2011 International Conference on Parallel Architectures and Compilation Techniques

Optimizing Regular Expression Matching with SR-NFA on Multi-Core Systems*

Yi-Hua E. Yang
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: yeyang@usc.edu

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: prasanna@usc.edu

***Abstract*—Conventionally, regular expression matching (REM) has been performed by sequentially comparing the regular expression (regex) to the input stream, which can be slow due to excessive backtracking [21]. Alternatively, the regex can be converted to a deterministic finite automaton (DFA) for efficient matching, which however may require an extremely large state transition table (STT) due to exponential state explosion [17, 27]. We propose the *segmented regex-NFA* (SR-NFA) architecture, where the regex is first compiled into modular nondeterministic finite automata (NFA), then partitioned, optimized, and matched efficiently on modern multi-core processors. SR-NFA offers attack-resilient multi-gigabit per second matching throughput, does not suffer from either backtracking or state explosion, and can be rapidly constructed. For regex sets that construct a DFA with moderate state explosion, *i.e.*, on average 200k states in the STT, the proposed SR-NFA is 367k times *faster* to construct and update and use 23k times *less* memory than the DFA approach. Running on an 8-core 2.6 GHz Opteron platform, our prototype achieves 2.2 Gbps average matching throughput for regex sets with up to 4,000 SR-NFA states per regex set.**

***Index Terms*—Regular expression, NFA, bit-level parallelism, thread-level parallelism, multi-core processor**

I. Introduction

High-speed deep packet inspection, content filtering and data mining are becoming more pervasive in the Information Age. Regular expression matching (REM) is a core capability of these functions, where pre-defined patterns are matched against continuous input streams. Recent network intrusion detection systems [1, 2, 4] rely on REM with dynamic sets of complex regular expressions (regexes) against high bandwidth and potentially malicious input streams. Such REM activities can tax the underlying network processing system heavily in both processing cycles and memory usage.

Conventionally, REM has been implemented in software [3] where regex symbols are compared sequentially to the input characters; such performance is highly input and regex dependent and can be subject to the *"backtracking"* attack [21]. Alternatively, REM can be performed by a finite automaton, which is free of the backtracking behavior and can find multiple and overlapping regex matches concurrently in the input. A regex of length n over alphabet Σ can be compiled into a *nondeterministic finite automaton* (NFA) with $O(n)$ states and at most $O(n^2)$ transitions [16]. At run-time, the NFA maintains the activeness of its states, each of which can ϵ-transition to zero or more states. In total the NFA makes from $O(1)$ to $O(n^2)$ state transitions per input character.

The NFA can be further converted to a *deterministic finite automaton* (DFA), which maintains only one state number at run-time. For each input character, the DFA looks up the state transition table (STT) for one state transition, performs a small (constant) amount of computation, and transitions to the target state. The matching throughput of DFA-based REM is thus limited by the (random-access) latency of the STT access. For simple regexes whose STT can fit mostly in the processor cache, DFA-based REM usually achieves very good matching throughput on multi-core systems.

For some regexes, converting the $O(n)$-state NFA to a DFA can generate an excessively large STT with up to $O(2^n)$ states and $O(2^n|\Sigma|)$ transitions, a phenomenon known as the exponential state explosion [17, 27]. In these cases, DFA-based REM becomes impractical on multi-core systems for several reasons. First, a regex composed of several hundred symbols ($n > 100$) can require a DFA STT with multi-gigabytes of memory, which may not be feasible on platforms with limited memory size (such as an on-line NIDS). Second, an input that causes irregular state accesses across the entire STT can cause cache thrashing and be used as an performance based attack to the REM system. Third, due to (random-access) memory bandwidth competition, the throughput of a state-exploded DFA may not scale well in number of cores on a shared-memory multi-core system (as shown in [19] before applying input-specific training and optimizations). Finally, the NFA-to-DFA conversion can be both computation and memory expensive, making it difficult to update the REM solution.

Several techniques [7, 13, 22] have been proposed to compress the STT by introducing nondeterminism to the DFA. These nondeterministic features usually require more complex run-time processing and multiple random memory accesses per input character; both can reduce matching throughput. The compression ratio is usually highly regex-dependent, and the compressed DFA size still much larger than the original NFA. The DFA compression algorithm can be complex (*e.g.*, quadratic in the number of DFA states [7]), making regex update even more difficult.

Due to the above reasons, we believe NFA-based REM is a valuable complement to DFA-based REM on multi-core

* Supported by U.S. National Science Foundation under grant CCR-1018801.

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.73

Table I: Commonly used basic and extended regex operators.

Op.	Name	Example	Description
$\cdot$	Concatenation	$q_1 q_2$	q_2 right after q_1
$\vert$	Union	$q_1 \vert q_2$	Either q_1 or q_2
$\star$	Kleene closure	$q*$	q zero or more times
$+$	Repetition	$q+$	q one or more times
?	Optionality	$q?$	q zero or one times
{*m*,*n*}	Constrained rep.	$q\,\{m,n\}$	q in m to n times
[...]	Character class	[a $-$ c]	Either a, b or c
[^...]	Inv. char. class	[^\r\n]	Neither \r nor \n

systems. While most simple regexes can be handled by a DFA, those complex regexes that would cause exponential state explosion as DFAs can be processed by an efficient NFA architecture:

1) We propose the *segmented regex-NFA* (SR-NFA) architecture for REM that explots both bit and thread-level parallelism on multi-core systems.
2) We focus on fast REM compilation, small memory footprint and scalable matching throughput in number of cores. As a result, an SR-NFA with thousands of states can be constructed entirely onto the level-1 cache of a modern processor in a few millisecond.
3) We design optimizations to reduce memory and computation complexity of SR-NFA.

For regexes that would cause severe state explosion as DFA, the proposed SR-NFA architecture reduces memory usage from tens gigabytes to several kilobytes and construction time from hours to seconds, and achieves 1.6 ~ 2.5 Gbps attack-resilient throughput matching 200 ~ 3,200 regex symbols.

Section II gives the background on regular expression matching. Basic concept and data structures of the SR-NFA architecture are described in Section III; mapping and optimization on multi-core processors are discussed in Section III-D. A prototype implementation is evaluated in Section V. Section VI goes over related works, while Section VII concludes the paper and discusses future work.

II. Background

A. Regex, REM, and Match Ratio

By definition, a regular expression (*regex*) describes a regular language over a fixed alphabet. Three basic operators, *concatenation* ($\cdot$), *union* ($\vert$) and *Kleene closure* ($\star$), provide the facility to combine character symbols to form arbitrary regex patterns. In addition, most regular expression matching (REM) software also support several extended operators offering convenient representations for complex pattern. Table I lists the commonly supported regex operators.[1]

Definition 1: Given a regular expression (regex) r which defines the regular language $L(r)$ over alphabet Σ, and an input character sequence $X = [x_1 \cdots x_w], x_i \in \Sigma$. The process of regular expression matching (REM) for r searches X for all sub-sequences $X^{(j)} \subseteq X$, $1 \leq j \leq m$, such that $X^{(j)} \in L(r)$. We say the REM process matches *regex* r against *input* X of length w and produces a set of m matches $\{X^{(1)}, \ldots, X^{(m)}\}$.

B. REM by Sequential Comparison

REM has been implemented as software libraries [3] where the regex symbols are sequentially compared to the input characters to search for a match condition. When there are more than one possible regex symbols matching the input character (usually due to the Kleene closure or union operators), the alternative paths are selected one at a time in either a depth-first (greedy) or a breadth-first (lazy) manner.

If, following the tentatively selected search path, a regex symbol cannot accept an input character at some later point in time, the matching progress must be *backtracked* to the original selection point and a different path is selected. Alternative search paths may continue to fail, which can induce a large number of backtracks for regexes with multiple Kleene closures and unions over complex sub-patterns. Due to the backtracking behavior, a regex with K *multi-match* segments can require $O(K^2)$ times computation of a single segment to determine whether a match exists in the input [21]. As shown in [21], the backtracking behavior poses a serious performance-based attack on the REM system that utilizes the sequential comparison.

C. REM by Nondeterministic Finite Automata

REM can also be performed by an NFA through the following procedures.

Definition 2: To match a regex r against a sequence of input characters X by an NFA:

1: Obtain the *matching regex* r' by prefixing r with "`.*`" (Kleene closure over an *any-character*).
2: Construct an NFA to accept $L(r')$, following the McNaughton-Yamada construction [16].
3: Feed the characters in X to the NFA. A match is found when a *match state* is reached.

Step 1 above allows the resulting automata to search for r anywhere inside X. As a result, the NFA constructed to accept $L(r')$ in step 2 is different (and more complex) than an NFA that accepts $L(r)$.[2] As long as the finite automaton accepting $L(r')$ is continuously fed with the input characters, *all* matches (sub-sequences which are members of $L(r)$) in the input will be reported by step 3. As such, NFA-based REM is functionally more powerful than the serial REM.

With NFA-based REM, multiple states can be *active* at the same time, while each active state can make transitions to zero or more states per input character. Although the number of states is linear in the length of the regex [9], high memory/computation bandwidth per input character can be required to access/process the transitions from all active states.

[1] Some software REM features such as backreference and recursion do *not* produce regular languages. They are not the focus of this work.

[2] Special operators (*e.g.*, ^ and $, respectively) may be used to force the search to begin and end at input boundaries (usually a newline or the EOF character). For simplicity and without loss of generality, we assume these conditions are taken care of when obtaining the matching regex r'.

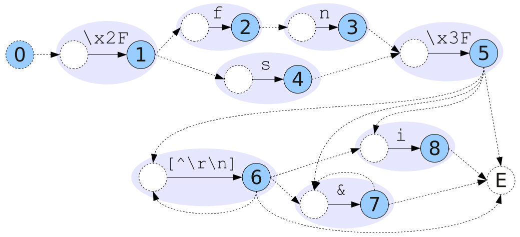


Figure 1: A modular NFA for the regular expression “\x2F(fn|s)\x3F[^\r\n]*(i|&*)”.

This makes NFA-based REM inefficient when implemented straight-forwardedly on processor-based systems.

D. REM by Deterministic Finite Automata

The NFA can be further converted to a DFA using the *subset construction* algorithm, which traverses the NFA with all possible input and find any subset of concurrently active NFA states. Each such (unique) subset of NFA states is assigned to a new DFA state.

With DFA-based REM, only one DFA state is active at any time. For each input character, the DFA makes one access to the state transition table (STT), performs a constant amount of computation, and finds a single transition target. While being computationally efficient (and optimal), the DFA can suffer from *state explosion* [12, 22, 27] during its construction, where the number of states required by the DFA is quadratically or exponentially larger than the original NFA [17, 27]. State explosion can be caused by certain regex patterns, or by combining several non-exploding patterns in one DFA [27].

Without state explosion, DFA-based REM usually offers better throughput performance than NFA-based REM on multi-core systems. This is especially true when the size of the DFA STT is in the same order of magnitude as the on-chip cache size of the multi-core processor, in which case most accesses to the STT are served quickly without accessing external memory. When there is state explosion during the DFA construction, DFA-based REM can become impractical due to the large memory requirement, severe cache thrashing, and extremely high construction and update complexity.

III. SR-NFA Architecture

A. Modular NFA Construction

To compile a regex r into SR-NFA, we first obtain a *modular* NFA for r using the modified McNaughton-Yamada (MMY) construction [26]. Figure 1 shows a modular NFA constructed by MMY for regex “\x2F(fn|s)\x3F[^\r\n]*(i|&*)”. With MMY, each regex symbol always generates a pair of nodes in the modular NFA; one node in the pair has ϵ fan-in and labeled fan-out transitions, while the other one has labeled fan-in and ϵ fan-out transitions. Each pair of nodes corresponds to a *state* of the modular NFA.

The MMY construction can be graphically described as Figure 2. Each oval represents the a sub-NFA, with both p and q representing sub-regexes. Each dashed line represents an ϵ-transition connecting the output of one sub-NFA to the input of another. In the lower half of Figure 2, we further extend the MMY construction to directly handle the extended regex operators (see Table I). Both optionality (?) and repetition (+) are special cases of the Kleene closure where the backward and forward ϵ-transitions, respectively, are omitted. Several cases of constrained repetition ({m,n}) are possible, depending on the relative values of m and n (whether they are equal to each other, or to zero or infinity); we only show the general case where $0 < m < n < \infty$. Handling these extended operators directly helps to reduce the time and memory complexity of the NFA construction.

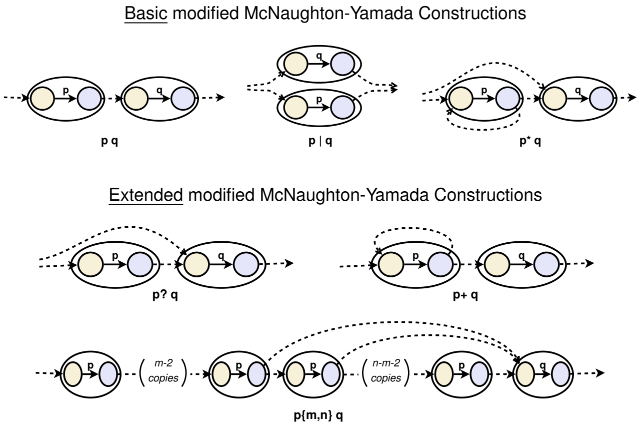

Basic modified McNaughton-Yamada Constructions
p
q
p q
p | q
p* q
Extended modified McNaughton-Yamada Constructions
p? q
p+ q
m-2 copies
n-m-2 copies
p{m,n} q

Figure 2: Graphical description of the basic (*upper*) and extended (*lower*, supporting ?, + and {m,n}) MMY constructions.

	2	3	4	5	6	7	8	E
1	1		1					
2		1						
3				1				
4				1				
5					1	1	1	1
6					1	1	1	1
7						1	1	1
8								1

Figure 3: The state-reachability matrix of a single-segment SR-NFA for Figure 1.

B. Single-Segment SR-NFA

We first describe an SR-NFA with only one segment. We observe that any modular NFA with n states can be fully described in two data structures:

- An $n \times n$ state-reachability matrix, which records potential state transitions between every pair of states.
- An n-element character-acceptance vector, which records the set of character values accepted by every NFA state.

1) State-Reachability Matrix (SRM): Figure 3 shows the state-reachability matrix (SRM) for the modular NFA in Fig-

Hex(char)	1	2	3	4	5	6	7	8
0A						0		
0D						0		
26 (&)						1	1	
2F	1					1		
3F					1	1		
66 (f)		1				1		
69 (i)						1		1
6E (n)			1			1		
73 (s)				1		1		

Figure 4: The character-acceptance matrix of the SR-NFA in Figure 1. All unspecified rows have value `0`'s except at column 6, where they have value `1`'s.

ure 1. Each '1' at row h and column k in the SRM represents an ϵ-transition going from state h to state k.

At run time, each active NFA state will perform one SRM lookup to find the vector of its ϵ-transition targets. Such vectors from all active stateas are bit-OR'd together to form the vector of "potentially active" states prior to the input character matching. Suppose p states are active concurrently at run time, it takes p accesses to the SRM and p bit-OR operations to find the next vector of potentially active states.

The memory complexity of an SRM with n states is exactly n^2 bits or $\lceil n^2/8 \rceil$ bytes.

2) Character-Acceptance Matrix (CAM): Figure 4 shows the character-acceptance matrix (CAM) for the modular NFA in Figure 1. A '1' at row c and column k in the CAM shows that character c is accepted by state k in the NFA.

Note that instead of creating a vector with n (variable-length) lists of character values, one list for each state, we use a *matrix* of size $256 \times n$ to encode the acceptance of every character value (total 256 values) at every state (total n states). In the worst case, this can increase the memory complexity by $256/8 = 32$ times, suppose each state accepts only a single (8-bit) character value. However, due to the (common) use of character classes in real-life regexes, an NFA state often accepts a (custom-defined) character class with tens and even hundreds of values. In addition, the matrix representation allows us to perform character matching for all n states in a single row-access to the CAM followed by one bit-AND operation.

For a modular NFA of n states, the CAM has memory complexity of $256 \times n$ bits of $32 \times n$ bytes.

3) Single-Segment Operation: Algorithm 1 describes the run-time algorithm of single-segment SR-NFA.

Algorithm 1 Single-segment SR-NFA Operation.

Require: A state-reachability matrix R; a character-acceptance matrix A; an initial state vector $\bar{q}$ and a match state vector $\bar{m}$.

Ensure: Regular expression matching (REM) for regex r corresponding to $(R, A, \bar{q}, \bar{m})$

```
The SR-NFA maintains a bit-vecotr $\bar{v} = [v_1 \cdots v_n]$, where each $v_i$ corresponding to one NFA state.
for each input character $c$ do
    $\bar{v} \leftarrow \bar{v}$ OR $\bar{q}$ {initial states from NFA root(s)}
    $\bar{v} \leftarrow \bar{v}$ AND $A(c)$ {character matching}
    $\bar{t} \leftarrow [0 \cdots 0]$ {a temporary states}
    for each $v_i = 1$, $1 \leq i \leq n$ do
        $\bar{t} \leftarrow \bar{t}$ OR $R(v_i)$
    end for
    $\bar{v} \leftarrow \bar{t}$ {update run-time states}
    $\bar{t} \leftarrow \bar{t}$ AND $\bar{m}$
    if $\bar{t} \neq [0 \cdots 0]$ then
        Report $\bar{t}$ as match(es)
    end if
end for
```

C. Data Structures for Multi-Segment SR-NFA

With only one segment in the SR-NFA, the memory complexity of the SRM grows quadratically in the number of states. In addition, each bit-AND and bit-OR on the state vector becomes a proportionally more complex operation. The size of the SRM can become exceesively large for a single-segment SR-NFA with a few thousand states. On the other hand, the SRM is usually very sparse in the NFAs for most real-life regexes.

To improve memory efficiency, we partition large SR-NFA into multiple segments of the same length. Ideally, each state in one segment only transitions to none or few states in other segments, with the length of the semgents matching the size of the longest word in the multi-core processor. Algorithm 2 describes the heuristics we use to segment a large SR-NFA. Note that due to the recursive nature of the MMY construction (see Section III-A), the SR-NFA has a rather "clean" structure with each state transitioning either *forward* or *backward* to a relatively small number of target states. We take advantage of this "natural order" of states when partitioning the SR-NFA into multiple segments.

After the partitioning, each segment has its own SRM to handle the intra-segment ϵ-transitions. In addition, two types of pseudo-states and transition matrices are added to each segment to handle cross-segment ϵ-transitions:

- A set of *forward-pseudo* states and a *forward-transition matrix* (FTM), which relay forward ϵ-transitions to the next segment (usually caused by long chains of cancatenations and/or wide unions in the regex).
- A set of *backward-pseudo* states and a *backward-transition matrix* (BTM), which relay backward ϵ-transitions to the previous segment (always caused by Kleene closures around large sub-regexes).

To make cross-segment transitions, the SRM of each segment is slightly widened to cover the forward and backward-pseudo states in that segment. When a state needs to make a cross-segment transition, it first transitions to a forward-pseudo (backward-pseudo) state, which then relays the transition to the next (previous) segment by accessing the corresponding

Algorithm 2 SR-NFA segmentation algorithm.

```
for each state do {find its "natural order"}
    First based on the forward $\epsilon$-transitions caused by the concatenate operators.
    Then based on the backward $\epsilon$-transitions caused by the $\star$ or $+$ operators.
    Finally based on the $\epsilon$-transitions caused by the union or other operators.
end for
for each state in above order do {find state number}
    Number the state in its natural order.
    If two states were not relatively ordered, then either one can have a lower state number.
end for
for each state in increasing state number do
    if the current segment is full then
        Create a new segment.
        Use the new segment as current segment.
    end if
    Add the state into the current segment.
end for
```

row in the FTM (BTM).

More importantly, the FTM (BTM) is also widened to let the pseudo state transition to all normal and pseudo states in the next (previous) segment. This allows a forward (backward) transition to go across multiple segments by passing through multiple pseudo states.

Specifically, assume the SR-NFA is partitioned into s segment, where each segment $i, 0 \leq i \leq s-1$, has n_i normal states, f_i forward-pseudo states and b_i backward-pseudo states. The SRM of segment i is extended to size $n \times (n_i + f_i + b_i)$, with additional columns corresponding to the forward and backward-pseudo states. The FTM of segment i is a bit matrix of size $f_i \times (n_{i+1} + f_{i+1})$, allowing the forward-pseudo states of segment i to ϵ-transition to both normal and forward-pseudo states of segment $i+1$. Similarly, the BTM of segment i has size $b_i \times (n_{i-1} + b_{i-1})$, allowing ϵ-transitions to both normal and backward-pseudo states of segment $i-1$.

Figure 5 shows a multi-segment SR-NFA for the modular NFA in Figure 1. Each segment in Figure 5 has up to 3 normal states and 2 forward-pseudo states. For clear presentation we did not use backward-pseudo states in this example; the backward transitions are handled in the same way (although in a revsered direction) as the forward transitions. Figure 6 shows the per-segment SRM and FTM to implement the multi-segment SR-NFA in Figure 5. Note that instead of using an 8×8 SRM as in Figure 3 for the single-segment SR-NFA, here we have 5 much smaller matrices resulting in over 40% reduction of memory usage.

D. Multi-Segment SR-NFA Processing

To update the run-time states for REM on a multi-segment SR-NFA, three types of processing are performed iteratively, one iteration per input character.

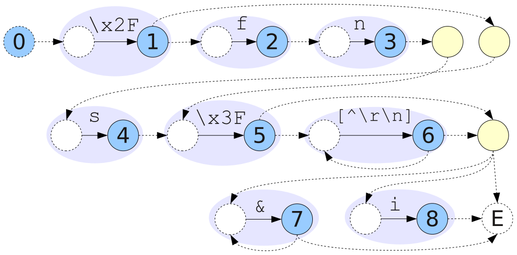


Figure 5: The segmented SR-NFA converted from Figure 1 with automatically introduced forward-pseudo states.

Segment 0: states 1 through 3

Reachability matrix

	2	3	F1	F2
1	1			1
2		1		
3			1	

Forwarding matrix

	4	5	6	F3
F1		1		
F2	1			

Segment 1: states 4 through 6

Reachability matrix

	5	6	F3
4	1		
5		1	1
6		1	1

Forwarding matrix

	7	8	E
F3	1	1	1

Segment 2: states 7 and 8

Reachability matrix

	7	8	E
7	1		1
8			1

Forwarding matrix

(None needed)

Figure 6: Per-segment reachability and forwrading matrices for the segmented SR-NFA in Figure 5.

1) Intra-segment processing (ISP) : Two operations are performed by ISP: character matching and intra-segment transitions. Both are handled in the same way as described in Algorithm 1. The only difference is that the number of colums in the state-reachability matrix (SRM) is increased to include both forward-pseudo and backward-pseudo states as transition targets. On the other hand, the pseudo-states do *not* take place in the character-acceptance matrix (CAM). The pseudo-state values are cleared at the character matching step.

With the number of states per segment matching the length of the processor word, the worst-case time complexity of ISP per input character becomes linear in the number of segments. Furthermore, only the active states in a segment need to be processed, and in practice the number of concurrently active states is usually much smaller than the total number of states in the segment. With the help of bit manipulation instructions such as BSR (bit scan reverse) and LZCNT (leading zero count), the time complexity of ISP per input character can be reduced to the number of active states in the segment.

Because ISP only concerns state transitions within individual segments, for each input character, ISP for all segments can be performed independently in any order.

2) Forward-segment processing (FSP): FSP handles ϵ-transitions from one segment to a following segment between two iterations of ISP. For each segment, FSP must be performed after ISP, which produces the initial set of active forward-pseudo states (if any) in that segment.

A forward ϵ-transition can go across multiple segments by passing through multiple forward-pseudo states in consecutive segments. An active forward-pseudo state is cleared after it makes its ϵ-transitions. Eventually, every forward-pseudo state will ϵ-transition to some normal states in the following segments, upon which time the current iteration of FSP is finished. This can be performed efficiently by sweeping through the SR-NFA from the first segment till the last segment. Specifically, we first process the active forward-pseudo states in segment 0, making any ϵ-transitions to the (normal and forward-pseudo) states in segment 1; then, we process the active forward-pseudo states in segment 1, making any ϵ-transitions to the states in segment 2; and so on, until we reach the last segment.

The time complexity of FSP per input character is bounded by the total number of segments times the maximum number of active forward-pseudo states per segment. In practice, the number of active forward-pseudo states at run time is usually much less than the number of active states; the total computation complexity of FSP NFA is usually less than that of ISP. However, due to the strong (sequential) dependency between FSP on consecutive segments, there is much less instructio-level parallelism (ILP) in FSP than in ISP.

3) Backward-segment processing (BSP): BSP is similar to FSP except BSP handles ϵ-transitions in the reverse direction, from one segment to a previous segment. . BSP also must be performed on any segment after ISP, which will produce the initial set of backward-pseudo states active in that segment.

To relay potentially long backward ϵ-transitions across multiple segments, BSP is performed from the last segment back to the first segment in (reversed) consecutive order. In practice, the time complexity of BSP per input character is even smaller than that of FSP, since there is usually very few long-range backward ϵ-transitions (always caused by Kleene closures over large sub-regexes) in real-world SR-NFAs.

Although BSP must be performed sequentially on consecutive segments in the reverse order, it can be performed in parallel to FSP with proper segment locking mechanism.

IV. SR-NFA Optimizations

A. Specialized STR *and* REP *Segments*

Even with a multi-segment SR-NFA, the various matrices, especially the state-reachability matrix (SRM) and the character-acceptance matrix (CAM) can still be sparse for two common types of regex sub-patterns:

- *String*: A sequence of characters concatenated one after another. *E.g.*, "`Authentications\s:`".
- *Repetition*: A single character class is repeated a large number of times. *E.g.*, "`[^\r\n]{1024}`".

We notice that in the case of *string* sub-pattern, every state within its span implicitly ϵ-transitions to the next adjacent state, thus the SRM is not needed. In the case of *repetition* sub-pattern, all states within its span accept the same character class, thus neither the SRM nor the CAM is needed.

Recall that with k-byte segments, a set of n states can take $\lceil k \times n \rceil$ bytes in the SRM and $32 \times k$ bytes in the CAM. Thus representing sub-strings or sub-repetitions in normal SR-NFA segments can waste a lot of memory for unused space in SRM and CAM. Such memory inefficiency can boast the SR-NFA size and reduce its critical cache performance. To alleviate this problem, we design two types of specialized segments, STR and REP, to handle the sub-strings and sub-repetitions respectively.

1) Basic STR *and* REP *optimizations:* Instead of occupying one normal segment bit by every sub-string state, we create an STR segment of n bits to hold a sub-string of n states. In addition, we add three STR-related special bits to the normal segment: an *entry* bit signalling when and where to active the first state in the sub-string, an *valid* bit showing the "activeness" of the sub-string, and an *exit* bit specifying when to check for the output condition of the sub-string. The STR segment has no SRM, but is associated with its own CAM.[3]

At run-time, whenever the entry bit for an STR segment is activated, the least-significant bit in the STR segment is set to 1. For each input character, the STR segment is simply left-shifted by 1 bit before the segment is matched to the input character. When a '1' reaches the most-significant bit in the STR segment, the *exit* bit for the STR segment is set to '1' in the normal segment, from which transitions to other states in the normal segment can commence.

Similarly, we create an REP segment of n bits to hold a sub-repetition of n states. The REP segment requires neither SRM or CAM; the entire REP segment can be matched to the input character as a single bit in the normal segment. This results in even greater savings in both memory and computation complexity.

2) Advanced STR *and* REP *optimizations:* On top of the basic optimizations described above, we further perform more sophisticated optimizations to map the STR and REP segments more efficiently onto the processor architectures.

First, we notice that there can be many short (< 10 symbols) sub-strings in a regex, resulting in numerous "call-outs" to the STR processing. To handle such cases more efficiently on modern processors with long words, we design mechanisms to "merge" multiple short sub-strings into one long (64-bit) STR segment in byte granularity. The entire STR segment is processed as a single unit, while each short sub-string still maintains its "identity" through the valid bit associated with it in the normal segment.

For sub-repetitions, it is the opposite scenario. Often a character class is repeated hundreds of times. Storing long and variable-length bit vectors is rather computationally expensive in most processor architectures. Instead, we dissect a long sub-repetition into multiple REP segments, each with the length of

[3] In REM most strings are compared in a caseless manner, while some have complex character classes. Using CAM offers speed advantage by comparing the sub-string to the input character in parallel.

a processor word. This allows multiple long sub-repetitions to be stored and processed in a uniform word array.

These two optimizations lay the groundwork for our final optimization technique, an "activeness" bitmap for both types of specialized segments. Because all STR or REP segments now have the same (processor word) length and are processed in a uniform way, we can use a bitmap to represent their activeness, one bit per segment. By paying a little extra memory and computation, the bitmap allows the REM process to skip the non-active segment processing quickly using the LZCNT and BSR instructions (see end of Section III-D1).

B. Merging multiple regexes

Many real-world regexes can be short after removing long sequences of sub-strings and sub-repetitions. Over one-third regexes used by Snort IDS [4] consists of less than 40 characters and compiles to no more than 40 NFA states. One way to improve the throughput performance of multi-pattern REM is to merge multiple short regexes into a single segmented SR-NFA. This is especially beneficial if various regexes share a common prefix, for which only one set of SR-NFA states are needed.

Eventually, however, various merged regexes will diverge to different match states. If M regexes are merged to be processed together, then the resulting SR-NFA shall have a tree-like structure with M leaves. To handle such match-state divergence, the SR-NFA can use a tree-like segmentation structrue where all segments are organized in a binary tree, rather than a linear sequence. In an s-segment SR-NFA with tree-like structure, each segment $i, 1 \leq i < s/2$, can make two types of forward transitions: one to segment $2i$ and the other to segment $2i+1$; each segment $j, 1 < j \leq s$, can make backward transitions only to segment $\lfloor j/2 \rfloor$. Segments numbered from $\lceil s/2 \rceil$ to s are "leaf" segments, whose forward-pseudo states are overloaded as the special match states.

Figure 7 graphically shows an example where 9 regexes (r1–r9), some sharing common prefixes, are merged into a segmented SR-NFA with 7 segments (s1–s7) using the tree-like segmentation structure. Inside the segments, each colored stripe on the left represents a sub-pattern of a particular regex; each colored bar on the right represent a forward-pseudo state for some sub-pattern. The dotted arrows show the sequence of segments and forward-pseudo states traversed in order to match regex r1 (purple) and r6 (brown), respectively.

Note that *merging* multiple regexes is different from *unioning*. With merging, each regex still maintains its identity; for example, matching r1 in Figure 7 remains distinguishable from matching r9. With unioning, however, multiple regexes become a single, indistinguishable one. This difference is especially important for deep packet inspection type of applications where matched regexes have critical individual significance.

C. Thread-level Parallelism for Multi-Regex Matching

Modern multi-core processors are built with increasingly large numbers of cores. Each core is often equipped with a dedicated level-1 (L1D) and/or leve-2 (L2D) data cache, ranging from a few tens to several hundred kilobytes in size. In order to achieve scalable performance, it is especially important to take advantage of the on-chip cache resources and to fully utilize the available cores on the multi-core processors. Below we describe two dimensions of thread-level parallelism that can be exploited by our SR-NFA architecture on a generic class of multi-core systems.

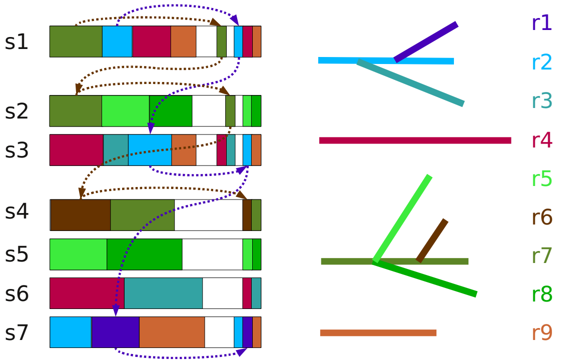


Figure 7: Example of 9 regexes (r1–r9) merged into a 7-segment (s1–s7) SR-NFA with the tree-like structure.

1) Parallelism with multiple regexes: Given the relatively small size of most real-life SR-NFA, we can run a different SR-NFA on every core in a multi-core system without increasing much bandwidth pressure to the (shared) memory subsystem. This allows us to scale up the number of concurrently matched regexes proportionally to the total number of cores without sacrificing the matching throughput.

For example, asssume each segment in the SR-NFA has up to 40 normal states, 16 forward-pseudo states and 8 backward-pseudo states. According to Section III, the run-time states of each segment will be 8 bytes (64 bits) in length; the per-segment SRM will take $40 \times 8 = 320$ bytes and the CAM $256 \times 5 = 1,280$ bytes. Plus the FTM $16 \times 8 = 128$ bytes and the BTM $8 \times 8 = 64$, the entire segment can fit into 2 KB of cache memory. The use of STR and REP segments can further increase the total number of NFA states handled with little memory requirement. Using the SR-NFA architecture, most real-world regexes (even those compiled into hundreds of NFA states) can be mapped entirely onto several tens kilobytes of level-1 data cache in modern microprocessors.

2) Parallelism with multiple inputs: It is possible for a multi-core system to process a single SR-NFA in multiple cores with different inputs. Due to the small memory footprint of the SR-NFA, having more cores accessing multiple SR-NFAs will not pressure the external memory. Since the number of cores in modern multi-core systems is increasing at a much higher speed than the number of memory channels, SR-NFA can achieve a much better throughput scaling for matching regexes that would otherwise be converted to a large (state-exploded) DFA.

Table II: Memory usage for various segment sizes

Seg. type (bits)	32 22:7:3	48 33:11:4	64 44:15:5	STR 64	REP 64
CAM	704	1056	1408	2048	N/A
SRM	88	198	352	N/A	N/A
FTM+BTM	40	90	160	N/A	N/A
Total	832	1344	1920	2048	0
Bytes/state	**4.7**	**5.1**	**5.5**	**4**	**0**

V. Performance Evaluation

A. Resource Usage

Table II shows the amount of memory used for segments of various types and lengths in our SR-NFA. We implemented 32, 48, and 64-bit normal segments, as well as 64-bit STR and REP segments. Larger segments trade memory efficiency off for matching capacity. For example, by increasing segment size from 32 bits to 64 bits, we can process a 2x larger SR-NFA with roughly the same throughput; on the other hand, memory usage increases from 4.7 to 5.5 bytes/state.

While both specialized (STR and REP) segments are processed 64 bits at a time, their designs (Section IV-A) allow each segment to be utilized with single-byte granularity (*i.e.*, each 64-bit STR or REP segment can store up to 8 short sub-strings or sub-repetitions, respectively).

Although the 64-bit STR segment has a relatively large memory size, its run-time processing is much simplified. There is no SRM/FTM/BTM accesses needed, significantly lowering the memory bandwidth required to process these segments. The REP segment is essentially "free" in terms of memory footprint (except the run-time state of 64 bits per segment). It has even lower processing complexity than the STR segment, as discussed in Section IV-A.

B. Construction Time

The REM solution construction time (or compilation time) is a metric often ignored in the literature. However, in real-world applications, it is usually one of the most important metric for the usability of a REM program. In a dynamic setting where regexes can be continually updated, or where different subsets of regexes can be used to match against various inputs, the ability to quickly construct an REM solution while still offering good performance becomes important.

Figure 8 shows the average time it takes to compile Snort regexes incrementally into the SR-NFA architecture using Algorithm 2. Analytically, each step in Algorithm 2 has time and space complexity no worse than linear in total number of states. In practice, compiling a regex (to 20 ~ 4,000 states) takes less than 0.3 ms for all the tested real-life regexes used by Snort. The sub-millisecond construction allows our SR-NFA architecture to be updated at run time.

C. Throughput Performance

We evaluated our SR-NFA prototype on a dual-socket quad-core Opteron 2382 server. For evaluation, we used 64-bit normal and specialized (STR and REP) segments. Each normal segment consists of 44 character matching state, 14 forward-pseudo 4 backward-pseudo states. Not all pseudo states are utilized in most segments. The fixed number of pseudo states helps us simplify the programming and slighly improves the run-time performance.

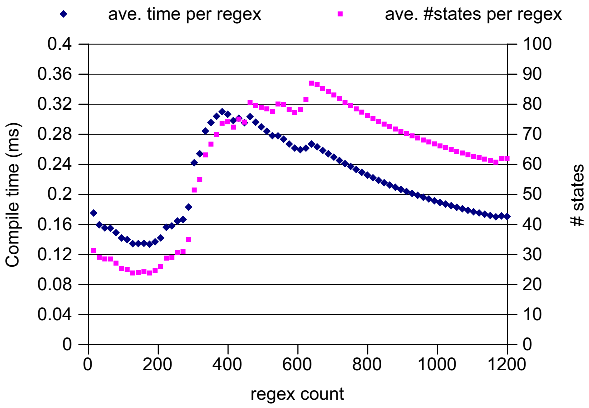


Figure 8: Compilation time (in milliseconds) per regex to build an *incrementally* larger SR-NFA for the set of "regular" regexes used by Snort rules.

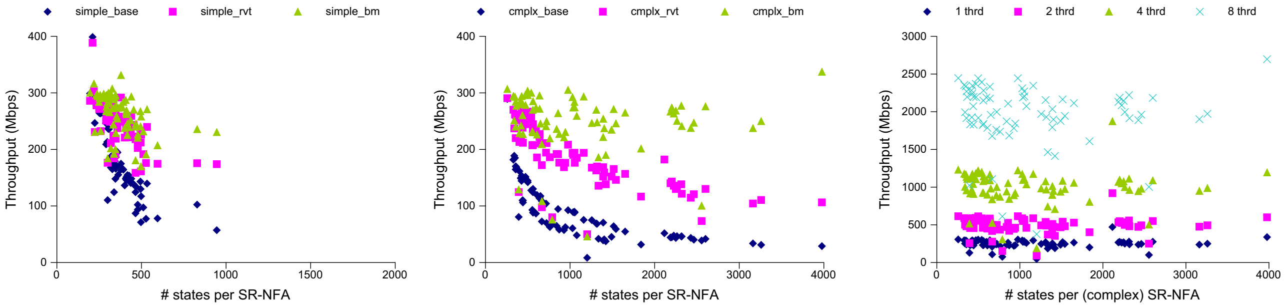
We partition 1134 real-life regexes from the Snort rulesets in two groups:

1) Those "simple" sets of regexes which can be compiled to DFAs with little or moderate state explosion. Each of these DFAs has an STT taking less than 1 GB memory and can be constructed in less than half an hour.
2) Those "complex" sets of regexes which are compiled to DFAs with severe state explosion, resulting in multi-gigabytes STT size and taking hours for compilation (some could not even be practically completed).

In both groups above, a regex set consists of 2 to 7 regexes and is compiled to an SR-NFA with 80 to 4,000 states. In total we created 280 set, or on average 4 regexes per set. While 4 regexes per set is a relatively small number, we note that such arrangement is actually useful in real-life scenarios:

- In practice, we usually do not need to match a large set of regexes against a single input (*e.g.*, an Internet traffic flow). Instead, once the type of the input is identified, a small set of regexes associated with the type is used.
- For all complex and even some simple regex sets, adding more regexes results in exponentially larger STT and long compilation time. Thus matching in small subgroups may be the only choice (if at all possible) for these regex sets.
- Most simple regex sets do not cause state explosion when compiled to DFA individually, but will when compiled with other simple sets. Without compiling the regexes, it is not always easy to tell *in advance* whether a set of regexes *will* cause state explosion.

We implemented SR-NFA with basic configuration (base), with the specialized STR and REP segments (rvt), and with



(a) "Simple" regexes

(b) "Complex" regexes

(c) Throughput scaling

Figure 9: Throughput optimization and scaling of SR-NFA for (a) regexes with little state explosion (simple); (b) regexes with high degrees of state explosions (complex); (c) complex regex matching with 1, 2, 4 and 8 threads.

Table III: Performance comparison of SR-NFA with improved sequential matching (ISM) and DFA approaches.

(average)	Throughput	Mem. size	Compile time
SR-NFA	2.2 Gbps	8.6 KB	0.23 ms
ISM [21]	~1.2 Gbps	n/a	n/a
DFA	<0.7 Gbps	202 MB	84.4 sec

additional "activeness" bitmaps (bm) as discussed in Section IV-A. As shown in Figure 9, our optimization techniques improve throughput performance across all regexes of different sizes and classes. (A few optimized data points are off the chart and not shown in the plots.) In fact, our optimizations show more throughput improvements in complex regex sets than in simple regex sets. This makes the SR-NFA architecture even more attractive for matching those regex sets that would result in severe state explosions.

In Figure 9c we demonstrate the throughput scaling of the SR-NFA architecture. We take the complex regex sets, compiled with all optimization techniques, and run on 1 to 8 cores in the 8-core server. On average we are able to achieve 1.6 Gbps to 2.5Gbps with 8 threads. The close-to-perfect throughput scaling in number of threads is expected since the largest SR-NFA has a data structure much smaller than the 64 KB L1D cache on the Opteron processor.

D. Performance Comparison

Table III compares performance of SR-NFA with improved sequential matching approach in [21], as well as a straightforward DFA-based REM. Throughput values from [21] are scaled *linearly* in clock rate (2.0 $\to$ 2.6 GHz), pipeline width (2 $\to$ 3 way) and number of cores (1 $\to$ 8) to the newer CPU. The SR-NFA results are averages of the complex regex sets (from Figure 9c). The DFA results are averages of regex sets that compile to DFAs with an STT between 25 MB and 1 GB in size (about 30% of all the regex sets we tested).

Note that we did not compare with all previously proposed REM approaches for two important reasons. First, some previous REM solutions were based on or improved from DFA. When there is little or no state explosion, those DFA-based solutions can achieve very good throughput and are the preferred choice for REM; the SR-NFA, on the other hand, should be used to handle cases where severe state explosion makes DFA-based REM impractical. Second, a few prior work (*e.g.* [14, 18]) propose simplified pattern matching mechanisms that do not handle full regular expression capabilities.[4] These solutions can still be useful to match "regexes" known to have the limited features. However, they are not considered full regular expression matching solutions in this study.

VI. Related Works

NFA-based REM was initially implemented in a circuit-based architecture [9], where an n-character regex is converted to an n-state NFA and mapped to an integrated circuit using no more than $O(n)$ circuit area. Sidhu and Prasanna [20] later proposed to construct NFA-based REM circuits on field-programmable gate arrays (FPGA) [20]. Optimizations such as input/output pipelining [11], common-prefix extraction [8, 11], temporal [24] and spatial [25] multi-character matching, and centralized [8, 15] and memory-assisted [25] character decoding, were applied to further improve the matching throughput and resource utilization. While achieving high matching throughput (~10 Gbps) over large number (>1k) of patterns, the Achilles' heel of NFA-based REM on FPGA is the difficult circuit-based implementation.

DFA-based REM utilizes relatively simple operations on processor-based architectures and can achieve very good throughput with simple regexes [23]. However, state-space explosion of the resulting DFA can greatly increase the amount of required memory, which could in turn impact the memory performance and even the practical feasibility of the solution. Various techniques were proposed to compress the state transition table (STT) of the DFA with nondeterministic features [7, 10, 12, 13, 22]. These techniques, however, can be

[4] It can be shown, as discussed in [5], that the simplified pattern construction does not genrate all regular languages. A proof however is out of the scope of this paper.

computationally expensive and are often designed *a-posteriori* to a particular set (or type) of regexes.

High-performance REM software was proposed in [18], which matches pattern in a memory-based approach similar to the Generalized Aho-Corasick algorithm [14]. While reporting very high throughput, their software accepts only a simplified set of regex construction; it does not allow unions of arbitrary sub-regex, and accepts Kleene closures only over single *any-value* characters.

Partitioning a REM into multiple smaller parts is a well-known technique. In [12], a regex is split into a prefix and a suffix to avoid or reduce state explosion in the prefix part. A similar technique was also used in [6] to partition multiple regexes into a head-DFA plus individual tail-NFAs. Both of these works relied on the NFA-to-DFA conversion and various DFA compression techniques to implement REM on shared-memory multi-core architectures. As such, the regexes and their prefixes were chosen carefully (and heuristically) to minimize the DFA state explosion.

VII. Conclusion and Future Work

We propose the compilation and optimization of arbitrarily complex regexes into a novel SR-NFA architecture. SR-NFA has very fast construction time and small memory footprint, offers attack-resilient matching throughput, and can be mapped efficiently on processor architectures. Future multi-core processors are expected to have larger number of cores, wider SIMD length (*e.g.* 256-bit AVX), and higher thread-level parallelism in a shared memory system. By simply riding on the wave of hardware improvement, and with a more optimized implementation, we expect continual performance improvement in REM using the proposed SR-NFA architecture.

References

[1] Bro Intrusion Detection System. http://bro-ids.org.

[2] Clam AntiVirus. http://www.clamav.net/.

[3] PCRE: Perl Compatible Regular Expression. http://www.pcre.org/.

[4] SNORT. http://www.snort.org/.

[5] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972.

[6] Michela Becchi and Patrick Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *ACM CoNEXT*, pages 1–12, 2007.

[7] Michela Becchi and Patrick Crowley. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, pages 145–154, 2007.

[8] João Bispo, Ioannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Proc. IEEE Intl. Conf. on Field Programmable Technology (FPT)*, pages 119–126, December 2006.

[9] Robert W. Floyd and Jeffrey D. Ullman. The Compilation of Regular Expressions into Integrated Circuits. *Journal of ACM*, 29(3):603–622, 1982.

[10] J. Grosch. Efficient generation of lexical analyzers. *Software–Practice & Experience*, 19(11):1089–1103, 1989.

[11] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proc. IEEE Sym. on Field-Programmable Custom Computing Machines (FCCM)*, page 111, 2002.

[12] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, pages 155–164, 2007.

[13] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *SIGCOMM Computer Communication Review*, 36(4):339–350, 2006.

[14] Tsern-Huei Lee. Generalized aho-corasick algorithm for signature based anti-virus applications. In *Proc. Intl. Conf. on Computer Communications and Networks (ICCCN)*, 2007.

[15] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of Regular Expression Pattern Matching Circuits on FPGA. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pages 12–17, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[16] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Trans. on Comput.*, 9(1):39–47, March 1960.

[17] A. R. Meyer and M. J. Fischer. Economy of description by automata, grammars, and formal systems. In *Proc. 12th Sym. on Switching and Automata Theory (SWAT '71)*, pages 188–191, Washington, DC, USA, 1971. IEEE Computer Society.

[18] Davide Pasetto, Fabrizio Petrini, and Virat Agarwal. Tools for very fast regular expression matching. *Computer*, 43:50–58, 2010.

[19] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Exact Multi-pattern String Matching on the Cell/B.E. Processor. In *Computing Frontiers*, pages 33–42, 2008.

[20] R. Sidhu and V.K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. IEEE Sym. on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, 2001.

[21] R. Smith, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *Proc. 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 89–98, Dec. 2006.

[22] Randy Smith, Cristian Estan, and Somesh Jha Shijin Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *ACM SIGCOMM*, August 2008.

[23] Giorgos Vasiliadis, Michalis Polychronakis, Spyros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *RAID*, pages 265–283, 2009.

[24] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-Speed Regular Expression Matching Engine Using Multi-Character NFA. In *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 697–701, Aug. 2008.

[25] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. ACM/IEEE Sym. on Architectures for Networking and Communications Systems (ANCS)*, November 2008.

[26] Yi-Hua E. Yang and Viktor K. Prasanna. Software Toolchain for Large-Scale RE-NFA Construction on FPGA. *Intl. Journal of Reconfigurable Computing*, 2009:10, 2009.

[27] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, pages 93–102, 2006.