Fast and Scalable Pattern Matching for Content Filtering

Sarang Dharmapurikar
Washington University in St. Louis.
sarang@arl.wustl.edu

John Lockwood
Washington University in St. Louis.
lockwood@arl.wustl.edu

ABSTRACT

High-speed packet content inspection and filtering devices rely on a fast multi-pattern matching algorithm which is used to detect predefined keywords or signatures in the packets. Multi-pattern matching is known to require intensive memory accesses and is often a performance bottleneck. Hence specialized hardware-accelerated algorithms are being developed for line-speed packet processing. While several pattern matching algorithms have already been developed for such applications, we find that most of them suffer from scalability issues. To support a large number of patterns, the throughput is compromised or vice versa.

We present a hardware-implementable pattern matching algorithm for content filtering applications, which is scalable in terms of speed, the number of patterns and the pattern length. We modify the classic Aho-Corasick algorithm to consider multiple characters at a time for higher throughput. Furthermore, we suppress a large fraction of memory accesses by using Bloom filters implemented with a small amount of on-chip memory. The resulting algorithm can support matching of several thousands of patterns at more than 10 Gbps with the help of a less than 50 KBytes of embedded memory and a few megabytes of external SRAM. We demonstrate the merit of our algorithm through theoretical analysis and simulations performed on Snort's string set.

Categories and Subject Descriptors

C.2.3 [**Internetworking**]: Network Monitoring

General Terms

Algorithms, Design, Performance, Security

Keywords

Content Filtering, Pattern Matching, Network Intrusion Detection, Bloom Filters

1. INTRODUCTION

Several modern packet processing applications including Network Intrusion Detection/Prevention Systems (NIDS/NIPS), Layer-7 switches, packet filtering and transformation systems perform deep packet inspection. Fundamentally, all these systems attempt to make sense of the application layer data in the packet. One of the most frequently performed operation in such applications is searching for predefined patterns in the packet payload. A web server load balancer, for instance, may direct a HTTP request to a particular server based on a certain predefined keyword in the request. Signature-based NID(P)S looks for the presence of the predefined signature strings deemed harmful to the network such as an Internet worm or a computer virus in the payload. In some cases the starting location of such predefined strings can be deterministic. For instance, the URI in a HTTP request can be spotted by parsing the HTTP header and this precise string can be compared against the predefined strings to switch the packet. In certain cases one doesn't know where the string of our interest can start in the data stream making it imperative for the system to scan every byte of the payload. This is typically true of the signature based intrusion detection systems such as Snort [1].

Snort is a light-weight NIDS which can filter packets based on predefined rules. Each Snort rule first operates on the packet header to check if the packet is from a source or to a destination network address and/or port of interest. If the packet matches a certain header rule then its payload is scanned against a set of predefined patterns associated with the header rule. Matching of one or multiple patterns implies a complete match of a rule and further action can be taken on either the packet or the TCP flow. The number of patterns can be in the order of a few thousands. Snort version 2.2 contains over 2000 strings.

In all these applications, the speed at which pattern matching is performed critically affects the system throughput. Hence efficient and high-speed algorithmic techniques which can match multiple patterns simultaneously are needed. Ideally we would like to use techniques which are scalable with number of strings as well as network speed. Software based NIDS suffer from speed limitations, they can not sustain wire-speed of more than a few hundred mega bits per second. This has led the networking research community to explore hardware based techniques for pattern matching. Several interesting pattern matching techniques for network intrusion detection have been developed, a majority of them produced by the FPGA community. These techniques use the reconfigurable and highly parallel logic resources on an FPGA to contrive a high-speed search engine. However, these techniques suffer from scalability issues, either in terms of speed or the number of patterns to be searched, primarily due to the limited and expensive logic resources.

We present an algorithm that is scalable in terms of network speed, number of patterns and pattern length. Our algorithm modifies the classic Aho-Corasick algorithm (see [2]) to allow it to consider an alphabet consisting of symbols formed by group of characters instead of a single character. We show how a regular Aho-Corasick automaton can be transformed into an automaton which is suitable for matching multiple characters at a time as opposed to single character. Moreover, we also show how this transformation allows us to *parallelize* the Aho-Corasick algorithm. Once we parallelize the Aho-Corasick automaton, we can use multiple instances of it to achieve a required speedup by advancing the text stream by multiple characters at a time. Most importantly, we show how each automation can be implemented using Bloom filters [4] which help us suppress off-chip memory access to a great extent. When a string of interest appears in the text stream our machine performs the necessary memory accesses and hence slows down the string matching process. However, since the typical text stream in the context of NIDS rarely contain strings of interest, our algorithm can maintain the desired speedup for such a text stream.

The rest of the paper is organized as follows. In the next section we summarize the related work in hardware-based multi-pattern matching. Then, in Section 3 we begin with an introduction to the Aho-Corasick algorithm and highlight the problems it suffers from. In Section 4 we illustrate our algorithm and analyze its performance in Section 5. We report the simulation results with the string-set extracted from Snort NIDS in Section 6. Finally Section 7 concludes the paper.

2. RELATED WORK

Multi-pattern matching is one of the well studied classical problems in computer science. The most notable algorithms include Aho-Corasick (explained in the next section) and Commentz-Walter algorithms which can be considered as the extension of well-known KMP and Boyer-Moore single pattern matching algorithms respectively [7]. Both these algorithms are suitable only for software implementation and suffer from throughput limitations. The current version of Snort uses an optimized Aho-Corasick algorithm.

In the past few years, several interesting algorithms and techniques have been proposed for multi-pattern matching in the context of network intrusion detection. The hardware-based techniques make use of commodity search technologies such as TCAM [13] or reconfigurable logic/FPGAs [6][11][3][10] Some of the FPGA based techniques make use of the on-chip logic resources to compile patterns into parallel state-machines or combinatorial logic. Although very fast, these techniques are known to exhaust most of the chip resources with just a few thousand patterns and require bigger and expensive chips. Therefore, scalability with pattern set size is the primary concern with purely FPGA-based approaches.

An approach presented in [5] uses FPGA logic with embedded memories to implement parallel Pattern Detection Modules (PDMs). PDMs can match arbitrarily long strings by segmenting them in smaller substrings and matching them sequentially.

The technique proposed in [11] also seeks to accelerate the Aho-Corasick automation by considering multiple characters, however, our underlying implementation is completely different. While they use suffix matching, we use prefix matching with multiple machine instances. Moreover, their implementation uses FPGA lookup tables and hence is limited in the pattern set size where as our implementation is based on Bloom filters, which are memory efficient data structures.

From the scalability perspective, memory-based algorithms are attractive since memory chips are cheap. Unfortunately, while using memory-based algorithms, the memory access speed becomes a bottleneck. A highly optimized hardware-based Aho-Corasick algorithm was proposed in [12]. The algorithm uses a bit-map for compressed representation of a state node in Aho-Corasick automaton. Although very fast even in the worst case (8 Gbps scanning rate), the algorithm assumes the availability of excessively large memory bus such as 128 bytes to eliminate the memory access bottleneck and would suffer from power consumption issues.

A Bloom-filter based algorithm proposed in [8] makes use of a small amount of embedded-memory along with commodity off-chip memory to scan a large number of strings at high speed. Using on-chip Bloom filters, a quick check is done on the payload strings to see if it is likely to match a string in the set. Upon a Bloom filter match, the presence of the string is verified by using a hash table in the off-chip memory. The authors argue that since the strings of interest are rarely found in the packets, the quick check in Bloom filter reduces more expensive memory accesses and improves the overall throughput greatly. However, since the algorithm involves hashing over a maximum length pattern size text window, it does not scale for arbitrarily long strings (100s of bytes). It is reported that up to 16 bytes is a feasible pattern length for a high-speed implementation. As we will see, our algorithm combines the techniques in [8] with Aho-Corasick algorithm to get rid of the string length limitation.

A TCAM based solution for pattern matching proposed in [13] breaks a long patterns into shorter segments and keeps them in TCAM. A window of characters from the text is looked up in the TCAM and upon a partial match, the result is stored in a temporary table. The window is moved forward by a character and the lookup is executed again. At every stage, the appropriate partial match table entry is taken into account to verify if a complete string has matched. The authors deal with the issue of deciding a suitable TCAM width for efficient utilization of TCAM cells. They also use TCAM cleverly for supporting wild card patterns, patterns with negations and correlated patterns. Although this technique is very fast, being TCAM based, it suffers from other well known problems such as excessive power consumption and high cost. Further, the throughput of this algorithm is limited to a single character per clock tick. Scanning multiple characters at a time would require multiple TCAM chips.

3. AHO-CORASICK ALGORITHM

In multi-string matching problem, we have a set of strings S and we would like to detect all the occurrences of any of the strings in S in a text stream T. We will denote by $T[i...j]$ the character sequence from i^{th} character to j^{th} character of stream T. For a given set of strings, the Aho-Corasick algorithm constructs a finite automaton. This finite automaton can be a Deterministic Finite Automaton (DFA) or a Non-deterministic Finite Automaton (NFA). For our purpose, we will focus on NFA version of the algorithm since that is the one we will improve upon. Figure 1 shows an example of a constructed NFA for a set of strings.

Pattern matching is easy: given a current state in the automaton and the next input character, the machine checks to see if the character causes a failure transition. If not, then it makes a transition to the state corresponding to the character. Otherwise, it makes a $failure$ transition. In case of a failure transition the machine must reconsider the character causing the failure for the next transition and the same process is repeated recursively until the given character leads to a non-failure transition. Note that eventually there will be one non-failure transition since all the transitions from q_0 are always non-failure transitions and by falling down the chain of failure transitions, machine can reach state q_0 in the worst case. Only after

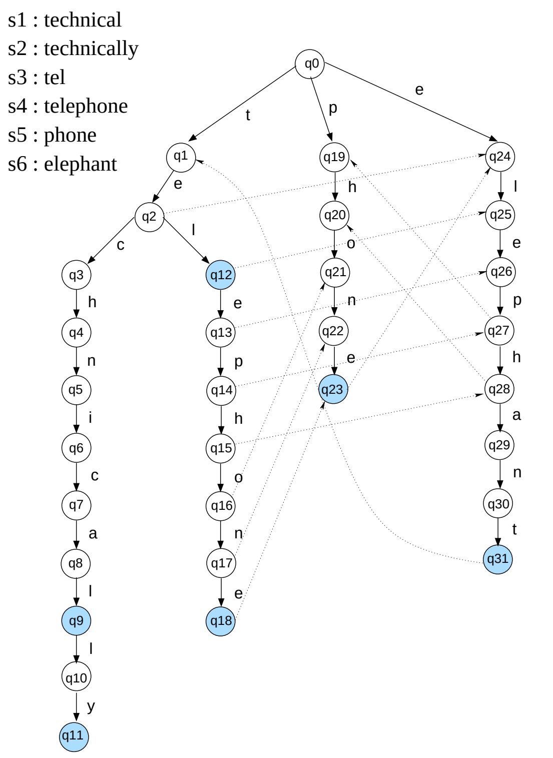

Figure 1: Building an Aho-Corasick NFA for a set of strings. Failure transitions to only non q_0 states are shown for the purpose of clarity. All the other states make a failure transition to q_0

the transition settles to a particular state, the next character from the text can be considered.

The first fundamental problem Aho-Corasick algorithm suffers from is a high memory access requirement. At least one memory access is needed to read the state node on each input character. Furthermore, the sequential failure transitions can cause more memory accesses. In the worst case, the average number of memory accesses required per input character is two. Therefore, given the high latency and slow speed of commodity memory chips, using them to implement Aho-Corasick algorithm can severely degrade the throughput of the system.

The second problem we observe in the regular Aho-Corasick algorithm is that it can not be readily parallelized. Hence, we are forced to consider only one character at a time from the text stream no matter how much logic and memory resources are available. The processing of one character per clock cycle of the system clock can create a bottleneck for high speed networks.

A naive approach to improve the throughput by scanning multiple characters at a time is to simply use multiple instances of the automaton working in parallel each consuming a single character at a time. While this is implementable, due to the memory requirement of each automaton, we would need separate external memory chips for each of the automaton to achieve the desired speedup. Multiple memory chips is not an attractive solution due to the high cost, the larger number of pins to interface the chips and the resulting power consumption. Moreover, with such an implementation, a single TCP flow always needs to be handled by only one of the automata which can potentially lead to an imbalanced throughput across flows.

We describe a hardware based implementation of our modified version of Aho-Corasick algorithm which attacks the two problems mentioned above. First we parallelize the Aho-Corasick algorithm and then use on-chip Bloom filters to suppress the memory accesses to off-chip memory.

4. A SCALABLE AND HIGH-SPEED ALGORITHM

4.1 Basic Ideas

We reorganize the Aho-Corasick automaton to consider k characters at a time. While executing this NFA, given the current state of the NFA, we directly jump to a state to which we would eventually go by considering one character at a time in the next k characters.

With respect to Figure 1 and assuming that we would like to consider $k = 4$ characters at a time, if the machine is in the state q_0 and a string "tech" is given as an input then it jumps to state q_4 and continues its execution from this state by looking at next k characters. If the next four characters are "nica" then it jumps to q_8. While in q_4, if any other string (e.g. "nice") is seen then we simply make a failure transition to the failure state associated with q_4 since we know that this results in a failure transition eventually. Thus, we do not need to track the states sequentially all the way up to q_7 by considering the characters 'n', 'i' and 'c' and finally make a failure transition due to 'e'. From the failure state of q_4 we can reconsider the same k characters again.

By comparing the next k characters with all the valid k character segments associated with a given state we can find out if the machine eventually goes to a non-failure state or a failure state. For instance, the valid 4-character strings associated with state q_0 are "tech", "tele", "phon" and "elep" which lead the machine to states q_4, q_{13}, q_{22}, and q_{27} respectively. Likewise, "nica" is a valid 4-character segment associated with q_4 which leads the automaton to q_8.

Thus, in our new NFA we treat a group of $k-$characters as a *single symbol*. The new NFA essentially jumps k characters ahead. We call this variation of the algorithm Jump-ahead Aho-CorasicK NFA (JACK-NFA). Tracking the states in this fashion, however, eventually requires us to match fewer than k characters to detect the string completely. For instance, after matching "tech" and "nica" we go to q_8. From this state we need a match for either "l" or "lly" to detect an entire valid string and these segments contain less than four characters. We refer to these segments as *tails* associated with strings. "l" and "lly" are tails of "technical" and "technically" respectively. The JACK-NFA for the example string set is shown in Figure 2.

Briefly, our algorithm first matches the longest prefix of strings having a length of multiple of k characters using JACK-NFA. After this prefix is found to be matching, we check to see if the tails associated with that particular string match with any prefix of the next $k-$ character text. When a matching tail is found in the next k characters, we say that a string matches completely. When we scan k characters to look for a matching tail, we can start from the longest prefix of these k characters and move towards a shortest prefix. We can stop when we find a longest matching prefix. By keeping the information regarding any shorter prefix that should match with this longer prefix the algorithm can correctly match all the patterns. For instance, when the machine is in state q_5 and upon inspecting next four characters if it finds a match for "lly" then it can stop right there and report a match for strings "technical" and "technically". It doesn't need to match shorter prefix "l". Likewise, when the machine is in q_0 and finds a match for the four characters

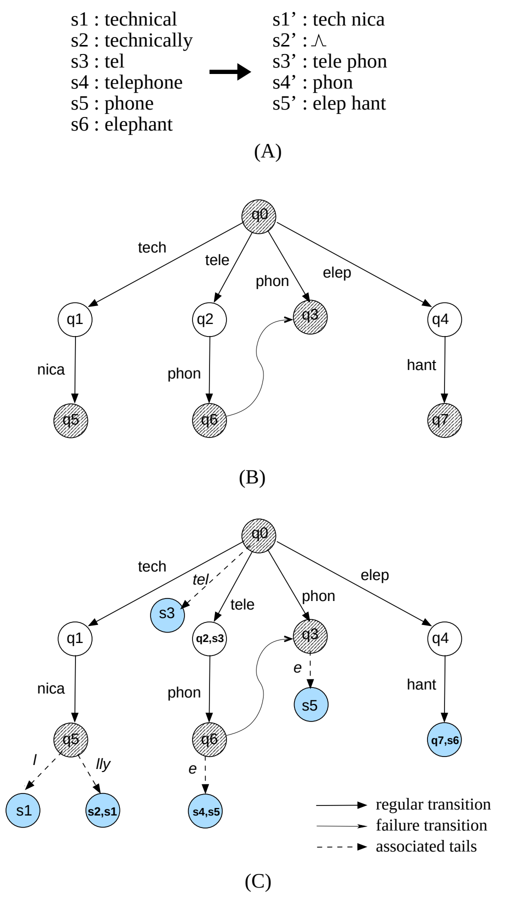

Figure 2: (A) The original string set and modified string set. The space between the k character boundary is is shown as a demarcation (B) Jump-ahead Aho-Corasick (JACK) NFA. The nodes with pattern indicate a state corresponding to a matching substring. Failure transition to only non q_0 states are shown. Failure transitions of rest of the states are to q_0 (C) JACK-NFA with tails associated with states.

"tele" then it can directly jump to state q_2 and also report a match for the string "tel". It doesn't

Thus, at any state, the machine looks at next k characters and looks for a match for all of them (in which case it makes a transition to another state) or looks for a matching prefix of k characters (for tail matching). Therefore, it is the same as looking for the longest matching prefix of $k-$character substring where the prefixes to be searched are specific to a state and they change when the state changes.

To match the string correctly, our machine must capture it at the correct $k-$character boundary. If a string were to appear in the middle of the k character group, it will not be detected. For instance, if we begin scanning text "xytechnical..." then the machine will view the text as "xyte chni cal.." and since "xyte" is not the beginning of any string and is not a valid 4-character segment associated with state q_0, the automaton will never jump to a valid state causing the machine to miss the detection. Thus, we must ensure that the strings of interest appear at the correct byte boundary. To do this, we deploy k machines each of which scans the text with one byte offset. In our example, if we deploy 4 machines then the first machine scans the text as "xyte chni cal.." whereas the second machine scans it as "ytec hnic al.." and the third machine scans it as "tech nica l.." and finally the fourth machine scans it as "echn ical ...". Since the string appears at the correct boundary for the third machine, it will be detected by it. Therefore, by using k machines in parallel we will never miss detection of a string.

These k machines need not be *physical machines*. They can be four *virtual machines* emulated by a single physical machine. We illustrate this concept with help of Figure 3. In this figure we use $k = 4$ virtual machines each scans the text by a character offset with respect to the neighbor machines.

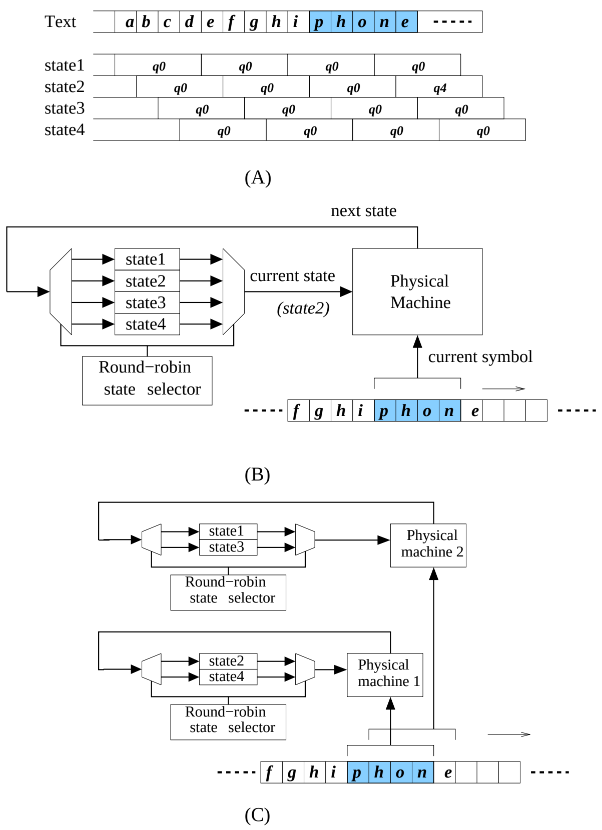

Figure 3: Illustration of Virtual Machines. (A) Each virtual machine i maintains its $state_i$. This state is updated every k iterations. In each iteration we update k states corresponding to k virtual machines. (B) Machines virtual since only the $state$ variable of each machine is independent. The component that updates the state is the same for all the states. We call it physical machine. (C) Multiple virtual machines can be implemented using the same physical machine. The figure shows that machine 1 and 3 are implemented by one physical machine and machine 2 and 4 by another. This gives a speedup of two

To implement these virtual machines, we maintain k independent states, $state_1$ to $state_k$ and initialize each to q_0. When we start scanning the text, we consider first k characters $T[1...k]$, and compute a state transition of $state_1$ and assign the next state to the $state_1$. Then we move a character ahead and consider characters $T[2...k+1]$ and update the $state_2$ with these characters as the next symbol. In this way, we keep moving a byte ahead and update the state for each virtual machine. After having considered first $k-1$ bytes, bytes $T[k...2k-1]$ will be considered to update the last virtual state machine and then we would complete a cycle of updates. Now we consider characters $T[k+1...2k]$, and update machine 1. We repeat this round-robin updates of each virtual machine. This ensures that each character is considered only once by a virtual machine and each machine will always look at k characters at a time. Clearly, one of the machines will always capture the strings at the correct boundaries and detect them. Note how machine 2 makes a transition from q_0 to q_4 after receiving the symbol $phon$ at the correct boundary. After reaching in q_4 and considering the next four characters, the machine outputs $phone$.

With virtual machines, we still consume only a character at a time. We gain the speedup by using multiple physical machines to implement the virtual machines. Note that the virtual machine operation is independent from each other and can be implemented in parallel. Moreover, the number of physical machines need not be equal to number of virtual machines; they can be less than the number of virtual machines where each physical machines implements multiple virtual machines. To shift the text stream by r characters at a time, we use r physical machines to emulate k virtual machines where $r \leq k$. This can be illustrated with Figure 3(C). If we consider $k = 4$ virtual machines which keep $state_1$ to $state_4$ and would like a speed up of just two then two machines can be used. In the first clock cycle, two machines update $state_1$ and $state_2$ and shift the stream by two characters. In the next clock cycle, they update $state_3$ and $state_4$ and shift the stream again by two characters. The same procedure repeats.

To reduce memory accesses in each cycle, we use on-chip Bloom filters containing compressed transition tables of the machine. When we look up the transition table to get the next state from the current state and current symbol, we first check the on-chip Bloom filters to see if the state transition should be successful or a failure. Only for successful transitions we need to look up the input key in the off-chip table.

The exact details of the algorithm and data structures are explained in the next subsection.

4.2 Data structures and Algorithm

The challenge is to implement each “physical machine” shown in Figure 3 such that it requires very few memory accesses to execute a state transition. We begin by representing the JACK-NFA using a hash table. Each table entry consists of a pair $\langle state, substring\rangle$ which corresponds to a transition edge of this NFA. For instance $\langle q_0, tel\rangle$, $\langle q_0, phon\rangle$ etc. are the transition edges in the NFA. This pair is used as a key for hash table. Associated with this key we keep the tuple $\{NextState, MatchingStrings, FailureChain\}$ where $FailureChain$ is the set of states the machine will fall through if it fails on $NextState$. For instance, the tuple associated with the key $\langle q_0, tele\rangle$ is $\{q_2, s_3, q_0\}$ which means that the machine goes from state q_0 to q_2 with substring $tele$ and from q_2 it can fail to q_0. When it is in q_2, it implies a match for string s_3. Likewise, the tuple associated with $\langle q_2, phon\rangle$ is $\{q_6, NULL, q_3, q_0\}$ implying that there are no matching strings at q_6 and upon a failure from q_6, the machine goes to q_3 from where it can fail to q_0. The entire transition table for the example JACK-NFA is as shown in Table 1. It should be noted that the final state of each FailureChain is always q_0. Secondly, the keys in which the substring are tails don’t lead to any actual transition and neither they have any failure states associated.

This table can be kept in the off-chip commodity memory. We now express our algorithm for *just a single physical machine emulating k virtual machines* (Figure 3(B)) with the pseudo-code shown in the Figure 4.

j denotes the virtual machine being updated and i denotes the current position in the text. All the virtual machines are modified in the round robin fashion (expressed by the mod k counter of line 12). All the states corresponding to the virtual machines are ini-

$\langle state, substring\rangle$	Next State	Matching Strings	failure chain
$\langle q_0, tech\rangle$	q_1	NULL	q_0
$\langle q_0, tele\rangle$	q_2	s_3	q_0
$\langle q_0, phon\rangle$	q_3	NULL	q_0
$\langle q_0, elep\rangle$	q_4	NULL	q_0
$\langle q_1, nica\rangle$	q_5	NULL	q_0
$\langle q_2, phon\rangle$	q_6	NULL	q_3, q_0
$\langle q_4, hant\rangle$	q_7	s_6	q_0
$\langle q_0, tel\rangle$	NULL	s_3	NULL
$\langle q_3, e\rangle$	NULL	s_5	NULL
$\langle q_5, l\rangle$	NULL	s_1	NULL
$\langle q_5, ly\rangle$	NULL	s_2, s_1	NULL
$\langle q_6, e\rangle$	NULL	s_4, s_5	NULL

Table 1: Transition table of JACK-NFA. This table can be implemented in the off-chip memory as a hash table.

```
DetectStrings
1.  $j \leftarrow 1$,$i \leftarrow 1$
2.  for ($l = 1$ to $k$) $state_l \leftarrow q_0$
3.  while (text available)
4.      $x = T[i...i + k - 1]$
5.      $\{state, strings\} \leftarrow$ LPM($state_j, x$)
6.      report $strings$
7.      $l \leftarrow 0$
8.      while ($state = NULL$)
9.          $\{state, strings\} \leftarrow$ LPM($state_j.f[l$++$], x$)
10.         report $strings$
11.     $state_j \leftarrow state$
12.     $i \leftarrow i + 1$,$j \leftarrow (j + 1 \mod k)$
```

Figure 4: Algorithm for detecting strings.

tialized to q_0 (line 2). To execute JACK-NFA, we consider next k characters from the text (line 4) and search for the longest matching prefix associated with the current state of the virtual machine being updated, $state_i$ (line 5). Shortly we will see how exactly we perform LPM with very few memory accesses. For now, assume that the LPM process returns a set of matching strings, the next state of the machine and the failure chain associated with the next state. If the next state is valid, we update the state of the current virtual machine (line 11). If the next state is NULL then we execute the same procedure with each of the states in the failure chain (line 9-11) starting from the first (line 8). We stop falling through the failure chain once we find a successful transition from one of them. When we perform LPM for failure states, we report the matching strings at each failure node as we trickle down the failure chain (line 10).

Now, consider the $LPM(q, x)$ process to find the longest matching prefix of x which is associated with q. This could be easily performed by probing the hash table with the keys $\langle q, x[1...i]\rangle$ starting from the longest prefix, $x[1...k]$. We could continue probing until we find a matching prefix. If none was found then we could return NULL. However this naive process will require k memory accesses in the worst case for as many hash probes. However, for applications such as NIDS, the true matches are rare and most of the time the machine results in a failure transition. We can use Bloom filters to filter out unsuccessful searches in the off-chip table. We first group all the keys containing the prefixes of the same length. For instance, the keys $\langle q_5, l\rangle$, $\langle q_3, e\rangle$, $\langle q_6, e\rangle$ form one group. $\langle q_0, lly\rangle$ forms another group and so on. Then we store all the keys of a group in one Bloom filter which is implemented using a small amount of on-chip memory. Since k groups are possible with k unique prefix lengths, we maintain as many parallel Bloom filters each of which corresponds to a unique prefix length as shown in Figure 5.

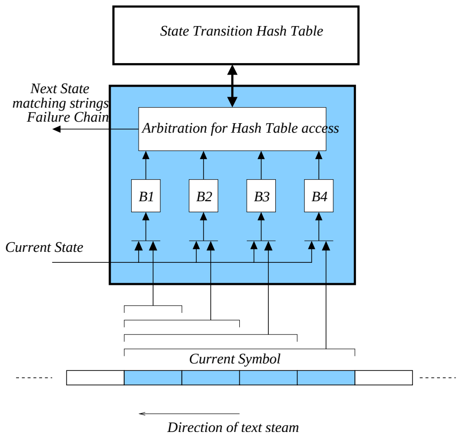

Figure 5: Implementation of a physical machine. For this figure $k = 4$. The pair $\langle state, prefix\rangle$ is looked up in the associated Bloom filter before off-chip table accesses.

Before, we look up the pair $\langle q, x[1...i]\rangle$ in the off-chip table, we probe the on-chip Bloom filter corresponding to length i. In total k parallel queries are performed to Bloom filters and the bit array of their results, *match* vector, is obtained. Since we can engineer a Bloom filter to give the result in a single clock cycle (see [8]), we immediately know which of the k keys are a possible match by looking at the bits in $match$ vector. We, then walk through the $match$ vector from the longest to the shortest prefix and execute the hash table lookup, which we refer to as HTL($\langle q, x[1...i]\rangle$), for a prefix showing a match in the filter. These operations are described through the pseudo-code given in the Figure 6.

```
LPM $(q, x)$
1.  for ($i = k$ downto 1)
2.      $match[i] \leftarrow$ BFL$_i$($\langle q, x[1...i]\rangle$)
3.  for ($i = k$ downto 1)
4.      if ($match[i] = 1$)
5.          $\{\langle q', y[1...i]\rangle, next, strings\} \leftarrow$ HTL($\langle q, x[1...i]\rangle$)
6.          if ($\langle q', y[1...i]\rangle = \langle q, x[1...i]\rangle$)
7.              return $\{next, strings\}$
8.  if ($q = q_0$) $next \leftarrow q_0$
9.  else $next \leftarrow NULL$
10. return $\{next, NULL\}$
```

Figure 6: Algorithm for the Longest Prefix Matching.

The speed advantage comes from the fact that the operations of line 1-2 can be executed in parallel and a result can be obtained in a single clock cycle. Furthermore, $match$ vector is usually empty with the exception of rare false positives and true positives. Therefore, we hardly need to access the hash table. Note that in the end if no matching prefix is found and if the current state is q_0 then we return q_0 as the next state since this is a failure but there are no failure states for q_0.

5. ANALYSIS

Now we analyze the performance of our algorithm in terms of the number of memory accesses performed after processing t characters of the text. The number of $k-$character symbols seen by the first machine is $\lceil t/k\rceil$. Since the second machine scans it with a byte offset, the number of symbols it sees is $\lceil (t-1)/k\rceil$. Likewise, the number of symbols, y_i seen by the i^{th} machine is

$$y_i = \begin{cases} \left\lceil \frac{t-(i-1)}{k} \right\rceil & \text{for } t \geq i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Before we analyze the algorithm, we will derive an expression for the number of memory accesses required to execute an LPM. Recall that in the LPM algorithm, we start inspecting the $match$ vector from the longest to the shortest prefix and execute a hash table search for any bit that is set. If the bit was set due to false positive of Bloom filter we waste a memory access and proceed to search the next matching prefix. If the bit is set due to a true match then we stop after the hash table search. Let T_i denote the cumulative number of memory accesses spent in hash table accesses from bit i down to bit 1. Let p_i denote the probability that the input to filter i was a true string (also known as true positive probability or successful search probability). Finally, let f_i denote the false positive probability of Bloom i. With these notations, we can express T_i as follows

$$T_i = p_i + (1-p_i)(f_i + T_{i-1}) \tag{2}$$

with boundary condition $T_0 = 0$. This equation says that with probability p_i we make one memory access for successful hash table search and with probability $(1-p_i)$ we make an unsuccessful hash table search if filter shows a false positive with probability f_i and also proceed to the next bit. The total time spent in LPM is T_k. This recursive relation will be used in further analysis.

A special case of this equation in which $p_k = 0$ will be used later. In this case, we know that the filter k did not have a true input. This is the case in which a machine fails from a given state and starts evaluating failure states. We will denote the cumulative number of memory accesses for this particular case as T'_k which is

$$T'_k = f_k + T_{k-1} \tag{3}$$

where T_{k-1} is given by the recursive relation of Equation 2.

We will now prove the following theorem

Theorem: $T_i \leq 1 + \sum_{j=2}^{i} f_j$ for $i \geq 2$ and $T_1 \leq 1$

proof: The proof is by induction on i.

For $i = 1$,

$$\begin{aligned} T_1 &= p_1 + (1-p_1)(f_1 + T_0) \\ &= f_1 + p_1(1-f_1) \\ &\leq 1 \end{aligned}$$

For $i = 2$,

$$\begin{aligned} T_2 &= p_2 + (1-p_2)(f_2 + T_1) \\ &\leq p_2 + (1-p_2)(f_2 + 1) \\ &\leq 1 + f_2 \end{aligned}$$

For $i = 3$,

$$\begin{aligned} T_3 &= p_3 + (1-p_3)(f_3 + T_2) \\ &\leq p_3 + (1-p_3)(f_3 + 1 + f_2) \\ &\leq 1 + f_3 + f_2 \end{aligned}$$

Now we will assume that the result holds for j, i.e.

$$T_j \leq 1 + \sum_{l=2}^{j} f_l$$

Hence,

$$\begin{aligned} T_{j+1} &= p_{j+1} + (1-p_{j+1})(f_{j+1} + T_j) \\ &\leq p_{j+1} + (1-p_{j+1})(f_{j+1} + (1 + \sum_{l=2}^{j} f_l)) \\ &\leq (1 + \sum_{l=2}^{j+1} f_l) \end{aligned}$$

Hence the proof. ●

As a result of this theorem, the following holds

$$T'_k \leq 1 + \sum_{i=2}^{k} f_i \qquad (4)$$

We will analyze the performance for three different cases:

- Worst case text: the text that triggers the most pathological memory accesses pattern
- Random text: the text composed of uniformly randomly chosen characters
- Synthetic text: Text is random but with some concentration of the strings of interest

5.1 Worst Case Text

To evaluate the worst case behavior, let's assume that the JACK-NFA has gone down to a state at depth d after which it falls through the failure chain by consuming the next symbol (The depth of state q_0 is 0). Thus, at this state, when it consumes a symbol, it needs T'_k memory accesses for LPM. Further, the worst case length of a failure chain associated with a state having a depth d is d, including the state q_0. We execute a LPM on each of these states, requiring $T'_k d$ memory accesses. Thus, after a failure from depth d state, in the worst case we require $T'_k d + T'_k = T'_k(d+1)$ memory accesses.

To reach to a state with depth d, the machine must have consumed at least d symbols. Moreover, for each of these symbols it executes a LPM which stops at the first memory access itself returning a match for a $k-$character symbol (and not less than k) since only a successful $k-$character symbol match pushes the machine to the next state. This implies that to reach a depth d state, exactly d memory accesses are executed after consuming d symbols. Finally, $d+1^{th}$ symbol causes failure and subsequently $T'_k(d+1)$ memory accesses as explained above. Hence, the worst case memory accesses for machine i after consuming y_i characters can be expressed as

$$w_i = y_i(T'_k + 1) - 1$$

The total number of worst case memory accesses by all the k machines after processing t character text can be expressed as

$$\begin{aligned} W &= \sum_{i=1}^{k} w_i = \sum_{i=1}^{k} (y_i(1+T'_k) - 1) \\ &= \sum_{i=1}^{k} \left(\left\lceil \frac{t-(i-1)}{k} \right\rceil (1+T'_k) - 1 \right) \end{aligned}$$

If $t >> i$ i.e. the number of characters consumed is more than the number of machines then we have $t-(i-1) \approx t$. If $t >> k$ i.e. the number of characters consumed is greater than the symbol size then $\lceil t/k \rceil \approx t/k$. Therefore,

$$W \approx \sum_{i=1}^{k} \left(\frac{t}{k}(1+T'_k) - 1 \right) = t(1+T'_k) - k$$

Using Equation 4, we have

$$W \leq t(2 + \sum_{i=2}^{k} f_i) - k < t(2 + \sum_{i=2}^{k} f_i)$$

and the worst case memory accesses per character, M_w,

$$M_w = 2 + \sum_{i=2}^{k} f_i \qquad (5)$$

By keeping the false positive probabilities f_i moderately low, we can reduce the factor $\sum_{i=2}^{k} f_i$. It should be recalled that the worst case memory accesses for the original Aho-Corasick is $2t$. Thus, our technique doesn't provide any gain over the original Aho-Corasick in the worst case scenario. However, as we will see in the next two subsections, the average case text, which is what is expected to be seen in a typical network traffic, can be processed very fast.

5.2 Random Text

We would like to know how our algorithm performs when the text being scanned is composed of random characters. It should be recalled that our state machine makes a memory access whenever a filter shows a match either due to a false positive or a true positive. Hence the performance depends on how frequently the true positives are seen. Let b_l^q denote the number of l character branches sprouting from state q. With respect to Figure 2, $b_4^{q_0} = 4$, $b_3^{q_0} = 1$. While the machine is in state q, the probability of spotting one of its l character branches in next k-character symbol from the text is $p_l^q = b_l^q/256^l$. Since, the average case evaluation of the algorithm depends on the true positive probability, we must make some assumptions regarding the value of branching factor for states. We assume that $b_l^q << 256^l$. This is clearly justifiable for values of $l \geq 3$ where $256^3 = 16M$ and the practical values of b_l^q are less than a few thousand. For instance, when we constructed the JACK-NFA with Snort string set we found that $b_4^{q_0} = 1253$ which was also the maximum. The rest of the states had $b_4^q \leq 4$. Hence, for Bloom filters corresponding to length $l \geq 3$ the probability of a true positive, $b_l^q/256^l$ can be considered negligibly small for practical purposes. Further, we also assume that we have very few $2-$character strings in our set and there are no single character strings. With these assumptions $p_2^{q_0} \approx 0$ and $b_1^{q_0} = 0$. Since for all i, $p_i^{q_0} \approx 0$, the JACK-NFA never leaves state q_0 (in other words, it leaves state q_0 very rarely, like once every million characters scanned causing negligibly small number of memory accesses). Therefore, all the memory accesses performed while scanning random text are due to the false positives shown by Bloom filters. Formally, by substituting $p_i = 0$ in Equation 2, we obtain the average number of memory accesses each machine performs while scanning a single symbol as $T_k = \sum_{i=1}^{k} f_i$ Since there are k characters in one symbol, the average memory accesses per machine per character is T_k/k. Since there are k machines, the average memory access per character, $M_a = T_k$:

$$M_a = \sum_{i=1}^{k} f_i \tag{6}$$

By keeping the false positive probability of each Bloom filter moderately low, the overall memory accesses can be reduced greatly. Therefore, the random text can be scanned quickly with hardly any memory accesses. We will shortly consider Bloom filter design for low false positives and evaluate it with Snort strings.

5.3 Synthetic Text

In subsection 5.1 we considered the most pathological text which causes two memory accesses per character and in subsection 5.2, we assumed that the text was composed of random characters. However, these two cases are two extremes of what is seen in reality. Typically, the strings to be searched occur with some finite frequency. In Snort, the strings are searched in the packet payload only when the packet header matches a certain "header rule". Hence, although certain strings are commonly seen in the data stream, it is of interest or is said to occur truely only when it appears within a particular context. To quantify the frequency of appearance of the strings in mostly random text, we define a new parameter called *concentration* which we denote by c, as the ratio of the number of string characters in the text to the total number of text characters. For instance, if we spot a 10 character long string of our set in the 1000 character text then the concentration is $c = 10/1000 = 0.01$. We use this value of concentration to model our text input for further analysis. This value may seem rather arbitrary, however, our experiments with Snort indicate that typically concentration is quite smaller than what we have assumed: approximately 1/20000. To be conservative, we assume that when the string appears in the text (with concentration c), it triggers the most pathological memory accesses pattern of two memory accesses per character (M_w). Otherwise, for random characters (with concentration $1-c$), the memory accesses are given by M_a. We can represent the average number of memory accesses, M_s, for a synthetic text as approximately

$$\begin{aligned} M_s &= M_w c + M_a(1-c) \\ &= (2 + \sum_{i=2}^{k} f_i)c + (1-c)\sum_{i=1}^{k} f_i \end{aligned} \tag{7}$$

On the one hand, the assumption of two memory accesses per string character tends to over estimate the value of M_s, on the other hand practical Bloom filters with a given configuration of memory and hash function show a higher false positive rate which causes M_s to be under estimated. In order to get a sense of actual value of M_s we simulate our algorithm over Snort strings and synthetic text with a given concentration of strings.

6. EVALUATION WITH SNORT

We used the Snort version 2.0 for our experiments which has 2280 content filtering rules. The total number of strings associated with all these rules is 2259. The string length distribution is as shown in Figure 7.

We simulated our algorithm with $k = 16$. Remember that the larger the k, smaller is the total number of segments and tails. Ideally we would like the k to be the largest string length. However, since calculation of hash over such a long string is not practical and it would require several Bloom filters to be operating in parallel, we resort to the Aho-Corasick, which is indeed the motivation behind our work. The experiments results in [8] indicate that $k = 16$ is

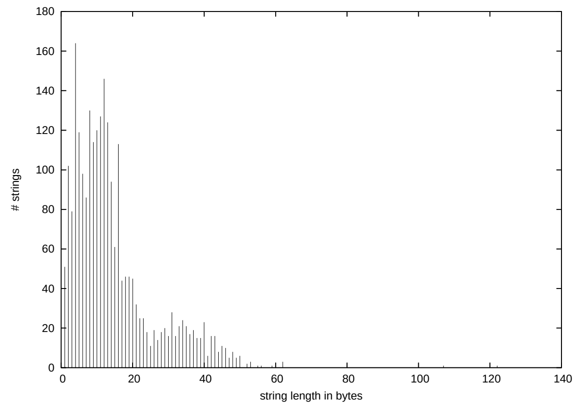

Figure 7: String length distribution. Maximum string length is 122 bytes.

a feasible value. Secondly, around 70% strings are within 16 characters and hence don't need to be broken up (these end up as the tails associated with the state q_0) if $k = 16$. With a JACK-NFA constructed from Snort strings, the resulting number of items, n_i, in i^{th} Bloom filter within an engine of k Bloom filters is as shown in the Table 2.

It is noteworthy that the number of items in k^{th} Bloom filter is larger than the items in each of the other filters. This is expected since all the strings are first broken up in the segments of k characters and thus creating maximum number of items of this length whereas segments of any shorter lengths are just left over tails. The amount of memory to be allocated to each filter depends on the number of hash functions, h, used per Bloom filter. We used two values of hash functions $h = 8, 16$. For a Bloom filter i with number of hash functions h and number of strings n_i, the optimal amount of memory m_i to be allocated is given by the formula [9]:

$$m_i = 1.44 h n_i \tag{8}$$

We round this number to the next power of 2 since usually the embedded memories are available in the power of 2 sizes. The memory allocated to all the Bloom filters for each configuration of hash functions is also shown in the Table 2. The false positive probability of the Bloom filter constructed in this way is given by the formula [9]:

$$f_i = \left(1 - e^{-n_i h / m_i}\right)^h \tag{9}$$

These theoretical false positive probabilities are shown in the table. We created a synthetic text with a string concentration of $c = 0.01$. To achieve this, we scanned all the strings with which we created the JACK-NFA one by one with random characters interspersed in between two strings. The total number of random characters inserted were 1000 times the total characters in the string set. The total number of memory accesses and accesses per character were noted. The observed false positive probability of each Bloom filter was also obtained. The values are as shown in the Table 2. The table clearly indicates that the observed false positive rate of each Bloom filter is very close to the one predicted theoretically.

In scanning a large text with t characters, we spend $tM_s\tau$ time in external memory accesses where τ is the average time for one memory access. Moreover, with r physical engines, we spent t/r clock ticks of the system clock in just shifting the text. If F is the

		hash functions $h = 8$			hash functions $h = 16$		
i	n_i	m_i (bits)	observed f_i	theoretical f_i	m_i (bits)	observed f_i	theoretical f_i
1	70	1024	0.000000	0.000991	2048	0.005719	0.000001
2	76	1024	0.000972	0.001614	2048	0.000000	0.000003
3	146	2048	0.001092	0.001273	4096	0.000000	0.000002
4	228	4096	0.000290	0.000278	8192	0.000000	0.000000
5	173	2048	0.003331	0.003389	4096	0.000010	0.000011
6	138	2048	0.001240	0.000909	4096	0.000000	0.000001
7	127	2048	0.000490	0.000547	4096	0.000000	0.000000
8	172	2048	0.002490	0.003281	4096	0.000010	0.000011
9	131	2048	0.000620	0.000662	4096	0.000000	0.000000
10	156	2048	0.001720	0.001879	4096	0.000120	0.000004
11	159	2048	0.002700	0.002098	4096	0.000000	0.000004
12	172	2048	0.003859	0.003281	4096	0.000010	0.000011
13	155	2048	0.001600	0.001810	4096	0.000000	0.000003
14	123	2048	0.000490	0.000448	4096	0.000020	0.000000
15	94	2048	0.000070	0.000080	4096	0.000000	0.000000
16	968	16384	0.000180	0.000405	32768	0.000000	0.000000

Table 2: Results of Bloom filter construction.

system clock frequency then the total average time spent in scanning t character text is $tM_s\tau + t/rF$ giving us an average throughput of

$$R = \frac{8t}{tM_s\tau + \frac{t}{rF}} = \frac{8}{M_s\tau + \frac{1}{rF}} \quad bps \tag{10}$$

In order to store the off-chip table we will assume the use of a 250MHz , 64bit wide, QDRII-SRAM which are available commercially. We also assume $F = 250MHz$, since latest FPGAs such as Virtex-4 from Xilinx can operate at this frequency which can also operate in synchronization with the off-chip memory. In order to calculate τ we must know the size of the off-chip table entry. We assume 4 bytes for state representation. Also, Instead of keeping the list of all the matching string IDs, we will keep a pointer to such list which can be implemented in a different memory or the same memory. A pointer will make the table entry more compact. Secondly, with $k = 16$ character symbols, the resulting JACK-NFA exhibits a failure chain length of just one, i.e. failure to state q_0 for more than 97% of the states. Hence we will keep space for only one failure state or alternatively a pointer to a chain of states if there are more than one states in the failure chain. Therefore, each table entry can be represented in a 32 byte data structure (4+16 bytes for $\langle state, symbol \rangle$ + 4 bytes for NextState + 4 bytes for Matching Strings Ptr + 4 bytes for FailureChain/Ptr). In a carefully constructed hash table, we require approximately one memory accesses to read one table entry, i.e. two clock cycles of dual data rate 250 MHz 64 bit wide data bus. Hence, $\tau = 8ns$.

We use the values of the theoretical false positive probabilities of all the Bloom filters from Table 2 to compute the theoretical pessimistic average memory accesses per character using Equation 7. Substituting it in Equation 10 we obtain the theoretical throughput value. At the same time we observe the average memory accesses per character during the simulation and use it to compute the observed throughput. These results are shown in the Table 3.

We see from the table that the observed throughput has always been better than the theoretically predicted due to some pessimistic assumptions involved in deriving the theoretical throughput. The results also indicate that we can construct an architecture to scan data at the rates as high as 10 Gbps with just 376 Kbits of on-chip memory. Moreover, we also observe that increasing the number of hash functions gives diminishing returns since with the same amount of memory, an architecture with 8 hash functions per filter can do a better job than the architecture with 16 hash functions per filter. Thus, in this particular case, it is feasible to build several less perfect engines (i.e. with high false positive rate) than to build fewer but near perfect engines (i.e. with very low false positive rate).

7. CONCLUSIONS

We have presented a hardware accelerated version of the Aho-Corasick multi-pattern matching algorithm for network content filtering and intrusion detection applications. We modify the original algorithm to scan multiple characters at a time. We then suppress the *unnecessary* memory accesses using Bloom filters, to speed up the pattern matching. The *logic* resources required to implement our algorithm in hardware are independent of the number of patterns or pattern lengths since it primarily makes use of embedded on-chip memory blocks in VLSI hardware. Due to its reliance on only embedded memory, we argue that it is more scalable in terms of speed and the size of the pattern set, when compared to other hardware approached based on FPGA and TCAM. It is also scalable in terms of pattern length, unlike the earlier Bloom filter based approaches which worked well for only short patterns. With less than 50 KBytes of on-chip memory, our algorithm can scan more than 2000 Snort patterns at more than 10 Gbps data rate.

8. ACKNOWLEDGMENTS

This research was supported by a grant from Global Velocity. The authors of this article have equity and can receive royalty from a license of this technology. John Lockwood is a co-founder and consultant to Global Velocity.

9. REFERENCES

[1] *http://www.snort.org*.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

# Engines (r)	# hash functions(h)	Total on-chip memory bits $\sum m_i$	Theoretical Throughput (Gbps)	Observed Throughput (Gbps)
1	8	47104	1.84	1.88
2	8	94208	3.41	3.57
4	8	188416	5.95	6.45
8	8	376832	9.48	10.8
1	16	94208	1.92	1.96
2	16	188416	3.70	3.87
4	16	376832	6.89	7.49
8	16	753664	12.1	14.1

Table 3: The evaluation algorithm DetectString with Snort string set. A synthetic text was generated with string concentration of one true string in every 100 characters. System operates at a speed of $F = 250MHz$. An off-chip QDRII-SRAM operating at the same frequency was assumed to be available for storing the hash tables. The total number of strings considered were 2259.

[3] Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 223–232. ACM Press, 2004.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM*, 13(7):422–426, May 1970.

[5] Y. Cho and W. Mangione-Smith. Fast reconfiguring deep packet filter for 1+ Gigabit network. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2005.

[6] C. R. Clark and D. E. Schimmel. Scalable multi-pattern matching on high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[7] D. L. Stephen. String Searching Algorithms. In *Lectures Notes Series on Computing*, volume 3, 1994.

[8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood. Deep packet inspection using parallel Bloom filters. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2003.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[11] Y. Sugawara, M. Inaba, and K. Hiraki. Over 10 Gbps string matching mechanism for multi-stream packet scanning systems. In *Field Programmable Logic and Application: 14th International Conference, FPL*, Antwerp, Belgium, Aug. 2004. Springer-Verlag.

[12] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE Infocom*, Hong Kong, China, Mar. 2004.

[13] F. Yu, R. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *IEEE International Conference on Network Protocols (ICNP)*, Berlin, Germany, Oct. 2004.