matching at line rate can be categorized by distinct implementation targets: FPGA-based implementations [22][23][24] [25][26] and approaches suitable for deployment on a general-purpose processor or on ASIC hardware [15][16][17][18][19][20][21]. The extended automaton proposed in this paper can be applied to all these implementation scenarios. However, we reserve the evaluation on FPGAs for future work.

The two main advantages of FPGAs reside in their intrinsic reconfiguration capability and parallelism. The first aspect is less significant for the problem at hand because, currently, the update frequency in practical rule-sets is relatively low. Parallelism can be (and is) exploited to run several regular expression engines—either in the DFA or in the NFA form—at the same time. FPGA-specific solutions have the following main disadvantages. First, they operate at a lower clock frequency compared with general-purpose processors and ASIC solutions; and, second, they lack in scalability because the number of supportable rules is a function of FPGA resources rather than external memory.

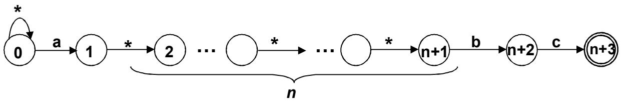

Figure 1: NFA accepting RegEx .a.{n}bc***

An interesting NFA implementation for FPGAs is described by Sidhu and Prasanna [22]. The main idea is to encode each state in a flip-flop to allow each character to be processed in constant time. However, Franklin et al. [23] show how, in practice, the performance of FPGA-based NFA designs can decrease rapidly as the character density increases. A recent design proposed by Mitra et al. [26] shows that up to 250 PCREs from Snort can be accommodated on a single FPGA. Clearly, this may make the deployment of the whole Snort rule-set costly and cumbersome.

The main advantages of algorithmic approaches suitable for implementation on general-purpose processors and ASIC hardware are generality, versatility and availability of higher clock frequencies. In this context, memory storage and bandwidth requirements represent the main issues.

A substantial body of research work has focused on compression techniques aimed at reducing the amount of memory needed to represent DFAs. In particular, Kumar et al. [17] proposed an algorithm to compress a DFA through the introduction of default transitions, a generalization of the failure pointer concept presented in the classical Aho-Corasick algorithm for string-matching [1]. Their work is based on the idea of trading of memory storage requirement with processing time. A more general and less complex algorithm to achieve the same goal was recently proposed by Becchi et al. [19]. By restricting default transitions to be backward directed, they can achieve a better worst case bound on the processing time, while offering the same compression degree of [17]. Unfortunately, as mentioned in the introduction and as detailed in [16][20], pure DFA-based approaches cannot deal with the complexity of regular expressions present in practical rule-sets. In fact, the presence of large character class repetitions may make it impossible to practically compute a DFA. While the problem can be limited in the case of unconstrained repetitions through regular expression partitioning [16][25] (by sustaining greater memory bandwidth), counting constraints still remain an open issue. To get a concrete sense of this issue, it can be observed that the above compression algorithms have been tested on very limited rule-sets: either on large sets of simple patterns or on sets containing fewer than ten-to-twenty complex regular expressions.

Two recent works [20][21] aim at overcoming the limitations of pure DFA- and NFA-based solutions in the context of traditional regular expressions.

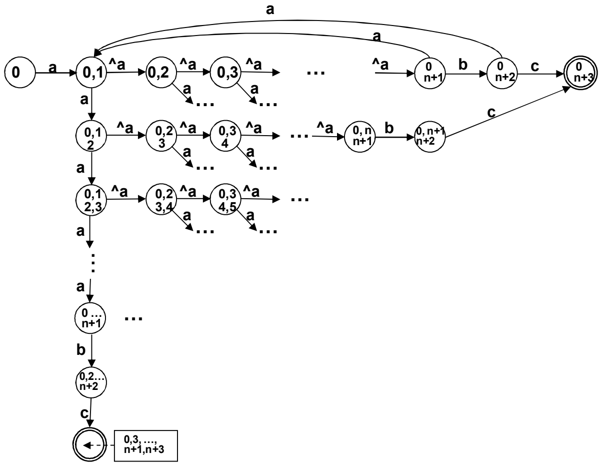

Figure 2: Sketch of DFA accepting RegEx .a.{n}bc***

In particular, in [20] Becchi and Crowley propose a hybrid-FA with a memory storage requirement comparable to that of an NFA and a memory bandwidth requirement dependent on the number of regular expressions presenting large character class repetitions. The basic idea is to perform *partial* NFA-to-DFA conversion, thus preventing state explosion from happening. The outcome is a hybrid automaton consisting of a head-DFA (on which the compression techniques discussed above can be applied) and several tail-automata, either in NFA or in DFA form. The work presented in [20], however, does not fully elaborate the problem of counting constraints; in this paper we show how the proposed extended-automaton can be combined with the hybrid-FA proposed in [20].

In [21] Kumar et al. propose HFA, a history-based automaton. Specifically, the proposal uses conditional transitions and a "history" data structure to limit the size of the underlying state machine. In fact, by these means an HFA avoids the duplication of entire groups of DFA states that would result from the presence of large character class repetitions in the regular expression set. Counters on states with auto-transitions are introduced to support counting constraints within the HFA. However, because HFAs use a single counter instance for each constrained repetition, they are not functionally equivalent to the original NFA. In other words, HFA may fail to recognize some match situations (see Section 3 for more details). Additionally the successful application of HFA is subject to constraints on the underlying regular expressions. Finally, the need for querying and updating the history data structure may affect the performance, especially if HFAs are used in the fast path.

To our knowledge, our work is the first to propose an automaton addressing the problem of back-references. To this end, it is worth mentioning how the problem is faced within the string-processing arena, that is, within tools like *grep*, *awk*, *Perl* and so on.

As explained in [3], string-processing tools are based on either a *text-directed* or a *regex-directed* engine. In either case, the regular expression under consideration is

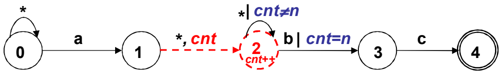

Figure 3: Counting-NFA accepting RegEx .*a.{n}bc

represented with a tree-like data structure. Text-directed engines have basically an NFA-like operation: they process each input character only once, traverse the tree-like data structure in breadth-first fashion and keep all ongoing matches active. Regex-directed engines, on the other hand, perform a depth-first traversal of the tree-like data structure and *back-track* in the input text when the end of a branch is reached without detecting a match. The way back-tracking is performed depends on the engine and on the way the regular expression is written. Because a character may be processed an undefined number of times, the performance of regex-directed engines are non-deterministic. Extensions to the regular expression language (lazy and greedy quantifiers, atomic groups, positive and negative look-around, and others) are introduced to speed up the matching operation of regex-directed engines.

Back-references are the only feature that extends the expressive power of regular expressions, and are not simply introduced to speed up regex-directed engines. Handling back-references implies *remembering* already processed portions of the input text. Therefore, back-references cannot be directly supported through finite automata, which are intrinsically memory-less. As a consequence, back-references currently are not supported through any text-directed engine. The implications of this fact are the following: First, within string-processing tools, back-references are always handled by back-tracking in the input text. Second, since regex-directed engines are meant to process a single regular expression at a time, if several regular expressions must be matched, *the input text will be reprocessed for each pattern under consideration*.

3. HANDLING COUNTING CONSTRAINTS

In this section, we present the extension to finite automata introduced to handle constrained repetitions in an effective way.

As a motivating example, let us consider the regular expression .**a.{n}bc*. The initial .* sub-pattern tells us that pattern *a.{n}bc* can occur at any position of the input text. Thus, the input text will be inspected for the occurrence of character *a* followed by *n* characters and by the *bc* substring. There are no restrictions on the characters separating *a* from *bc*, provided that they be exactly *n*.

In this section, we first show the NFA and the DFA that accept the considered regular expression, and discuss their benefits and limitations. Second, we propose the counting counterparts, focusing on the need for maintaining functional equivalence. In other words, the proposed counting automata must accept all the input strings accepted by traditional solutions and no others.

The NFA accepting .**a.{n}bc* is represented in Figure 1. As can be seen, when the cardinality of the counting constraint *n* is large, the size of the NFA (in terms of number of states) is linear in *n*. The basic problem of an NFA representation resides in the fact that, during operation, many states can be active in parallel, leading to a high memory bandwidth requirement and/or processing time. In fact, the behavior of NFAs representing regular expressions with counting constraints may approach the worst-case complexity *O(n)* per input character, a worst-case bound never achieved otherwise.

As an example, let us assume to process an input string of the form *aaaaaaaaaaaa...aaaabc*. Because state *0* is always active, transition $0 \rightarrow 1$ is triggered upon receiving symbol *a*, and all transitions up to state $n+1$ happen on any character, so eventually all states from 0 to $n+1$ will be active in parallel. In the general case, the number of parallel activations can approach the ratio between the number of NFA states and the length of the prefix preceding the counting constraint. This may result in unacceptable memory bandwidth requirements and processing time.

As is well-known from theory [2], a DFA representation can be used to keep the processing time complexity down to one state traversal per input character. A DFA can be built from an NFA through the subset construction operation, which associates a DFA state to each set of NFA states reached in parallel upon processing a given character. Because there can be 2^N distinct subsets for a set of *N* elements, there may be exponential state blowup when transforming an NFA into the corresponding DFA. In practice, this does happen only in some situations, most notably with counting constraints.

Figure 2, showing part of the DFA for regular expression .**a.{n}bc*, exemplifies this fact. State numbering reflects subset construction on the NFA in Figure 1. As should be evident, state explosion is due to the need for representing all possible occurrences of the prefix (in this case pattern *a*) at any position of the counting constraint. Notice that a similar blowup would have occurred even if the counting constraint was on a range of characters including *a*, such as *[a-z]{n}*. Conversely, this problem does not occur when the repetition does not involve characters in the prefix. The situation becomes more complicated when different regular expressions are compiled into a single DFA. In this case, explosion is also due to accounting for occurrences of the remaining regular expressions within the counting constraint.

As a result, the memory storage requirement can become a bottleneck when representing regular expressions with counting constraints in DFA form. For high numbers of repetitions of *n*, DFAs may even be an infeasible solution. To solve the issues described above, we now introduce the concept of the counting automaton. Specifically, we first create a counting-NFA and then show how to extend the subset construction operation to build the corresponding counting-DFA. The basic idea is to replace the chain of counting states (states from *2* to *n+1* in Figure 1) with a single state incrementing a counter, and make the transitions out of this state conditional on the value of the counter.

This basic concept is complicated by observing that, to preserve functional equivalence between the original and the counting-NFA, *one instance of the counter is not enough*. To understand this fact, let us consider the above regular expression with *n=3,* and assume to process the input text *axaybzbc.* As can be observed, this string matches the given regular expression starting from the second occurrence of character *a*. If we assume a single counter instance, it will be set to *1* after processing *x*, to *2* after the second occurrence of *a,* and to *3* after considering character *y*. The match operation will proceed on *b* but fail on *z*. Finally, characters *b* and *c* will leave the automaton on state *0*, incorrectly reporting a mismatch.

The above behavior can be avoided by allowing *multiple instances* of the same counter. To this end, beside the increment operation, we need an allocation and a de-allocation action. While the increment operation acts in parallel on all active instances of the counter, allocation and de-allocation are instance-specific actions.

The resulting counting-NFA is represented in Figure 3. The dashed (red) transition *1→2* represents the allocation operation (which sets the value of the newly created instance of *cnt* to *0*). The dashed (red) state is the counting state: its increment action is performed on all active counter instances when the state is entered. The conditions following "|" make the corresponding transition conditional. In this example, we assume that de-allocation happens when the corresponding instance assumes value *n* (this aspect can be generalized). Hence, the instance with value *n* will be de-allocated after traversing transition *2→3.*

The demonstration of the counting-NFA traversal with the above input text is represented in Table 1 (columns 1-3). Counter instances are numbered in order of creation. As can be observed, a second instance is initiated upon processing character *y*. The next *b* will trigger transition *2→3* because on the condition of instance cnt_1, as well as *2→2* because cnt_2 is not equal to *n*. The presence of cnt_2 leads to correct operation (a match is eventually reported as accepting state *4* is entered).

In the case of a counting constraint of cardinality *n*, up to *n* counter instances may be active in parallel. This could potentially affect the memory bandwidth requirement/processing time, again leading to an unacceptable worst case behavior. However, we observe that *the number of memory operations to be performed upon processing a character can be made independent of the number of active counter instances*. Specifically: i) using a differential representation, *each counter update may effectively involve just two instances* (the oldest and the new one); ii) *the evaluation of the condition can be performed using at most two instance values*. Let us describe these two points in detail.

Table 1: Example of counting-NFA/DFA traversal.

Char	***Active states***	***Counter instances***	***Condition (n=3)***
a	0,1		
x	0,2	cnt_1=1	cnt ≠ n
a	0,1,2	cnt_1=2	cnt ≠ n
y	0,2	cnt_1=**3**, cnt_2=1	cnt = ⊥
b	0,2,3	cnt_2=2	cnt ≠ n
z	0,2	cnt_2=**3**	cnt = n
b	0,3		
c	0,4		

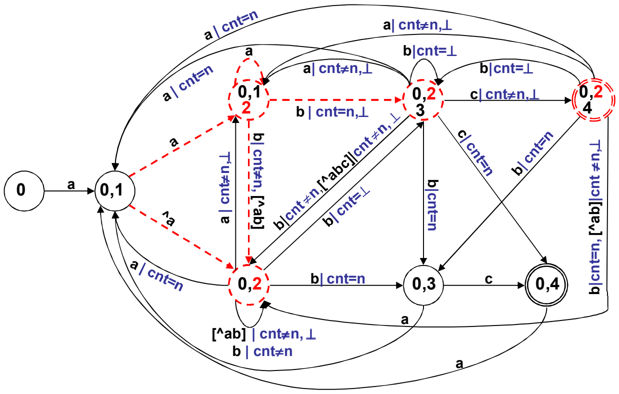

Figure 4: Counting-DFA accepting RegEx .*a{n}bc. Dashed states contain the *cnt*++ action; dashed transitions trigger a counter instantiation; transitions not shown lead to state 0.

i) **Differential representation** - Since the increment operation acts in parallel on all the counter instances, the difference between them will remain constant over execution. At each step, it will be sufficient to store the value of the largest instance (the oldest one), whereas the others can be recorded as a delta between their value and the one of the previously created instance. When a counter instance gets de-allocated, the value of the one which follows can be restored from the delta information.

ii) **Condition evaluation** – It is known that the oldest instance (say *cnt'*) is the largest, and that all instances differ in value. Therefore, if *cnt'* differs from *n*, then all instances are different from *n* and *cnt≠n* is verified. If *cnt'* is equal to *n*, then *cnt=n* is true. If, in the latter case, (at least) an additional instance of the counter exists, then it is necessarily different from *n*, and the condition *cnt≠n* is also verified. Therefore, knowing the value of *cnt'* and whether it is single instance is enough to evaluate the conditions and determine the transitions to be taken.

By generalizing the subset construction operation it is possible to derive the counting-DFA represented in Figure 4 from the counting-NFA shown in Figure 3. For readability, counter instantiation and increment are just represented by dashing (and red coloring) the transitions and states where these actions take place.

It can be noted that, in both cases, *the number of states is independent of n*. This makes counting automata

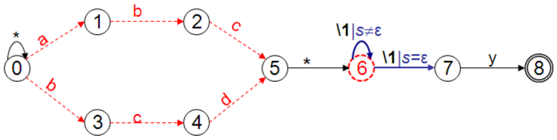

Figure 5: Extended-FA for RegEx .*(abc|bcd).\1y

attractive particularly for high values of n, for which a pure DFA would not be a feasible solution.

Subset construction is generalized as follows: First, if an NFA state S triggers an action *act* (in the example, state 2 triggers action *cnt++*), then all DFA states whose subset contains S are assigned *act*. Second, if a counter is instantiated upon NFA transition $S_1 \rightarrow S_2$ (1→2 in the example), the same happens for all DFA transitions connecting a state containing S_1 to a state containing S_2 in their subset. Finally, transitions can possibly be conditional upon the value of the counters.

Being deterministic, the automaton must be built to trigger *one and only one* transition for each (state, character) pair. To this end, three conditions on the counter will be considered: i) $cnt \neq n$, ii) $cnt=n$ and iii) $cnt=\perp$. The first means that the oldest (active) instance *cnt'* is different from n, the second that *cnt'* is single instance equal to n, and the third that *cnt'* is equal to n but at least a second instance of the counter exists. In other words, $cnt=\perp$ whenever both $cnt=n$ and $cnt \neq n$ hold *at the same time* in the corresponding counting-NFA. The condition is represented in the last column in Table 1; notice that the represented traversal holds also for the counting-DFA.

The concepts above can be easily generalized to character range repetitions, sub-expression repetitions and *{n,m}, {n,+}-like* counting constraint.

4. HANDLING BACK-REFERENCES

In this section we introduce back-references and show how to extend an NFA to handle them. We intend to modify the NFA operation as little as possible.

A back-reference in a regular expression refers to some sub-expression enclosed within capturing parentheses, and indicates that the referred sub-expression can be matched later within the regular expression itself. By convention, capturing parentheses are numbered by numbering their open parentheses from left to right. The back-reference to the *j*-th captured sub-expression is indicated by a back-slash (\) follow by *j* (\j). Back-references add expressive power to regular expressions when the referred sub-expressions are not exact match strings.

As an example, let us consider the extended regular expression *a(bc|d)e([a-z])\1([1-9])f* (from now on, let us imply the initial .* sub-pattern).

$$a\underbrace{(bc\,|\,d)}_{\backslash 1}e\underbrace{([a-z])}_{\backslash 2}\backslash 1\underbrace{([1-9]+)}_{\backslash 3}f$$

As shown above, the order of the capturing parentheses implies the sub-expression numbering, which is then used to perform back-references. It can be observed that each sub-expression contains an alternative. Moreover, the length of the substring matching the captured sub-expression is not necessarily fixed or known *a priori* (see sub-expressions 1 and 3).

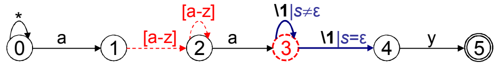

Figure 6: Extended-FA for RegEx .*a([a-z]+)a\1y

Once encountered, the back-reference refers to the specific substring in the input text which first matched the given sub-expression. As an example, the above regular expression will match text *abceabc123f*, but won't match *abcead123f*. In fact, in the second case, when the back-reference *\1* is encountered the current input (*d*) does not match the input text captured when the referenced sub-expression was first matched (*bc*).

From this example it should be clear that a machine processing back-references needs to *record* matched portions of the input string. This cannot be performed through a finite state machine, which is intrinsically memory-less.

One could consider augmenting a NFA with *tags* capturing the beginning (and the end) of a match as proposed by Laurikari [5] to solve the problem of determining the position of a match or of a sub-match in linear time. However, the problem is more complicated. In fact, there exist situations where the start and the end of a sub-match are not unequivocally defined: for example, a captured sub-expression may have a variable length or may overlap with the preceding/following characters in the pattern.

As an example, let us consider the two following regular expressions with back-references: (1) *(abc|bcd).\1y* and (2) *a([a-z]+)a\1y*. Pattern (1) matches either *abc* or *bcd* followed by any character, followed by the previously matched sub-pattern, followed by character *y*. Pattern (2) matches *a*, followed by a substring of any length composed by *[a-z]* characters, followed by *a*, followed by the previously matched sub-string and finally by character *y*. In the first case, the complexity arises by two facts: first, the two alternatives in the referenced sub-expression (*abc* and *bcd*) overlap; second, the captured sub-expression overlaps with the following character, which is a wildcard. In the second case, the complexity is due to the overlapping of the back-referenced sub-expression with both the preceding and the following characters (*a*), and by the fact that its length is variable.

The text *abcdabcdy* matches regular expression (1) starting from the second character, and not from the first. In fact, even if *abc* is matched and correctly back-referenced, its second occurrence is not followed by *y*. The text *babacabacy* matches regular expression 2 from the second character, and the matched sub-text contains one *a*.

Our solution aims at preserving the NFA operational model. In other words: *(i)* both finding all possible subsequent matches as well as stopping at the first match should be allowed; *(ii)* each input character should be processed only once (possibly against a set of active states); and *(iii)* the parallel processing of different regular expressions should be permitted.

To this end, we extend the NFA as follows:

- Each back-reference in the regular expression set is associated with a unique identifier. Thus, it is irrelevant whether two back-references belong to the same or to different patterns.
- The transitions implementing back-referenced sub-expressions are augmented with a *tag* indicating that, during traversal, the input text must be recorded. Each tag is associated with the corresponding back-reference identifier.
- Each active state can be associated with a set of matched substrings MS_k for each back-reference \k. This is performed as follows. *(i)* When a transition $S_x \rightarrow S_y$ is taken, the set MS_k associated to S_x gets moved to S_y. *(ii)* If the taken transition is tagged *k*, the current input character is appended to the strings in MS_k.
- If a back-reference \k originates from state S_j, S_j is *consuming:* when active, all the strings in its MS_k are processed and shortened (one character at a time)
- Two special conditional transitions representing the back-reference are created. If the input character matches some string in MS_k then: *(i)* transition $S_j \rightarrow S_{j+1}$ is taken if the corresponding string is consumed completely; *(ii)* transition $S_j \rightarrow S_j$ is taken otherwise.

To clearly understand, let us consider the described scheme in practice. For the sake of example, Figure 5 and Figure 6 show the extended-FA corresponding to the regular expressions considered above. Tagged transitions as well as consuming states are dashed (and red); transitions on back-references are represented (in blue) along with the corresponding condition.

Let us first assume to traverse the extended-FA in Figure 5 with the input text *abcdabcdy*. The corresponding sequence of {*state*(MS_1)} activations is the following:

a: 0 – 1(*a*)
b: 0 – 3(*b*) – 2 (*ab*)
c: 0 – 4(*bc*) – 5 (*abc*)
d: 0 – 5 (*bcd*) – 6(*abc*)
a: 0 – 1(*a*) – 6 (*bcd,bc*)
b: 0 - 3(*b*) – 2(*ab*) – 6(*cd, c*)
c: 0 – 4(*bc*) – 5(*abc*) – 6(*d*) - 7
d: 0 – 5(*bcd*) – 6(*abc*) - 7
y: 0 – 6(*bcd*) – 8

Similarly, traversing the extended-FA in Figure 6 with the input text *babacabacy* will lead to the following operation:

b: 0
a: 0 - 1
b: 0 – 2(*b*)
a: 0 – 1 – 2(*ba*) – 3(*b*)
c: 0 – 2(*c, bac*)
a: 0 – 1 – 2(*ca,baca*) – 3(*c,bac*)
b: 0 – 2(*b, cab, bacab*) – 3(*ac*)
a: 0 – 1 – 2(*ba, caba, bacaba*) – 3(*b, c, cab, bacab*)
c: 0 – 2(*c,bac,cabac, bacabac*) – 3(*ab*) – 4
y: 0 – 2(*cy, bacy, cabacy, bacabacy*) - 5

As can be observed, the scheme ensures correct operation. This is essentially due to the correct and full propagation of the partial match information among states.

Note that storing all the matched substrings as shown in the example leads to conspicuous information replication. In practice, the MS_k contains a set of pairs *(starting position, ending position)* pointing to the matched substrings in the input text. Even so, the worst-case bound in the number of pointers needed for each back-reference is $O(m^2N_{NFA})$, where *m* is the length of the input text. It must be mentioned that this worst-case bound applies when the referenced substring has variable length (as in the second example). Similar patterns must be advised against since they may open the way to algorithmic attacks also if back-tracking (i.e., a regex-directed engine) is used.

A way to control the size of the MS_k (and thus, to keep the memory required to store partial matching information under a given bound) can be derived from the following observation. The active states of an NFA can in principle be processed independently and at different times. Therefore, the processing of particular NFA state activations can be deferred, while waiting for other activations to terminate and free memory. Let us consider processing the 7th input character (that is, the third *b*) in the second example. It is possible to postpone the consideration of states 0 and 1 and to evaluate only the activations in states 2 and 3. States 0 and 1 can be processed on the 7th character of the input text at a later time (that is, when the other activations die or enough memory is released). Note that this corresponds to combining a breadth-first with a depth-first NFA traversal, effectively introducing some back-tracking. A full depth-first traversal would correspond to a pure back-tracking approach.

Finally, it is worth highlighting the similarities in our handling of counting constraints and back-references. In both cases, the NFA has been augmented with some state information. The basic semantics of the NFA operation is unmodified. Transitions activating counters are conceptually equivalent to tagged transitions recording text: they trigger an *action* on a particular counter/back-reference (typically leading to the storing of state information). Conditional transitions are also semantically equivalent in the two scenarios: they depend on whether the stored data (a counter or a matched substring) have been "consumed." Counting states are similar to consuming states, in that both modify the state information associated to them. The procedure to transform an extended NFA with back-references into its deterministic counterpart is equivalent to that seen for counting constraints. More important, the two mechanisms can be combined and unified into a single extended-FA having: *(i)* tagged transitions on counters and back-references, *(ii)* conditional transitions on counters and back-references, *(iii)* counting and consuming states.

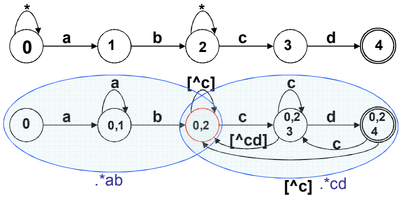

Figure 7: NFA and DFA for RegEx .ab.*cd***

5. COMPACTING AN EXTENDED-FA

The methodology presented above allows limiting the number of states necessary to represent regular expressions containing counting constraints and handling back-references. However, this comes at the cost of the introduction of conditional transitions, which must be represented in an efficient way. In particular, we aim at an encoding scheme that allows applying compression techniques proposed in the context of pure DFAs, naming default transitions [17][19] and character classes [19].

A simple technique to represent conditional transitions consists of using a *character translation unit,* which operates as follows: *(i)* every character *c* never appearing in a conditional transition is associated a single symbol, *(ii)* every character appearing in a conditional transition is associated multiple symbols, one for each possible condition of the counter. Notice that, in case of a single counting constraint, this corresponds to expanding the alphabet $\sum$ of cardinality $|\sum|$ to an alphabet $\sum'$ of cardinality $k|\sum|$, *k* being equal to 3 or 5 depending on the nature of the counting constraint (*{n}* versus *{n,m}*).

Once this preliminary character translation has been performed, the resulting DFA can be compressed by standard techniques. In particular, character classes can be used to reduce the size of the alphabet. Specifically, a set of symbols $c_1, c_2, \ldots, c_k$ can be merged into a single class if, for any state *s*, they lead to the same next state, that is, $\delta\ (s, c_1) = \delta\ (s, c_2) = \ldots = \delta\ (s, c_k)$, *δ(state,char)* being the state transition function. Notice that the next state can vary across the different *s* in the DFA. At the end of this operation, each character class will be represented by a single symbol in a new, reduced alphabet $\sum''$. This technique is especially effective in compressing sets of characters that do not appear in the compiled regular expressions or that are treated in homogeneous ways across the DFA (e.g., case insensitive regular expressions). In [19] an algorithm to perform character class translation in $O(n^2)$ time, *n* being the number of states in the DFA, is presented.

A second technique proposed in [17] and refined in [19] consists of using *default transitions* to eliminate the transition redundancy typically present in DFAs. Specifically, if two states s_x and s_y have *k* outgoing transitions in common (that is, $\delta(s_x, c_i) = \delta(s_y, c_i)$ for *k* different characters c_i), then those *k* transitions can be removed from one of the two states, say s_y, by introducing a default transition from s_y to s_x. After performing this compression, only $|\sum|-k$ labeled and a default (unlabeled) transition will be needed to fully specify the behavior of the automaton in state s_y. When processing an input text, the default transition will be taken if the current state does not contain a labeled transition on the input character. Moreover, the traversal of a default transition won't cause input character consumption. Clearly, this technique can be applied to an extended-DFA after the described character translation (possibly incorporating character classes) has been performed.

Two observations—one about character translation and the other about default transitions—suggest how the two above techniques can be particularly suitable to extended-DFAs.

First, as can be observed in Figure 4, extended-DFAs consist of two categories of states: *standard* and *counting/consuming* states. The former do not have the NFA counting/consuming-state (state 2 in the example of Figure 3) in their subset; the latter do. As a consequence, standard states do not present any conditional transitions. This fact suggests that compression can be more efficient if *two distinct character translations*—one for standard and one for counting states—are used. To efficiently determine which translation to perform during operation, the two groups of states can be laid out in two distinct memory regions.

Second, let us assume to have a counting constraint with a limited number of repetitions, such to allow a DFA of reasonable size. As can be observed in Figure 2, the counting constraint originates a high number of states with limited redundancy. In fact, each of them tends to transition "forward" to a distinct set of states. Conversely, the corresponding extended-DFA, besides being smaller, exhibits more redundancy. In fact, the counter eliminates the need for all the distinct forward transitions present in the pure DFA counterpart. To get an intuition of this fact, observe the incoming transitions to the counting states in Figure 2, and compare them with the DFA states having NFA-state 2 in their subset in Figure 4.

6. COMPILING SEVERAL REGULAR EXPRESSIONS INTO AN AUTOMATON

The methodology described above can be applied to any number of counting constraints/back-references, whether they appear in a single regular expression or in different regular expressions compiled into a single automaton. However, as the number of counting constraints and back-references increases, two problems arise: first, the induced alphabet gets larger; second, the size of the DFA (in terms of number of states) can prohibitively increase. These two problems are related; we first analyze their cause (Section 6.1) and then propose an algorithmic solution with important architectural implications (Section 6.2).

6.1 The problem

To understand the issues above, let us consider a simpler problem: the compilation of different regular expressions containing dot-star conditions (i.e., ".*" sub-patterns) into a single DFA.

Figure 7 represents the NFA and the DFA accepting regular expression *.*ab.*cd*. As can be seen, the DFA can be divided into two sub-DFAs: the first accepting *.*ab* and the second accepting *.*cd*. The NFA state representing the second .* sub-pattern (state 2) is transformed into DFA state 0-2, which represents the match of the first sub-expression and the beginning of the match of the second one. No transition from the second sub-DFA falls beyond state 0-2.

States representing dot-star conditions, which we will call special states, have an important implication when compiling multiple regular expressions. Specifically, if a regular expression RE_1 containing a dot-star is compiled with a regular expression RE_2, the sub-DFA representing the match of RE_2 is duplicated: one instance will start at state 0 and one instance will start at the special state.

If many regular expressions containing dot-star sub-patterns are compiled, the situation gets more complex. In Figure 8 we assume to have two regular expressions of this kind, say RE_1 (.*RE_{1a}.* RE_{1b}) and RE_2 (.*RE_{2a}.* RE_{2b}). Special states are represented with (red) shaded filling. As can be seen, beside the special states representing the match of .*RE_{1a} and .*RE_{2a}, we will have an additional special state representing the parallel match of both those sub-expressions, with a consequent additional complexity in terms of sub-DFA replication. Clearly, when more regular expressions containing dot-star conditions are compiled, the number of possible special state combinations will affect the complexity and the size of the resulting DFA. Note that this concept applies also to patterns containing repetitions of large character ranges, of the form *[^$c_1c_2...c_k$]**.

The same considerations hold for extended automata. In fact, counting/consuming states behave like special states (they have an auto-loop on a large character class). This fact has two implications. First, compiling different regular expressions with counting constraints/back-references can lead to sub-DFA replication, with consequent increase in the extended-DFA size. Second, the combination of multiple NFA counting/consuming states into a single DFA state implies outgoing transitions conditional on multiple distinct counters/back-references. This, in turns, translates into an added alphabet translation complexity and, ultimately, into a larger alphabet.

6.2 The solution

One way to avoid this effect is to isolate each counting constraint/back-reference/dot-star condition from the other. This can be accomplished by using a hybrid-FA, as described in [20]. Specifically, the hybrid-FA can be built as follows. The subset construction operation (that is, NFA to DFA transformation [2]), started at the entry state s_0 should be interrupted on each special state. Second, each NFA special state should be treated as an entry state to a separate tail-DFA: that is, a distinct subset construction operation should be initiated on every special state. The outcome of this procedure will be: *(i)* a single *head-DFA* not containing any counting operation/back-references, except for counter instantiation and initialization and substring recording; *(ii)* a set of *tail-DFAs*, one for each counting constraint/back-reference/dot-star condition. Notice that dot-star condition expansion is less critical because it can increase the size of the head-DFA but not the one of the alphabet. Also, it is possible to keep the tail-automata in NFA format, and interrupt subset construction before special states. In this case, tagged transitions can be also kept entirely in the NFA part.

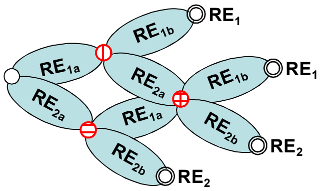

Figure 8: Exemplification of DFA obtained by compiling RegEx RE_1=.*RE_{1a}.*RE_{1b} and RE_2=.*RE_{2a}.*RE_{2b}

Figure 9 represents this operation on a simple example containing a single counting constraint. In particular, Figure 9(1) shows the counting-NFA, Figure 9(2a) the head-DFA and Figure 9(2b) the counting tail-DFA for regular expressions *.*ab.{n}cd* and *.*de*. When creating the head-DFA, subset construction is interrupted upon encountering the counting state 3. At this point, state 0-3 is created. However, sub-state 3 is ignored to the end of computing the outgoing transitions from state 0-3; its label is used only to distinguish the newly created state from state 0 and to link the tail-DFA. Separately, subset construction is performed starting from state 3, thus creating the represented tail-DFA.

When processing the input text, the head-DFA is always active: each input character will trigger a state transition on it. The tail-DFA will be activated every time the head-state 0-3 is traversed. Its deactivation will take place upon entering state Ø or an accepting state (we assume that multiple matches of the same regular expression are not of interest).

The operation of distinct tail-DFA machines is, in principle, independent. Furthermore, once the head-DFA has activated a tail-DFA, the two can execute in sequence or in parallel threads. Whatever execution model is assumed, the number of parallel activations of a given tail-DFA needed to ensure correct operation should be kept minimal to reduce memory bandwidth requirements.

In the case of Figure 9 (2b), a distinct activation of the tail-DFA is required each time head-state 1-3 is reached. Therefore, at any given time, the tail-DFA may be active in parallel on different states, which affects the number of

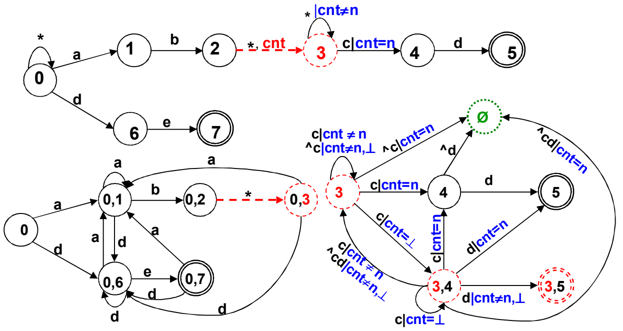

Figure 9: (1) counting-NFA, (2a) head-DFA, (2b) counting tail-DFA for regular expressions: (1) .*ab.{n}cd, (2) .*de. In the tail-NFA state Ø is a dead state. Transitions exiting the accepting state are omitted.

memory operations needed to process an input character. This can be avoided by transforming the tail-DFA as shown in Figure 10. Namely, the counting sub-state 3 is added to all states (except Ø) and equivalent states are reduced. Because a new activation of the tail-DFA always begins from the counting state 3, this ensures that, if the tail-DFA is already active, the new activation will be covered by the current active state. At the same time, the condition on the counter will prevent invalid transitions from being performed. This transformation can be applied to any counting tail-DFA to allow a *single activation* to preserve correct operation. Moreover, a similar concept can be applied to tail-DFAs built on dot-star sub-patterns.

7. MAPPING ON AN ARCHITECTURE

In this section we present a regular expression-matching architecture suitable for implementing the above scheme. In particular, we refer to the hybrid extended-FA proposed in Section 6 to deal with data-sets presenting a high number of regular expressions containing counting constraints, back-references and dot-star conditions. As mentioned above, the head-DFA and the tail-DFAs/NFAs can be processed in sequence within the same thread of execution or can be assigned to separate threads. In this architectural proposal we opt for the second alternative, and we assume to separate the head-DFA from the tail-DFAs/NFAs processing. All the active tail-DFAs, however, are handled by a single thread (even though the architecture can be easily modified to split them across multiple threads).

Several facts motivate this choice. First, and most important, the head-DFA engine is supposed to be always active, whereas the tail-DFAs/NFAs are triggered only upon traversal of a special state and can be deactivated later on. As a consequence, regular expressions not containing counting constraints, back-references and dot-star conditions will be entirely matched within the head-DFA. Thus, the head-DFA can be considered the common case to be implemented on the fast path, whereas the tail-FAs the exception to be offloaded. To this end, it can be observed that the head-DFA does not have to deal with counters and back-references and its tasks are therefore simpler and faster. Finally, notice that the next state information concerning head- and tail-FAs can be stored on different memory regions, and, in general, on different memory banks. The use of two threads allows greater parallelism in the memory accesses.

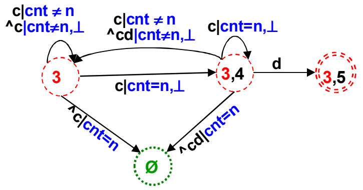

Figure 10: Transformed tail-DFA to allow single activation.

Figure 11 presents the logical view of the proposed architecture. Three memory banks are used: two for the head- and the tail-FA next state information, and one for the counters and the back-reference information. All DFAs are compressed through the techniques detailed in Section 5 [19] and are encoded using indirect addressing [27]. State identifiers carry the information about which labeled outgoing transitions are present, thus allowing one memory access per state traversal. As we will detail in Section 8, 32-bit state identifiers can be used, leading to 32-bit wide memory accesses to determine the next state. Counters present in common data-sets typically consist of less than 200 and can span till 1,024 repetitions. This suggests that two counter values can be stored in 20 bits and therefore be accessed through a single 32-bit wide memory access.

The operation of the head-DFA engine is straightforward. It translates the current state identifier and translated input character into a memory address, and queries the head-DFA memory for the next state. If a labeled transition (as opposed to the default transition) is taken, the input character is consumed. If the current state is accepting, a match is reported. Finally, if a special state is encountered, an entry is inserted in the *activation FIFO*, which represents the interface between the head- and the tail-engine. This entry carries the information about the special state and the character position within the input stream the transition refers to. This information is necessary because the two engines may proceed out of phase.

The operation of the tail-engine is more complex. All active tail-DFAs operate in lock-step: for each input character, the tail-engine must perform one consuming state transition on each active tail-DFA. To this end, the following operations must be performed. First, the counter data are extracted from memory. Such information is sent to the character translation unit along with the input character to determine the translated symbol. Second, the current state identifier, recorded in the *active tail-DFAs* table (stored in scratch memory), is decoded and the memory address of the next state extracted. Finally, this information is used to query the state memory. Additionally, if the first

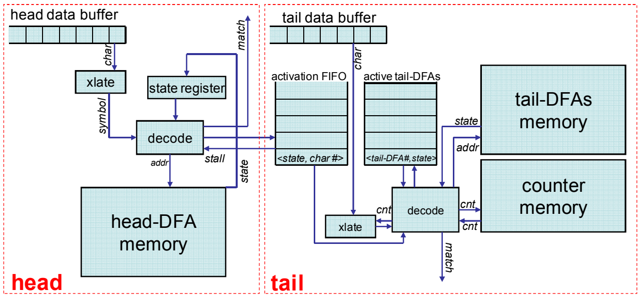

Figure 11: Logical view of the proposed architecture

character position stored in the activation FIFO corresponds to the current one, the corresponding FIFO entry is processed, the counter table updated and, if necessary, a new entry is added into the active tail-DFAs table.

The tail- and head-engine use distinct data buffers. To prevent the tail-engine from running ahead of the head-engine (which may cause counter inconsistencies), characters are transferred from the head to the tail data buffer upon consumption (that is, when a labeled transition is taken). An empty data buffer will stall the tail-engine. Additionally, a full activation FIFO will cause a back-pressure signal to be sent to the head-DFA, making it stall. Notice that no deadlock can take place in this condition. Finally, because each tail-DFA can have only a single activation at any given time, the size of the active tail-DFA table is bounded to the number of counting constraints/dot-star conditions/back-references in the rule-set.

8. EXPERIMENTAL EVALUATION

In this section, we briefly present the results of an experimental evaluation conducted on rule-sets extracted from Bro v0.9 [8], ClamAV r.0.91 [9] and Snort (snapshot from November 2007) [7]. Snort rules having common packet headers have been grouped together. Specifically we considered headers: *(i)* tcp $EXTERNAL_NET any $HOME_NET $HTTP_PORTS/8080/80 (*snort20*), *(ii)* tcp $EXTERNAL_NET any $HOME_NET 7777:7778/10202:10203/143/any (*snort76*), *(iii)* tcp $EXTERNAL_NET $HTTP_PORTS/8080/80 $HOME_NET any (*snort676*), and *(iv)* tcp $EXTERNAL_NET * $HOME_NET any (*snort702*). As far as Bro is concerned, we considered all payload patterns in the *sig-addendum* set.

The characteristics of these rule-sets, the size (number of states) of the corresponding automata, and their memory footprint are reported in Table 2.

The number of patterns in each set is reported in the data-set name. Note that these data-sets contain a significant number of counting constraints. Additionally, *snort676* and *snort702* present many back-references. In fact, most of them are complex (the back-referenced pattern is, for instance, *[a-zA-Z0-9]+*) and cannot therefore be resolved through enumeration.

Three kinds of automata are computed: the basic NFA (columns 4-5), the extended-NFA (columns 6-7) and its hybrid counterpart (columns 8-12). The rules with back-references could not be represented in a traditional NFA. For the sake of comparison, we extended the NFAs represented in columns 4-5 through our back-reference mechanism (extended-NFAs in columns 6-7 also use our counting constraints scheme).

The algorithm for hybrid-FA creation [20] was modified to move counting and consuming states to the tail-automata, and configured to keep the head-DFA size on the order of 30K states. NFAs have been reduced by collapsing common prefixes and DFAs have been compressed through the default transition creation algorithm described in [19].

As far as memory encoding is concerned, we used indirect addressing [27], which, as mentioned, allows one memory access per state traversal. States with many outgoing transitions cannot benefit from indirect addressing and are fully represented. We tested 32-bit and 64-bit wide state identifiers and obtained better results with the former (which we report). In NFAs, we split states with multiple transitions on the same character into multiple states connected through epsilon transitions as described in [27].

First, it can be observed that we were able to compile a large number of complex regular expressions, containing simple and repeated character ranges, disjunctions of sub-patterns, dot-star terms, counting constraints and back-references.

Second, using our extended-automata, the size of the NFA decreases. The effect is remarkable in terms of memory footprint. In fact, most of the reduced states have many outgoing transitions and would have therefore needed

Table 2: Characteristics and size of NFA, Extended-NFA and Extended-hybrid-FA for different data-sets. All sizes are computed using indirect addressing and 32-bit wide state identifiers.

Data-set	Characteristics		NFA		Extended-NFA		Extended-Hybrid-FA				
	# counters	# back-ref	# states	size (KB)	# states	size (KB)	# tails	head		tail	
								# states	size (KB)	# states	size (KB)
bro43	10	0	1,107	635	564	93	23	30,286	2,450	449	65
clamav440	90	0	26,940	9,042	18,281	349	76	20,614	156	4,188	126
snort20	4	0	1,508	1,013	498	82	30	30,438	3,312	491	76
snort76	46	0	54,135	53,303	2,395	2,045	495	30,607	16,268	2,267	1,933
snort676	37	181	62,812	21,684	47,870	7,133	251	32,261	15,917	47,683	6,949
snort702	48	172	65,043	23,406	48,270	7,381	261	33,285	8,551	47,954	7,070

a full representation.

Third, converting to a hybrid-FA representation decreases the memory bandwidth at the cost of an additional 156KB-16MB needed to hold the head-DFAs. As detailed in [20] and using default transitions as in [19], the total number of memory accesses per input character is *2*#head-DFAs* in the average case and *2*#head-DFAs + 2*#counters + 2*#tail-DFAs* in the worst case. The factor 2 on the DFAs is due to default transitions. These numbers are by far smaller than their NFA counterpart (number of NFA states). Note that the limited reported memory footprints make it possible to deploy the automata with SRAM in an ASIC implementation, allowing memory access rates in excess of 500MHz.

Finally, back-references are handled in the automaton without the need for invoking a PCRE engine on each partially matched pattern.

9. CONCLUSION

In conclusion, we propose an extended finite automaton suitable for representing regular expressions containing counting constraints and back-references.

When addressing counting constraints, the design aims to minimize memory storage and bandwidth requirements. Specifically, the size of an extended-FA is independent of the number of repetitions, and the number of memory accesses needed for each counter is independent of the number of active counter instances. When addressing back-references, the design aims to preserve the original NFA operating semantics while retaining efficient support for repeated substrings and counting constraints. Also, we showed how standard compression techniques can be applied to an extended-DFA. As a practical consideration, we analyzed the problem of compiling several regular expressions with problematic sub-patterns into a single automaton, and proposed a hybrid, comprehensive solution.

To the best of our knowledge, we have described the first high-speed automaton that can accommodate all the Perl-Compatible Regular Expressions present in the Snort network intrusion and detection system.

ACKNOWLEDGMENTS

The authors wish to thank Chris Clark from Compiler Resources, Inc. for his useful feedback. This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794 and by Intel's gifts.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," in Communications of the ACM, 1975.

[2] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979.

[3] J. E. F. Friedl, "Mastering Regular Expressions," Third Edition, O'Reilly, August 2006

[4] Perl Compatible Regular Expressions: http://www.pcre.org/

[5] Ville Laurikari, "NFAs with Tagged Transitions, Their Conversion to Deterministic Automata and Application to Regular Expressions", in SPIRE 2000

[6] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in 13th System Administration Conf., Nov 1999.

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

[8] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", in Computer Networks, 31(23-24), Dec. 1999

[9] ClamAV: http://www.clamav.net/

[10] Cisco Security Appliance. http://www.cisco.com. 2007.

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

[12] M. Altinel, M.J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information", in Proc. VLDB Conference 2000.

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

[14] J. Newsome et al., "Polygraph: Automatic Signature Generation for Polymorphic Worms", in IEEE Security & Privacy Symp., 2005.

[15] L. Tan, and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," in ISCA 2005.

[16] F. Yu et al., "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", in ANCS 2006

[17] S. Kumar et al., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in ACM SIGCOMM, Sept 2006.

[18] S. Kumar et al., "Advanced Algorithms for Fast and Scalable Deep Packet Inspection", ANCS 2006

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

[20] M. Becchi and P. Crowley, "A Hybrid Finite Automaton for Practical Deep Packet Inspection", in CoNEXT 2007.

[21] S. Kumar et al. "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," in ANCS 2007.

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

[23] R. Franklin et al., "Assisting Network Intrusion Detection with Reconfigurable Hardware," FCCM 2002.

[24] C. Clark et al., "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in FLP 2003

[25] B. Brodie, et al., "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in ISCA 2006.

[26] A. Mitra et al., "Compiling PCRE to FPGA for Accelerating SNORT IDS", in ANCS 2007

[27] Becchi et al., "A workload for evaluating deep packet inspection architectures," in IISWC 2008