Multi-Byte Regular Expression Matching with Speculation

Daniel Luchaup[1] Randy Smith[1] Cristian Estan[2⋆] Somesh Jha[1]

[1] University of Wisconsin-Madison, {luchaup,smithr,jha}@cs.wisc.edu
[2] NetLogic Microsystems, estan@netlogicmicro.com

Abstract. Intrusion prevention systems determine whether incoming traffic matches a database of signatures, where each signature in the database represents an attack or a vulnerability. IPSs need to keep up with ever-increasing line speeds, which leads 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 for 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.

Key words: low latency, parallel pattern matching, regular expressions, speculative pattern matching, multi-byte, multi-byte matching

1 Introduction

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 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 investigating the problem of *multi-byte* matching, or the problem of IPS concurrently scanning multiple bytes of a packet. We present a novel speculation-based method for multi-byte matching.

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.

⋆ Work done while at University of Wisconsin-Madison

Speculation has been used in several areas of computer science, especially computer architecture. Our speculative 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. This idea opens the door for an entire new class of parallel multi-byte matching algorithms.

This paper 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. 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. Section 3 presents an overview of SPPM, with details given in Section 4. 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 24% improvement over the traditional matching procedure. We 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.

2 Background

2.1 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[3] that takes M cycles[4]. Ideally the processor would schedule other operations while waiting for the result of the read from memory, but in Algorithm

[3] Assuming that the two indexes are combined in a single offset in a linear array
[4] 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?
1  state ← start_state;
2  for i = 0 to |I| do
3  |  input_char ← I[i];
4  |  state ← DFA[state][input_char];
5  |  if accepting(state) then
6  |  |  return MatchFound ;
7  |  end
8  end
9  return NoMatch ;
```

Algorithm 1: Traditional DFA matching.

1 each iteration is data-dependent on the previous one: the algorithm cannot proceed with the 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.

2.2 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.

3 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 also prefix closed, a restriction that will be lifted in Sec. 4.4. 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.

3.1 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 states 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

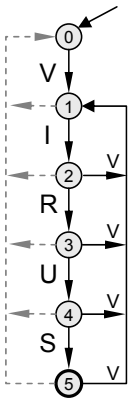

Fig. 1. DFA for `.*VIRUS`; dotted lines show transitions taken when no other transitions apply.

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								$2 \neq 0$,	0	0	0	0	0	0	0
step 10									$3 \neq 0$,	0	0	0	0	0	0
step 11										$4 \neq 0$,	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.

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 different 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.

3.2 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)

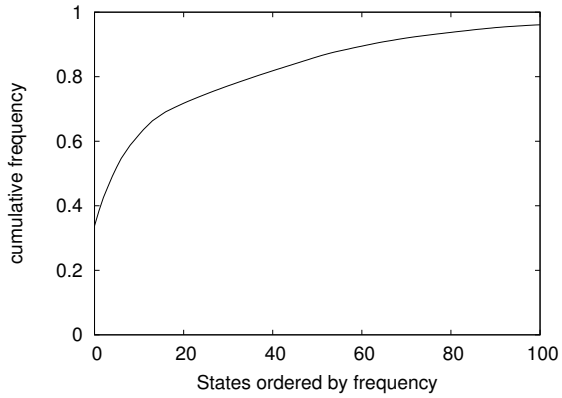

Fig. 3. The state frequency CDF graph for a PRE composed of 768 Snort signatures. Most of the scanning time is spent in a few hot states. The most frequent state accounts for 33.8% of the time and the first 6 most frequent states account for half the time.

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 doesn't 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 diminishes 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.

3.3 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.

4 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 3.1 and by introducing a simplified performance model for evaluating the benefits of speculation. After presenting basic SPPM algorithms for single-threaded software and for simple parallel hardware, we discuss variants that are not constrained by the simplifying assumptions. These generalized algorithms work with unconstrained regular expressions, return more information about the match, not just whether a match exists or not, and limit speculation to guarantee good worst-case performance.

4.1 Basic SPPM Algorithm

Algorithm 2 shows the pseudocode for the informal example from Sect. 3.1. 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

```
   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 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.

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 plat-

Table 1. Simplified performance model metrics (N is number of processors).

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

forms. 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 maximum-sized 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. Table 1 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.

```
   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 ← chunk_1[i];
 8     c_2 ← chunk_2[i];
 9     S_1 ← DFA[S_1][c_1];
10     S_2 ← DFA[S_2][c_2];
11     if accepting(S_1)||accepting(S_2) then
12         return MatchFound ;
13     history[len_1 + i] ← S_2;
14     i ← i + 1;
   // Validation as in Algorithm 2
15 ...
```

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

Performance of Algorithm 2: In the worst case, no match is found, and coupling between Primary and Secondary doesn't 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 $V \ll |I|/2$. 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.

4.2 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 since its performance is weaker on the platforms we evaluated, we only report results for the 2-chunk version.

4.3 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 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 $w_1 w_2$ 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.

```
   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 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 < ⌊|I|/N⌋ do
 7     forall 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 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 with N processing Units (PUs). Accepts PREs.

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 $\delta(S_1, w) = \delta(\delta(S_0, w_3), w) = \delta(S_0, w_3 w)$ we have $\delta(S_1, w)$ accepting $\Rightarrow \delta(S_0, w_3 w)$ accepting. Hence $w_3 w \in L$. Since L is prefix closed, $w_3 w \in L \Rightarrow w_2 w_3 w \in L \Rightarrow \delta(S_0, w_2 w_3 w)$ accepting. But $\delta(S_0, w_2 w_3 w) = \delta(\delta(S_0, w_2 w_3), w) = \delta(S_2, w)$. 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 $\delta(S_2, w) = \delta(\delta(S_0, w_2 w_3), w) = \delta(S_0, w_2 w_3 w)$ is accepting. Hence $w_2 w_3 w \in L$. Since L is prefix closed, $w_2 w_3 w \in L \Rightarrow w_1 w_2 w_3 w \in L$. We have $w_1 w_2 w_3 w \in L \Leftrightarrow \delta(S_0, w_1 w_2 w_3 w)$ is accepting. But, $\delta(S_0, w_1 w_2 w_3 w) = \delta(\delta(S_0, w_1 w_2 w_3), w) = \delta(S_1, w)$. 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: We define *validation region* 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:

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

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 $M_e \approx 50\%$ 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 $P_e \approx 100\%$ and $M_e \approx 100\%$.

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 of batching together packets of different sizes. Due to the complexity of performance in IPSs with parallel hardware, it 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.

4.4 Relaxing the Assumptions

Matching non-PRE expressions: The basic SPPM algorithms require prefix-closed expressions only because Secondaries are allowed to safely terminate the algorithm 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 the case when the string `bad` occurs at the beginning of the second chunk, not at the beginning of the input. The version of the algorithm described later in this section avoids the problem.

Returning more information about matched packets: 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 the DFA 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. It is straightforward to extend Algorithm 4 to deliver information about the match, but if the system requires information about the first match (or about all matches), we need a more elaborate modification.

The most general case is when the system requires an ordered list of all matches and accepts arbitrary regular expressions. We change the way Algorithm 4 handles matches: instead of returning immediately, each Secondary PU keeps a list of all the matches it finds. After validation, the individual lists are combined in an ordered list of all matches, but candidate matches found by PU_k at positions preceding the coupling position with PU_{k-1} are discarded. Note that since the common case in IPSs is that no matches are found, the overhead of the extra bookkeeping required is incurred only for a small fraction of the packets and the overall system performance is not affected.

Limiting inefficiency by bounding the validation cost: 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 the history buffer to H positions,

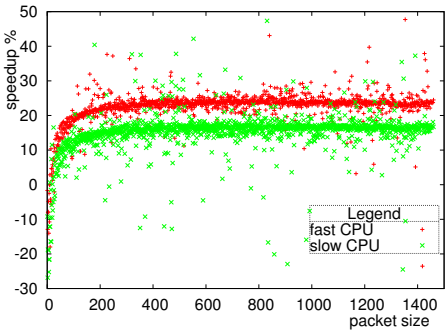

Fig. 4. Speedup of Algorithm 3 over the sequential DFA Algorithm

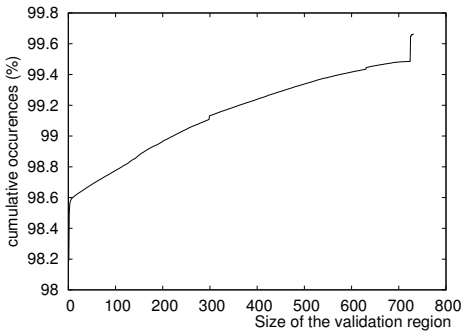

Fig. 5. CDF graph for the sizes of the validation region

and stop the validation stage for all PUs other than the primary when they reach the end of their history buffer. If H is large enough convergence may still happen (based on our experiments 40 would be a good value), but we bound the number of memory accesses performed during the validation stage to $H(N-2)$ for the $k-2$ non-primary PUs doing validation and $|I|-|I|/N$ for the primary. Thus $M \leq |I|(2-1/N)+H(N-2) < 2|I|+HN$ and $M_e > 1/(2+HN/|I|)$.

5 Experimental Evaluation

We compared results using speculative matching against the traditional DFA method. We used 106 DFAs recognizing a total of 1499 Snort HTTP signatures. 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. The most frequent packet sizes were 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.

5.1 Evaluation of Algorithm 3 (Software Implementation)

We implemented Algorithm 3 and measure its actual running time using Pentium performance counters. We ran experiments on two processors, an Intel Core 2 at 2.4GHz and a Pentium M at 1.5GHz. Compared to the traditional sequential algorithms we obtained speedups of 24% and respectively 13%. We explain the higher speedup for the faster processor by the larger gap between the processor speed and the memory latency. Figure 4 shows how the packet size influences the speedup for Algorithm 3: for packets smaller than 20 bytes speculation may result in slowdowns, but for packets larger than 50 bytes the speedup does not change significantly with packet size.

We also find that the validation typically happens quickly. For 98% of the packet validation happens after a single input byte is processed. Validation failed

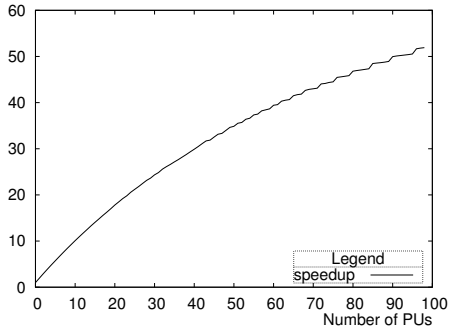

Fig. 6. Speedup for Algorithm 4

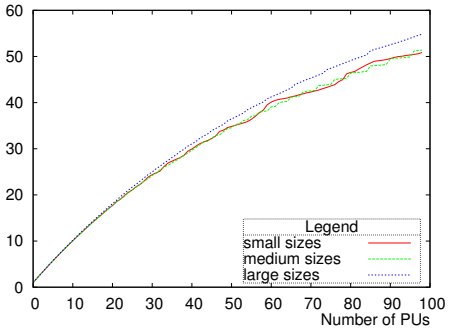

Fig. 7. Speedup by packet size

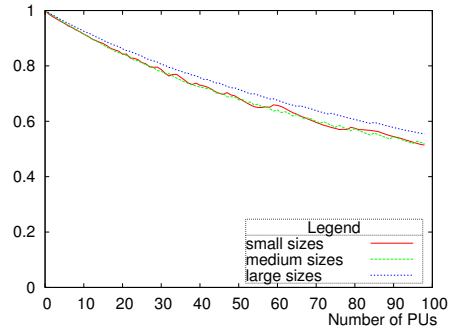

Fig. 8. Processing efficiency by packet size

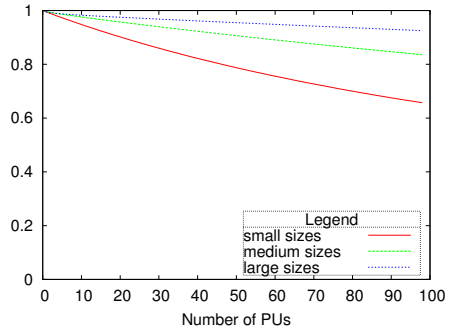

Fig. 9. Memory efficiency by packet size

for only 0.335% of the packets. Figure 5 shows the cumulative distribution of the sizes of the validation regions.

5.2 Evaluation of Algorithm 4

We evaluated Algorithm 4 for up to $N = 100$ processing units. We report speedups and efficiency based on our performance model which relies on the number of accesses to the DFA data structure (lines 9 and 18 of Algorithm 4). These metrics are described in Sect. 4.3 by equations 1. From Fig. 6 we see 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 $N = 100$. Figures 7, 8 and 9 show the speedup, processing efficiency and respectively memory efficiency for packets of various sizes: small (1-250), medium (251-1250) and large (1251-1500). The only notable difference is the low memory efficiency for small packets.

Figures 10 and 11 present the cumulative distributions for the sizes of the validation regions when $N = 10$. Figure 10 captures the sizes of all validation regions, which is relevant to memory efficiency. Figure 11 captures only the largest validation region for each packet, which is relevant to processing efficiency.

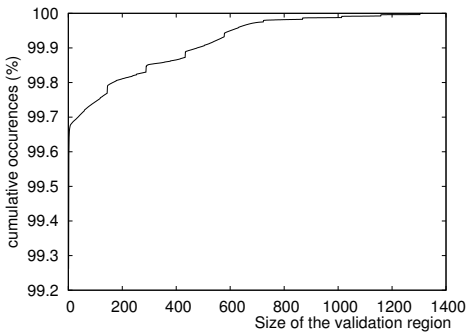

Fig. 10. Cumulative distribution of the sizes of validation regions

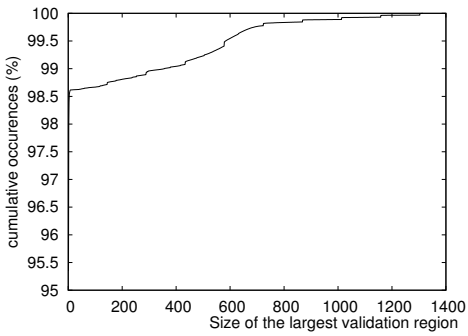

Fig. 11. Cumulative distribution of the size of largest validation regions

The average size for the validation regions is $V_\Sigma/(N-1) = 2.12$ and for the largest validation regions is $V_{max} = 8.24$. 99.26% of the validation regions were a single byte long and 95.35% of the packet had $V_{max} = 1$.

6 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 [27, 8, 25, 16, 26]. With the rise of attack techniques involving evasion [18, 19, 10, 21] and mutation [12], though, string-based signatures have more limited use, and modern systems have moved to vulnerability-based signatures written as regular expressions [28, 6, 24, 20]. In principle, DFA-based regular expression matching yields high matching speeds, but combined DFAs often produce a state-space explosion [22] with infeasible memory requirements. Many techniques have been proposed to reduce the DFA state space [22, 23], or to perform edge compression [15, 3, 13, 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 encodings.

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.* [14] 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 multi-core, 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 [17] derives an $O(log(n))$-time string matching algorithm using $O(n*length(string))$ processors. As with the above, 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.* [30] propose a TCAM-based scheme for matching simple regular expressions or strings. Weinsberg *et al.* [29] 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.

7 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 doesn't 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.

Our technique is the first method we are aware of that performs regular-expression matching in parallel. 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.

Acknowledgements: The authors 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. M. Jordan. Dealing with metamorphism. *Virus Bulletin Weekly*, 2002.
13. S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Securecomm*, Sep 2008.
14. 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.
15. 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.

16. 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.
17. J. Misra. Derivation of a parallel string matching algorithm. *Information Processing Letters*, 85:255–260, 2003.
18. V. Paxson. Defending against network IDS evasion. In *Recent Advances in Intrusion Detection (RAID)*, 1999.
19. T. Ptacek and T. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, Jan. 1998.
20. M. Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference*. USENIX, 1999.
21. U. Shankar and V. Paxson. Active mapping: Resisting nids evasion without altering traffic. In *IEEE Symp. on Security and Privacy*, May 2003.
22. 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.
23. R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.
24. R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, Oct. 2003.
25. 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.
26. L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA*, June 2005.
27. 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.
28. 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.
29. 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.
30. F. Yu, R. H. Katz, and T. Lakshman. Gigabit rate packet pattern-matching using tcam. In *In ICNP*, pages 174–183, 2004.