Speculative Parallel Pattern Matching

Daniel Luchaup*, Randy Smith, Cristian Estan, Somesh Jha

***Abstract*—Intrusion prevention systems determine whether incoming traffic matches a database of signatures, where each signature is a regular expression and represents an attack or a vulnerability. IPSs need to keep up with ever-increasing line speeds, which has lead to the use of custom hardware. A major bottleneck that IPSs face is that they scan incoming packets one byte at a time, which limits their throughput and latency. In this paper, we present a method to search for arbitrary regular expressions by scanning multiple bytes in parallel using speculation. We break the packet in several chunks, opportunistically scan them in parallel and if the speculation is wrong, correct it later. We present algorithms that apply speculation in single-threaded software running on commodity processors as well as algorithms for parallel hardware. Experimental results show that speculation leads to improvements in latency and throughput in both cases.**

***Index Terms*—low latency, parallel pattern matching, regular expressions, speculative pattern matching, multi-byte, multi-byte matching, parallel regular expression matching**

I. Introduction

Most Intrusion Prevention Systems (IPSs) match incoming traffic against a database of signatures, which are Regular Expressions (REs) that capture attacks or vulnerabilities. IPSs are a very important component of the security suite. For instance, most enterprises and organizations deploy an IPS. A significant challenge faced by IPS designers is the need to keep up with ever-increasing line speeds, which has forced IPSs to move to custom hardware. Most IPSs match incoming packets against signatures one byte at a time, causing a major bottleneck. In this paper we address this bottleneck by using speculation to solve the problem of *multi-byte* matching, or the problem of IPS concurrently scanning multiple bytes of a packet.

Deterministic Finite Automata (DFAs) are popular for signature matching because multiple signatures can be merged into one large regular expression and a single DFA can be used to match them simultaneously with a guaranteed robust performance of $O(1)$ time per byte. However, matching network traffic against a DFA is inherently a serial activity. We break this inherent serialization imposed by the *pointer chasing* nature of DFA matching using speculation.

This paper is the extended, journal version of the work in [18]. It makes the following contributions: we present Speculative Parallel Pattern Matching (SPPM), a novel method for DFA multi-byte matching which can lead to significant speedups. Our method works by dividing the input into multiple chunks and scanning each of them in parallel using traditional DFA matching. The main idea behind our algorithm is to guess the initial state for all but the first chunk, and then to make sure that this guess does not lead to incorrect results. The insight that makes this work is that although the DFA for IPS signatures can have numerous states, only a small fraction of these states are visited often while parsing benign network traffic. We use a new kind of speculation where gains are obtained not only in the case of correct guesses, but also in the most common case of incorrect ones yet whose consequences quickly turn out to still be valid. This idea opens the door for an entire new class of parallel multi-byte matching algorithms.

Section III presents an overview of SPPM, with details given in Sections IV and V. We present a single-threaded SPPM algorithm for commodity processors which improves performance by issuing multiple independent memory accesses in parallel, thus hiding part of the memory latency. Measurements show that by breaking the input into two chunks, this algorithm can achieve an average of 40% improvement over the traditional matching procedure. We also present SPPM algorithms suitable for platforms where parallel processing units share a copy of the DFA to be matched. Our models show that when using up to 100 processing units our algorithm achieves significant reductions in latency. Increases in throughput due to using multiple processing units are close to the maximum increase afforded by the hardware.

II. Background

A. Regular Expression Matching – a Performance Problem

Signature matching is a performance-critical operation in which attack or vulnerability signatures are expressed as regular expressions and matched with DFAs. For faster processing, DFAs for distinct signatures such as `.*user.*root.*` and `.*vulnerability.*` are combined into a single DFA that simultaneously represents all the signatures. Given a DFA corresponding to a set of signatures, and an input string representing the network traffic, an IPS needs to decide if the DFA accepts the input string. Algorithm 1 gives the procedure for the traditional matching algorithm.

Modern memories have large throughput and large latencies: one memory access takes many cycles to return a result, but one or more requests can be issued every cycle. Suppose that reading $DFA[state][input\_char]$ results in a memory access[1] that takes M cycles[2]. Ideally the processor would schedule other operations while waiting for the result of the read from memory, but in Algorithm 1 each iteration is data-dependent on the previous one: the algorithm cannot proceed with the

D. Luchaup and S. Jha are with the Department of Computer Sciences, University of Wisconsin at Madison, {luchaup,jha}@cs.wisc.edu

R. Smith is at Sandia National Laboratories in Albuquerque, NM, smithr@cs.wisc.edu. Work was done while at U.W. Madison.

C. Estan is at NetLogic Microsystems, estan@netlogicmicro.com. Work was done while at U.W. Madison.

[1] Assuming that the two indexes are combined in a single offset in a linear array

[2] On average. Caching may reduce the average, but our analysis still holds

```
Input: DFA = the transition table
Input: I = the input string, |I| = length of I
Output: Does the input match the DFA?
state ← start_state;
for i = 0 to |I| do
    input_char ← I[i];
    state ← DFA[state][input_char];
    if accepting(state) then
        return MatchFound ;
return NoMatch ;
```

Algorithm 1: Traditional DFA matching.

next iteration before completing the memory access of the current step because it needs the new value for the $state$ variable (in compiler terms, M is the Recurrence Minimum Initiation Interval). Thus the performance of the system is limited due to the pointer chasing nature of the algorithm.

If $|I|$ is the number of bytes in the input and if the entire input is scanned, then the duration of the algorithm is at least $M * |I|$ cycles, regardless of how fast the CPU is. This algorithm is purely sequential and can not be parallelized.

Multi-byte matching methods attempt to consume more than one byte at a time, possibly issuing multiple overlapping memory reads in each iteration. An ideal *multi-byte* matching algorithm based on the traditional DFA method and consuming B bytes could approach a running time of $M * |I|/B$ cycles, a factor of B improvement over the traditional algorithm.

B. Signature Types

Suffix-closed Regular Expressions over an alphabet Σ are Regular Expressions with the property that if they match a string, then they match that string followed by any suffix. Formally, their language L has the property that $x \in L \Leftrightarrow \forall w \in (\Sigma)^* : xw \in L$. All signatures used by IPSs are suffix-closed. Algorithm 1 uses this fact by checking for accepting states after each input character instead of checking only after the last one. This is not a change we introduced, but a widely accepted practice for IPSs.

Prefix-closed Regular Expressions (PREs) over an alphabet Σ are regular expressions whose language L has the property that $x \in L \Leftrightarrow \forall w \in (\Sigma)^* : wx \in L$. For instance, `.*ok.*stuff.*|.*other.*` is a PRE, but `.*ok.*|bad.*` is not, because the `bad.*` part can only match at the beginning and is not prefix-closed. In the literature, non-PRE signatures such as `bad.*` are also called *anchored* signatures. A large fraction of signatures found in IPSs are prefix-closed.

When we need to make an explicit distinction against PRE, as a notational convenience we use the term *General Regular Expressions* (GRE) for unrestricted, arbitrary regular expressions.

III. Overview

The core idea behind the *Speculative Parallel Pattern Matching* (SPPM) method is to divide the input into two or more chunks of the same size and process them *in parallel*. We assume that the common case is **not** finding a match, although speedup gains are possible even in the presence of matches. As is customary in IPSs, all our regular expressions are suffix closed. Additionally, at this point we only match REs that are prefix closed (PRE), a restriction that will be lifted in Sec. V. In the rest of this section we informally present the method by example, we give statistical evidence explaining why speculation is often successful, and we discuss ways of measuring and modeling the effects of speculation on latency and throughput.

A. Example of Using Speculation

As an example, consider matching the input I=`AVOIDS_VIRULENCE` against the DFA recognizing the regular expression `.*VIRUS` shown in Fig. 1. We break the input into two chunks, I_1=`AVOIDS_V` and I_2=`IRULENCE`, and perform two traditional DFA scans in parallel. A *Primary* process scans I_1 and a *Secondary* process scans I_2. Both use the same DFA, shown in Fig. 1. To simplify the discussion, we assume for now that the Primary and the Secondary are separate processors operating in lockstep. At each step they consume one character from each chunk, for a total of two characters in parallel.

To ensure correctness, the start state of the Secondary should be the final state of the Primary, but that state is initially unknown. We speculate by using the DFA's start state, State 0 in this case, as a start state for the Secondary and rely on a subsequent validation stage to ensure that this speculation does not lead to incorrect results. In preparation for this validation stage the Secondary also records its state after each input character in a *History* buffer.

Figure 2 shows a trace of the two stages of the speculative matching algorithm. During the **parallel processing stage**, each *step i* entry shows for both the Primary and the Secondary the new state after parsing the *i*-th input character in the corresponding chunk, as well as the history buffer being written by the Secondary. At the end of step 8, the parallel processing stage ends and the Secondary finishes parsing without finding a match. At this point the *History* buffer contains 8 saved states. During the **validation stage**, steps 9-12, the Primary keeps processing the input and compares its current state with the state corresponding to the same input character that was saved by the Secondary in the History buffer. At step 9 the Primary transitions on input 'I' from state 1 to state 2 which is different from 0, the state recorded for that position. Since the Primary and the Secondary disagree on the state after the 9-th, 10-th and 11-th characters, the Primary continues until step 12 when they agree by reaching state 0. Once this *coupling* between the Primary and Secondary happens, it is not necessary for the Primary to continue processing because it would go through the same transitions and make the same acceptance decisions as the Secondary. We use the term *validation region* to refer to the portion of the input processed by both the Primary and the Secondary (the string `IRUL` in this example). *Coupling* is the event when the validation succeeds in finding a common state.

In our case, the input is 16 bytes long but the speculative algorithm ends after only 12 iterations. Note that for differ-

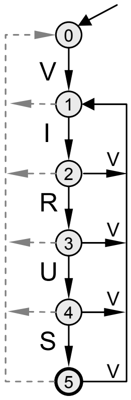


Fig. 1. DFA for `.*VIRUS`; dotted lines show transitions taken when no other transitions apply.

Input	A	V	O	I	D	S		V	I	R	U	L	E	N	C	E
step 1	**0**								**0**							
step 2		**1**							0	**0**						
step 3			**0**						0	0	**0**					
step 4				**0**					0	0	0	**0**				
step 5					**0**				0	0	0	0	**0**			
step 6						**0**			0	0	0	0	0	**0**		
step 7							**0**		0	0	0	0	0	0	**0**	
step 8								**1**	0	0	0	0	0	0	0	**0**
step 9									2≠0,	0	0	0	0	0	0	0
step 10										3≠0,	0	0	0	0	0	0
step 11											4≠0,	0	0	0	0	0
step 12												0=0,	0	0	0	0

Fig. 2. Trace for the speculative parallel matching of P=`.*VIRUS` in I=`AVOIDS_VIRULENCE`. During the parallel stage, steps 1-8, the Primary scans the first chunk. The Secondary scans the second chunk and updates the history buffer. The Primary uses the history during validation stage, steps 9-12, while re-scanning part of the input scanned by the Secondary till agreement happens at step 12.

ent inputs, such as `SOMETHING_ELSE..`, the speculative method would stop after only 9 steps, since both halves will see only state 0. The performance gain from speculative matching occurs only if the Primary does not need to process the whole input. Although we guess the starting state for the Secondary, performance improvements do not depend on this guess being right, but rather on validation succeeding quickly, *i.e.* having a validation region much smaller than the second chunk.

B. Statistical Support for Speculative Matching

In this section we provide an intuitive explanation behind our approach. We define a *default transition* to be a transition on an input character that does not advance towards an accepting state, such as the transitions shown with dotted lines in Fig. 1. If we look at Fig. 1, we see that the automaton for `.*VIRUS.*` will likely spend most of its time in state 0 because of the default transitions leading to state 0. Figure 2 shows that indeed 0 is the most frequent state. In general, it is very likely that there are just a few *hot* states in the DFA, which are the target states for most of the transitions. This is particularly true for PREs because they start with `.*` and this usually corresponds to an initial state with default transitions to itself.

For instance, we constructed the DFA composed from 768 PREs from Snort and measured the state frequencies when scanning a sample of real world HTTP traffic. Fig. 3 displays the resulting *Cumulative Distribution Function* (*cdf*) graph when the states are ordered in decreasing order of frequency. Most time is spend in a relatively small number of states. The most frequent state occurs in 33.8% of all transitions, and the first 6 states account for 50% of the transitions.

The key point is that there is a state that occurs with a relatively high frequency, 33.8% in our case. A back-of-the-envelope calculation shows that it is quite likely that both halves will soon reach that state. Indeed, assume a pure probabilistic model where a state S occurs with a 33.8% probability at any position. The chances for coupling due to state S at a given position are $0.338^2 = 0.114$. Equivalently, the chances that such coupling does not happen are $1 - 0.338^2 = 0.886$. However, the chances that disagreement happens on each of h consecutive positions are 0.886^h, which decreases quickly with h. The probability for coupling in one of 20 different positions is $1 - 0.886^{20} = 0.912$. Even if the frequency of a state S was 5% instead of 33.8%, it would take 45 steps to have a probability greater than 90% for two halves to reach state S. While 45 steps may seem high, it is only a tiny fraction, 3%, compared to the typical maximum TCP packet length of 1500 bytes. In other words, we contend that the length of the validation region will be small.

Note that the high probability of coupling in a small number of steps is based on a heavily biased distribution of frequencies among the N states of the DFA. If all states were equally probable, then the expected number of steps to coupling would be $O(N)$. This would make coupling extremely unlikely for automata with large numbers of states.

C. Performance Metrics

One fundamental reason why speculation improves the performance of signature matching is that completing two memory accesses in parallel takes less time than completing them serially. While the *latencies* of memories remain large, the achievable *throughput* is high because many memory accesses can be completed in parallel.

When we apply SPPM in single-threaded software settings, the processing time for packets determines both the throughput and the latency of the system as packets are processed one at a time. Our measurements show that SPPM improves both latency and throughput. When compared to other approaches using a parallel architecture, SPPM improves latency significantly and achieves a throughput close to the limits imposed by hardware constraints.

IV. Speculative Matching

Speculative Parallel Pattern Matching is a general method. Depending on the hardware platform, the desired output, the signature types, or other parameters, one can have a wide variety of algorithms based on SPPM. This section starts by formalizing the example from Section III-A and by introducing a simplified performance model for evaluating the benefits of speculation. Then we present basic SPPM algorithms for

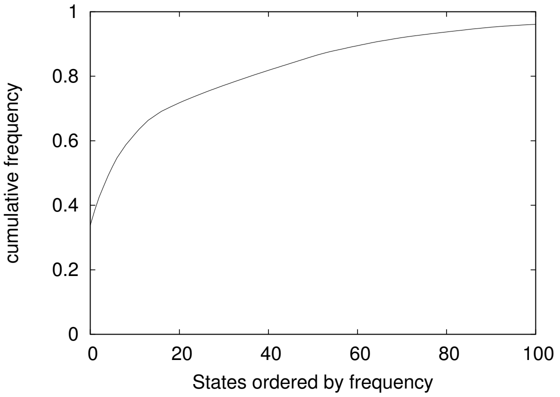


Fig. 3. The state frequency *cdf* graph for a PRE composed of 768 Snort signatures. The most frequent state accounts for 33.8% of the time and the first 6 most frequent states account for half the time.

Metric	Definition
Useful work	Number of bytes scanned, $\|I\|$
Processing latency (L)	Number of parallel steps/iterations
Speedup (S)	$S = \|I\|/L$
Processing cost (P)	$P = N \cdot L$
Processing efficiency (P_e)	$P_e = \|I\|/(N \cdot L)$
Memory cost (M)	Number of accesses to DFA table
Memory efficiency (M_e)	$M_e = \|I\|/M$
Size of validation region (V)	Number of steps in validation stage

Fig. 4. Simplified performance model metrics (N is number of processors).

```
   Input: DFA = the transition table
   Input: I = the input string
   Output: Does the input match the DFA?
   // Initialization stage
1  len ← |I| ;                                            // Input length
2  (len_1, len_2) ← (⌈len/2⌉, ⌊len/2⌋);                    // Chunk sizes
3  (chunk_1, chunk_2) ← (&I, &I + len_1);                  // Chunks
4  (S_1, S_2) ← (start_state, start_state);               // Start states
5  history[len_1 − 1] ← error_state ;                     // Sentinel
   // Parallel processing stage
6  for i = 0 to len_2 − 1 do
7      forall the k ∈ {1, 2} do in parallel
8          c_k ← chunk_k[i];
9          S_k ← DFA[S_k][c_k];
10         if accepting(S_k) then
11             return MatchFound ;
12     history[len_1 + i] ← S_2 ;                          // On Secondary
13     i ← i + 1;
   // Validation stage (on Primary)
14 while i < len do
15     c_1 ← I[i];
16     S_1 ← DFA[S_1][c_1];
17     if accepting(S_1) then
18         return MatchFound ;
19     if S_1 == history[i] then
20         break ;
21     i ← i + 1;
22 return NoMatch ;          // Primary finished processing
```

Algorithm 2: Parallel SPPM with two chunks. Accepts PREs.

single-threaded software and for simple parallel hardware. Section V shows variants that are not constrained by the simplifying assumptions.

A. Basic SPPM Algorithm

Algorithm 2 shows the pseudocode for the informal example from Sect. III-A. The algorithm processes the input in three stages.

During the **initialization stage** (lines 1-5), the input is divided into two chunks and the state variables for the Primary and Secondary are initialized. During the **parallel processing stage** (lines 6-13), both processors scan their chunks in lockstep. If either the Primary or the Secondary reach an accepting state (line 10), we declare a match and finish the algorithm (line 11). The Secondary records (line 12) the states it visits in the history buffer (for simplicity, the history buffer is as large as the input, but only its second half is actually used). During the **validation stage** (lines 14-21), the Primary continues processing the Secondary's chunk. It still must check for accepting states as it may see a different sequence of states than the Secondary. There are three possible outcomes: a match is found and the algorithm returns success (line 18), coupling occurs before the end of the second chunk (line 20) or the entire second chunk is traversed again. If the input has an odd number of bytes, the first chunk is one byte longer, and a sentinel is setup at line 5 such that the validation step will ignore it.

Correctness of Algorithm 2: If during the parallel processing stage the Secondary reaches the **return** at line 11, then the Secondary found a match on its chunk. Since our assumption is that we search for a prefix-closed regular expression, a match in the second chunk guarantees a match on the entire input. Therefore it is safe to return with a match.

If the algorithm executes the **break** at line 20, then the Primary reaches a state also reached by the Secondary. Since the behavior of a DFA depends only on the current state and the rest of the input, we know that if the Primary would continue searching, from that point on it would redundantly follow the steps of the Secondary which did not find a match, so it is safe to break the loop and return without a match.

In all the other cases, the algorithm acts like an instance of Algorithm 1 performed by the Primary where the existence of the Secondary can be ignored.
To conclude, Algorithm 2 reports a match if and only if the input contains one.

Simplified performance models: Our evaluation of SPPM includes actual measurements of performance improvements on single-threaded software platforms. But to understand the performance gains possible through speculation and to estimate the performance for parallel platforms with different bottlenecks we use a simplified model of performance. Because the input and the history buffer are small (1.5KB for a maximumsized packet) and are accessed sequentially they should fit in fast memory (cache) and we do not account for accesses to them. We focus our discussion and our performance model on the accesses to the DFA table. Figure 4 summarizes the relevant metrics.

We use the number of steps (iterations) in the parallel processing, $|I|/2$, and in the validation stage, V, to approximate the *processing latency*: $L = \frac{|I|}{2} + V$.

Each of these iterations contains one access to the DFA table. The latency of processing an input I with the traditional matching algorithm (Algorithm 1) would be $|I|$ steps, hence we define the *speedup* (latency reduction) as $S = \frac{|I|}{L} = \frac{|I|}{|I|/2+V} = \frac{2}{1+2V/|I|}$.

The *useful work* performed by the parallel algorithm is scanning the entire input, therefore equivalent to $|I|$ serial steps. This is achieved by using $N = 2$ processing units (PUs), the Primary and Secondary, for a duration of L parallel steps. Thus, the amount of processing resources used (assuming synchronization between PUs), the *processing cost* is $P = N \cdot L$ and we define the *processing efficiency* as $P_e = \frac{useful\ work}{processing\ cost} = \frac{|I|}{N \cdot L} = \frac{|I|}{2\cdot(|I|/2+V)} = \frac{1}{1+2V/|I|}$.

Another potential limiting factor for system performance is memory throughput: the number of memory accesses that can be performed during unit time. We define *memory cost*, M, as the number of accesses to the DFA data structure by all PUs, $M = |I| + V$. Note that $M \leq N \cdot L$ as during the validation stage the Secondary does not perform memory accesses. We define *memory efficiency* as $M_e = \frac{|I|}{M} = \frac{|I|}{|I|+V} = \frac{1}{1+V/|I|}$ and it reflects the ratio between the throughput achievable by running the reference algorithm in parallel on many packets and the throughput we achieve using speculation. Both P_e and M_e can be used to characterize system throughput: P_e is appropriate when tight synchronization between the PUs is enforced (e.g. SIMD architectures) and the processing capacity is the limiting factor, M_e is relevant when memory throughput is the limiting factor.

Performance of Algorithm 2: In the worst case, no match is found, and coupling between Primary and Secondary does not happen ($V = |I|/2$). In this case the Primary follows a traditional search of the input and all the actions of the Secondary are overhead. We get $L = |I|$, $S = 1$, $P_e = 50\%$, $M = 1.5|I|$, and $M_e = 67\%$. In practice, because the work during the iterations is slightly more complex than for the reference algorithm (the secondary updates the history), we can even get a small slowdown, but the latency cannot be much lower than that of the reference algorithm.

In the common case, no match occurs and $V \lll |I|/2$. We have $S = \frac{2}{1+2V/|I|}$, $P_e = \frac{1}{1+2V/|I|}$, $M = |I| + V/|I|$, and $M_e = \frac{1}{1+V/|I|}$, where $V/|I| \ll 1$. Thus the latency is typically close to half the latency of the reference implementation and the throughput achieved is very close to that achievable by just running the reference implementation in parallel on separate packets.

In the uncommon case where matches are found, the latency is the same as for the reference implementation if the match is found by the Primary. If the match is found by the Secondary, the speedup can be much larger than 2.

```
   Input: DFA = the transition table
   Input: I = the input string
   Output: Does the input match the DFA?
   // Initialization as in Algorithm 2
1  ...
6  for i = 0 to $len_2 - 1$ do
7      $c_1 \leftarrow chunk_1[i]$;
8      $c_2 \leftarrow chunk_2[i]$;
9      $S_1 \leftarrow DFA[S_1][c_1]$;
10     $S_2 \leftarrow DFA[S_2][c_2]$;
11     if accepting($S_1$)||accepting($S_2$) then
12         return MatchFound ;
13     $history[len_1 + i] \leftarrow S_2$;
14     $i \leftarrow i + 1$;
   // Validation as in Algorithm 2
15 ...
```

Algorithm 3: Single-threaded SPPM with two chunks. Accepts PREs.

B. SPPM for Single-threaded Software

Algorithm 3 shows how to apply SPPM for single-threaded software. We simply rewrite the parallel part of Algorithm 2 in a serial fashion with the two table accesses placed one after the other. Except for this serialization, everything else is as in Algorithm 2 and we omit showing the common parts. The duration of one step (lines 6-14) increases and the number of steps decreases as compared to Algorithm 1. The two memory accesses at lines 9-10 can overlap in time, so the duration of a step increases but does not double. If the validation region is small, the number of steps is little over half the original number of steps. The reduction in the number of steps depends only on the input and on the DFA whereas the increase in the duration of a step also depends on the specific hardware (processor and memory). Our measurements show that speculation leads to an overall reduction in processing time and the magnitude of the reduction depends on the platform. The more instructions the processor can execute during a memory access, the larger the benefit of speculation.

This algorithm can be generalized to work with $N > 2$ chunks, but the number of variables increases (e.g. a separate state variable needs to be kept for each chunk). If the number of variables increases beyond what can fit in the processor's registers, the overall result is a slowdown. We implemented a single-threaded SPPM algorithm with 3 chunks, but, on the platforms we evaluated, its performance was not satisfactory, so we only report results for the 2-chunk version.

C. SPPM for Parallel Hardware

Algorithm 4 generalizes Algorithm 2 for the case where N PUs work in parallel on N chunks of the input. We present this unoptimized version due to its simplicity.

Lines 2-5 initialize the PUs. They all start parsing from the initial state of the DFA. They are assigned starting positions evenly distributed in the input buffer: PU_k starts scanning at position $\lfloor (k-1) * |I|/N \rfloor$. During the **parallel processing stage** (lines 6-13) all PUs perform the traditional DFA processing for their chunks and record the states traversed in history (this is redundant for PU_1). The first $N-1$ PUs participate

```
Input: DFA = the transition table
Input: I = the input string (|I| =input length)
Output: Does the input match the DFA?
1  len ← |I|;
2  forall the $PU_k$, k ∈ $\{1..N\}$ do in parallel
3      index$_k$ ← start position of k-th chunk;
4      state$_k$ ← start_state;
5  history$[0..len-1]$ ← error_state;                    // sentinel
   // Parallel processing stage
6  while index$_1 < \lfloor |I|/N \rfloor$ do
7      forall the $PU_k$, k ∈ $\{1..N\}$ do in parallel
8          input$_k$ ← I[index$_k$];
9          state$_k$ ← DFA[state$_k$][input$_k$];
10         if accepting(state$_k$) then
11             return MatchFound ;
12         history[index$_k$] = state$_k$;
13         index$_k$ ← index$_k + 1$;
14 forall $PU_k$, k ∈ $\{1..N-1\}$ do in parallel active$_k$ ← true ;
15 while there are active PUs do
16     forall the $PU_k$ such that (active$_k$ == true) do in parallel
17         input$_k$ ← I[index$_k$];
18         state$_k$ ← DFA[state$_k$][input$_k$];
19         if accepting(state$_k$) then
20             return MatchFound ;
21         if history[index$_k$] == state$_k$ OR index$_k$ == len $-$ 1
           then
22             active$_k$ ← false;
23         else
24             history[index$_k$] = state$_k$;
25             index$_k$ ← index$_k + 1$;
26 return NoMatch ;
```

Algorithm 4: SPPM procedure for matching PREs with N processing units.

in the **validation stage** (lines 14-25). A PU stops (becomes inactive) when *coupling* with the right neighbor happens, or when it reaches the end of the input. Active PUs perform all actions performed during normal processing (including updating the history).

The algorithm ends when all PUs become inactive.

Linear History Is Relatively Optimal: Algorithm 4 uses a linear history: for each position in the input, exactly one state is remembered – the state saved by the most recent PU that scanned that position. Thus PU_k sees the states saved by PU_{k+1}, which overwrite the states saved by $PU_{k+2}, PU_{k+3}, ..., PU_N$.

Since we want a PU to stop as soon as possible, a natural question arises: would PU_k have a better chance of *coupling* if it checked the states for *all* of $PU_{k+1}, PU_{k+2}, ..., PU_N$ instead of just PU_{k+1}? Would a 2-dimensional history that saves the set of all the states obtained by preceding PUs at a position offer better information than a linear history that saves only the most recent state? In what follows we show that the answer is *no*: the most recent state is also the most accurate one. If for a certain input position, PU_k agrees with any of $PU_{k+1}, PU_{k+2}, ..., PU_N$ then PU_k must also agree with PU_{k+1} at that position. We obtain this by substituting in the following theorem $chunk_k$ for w_1, the concatenation of chunks $k+1$ to $k+j-1$ for w_2 and any prefix of $chunk_{k+j}$ for w_3. We use the notation w_1w_2 to represent the concatenation of strings w_1 and w_2; and $\delta(S, w)$ to denote the state reached by the DFA starting from state S and transitioning for each character in string w.

Theorem 1 (monotony of PRE parsing): Assume that DFA is the minimized deterministic finite automaton accepting a prefix-closed regular expression, with S_0 = the start state of the DFA. For any w_1, w_2, w_3 input strings we have: $\delta(S_0, w_1w_2w_3) = \delta(S_0, w_3) \Rightarrow \delta(S_0, w_1w_2w_3) = \delta(S_0, w_2w_3)$.

Proof: Let $S_1 = \delta(S_0, w_1w_2w_3) = \delta(S_0, w_3)$ and $S_2 = \delta(S_0, w_2w_3)$. Assume, by contradiction, that $S_1 \neq S_2$. Since DFA is minimal, there must be a string w such that only one of $\delta(S_1, w)$ and $\delta(S_2, w)$ is an accepting state and the other one is not.
Assume L = the language accepted by the DFA.
We have two cases:

1) $\delta(S_1, w)$ accepting and $\delta(S_2, w)$ is not. Since $\delta(S_1, w) = \delta(\delta(S_0, w_3), w) = \delta(S_0, w_3w)$ we have $\delta(S_1, w)$ accepting $\Rightarrow$ $\delta(S_0, w_3w)$ accepting. Hence $w_3w \in L$. Since L is prefix closed, $w_3w \in L \Rightarrow w_2w_3w \in L \Rightarrow \delta(S_0, w_2w_3w)$ accepting. But $\delta(S_0, w_2w_3w) = \delta(\delta(S_0, w_2w_3), w) = \delta(S_2, w)$. Therefore $\delta(S_2, w)$ is accepting, which is a contradiction.
2) $\delta(S_2, w)$ is accepting and $\delta(S_1, w)$ is not. Then $\delta(S_2, w) = \delta(\delta(S_0, w_2w_3), w) = \delta(S_0, w_2w_3w)$ is accepting. Hence $w_2w_3w \in L$. Since L is prefix closed, $w_2w_3w \in L \Rightarrow w_1w_2w_3w \in L$. We have $w_1w_2w_3w \in L \Leftrightarrow \delta(S_0, w_1w_2w_3w)$ is accepting. But, $\delta(S_0, w_1w_2w_3w) = \delta(\delta(S_0, w_1w_2w_3), w) = \delta(S_1, w)$. Therefore $\delta(S_1, w)$ is accepting, which is also a contradiction.

Both cases lead to contradiction, so our assumption was wrong and $S_1 = S_2$. ∎

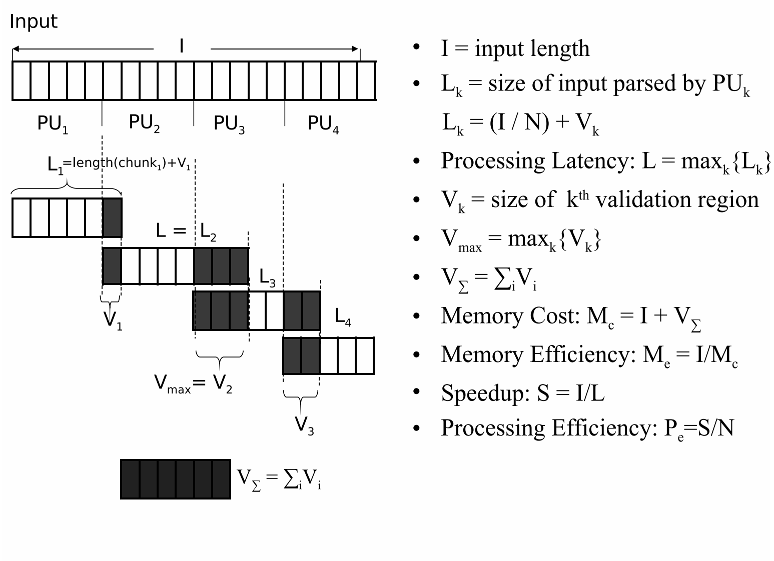
Performance of Algorithm 4:

For validation region k, we define V_k as the portion of the packet processed by PU_k during validation, so it can go beyond the end of chunk $k+1$. Let V_k be the length of the validation region k, $V_{max} = \max_{k=1}^{N} V_k$, and $V_\Sigma = \sum_{k=1}^{N} V_k$.

We get the following performance metrics (see Fig.5) :

$$\begin{array}{ll} \text{processing latency} & L = \frac{|I|}{N} + V_{max} \\ \text{speedup} & S = \frac{|I|}{L} = \frac{N}{1+N \cdot V_{max}/|I|} \\ \text{processing cost} & P = N \cdot L \\ \text{processing efficiency} & P_e = \frac{|I|}{P} = \frac{1}{1+N \cdot V_{max}/|I|} \\ \text{memory cost} & M = |I| + V_\Sigma \\ \text{memory efficiency} & M_e = \frac{|I|}{M} = \frac{1}{1+V_\Sigma/|I|} \end{array} \qquad (1)$$

In the worst case (no coupling for any of the chunks) $V_k = |I| - k|I|/N$ (ignoring rounding effects), $V_{max} = |I|(1-1/N)$ and $V_\Sigma = (N-1)|I|/2$ which results in a latency of $L = |I|$ (no speedup, but no slowdown either), a processing efficiency of $P_e = 1/N$, and a memory efficiency of $M_e \approx 2/N$. Note that the processing efficiency and the memory efficiency do not need to be tightly coupled. For example if there is no coupling for the first chunk, but coupling happens fast for the others, the latency is still $L = |I|$ and thus $P_e = 1/N$, but



- I = input length
- L$_k$ = size of input parsed by PU$_k$
  L$_k$ = (I / N) + V$_k$
- Processing Latency: L = max$_k$\{L$_k$\}
- V$_k$ = size of kth validation region
- V$_{max}$ = max$_k$\{V$_k$\}
- V$_\Sigma$ = $\sum_i$V$_i$
- Memory Cost: M$_c$ = I + V$_\Sigma$
- Memory Efficiency: M$_e$ = I/M$_c$
- Speedup: S = I/L
- Processing Efficiency: P$_e$=S/N

Fig. 5. Performance Metrics for one packet with I payload bytes.

$M_e \approx 50\%$ as most of the input is processed twice. But our experiments show that for N below 100, the validation regions are typically much smaller than the chunks and the speedups we get are on the order of $S \approx N$ and efficiencies are high.

We note here that SPPM always achieves efficiencies of less than 100% on systems using parallel hardware: within our model, the ideal throughput one can obtain by having the PUs work on multiple packet in parallel is always slightly higher than with SPPM. The benefit of SPPM is that the latency of processing a single packet decreases significantly. This can help reduce the size of buffers needed for packets (or the fraction of the cache used to hold them) and may reduce the overall latency of the IPSs which may be important for traffic with tight service quality requirements. Furthermore systems using SPPM can break the workload into fixed-size chunks as opposed to variable-sized packets which simplifies scheduling in tightly coupled SIMD architectures where the processing cost is determined by the size of the largest packet (or chunk) in the batch. This can ultimately improve throughput as there is no need to batch together packets of different sizes. Due to the complexity of parallel hardware in IPSs with, the performance depends on the specifics of the system beyond those captured by our model whether SPPM, simple parallelization, or a mix of the two is the best way to achieve good performance.

V. Relaxing the Assumptions

A. *Anchored Regular Expressions*

Algorithm 4 requires signatures that are PRE in order to avoid false matches. This raises the issue of what to do with the remaining signatures which are *anchored*. There are three options: (1) treat them separately; or (2) devise a new algorithm (Sect. V-B) ; or (3) mix them with the prefix closed signatures and use Algorithm 4 with the cost of false positives. In this section we give an argument for the first option. If we partition the signature set into two sets, one containing only PRE and the other containing only anchored expressions, then for the PRE subset we can use Algorithm 4, and for the anchored subset a very fast matching algorithm based on rejecting states.

Algorithm 1 only checks for accepting states, so it needs to scan the entire input to declare that there is no match. However if the expression is anchored, usually only part of the input needs to be scanned. Consider the anchored regular expression `VIRUS`, which matches the string `VIRUS` only at the beginning of the input, in contrast with `.*VIRUS.*`. For `VIRUS` we only need to scan the first five characters in the input to tell if a match occurs or not. This relies on *rejecting states* which are states that are not accepting and have all transitions back to themselves. Once a DFA enters a rejecting state, it cannot exit it and therefore it cannot reach an accepting state. A minimized DFA has at most one rejecting state, which makes checking for it easy. Not every DFA has such states but the DFA corresponding to a set containing only anchored expressions is likely to have one.

Lemma 1 (existence of REJECTING states): If a DFA (not necessarily minimized) has rejecting states then the language accepted by the DFA contains no subset that is prefix closed.

Note that the reciprocal is not true (consider a DFA that accepts even length strings).

Proof: Let S_0 be the starting state and S_1 a rejecting state. Then there must be a string w such that $S_1 = \delta(S_0, w)$. Assume, by contradiction, that the DFA accepts Y, a prefix closed string set. Let $x \in Y$. Y is prefix closed $\Rightarrow wx \in Y \Rightarrow \delta(S_0, wx)$ is an accepting state. But $\delta(S_0, wx) = \delta(\delta(S_0, w), x) = \delta(S_1, x) = S_1$, because S_1 is a rejecting state without any outgoing transitions. Hence S_1 should be both a rejecting state and accepting, which is a contradiction. ■

Expressions such as `^VIRUS` match `VIRUS` at the beginning of the input or after a new line. `^VIRUS` can be separated into `VIRUS` which is anchored and `.*\x11VIRUS` which is prefix closed (`\x11` stands for newline). This distinction can

```
Input: DFA = the transition table
Input: I = the input string
Result: Does the input match the DFA?
state ← start_state;
for i = 0 to $length$(I) do
    input_char ← I[i];
    state ← DFA[state][input_char];
    if accepting(state) then
        return true ;
    if rejecting(state) then
        return false ;
return false ;
```

Algorithm 5: DFA matching anchored-only expressions which have a rejecting state

be performed for all signatures. Thus, the language described by the entire signature set is the union of the languages for two disjoint sets of signatures: one containing only anchored expressions, and one containing only PRE. For the later SPPM can be used. For the former, we can use Algorithm 5 which is Algorithm 1 modified to check for both accepting and rejecting states.

For the set of anchored signatures extracted from Snort, Algorithm 5 outperforms the traditional DFA algorithm by orders of magnitude. Such speedups require the existence of rejecting states and according to Lemma 1 this requires a separation between anchored and prefix closed expressions.

B. General Case: Matching General Regular Expressions (GRE)

The most general case is when the IPS uses unrestricted, general regular expressions (GRE) and it requires an ordered list of all matches. In this case, we must change the way Algorithm 4 handles matches.

The basic SPPM algorithms require prefix-closed expressions only because Secondaries are allowed to safely report a match if they reach an accepting state. For non-PRE such as `.*ok|bad`, the matches found by Secondaries (which start processing from the start state of the DFA) may be false matches such as in the case when the string `bad` occurs at the beginning of the second chunk, not at the beginning of the input. The SPPM version described in Algorithm 6 avoids this problem.

The separation of *parallel stage* and *validation stage* into separate loops in Algorithm 4 was meant for ease of understanding, but the two loops can be combined. This is the format used in Algorithm 6 which generalizes Algorithm 4 to handle GRE. The main difference is that matches are not reported immediately. Instead, a global flag, $potential\_match$, records that a potential match was found and scanning is continued.

Claim 1 (Invariant:same trace as a traditional DFA): When line 18 is reached each input byte was processed and the corresponding position in the history buffer holds the same state as that obtained by the traditional DFA algorithm after processing that position. Hence the history buffer contains exactly the same sequence (trace) of states that the traditional DFA matching would have produced (using the same input and transition table).

```
   Input: DFA = the transition table
   Input: I = the input string (|I| =input length)
   Input: STATES = starting states (only STATES[1]
          must be start_state)
   Output: Does the input match the DFA?
1  len ← |I|;
2  forall the $PU_k$, k ∈ {1..$N$} do in parallel
3      index$_k$ ← start position of k-th chunk;
4      state$_k$ ← STATES[$k$];
5      active$_k$ ← (index$_k$ < |I|);
6  history[0..$len - 1$] ← error_state;                // sentinel
7  potential_match ← false;
8  forall the $PU_k$ such that (active$_k$ == true) do in parallel
9      input$_k$ ← I[index$_k$];
10     state$_k$ ← DFA[state$_k$][input$_k$];
11     if accepting(state$_k$) then
12         potential_match ← true;           // but keep going
13     if history[index$_k$] == state$_k$ OR index$_k$ == len − 1 then
14         active$_k$ ← false;
15     else
16         history[index$_k$] = state$_k$;
17         index$_k$ ← index$_k$ + 1;
18 if potential_match == true then
       // history contains only valid states! In
          fact, the trace of a traditional DFA
19     index$_k$ ← start position of k-th chunk;
20     forall the $PU_k$, k ∈ {1..$N$} do in parallel
21         while index$_k$ < end position of k-th chunk do
22             if accepting(history[index$_k$]) then
23                 return MatchFound ;
24             index$_k$ ← index$_k$ + 1;
25 return NoMatch ;
```

Algorithm 6: SPPM with N processing Units (PUs). Matches GREs. The initial state for non-Primary PUs can be any state.

Proof: The initial division of the input in at most N chunks covers the entire input. Therefore for each input byte there is a chunk covering it. Consider the $i-$th input byte and let $chunk_k$ be the chunk containing it. PU_k becomes inactive only when one of two conditions are satisfied: PU_k reaches the end of the input, or PU_k couples with some PU_{k+l} (remember that $history$ is initialized to hold an invalid state as a sentinel in all positions). But the end of the input is at or after position i and coupling with PU_{k+l} can only happen after the starting position of $chunk_{k+l}$. Hence, position i is processed at least once. We use induction on i to prove that the resulting value of $history[i]$ is the same state as that obtained by the traditional DFA algorithm. This is clearly true for $i = 0$ since it falls in the Primary chunk. Assume that the property holds for $\forall i < n$ and we'll prove that it also holds for n. Let PU_p be the last PU that processed position n (there is at least one such PU according to the first half of this claim). If $p = 1$ then PU_p is the Primary which starts in the same initial state as the traditional DFA and obviously traverses the same sequence of states while active. If $p > 1$, then $\forall j < p : PU_j$ must have stopped before position n. Let m be the largest position processed by any $PU_j, j < p$. Since PU_{p-1} cannot stop before the beginning of $chunk_p$, it follows that position m was also processed by PU_p. Since the last PU to visit m did not visit the next position $m+1$, it follows that the last visitor of m coupled with PU_p at position m. By induction hypothesis $history[m]$ is the same as for a traditional DFA. Since $history[m]$ was the state produced by PU_p it follows that all states produced by PU_p after position m, position n included, are the same as for a traditional DFA. Hence the property holds for position n which concludes the induction proof. ■

After scanning the entire input, at line 18, we must decide if any potential matches are indeed real matches. For this, we simply look at the states saved in the history and report the accepting ones. This is sound according to Claim 1. The common case in IPSs is that no matches are found so the overhead for the extra bookkeeping is incurred only for a small fraction of the packets.

Algorithm 6 addresses two additional issues: flexibility in the choice of secondary starting states and matching semantics:
Flexibility in Secondary Starting States: The starting state for a Secondary no longer has to be the same as the initial state of the DFA. This allows for the choice of other convenient state such as the most frequent one, which in the presence of anchored expressions might not be the initial state.
Flexibility in Matching Semantics: The basic matching algorithm is often extended to return more information than just whether a match occurred or not: the offset within the input where the accepting state has been reached and/or the signature number for that matched (a single DFA typically tracks multiple signatures). Furthermore, multiple matches may exist as the reference algorithm may visit accepting states more than once. For example if one recognizes the two signatures `.*day` and `.*week` with a single DFA and the input is `This week on Monday night!`, we have a match for the second signature at the end of the second word and one for the first signature at the end of the fourth word. Since the $history$ buffer contains the same trace as that of a serial DFA, one can get either the first match or all matches by changing the search order at lines 20-24. Algorithm 6 can return any information about the matches that the traditional algorithm can.

C. Bounding the Validation Region

In the worst case speculation fails and the whole input is traversed sequentially. There is nothing we can do to guarantee a worst case latency smaller than I and equivalently a processing efficiency of more than $1/N$. But we can ensure that the memory efficiency is larger than $2/N$ which corresponds to the case where all PUs traverse the input to the end. We can limit the size of each validation region to V positions, and stop the validation stage for all PUs other than the primary when they reach that limit, as shown in Algorithm 7. If V is large enough convergence may still happen (see Sec.VI-F), but we bound the number of memory accesses performed during the validation stage to $(N-2)V$ for the $N-2$ non-primary PUs doing validation and $|I|-|I|/N$ for the primary. Thus $M \le |I|(2-1/N)+(N-2)V < 2|I|+NV$ and $M_e > 1/(2+NV/|I|)$.

```
1  ..... Input: VMAX = the maximum validation size.....
2  len ← |I|;
3  forall the $PU_k$, k ∈ $\{2..N-1\}$ do in parallel
4      end$_k$ ← $MIN($index$_{k+1}$ + VMAX $+ 1, len)$;
5  end$_1$ ← end$_N$ ← $len$;
   // ... other initialization ...
6  forall the $PU_k$ such that (active$_k$ == true) do in parallel
7      input$_k$ ← I[index$_k$];
8      state$_k$ ← DFA[state$_k$][input$_k$];
9      if accepting(state$_k$) then
10         potential_match ← true;        // but keep going
11     if history[index$_k$] == state$_k$ then
12         active$_k$ ← false;
13     else
14         history[index$_k$] = state$_k$;
15         index$_k$ ← index$_k$ + 1;
16         if index$_k$ == end$_k$ then
17             active$_k$ ← false;
18 ok_limit ← len;
19 forall the k, k ∈ $\{1..N-1\}$ do
20     if index$_k$ ≥ end$_k$ then
21         ok_limit ← end$_k$;
22         state ← state$_k$;
23         for i = ok_limit to len − 1 do
24             c ← I[i];
25             state ← DFA[state][c];
26             if accepting(state) then
27                 return MatchFound ;
28 if potential_match == true then
29     for i = 0 to ok_limit − 1 do
30         c ← I[i];
31         state ← DFA[state][c];
32         if accepting(state) then
33             return MatchFound ;
34 return NoMatch ;
```

Algorithm 7: SPPM with N PUs, using bounded validation region.Matches PRE.

VI. Experimental Evaluation

We compared results using SPPM against the traditional DFA method. There are many more variations of SPPM than we can cover here. But the cases that we do cover show that SPPM has very good potential for massive parallelization of pattern matching. The simple, single threaded version can achieve speedups of 40%, and these are larger on faster CPUs. Simulation of parallel SPPM show that the speedup can be almost linear in $N =$ the number of CPUs, even for values of N as large as 50. This is because typically validation happens within a few bytes. The generalization of SPPM to handle arbitrary GRE (not just PRE) comes at a performance cost which depends on the number of matches in the input (almost free, if there are no matches). Bounding the validation region is a good option to guard memory efficiency.

A. Experimental Setup

Payload: As input we extracted the TCP payloads of 175,668 HTTP packets from a two-hour trace captured at the border router of our department. Figure 6 shows the cumulative distribution function (*cdf*) for packet lengths. The average length was 1052 bytes. The most frequent packet sizes were

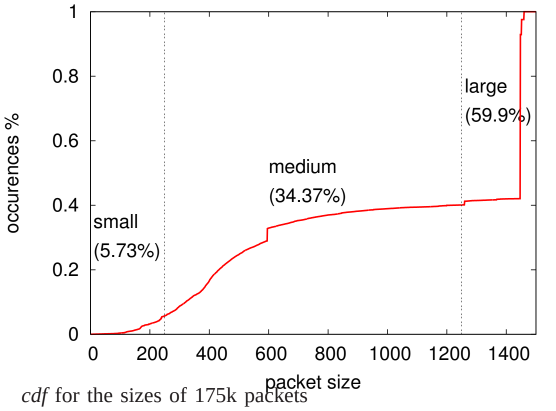


Fig. 6. *cdf* for the sizes of 175k packets

	Set1	Set2
Number of DFAs	106	1
Number of Signatures	639	811
Only PRE?	Yes	No
Number of States	217362	170265
Number of Matches	298	44060

Fig. 7. DFA sets used in experiments

1448 bytes (50.88%), 1452 bytes (4.62%) and 596 bytes (3.82%). Furthermore 5.73% of the packets were smaller than 250 bytes, 34.37% were between 251 and 1,250 and 59.90% were larger than 1,251.

Signatures: We used 1450 Snort HTTP signatures. Since a single DFA containing all signatures would not fit in the memory, an automated procedure inspired from [32] was used to divide them into 107 DFAs. These DFAs are grouped in two sets (see Fig.7):

- *Set1* contains 106 DFAs composed only of PREs.
- *Set2* a single DFA from a mixture of PRE and anchored signatures.

We treat *Set2* separately because it contains anchored signatures and it can be used with the basic PRE-only versions of SPPM only at the cost of reporting false matches. Nevertheless we still report results for this combination because it could be a valid design decision where false positives would later be discarded.

Match behavior: A common IPS behavior is to resume scanning after a match is handled and deigned non-malicious. This behavior can be approximated by always fully scanning the input. However, our default behavior is to return after the first match, as in Algorithm 1. In the few cases when we chose the alternative behavior of resumed scanning (for both the traditional algorithm and the speculative one), we explicitly state it. This only makes a difference for packets that contain a match.

B. Evaluation of Algorithm 3 (Single Threaded, Software Implementation)

We implemented four versions of Algorithm 3, the single threaded implementation which uses speculation to overlap memory accesses. Figure 10 shows the resulting speedups when compared to the traditional sequential algorithms[3]. The two "PRE" columns show the results for the basic version, intended to use only PRE as input. The two "GRE" columns show the results for the version modified to work with all GRE as described in Sect.V-B. For both versions we tried the

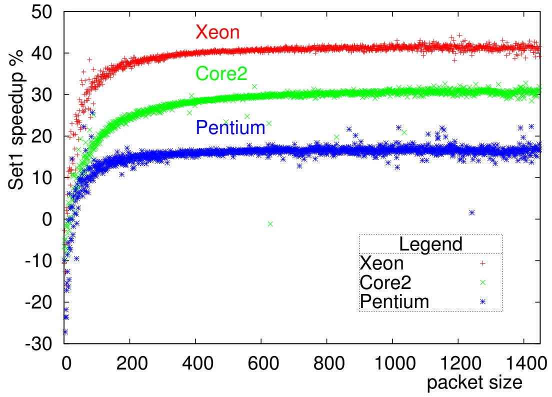


Fig. 8. Speedup of Alg. 3 (single threaded SPPM) over the sequential DFA Algorithm (on *Set1*)

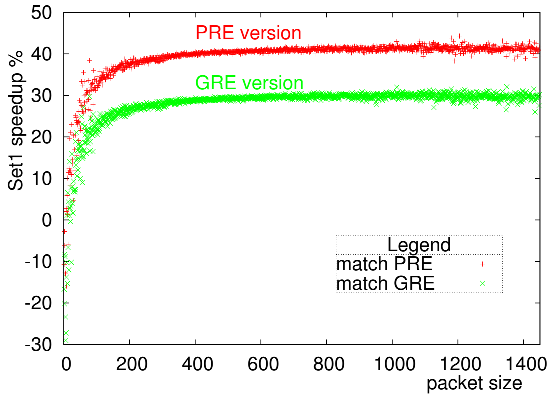


Fig. 9. Speedup of two variants of Alg. 3 (on Xeon/*Set1*): (1) matching only PRE, and (2) matching all GRE

(default) behavior which returns the first match and does not resume scanning, as well as the the behavior that resumes scanning after each match (distinction done by the "resume?" "YES/NO" columns in Fig. 10). We measured the actual running times using hardware performance counters and ran experiments on three architectures, a Pentium M at 1.5GHz, an Intel Core 2 at 2.4GHz and a Xeon E5520 at 2.27GHz. We explain the higher speedup on the more performing processors in Fig. 10 by the larger gap between the processor speed and the memory latency. Figure 8 shows how the packet size

[3] By using the corresponding behavior in terms of returning the first match or scanning the whole packet

		input: PRE		input: GRE	
	Sign.	resume?		resume?	
System	Set	NO	YES	NO	YES
Pentium	Set1	16.61	16.58	16.62	16.61
Pentium	Set2	21.27*	18.44*	-9.15	16.55
Core 2	Set1	30.61	36.65	16.79	26.18
Core 2	Set2	34.57*	32.79*	0.69	27.48
Xeon	Set1	41.41	44.45	29.96	34.58
Xeon	Set2	38.94	36.14*	5.02	29.65

Fig. 10. Speedup for versions of Alg.3 classified by the expected input (PRE vs. GRE), and by whether or not scanning is resumed after each match. Tests marked with '*' have 7 false positives.

Set	N	$\tilde{V}$	$\tilde{V}_{max}$	$V \leq 1$	$V \leq 2$	$V \leq 3$
Set1	2	4.81	4.81	98.51	98.82	98.88
Set1	10	2.12	8.24	99.26	99.58	99.64
Set1	50	1.24	9.09	99.50	99.84	99.89
Set1	100	1.12	9.29	99.54	99.87	99.93
Set2	2	45.97	45.97	47.27	75.67	81.75
Set2	10	15.99	86.33	51.78	83.51	90.29
Set2	50	4.78	96.56	53.23	85.85	92.70
Set2	100	3.26	98.50	53.52	86.27	93.16

Fig. 11. Validation region (V) statistics. N= Number of PUs. $\tilde{V}$ = average V size over all chunks. $\tilde{V}_{max}$ = average value of maximum validation size (V_{max}) in each packet. $V \leq K$ shows the percent of chunks for which validation occurs in K bytes.

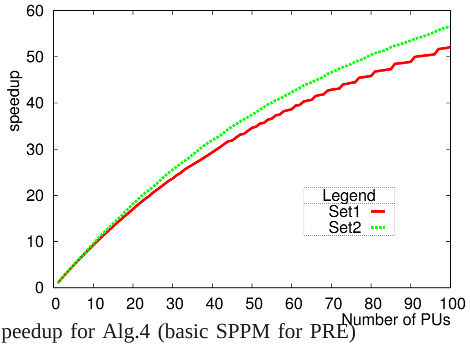


Fig. 12. Speedup for Alg.4 (basic SPPM for PRE)

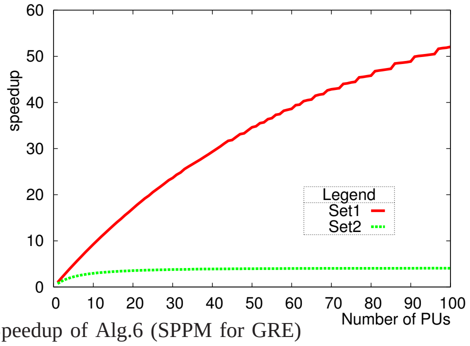


Fig. 13. Speedup of Alg.6 (SPPM for GRE)

influences the speedup for the PRE-only version of Algorithm 3 using *Set1*: for packets smaller than 20 bytes speculation may result in slowdowns. For packets larger than 150 bytes the speedup does not change significantly with the packet size. Each PRE-only version of the algorithm has a total of 7 false positives when used with on *Set2* (which is not PRE).

The generalization of Algorithm 3 to work with GRE comes at the cost of additional overhead. Figure 9 compares the speedup for the two versions by packet size and shows a clear decrease of about 10% for the general version, on the Xeon architecture. As discussed in Sect.VI-G this supports a model where anchored expressions are handled separrately and the basic PRE-only algorithm is used for the rest.

C. Evaluation of Algorithm 4 (Basic SPPM for Prefix Closed Regular Expressions) Using Simulation

We evaluated Algorithm 4 for up to $N = 100$ processing units. We used a simulation of parallel architecture and report speedups and efficiency based on our performance model which relies on the number of accesses to the DFA table (lines 9 and 18 of Algorithm 4). These metrics are described in Sect. IV-C by equations 1. Fig. 12 shows that speedup is almost linear up to $N = 20$ and it slowly diverges afterwords. The processing efficiency approaches 50% and the memory efficiency 90% by the time we reach $N = 100$ (see Figs. 14 and 16).

In [18] we show the influence of packet size on performance metrics. For space reasons we do not reproduce the graphs here, but as expected the algorithm performs better on larger packets. The greatest impact is observed for memory efficiency which degrades fast for small packets as N increases.

D. Validation Region

We found that the validation typically happens quickly. When $N = 10$ and all the DFAs in *Set1* are matched against the entire input, validation happens after a single byte for 99% of the chunks.

The scanning time for a packet is determined by V_{max} = the largest validation region in the packet (see Fig.5). Figure 11 shows for each signature set, and each $N = 2, 10, 50, 100$ the values for $\tilde{V}$ = average validation size over all chunks, and $\tilde{V}_{max}$ = average value of V_{max} over all packets. It also shows the percent of chunks for which validation happens within 1, 2 or 3 bytes. Figures 18 and 19 present the cumulative distributions for the sizes of the validation regions when $N = 10$. Figure 18 captures the sizes of all validation regions, which is relevant to memory efficiency. Figure 19 captures only the largest validation region for each packet, which is relevant to processing efficiency.

E. Evaluation of Algorithm 6 (SPPM for General Regular Expressions) Using Simulation

Figure 13 shows the speedup of Algorithm 6. On *Set1* it gets results almost identical to those for Algorithm 4. We explain this on the small number of matches. On *Set2* although Algorithm 6 exhibits speedups, it is greatly outperformed by Algorithm 4. Figures 15 and 17 show processor and memory efficiencies of Algorithm 6. As expected, on *Set2* these are lower than those for Algorithm 4.

F. Evaluation of Algorithm 7 (SPPM for PRE, with Bounded Validation Region) Using Simulation

We measured the performance of Algorithm 7 by limiting the validation region to various sizes. In theory, this cannot increase the processing efficiency (or the speedup). It can only improve the worst case for memory efficiency, and protect against certain algorithmic attacks. On our test data we

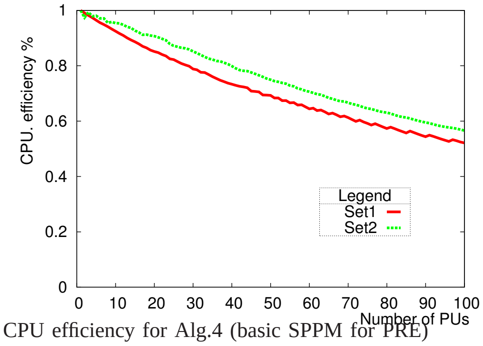


Fig. 14. CPU efficiency for Alg.4 (basic SPPM for PRE)

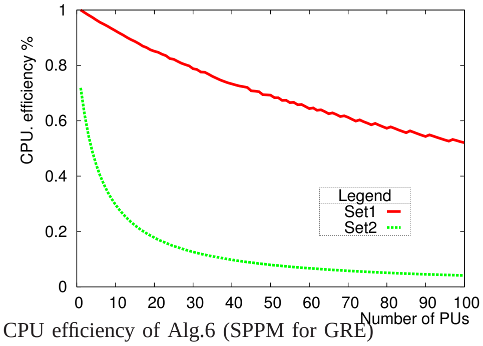


Fig. 15. CPU efficiency of Alg.6 (SPPM for GRE)

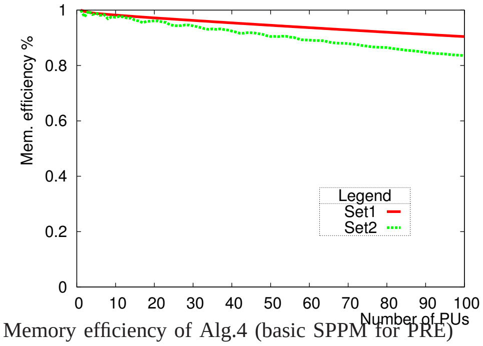


Fig. 16. Memory efficiency of Alg.4 (basic SPPM for PRE)

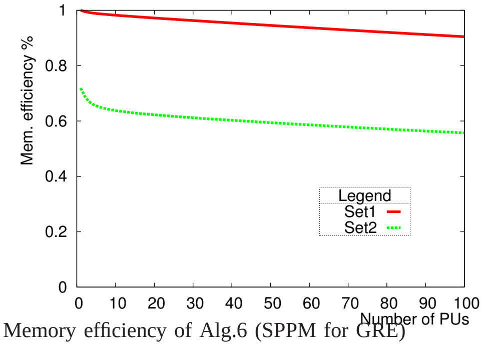


Fig. 17. Memory efficiency of Alg.6 (SPPM for GRE)

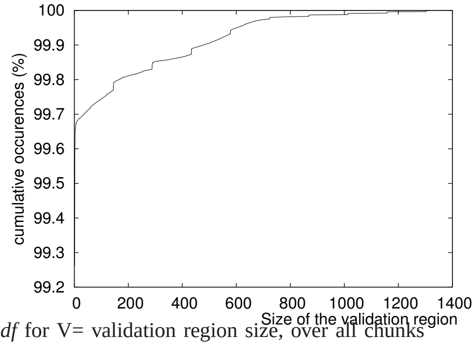


Fig. 18. *cdf* for V= validation region size, over all chunks

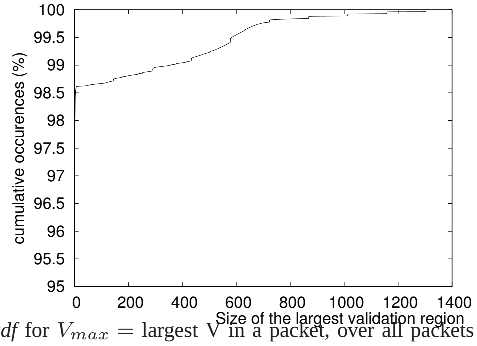


Fig. 19. *cdf* for V_{max} = largest V in a packet, over all packets

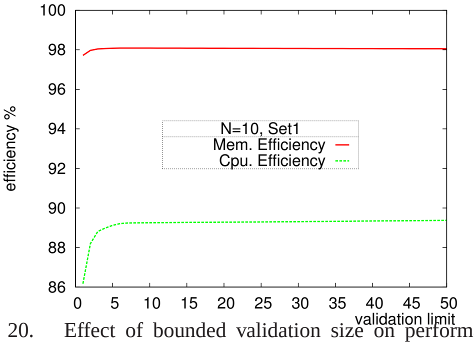


Fig. 20. Effect of bounded validation size on performance metrics for N=10

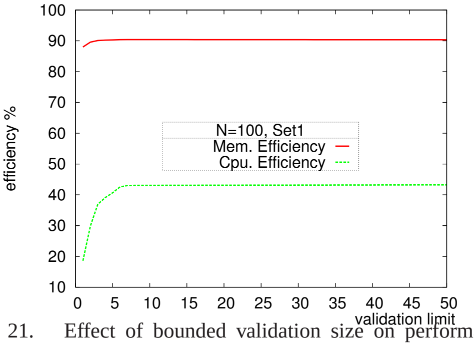


Fig. 21. Effect of bounded validation size on performance metrics for N=100

observed that if the limit is sufficiently large (about 10 bytes) than the memory and processor efficiency (and implicitly the speedup) are about the same as for unbounded memory (see Figs 20 and 21).

G. Anchored expressions and Rejecting States

We gathered all anchored signatures into one DFA as explained in Sect.V-A. We verified that the minimized DFA has indeed a rejecting state and then we scanned the input using Algorithm 5. Compared to the traditional algorithm, this version reduced the number of memory accesses by 99%. The actual matching time was reduced to 2%, that is a 50x speedup. Note that the difference between the PRE and GRE versions of SPPM can be much larger than 2% (see Figs. 9, 10 and *Set2* in Figs. 12 and 13). This supports the idea of partitioning the signatures in anchored and PRE, and handling them separately. This separation allows the selection of Algorithm 4 which outperforms Algorithm 6. Alternatively, Algorithm 4 could be used for all GRE with the risk of having false positives and having to handle them elsewhere. Because it is hard to quantify the computation done by an IPS when a possible match is reported, we do not explore this option further. Also, note that such false positives could also be used for algorithmic attacks to slow the IPS.

VII. Related Work

Signature matching is at the heart of intrusion prevention, but traditional matching methods have large memory footprints, slow matching times, or are vulnerable to evasion. Many techniques have been and continue to be proposed to address these weaknesses.

Early string-based signatures used multi-pattern matching algorithms such as Aho-Corasick [1] to efficiently match multiple strings against payloads. Many alternatives and enhancements to this paradigm have since been proposed [29], [8], [27], [17], [28]. With the rise of attack techniques involving evasion [20], [21], [10], [23] and mutation [13], though, string-based signatures have more limited use, and modern systems have moved to vulnerability-based signatures written as regular expressions [30], [6], [26], [22]. In principle, DFA-based regular expression matching yields high matching speeds, but combined DFAs often produce a state-space explosion [24] with infeasible memory requirements. Many techniques have been proposed to reduce the DFA state space [24], [25], or to perform edge compression [16], [3], [14], [9]. These techniques are orthogonal to our own, which focuses specifically on latency and can be readily applied to strings or regular expressions with or without alternative encoding.

Other work uses multi-byte matching to increase matching throughput. Clark and Schimmel [7] and Brodie *et al.* [5] both present designs for multi-byte matching in hardware. Becchi and Crowley [4] also consider multi-byte matching for various numbers of bytes, or *stride*, as they term it. These techniques increase throughput at the expense of changing DFA structure, and some form of edge compression is typically required to keep transition table memory to a reasonable size. Our work on the other hand reduces latency by subdividing a payload and matching the chunks in parallel without changing the underlying automaton. It would be interesting to apply speculative matching to multi-byte structured automata.

Kruegel *et al.* [15] propose a distributed intrusion detection scheme that divides the load across multiple sensors. Traffic is sliced at frame boundaries, and each slice is analyzed by a subset of the sensors. In contrast, our work subdivides individual packets or flows, speculatively matches each fragment in parallel, and relies on fast validation. Whereas Kruegel's work assumes individual, distinct network sensors, our work can benefit from the increasing availability of multicore, SIMD, and other n-way processing environments.

Parallel algorithms for regular expression and string matching have been developed and studied outside of the intrusion detection context. Hillis and Steele [11] show that an input of size n can be matched in $\Omega(log(n))$ steps given $n*a$ processors, where a is the alphabet size. Their algorithm handles arbitrary regular expressions but was intended for Connection Machines-style architectures with massive numbers of available processors. Similarly, Misra [19] derives an $O(log(n))$-time string matching algorithm using $O(n*length(string))$ processors. Again, the resulting algorithm requires a large number of processors.

Many techniques have been proposed that use Ternary Content addressable Memories (TCAMs). Alicherry *et al.* [2] propose a TCAM-based multi-byte string matching algorithm. Yu *et al.* [33] propose a TCAM-based scheme for matching simple regular expressions or strings. Weinsberg *et al.* [31] introduces the Rotating TCAM (RTCAM), which uses shifted patterns to increase matching speeds further. In all TCAM approaches, pattern lengths are limited to TCAM width and the complexity of acceptable regular expressions is greatly limited. TCAMs do provide fast lookup, but they are expensive, power-hungry, and have restrictive limits on pattern complexity that must be accommodated in software. Our approach is not constrained by the limits of TCAM hardware and can handle regular expressions of arbitrary complexity.

The work most closely related to ours is the parallel lexer from [12]. This was concurrent work with ours, which we were not aware at [18]. The core idea is similar to SPPM but their application domain is different: they use speculation to parallelize token detection. As opposed to SPPM they start the speculative matching a few bytes before the desired location, with the hope to reach a stable state by that point. In their case matches are frequent, and the language (tokens for some higher syntax) is simpler. We feel that speculation in [12] is justified by the fact that lexing is **memory-less** in the sense that the state at the beginning of a token is always the same no matter where parsing started, comments aside. Valid token beginnings are always coupling positions. In our case matches are infrequent. We give more insight on why the speculation works, prove that a linear history is efficient and give more implementation details and insight about coupling. Such insight is essential for the requirements of intrusion detection.

VIII. Conclusions

We presented speculative pattern matching method which is a powerful technique for low latency regular-expression matching. The method is based on three important observations. The first key insight is that the serial nature of the memory accesses is the main latency-bottleneck for a traditional DFA matching. The second observation is that a speculation that does not have to be right from the start can break this serialization. The third insight, which makes such a speculation possible, is that the DFA based scanning for the intrusion detection domain spends most of the time in a few hot states. Therefore guessing the state of the DFA at a certain position and matching from that point on has a very good chance that in a few steps will reach the "correct" state. Such guesses are later on validated using a history of speculated states. The payoff comes from the fact that in practice the validation succeeds in a few steps. A linear history is also essential for an efficient implementation of SPPM. It is also a key component for the ability to retrieve information about the matching states for arbitrary regular expressions, without sacrificing performance with excessive bookkeeping in the frequent case when matches are not found.

Our results predict that speculation-based parallel solutions can scale very well. Moreover, as opposed to other methods in the literature, our technique does not impose restrictions on the regular-expressions being matched. We believe that speculation is a very powerful idea and other applications of this technique may benefit in the context of intrusion detection.

Acknowledgment

We are thankful to the anonymous reviewers and to Mihai Marchidann for their constructive comments.

References

[1] A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, June 1975.

[2] M. Alicherry, M. Muthuprasannap, and V. Kumar. High speed pattern matching for network IDS/IPS. In *ICNP*, Nov. 2006.

[3] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS 2007*.

[4] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, December 2008.

[5] B. Brodie, R., and D. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, May 2006.

[7] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE FCCM*, Apr. 2004.

[8] S. Dharmapurikar and J. W. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Comm.*, 24(10):1781–1792, 2006.

[9] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro. An improved DFA for fast regular expression matching. *SIGCOMM Comput. Commun. Rev.*, 38(5):29–40, 2008.

[10] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Usenix Security*, Aug. 2001.

[11] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[12] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *HOTPAR*, 2009.

[13] M. Jordan. Dealing with metamorphism. *Virus Bulletin Weekly*, 2002.

[14] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Securecomm*, Sep 2008.

[15] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, May 2002.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM*, Sept. 2006.

[17] R. Liu, N. Huang, C. Chen, and C. Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.

[18] D. Luchaup, R. Smith, C. Estan, and S. Jha. Multi-byte regular expression matching with speculation. In *Recent Advances in Intrusion Detection: 12th International Symposium*, 2009.

[19] J. Misra. Derivation of a parallel string matching algorithm. *Information Processing Letters*, 85:255–260, 2003.

[20] V. Paxson. Defending against network IDS evasion. In *Recent Advances in Intrusion Detection (RAID)*, 1999.

[21] T. Ptacek and T. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, Jan. 1998.

[22] M. Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference*. USENIX, 1999.

[23] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *IEEE Symp. on Security and Privacy*, May 2003.

[24] R. Smith, C. Estan, and S. Jha. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, August 2008.

[25] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.

[26] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, Oct. 2003.

[27] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Int. Conf. on Field Programmable Logic and Applications*, sep. 2003.

[28] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA*, June 2005.

[29] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, pages 333–340.

[30] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.

[31] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker. High performance string matching algorithm for a network intrusion prevention system. In *High Performance Switching and Routing*, 2006.

[32] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of IEEE symposium on Architecture for networking and communications systems*, 2006.

[33] F. Yu, R. H. Katz, and T. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP*, pages 174–183, 2004.