IEEE Journal on Selected Areas in Communications: Oct. 2006, Volume: 24, Issue: 10, pp. 1781- 1792

Fast and Scalable Pattern Matching for Network Intrusion Detection Systems

Sarang Dharmapurikar, and John Lockwood, *Member IEEE*

***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 required for line-speed packet processing.**

We present hardware-implementable pattern matching algorithm for content filtering applications, which is scalable in terms of speed, the number of patterns and the pattern length. Our algorithm is based on a memory efficient multi-hashing data structure called Bloom filter. We use embedded on-chip memory blocks in FPGA/VLSI chips to construct Bloom filters which can suppress a large fraction of memory accesses and speed up string matching. Based on this concept, we first present a simple algorithm which can scan for several thousand short (up to 16 bytes) patterns at multi-gigabit per second speeds with a moderately small amount of embedded memory and a few mega bytes of external memory. Furthermore, we modify this algorithm to be able to handle arbitrarily large strings at the cost of a little more on-chip memory. We demonstrate the merit of our algorithm through theoretical analysis and simulations performed on Snort's string set.

I. 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 design 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 fast and scalable pattern matching algorithms based on the Bloom filter data structure [4]. A Bloom filter is a memory efficient approximate data structure used to represent a set of strings. It can be queried to verify if an input string belongs to the set of strings stored in it. We store the given set of strings in Bloom filters constructed in a high speed and parallel embedded memory blocks in FPGA. Using these on-chip Bloom filters, a quick check is done on the packet payload strings to see if any of them 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. Since the strings of interest are rarely found in the packets, the quick check in Bloom filter reduces more expensive memory accesses in a hash table search and improves the overall throughput greatly.

However, since the algorithm involves hashing over a maximum length pattern size text window, it works well only for short strings (up to 16 bytes long). In order to enable this algorithm to scale for arbitrary string lengths, we uniquely combine the classic Aho-Corasick multi-pattern matching algorithm with our technique. The resulting algorithm can support matching of several thousands of patterns at more than

10 Gbps with less than 50KBytes of embedded memory and a few megabytes of external SRAM.

The rest of the paper is organized as follows. In the next section we summarize the related work in hardware-based multi-pattern matching. In Section III we describe a simple algorithm based on Bloom filter to match a large number of short strings at very high speed. We provide the analysis of this algorithm in Section IV. We present the evaluation of this algorithm on Snort string set in Section V. Then in Section VI, we describe how the algorithm can be extended to handle arbitrarily long strings. The analysis and evaluation of the modified algorithm is provided in Sections VII and VIII respectively. Section IX concludes the paper.

II. Related Work

Multi-pattern matching is one of the well studied classical problems in computer science. The most notable algorithms include Aho-Corasick 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 [16] or reconfigurable logic/FPGAs [6][14][3][12] 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 [14] seeks to accelerate the Aho-Corasick automation by considering multiple characters at a time. Our approach in essence is similar to this approach 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 [15]. 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 TCAM based solution for pattern matching proposed in [16] 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.

III. A Simple Multi-Pattern Matching Algorithm

In a multi-pattern matching problem, we are given a set of strings $S = \{s_1, s_2, s_3, ..., s_n\}$ and *streaming* data T (which is alternatively called $text$). We would like to find out all the occurrences of any of the strings in S in T. We pre-process the strings in S and build a machine. We feed the streaming data to this machine which then reports matching strings whenever one is found.

Basically, string matching can be abstracted as a Longest Prefix Matching (LPM) problem. Let's denote by $T[i...j]$ the substring of T starting at location i and ending at j. Let's consider the set of strings S_l in which each string has a length l bytes. In order to check if any l byte string in T starting at location i, i.e. $T[i..(i+l-1)]$ matches any of the string in S_l, we simply need to *look up* $T[i...(i+l-1)]$ in S_l. A hash table can be used to perform this lookup quickly with an average of $O(1)$ complexity. However, such a table must be maintained for each set of strings with different lengths. Let L be the largest length of any string in S. When we consider a window of L bytes starting at location i, i.e., $T[i...(i+L-1)]$, we have L possible prefix strings $T[i...(i+L-1)], T[i...(i+L-2)], ..., T[i...i]$ each of which must be looked up individually in the corresponding hash table. After all these prefixes have been looked up we can move the window forward by a single byte such that now we consider bytes from location $i+1$ forming the window $T[(i+1)...(i+L)]$. The same hash table lookup procedure can be repeated for each prefix in this window. Thus, by looking up all the prefixes of a window and forwarding this window by a byte in each iteration we ensure that we scan the entire input to look for the predefined strings with fixed lengths. The hash table accesses can possibly be reduced if one of the prefixes is found to be matching. With each longer prefix string in the hash table, we maintain the list of shorter prefixes that should

match. Thus, when we start our hash table probes from the longest prefix, we do not need to continue them once we find a matching prefix.

Secondly, for applications such as network intrusion detection systems, most of the network data does not contain any string of interest. Hence, when we perform all the L hash table lookups per byte, most of these accesses result in unsuccessful searches. This gives us the opportunity to use Bloom filters to reduce the unsuccessful hash table accesses. Bloom filters are memory efficient and approximate data structures which can represent a set of strings. Such a filter can also be queried to check if a given string is stored in it or not. The filter can answer these queries quickly but with some false positives. With such a filter, a string can be represented approximately with just a few bits, irrespective of the original length of the strings. Due to this compact representation, it is feasible to maintain such a filter in the fast on-chip memory to enable quick lookup. See [9] for more information on the Bloom filters. For the further discussion, it suffices to assume that Bloom filter is a black box that can store a set of strings approximately with hashing and given an input string it can tell if it is present in the set stored in it with a certain false positive probability.

We maintain a separate Bloom filter corresponding to each hash table kept in the off-chip memory. This Bloom filter contains the set of all the strings in the corresponding hash table. Before we execute a hash table query, we first query the corresponding on-chip Bloom filter to check if the string under inspection is present in the off-chip hash table. We proceed to check the hash table only if the filter shows a match. Note that the verification in the off-chip hash table is necessary since Bloom filter can show a false match which will be identified only after hash table access. However, false positives being rare, most of the off-chip accesses are only due to true matches. Figure III shows the block diagram of the system. An array of parallel Bloom filters, each corresponding to a hash table, is shown. Each filter is queried with a prefix of the text window under inspection. The Bloom filters can be engineered to execute one query every clock cycle using the embedded memory Blocks in FPGAs [13]. If some Bloom filters show match for the corresponding prefix, we search those prefixes in the off-chip hash table serially starting from the longest matching prefix. We stop when a successful match is found.

The string matching algorithm is as shown in the pseudo-code of Figure 2.[1]

As described in this pseudo-code, to detect strings, we perform a Longest Prefix Matching (LPM) over a L byte window (line 3) and then advance this window by a byte (line 4). The Bloom filter Lookups (BFL) of line 1-2 in the LPM_1 algorithm are performed in parallel (in a single clock cycle). Since most of the time the match vector of line 3 is empty, the more expensive Hash Table Lookups (HTL) in the off-chip memory are avoided. Therefore, with rare true occurrences of strings in the data stream, our scheme effectively processes one byte per clock cycle. With a system clock of 250 MHz,

[1]Note that a similar algorithm has been applied to the IP route lookup problem in the past [9].

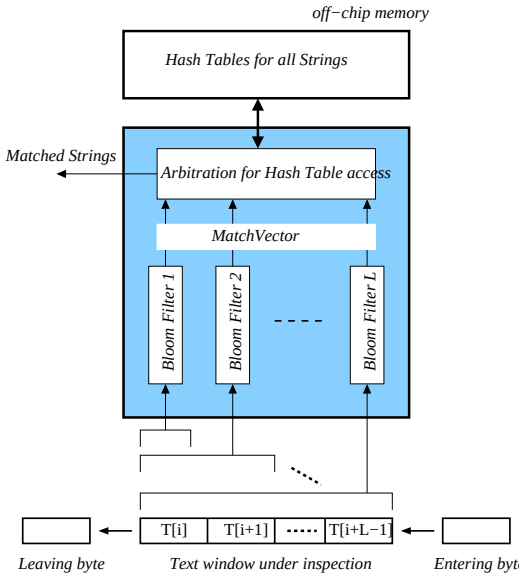

Fig. 1. A string matching machine consisting of multiple Bloom filters each of which detects strings of a unique length. The longest string is of length L. Upon a Bloom filter match, the string is looked up in the corresponding hash table

DetectStrings$_1$
1. **while** (text available)
2. $x = T[i...(i+L-1)]$
3. $\text{LPM}_1(x)$
4. $i++$

LPM$_1$ (x)
1. **for** $(i = k$ **downto** $1)$
2. $match[i] \leftarrow \text{BFL}_i(x[1...i])$
3. **for** $(i = k$ **downto** $1)$
4. **if** $(match[i] = 1)$
5. $\{y[1...i], info\} \leftarrow \text{HTL}(x[1...i])$
6. **if** $(y[1...i] = x[1...i])$
7. **return** $\{info\}$

Fig. 2. String matching algorithm which essentially performs the Longest Prefix Matching(LPM) over the text window.

this results in a throughput of 2 Gbps. Shortly we will consider the true positives and false positives and quantify their effect on the throughput.

The throughput can be improved further by simply deploying multiple identical machines in parallel, each of which scans the input window with a byte-offset. This is illustrated in Figure 3 in which we use $r = 4$ parallel machines. While first machine scans the input window $T[i...(i+L-1)]$, the remaining $r-1$ machines scan windows $T[(i+1)...(i+L)]$ to $T[(i+r-1)...(i+r+L-2)]$ as shown in the figure. Here, each machine can be equipped with its own off-chip hash table or alternatively can share a single hash table. In the first case, we can resolve the matches independently which certainly improves the throughput due to more off-chip memory bandwidth. In the latter case, all the matches are checked in the same hash table. However, since the false

matches of Bloom filters are rare (or can be reduced greatly) a single hash table suffices.

IV. Analysis

We model the efficiency of our architecture in terms of the number of memory accesses required per hash table probe. We will use the following notations to analyze the performance:

- f_i : false positive probability of Bloom filter i
- p_i : the probability that the prefix string of length i being inspected indeed belongs to the set of strings of length i (in other words successful search probability of hash table i or true positive probability of Bloom filter i).
- t_s : memory accesses required for a successful search in any hash table
- t_u : memory accesses required for an unsuccessful search in any hash table

A hash table is accessed for each string that shows a match. Clearly, an off-chip access is made either when a the string being queried is truly present in the hash table or when it is not present but Bloom filter shows a false match. Let T_i denote the cumulative memory accesses required in hash table accesses starting from Bloom filter i down to 1. The cumulative accesses can be expressed using the following recursive relation:

$$T_i = p_i t_s + (1 - p_i)(f_i t_u + T_{i-1}) \quad \text{for } i = B \text{ downto } 1 \quad (1)$$

with $T_0 = 0$. Starting from i^{th} Bloom filter, we execute a successful search in hash table with probability p_i and hence accesses required is $p_i t_s$. Otherwise (i.e. with probability $(1 - p_i)$ we get a false positive match and make an access thereby requiring accesses $f_i t_u$ and we proceed to the next filter result and repeat the whole procedure resulting in accesses T_{i-1}.

The memory accesses due to true positives can not be avoided. Hence if each window of input data under inspection contains a true string then it will result in a degraded throughput. This, in general, is not true for the purpose of NIDS string matching; the strings to be searched appear very rarely in the streaming data. With a carefully constructed hash table, the accesses required for a successful search and an unsuccessful search is approximately the same [10]. Hence we will assume that $t_u = t_s = 1$ one memory access to manipulate the hash table entry. Hence,

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

We can simplify the expression above by deriving a pessimistic upper bound on T_i through the following theorem:

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

This theorem essentially says that the cumulative average memory accesses spent in executing searches from a given Bloom filter downwards is worst when we have to execute the off-chip table access due to the false positive match of each Bloom filter (with probability f_j) and then our search ends at the last Bloom filter (corresponding to the shortest prefix) by consuming accesses T_1. Had the search ended at a longer prefix then we would have required fewer memory accesses. With this pessimistic assumption, the average memory accesses depend on the false positive probabilities of the Bloom filters (which can be tuned by allocating appropriate amount of memory and hash functions) and the true positive probability of just the last Bloom filter, p_1.

When we have L distinct Bloom filters in an engine, the cumulative accesses spent in scanning L byte window is simply T_L. Hence, we have

$$T_L \leq \sum_{j=2}^{L} f_j + T_1 = \sum_{j=2}^{L} f_j + p_1 + (1 - p_1) f_1 \quad (3)$$

When we have r parallel engines, the accesses required is simply rT_L consuming a total time $\tau r T_L$ where τ is the time required for one memory access while manipulating a hash entry. Apart from time $\tau r T_L$, we must spend a clock cycle in moving the window itself. Thus, the total time required to inspect a L byte window is $1/F + \tau r T_L$ where F is the system clock frequency. We will assume that the system clock frequency is the same as that of the off-chip SRAM memory used to store the hash tables. Since we shift r bytes per $\tau r T_L + 1/F$ seconds, the throughput of the system can be expressed as

$$R_r = \frac{8r}{\tau r T_L + 1/F} = \frac{8}{\tau T_L + 1/rF} \quad bits/s \quad (4)$$

The factor of 8 in this equation is due to byte to bit conversion. We will use this equation to compute the throughput of our system for various configuration. In the next section, we evaluate the performance of our algorithm using the Snort rule set.

V. Evaluation with Snort

We used the Snort version 2.0 for our experiments which has 2259 content filtering rules. The total number of strings associated with all these rules is 2412. The string length distribution is as shown in Figure 4.

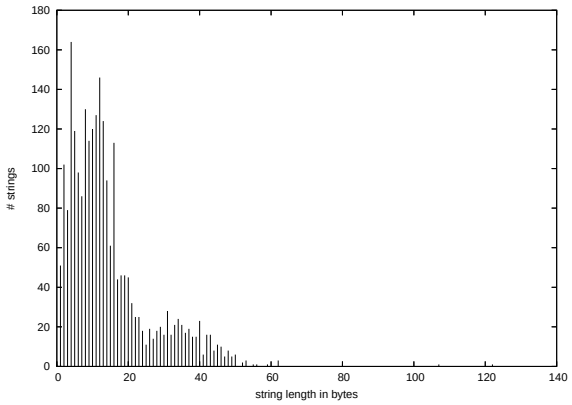

Fig. 4. String length distribution. Maximum string length is 122 bytes.

As this figure shows, Snort rule set contains strings with lengths up to 122 bytes. Since our algorithm requires maintaining a Bloom filter for each string length, we would require

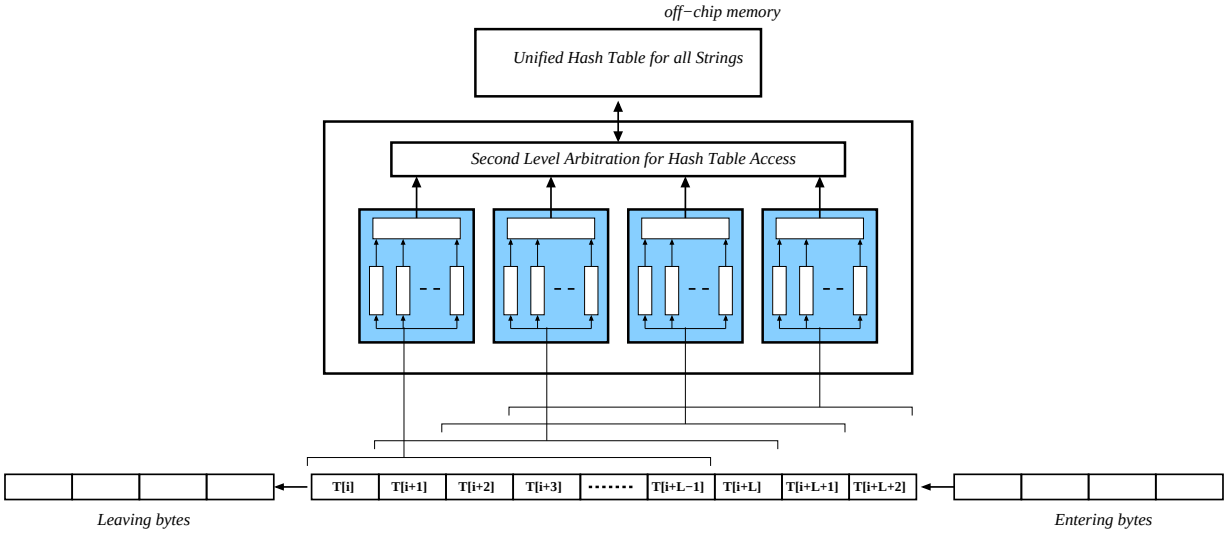

Fig. 3. Instantiating multiple parallel engines for better throughput

several Bloom filters as well as calculation of hash functions over the longest string. Both of these issues degrade the performance of this algorithm from the practical implementation perspective. Therefore the algorithm works well only when we have a small number of unique string lengths as well as short strings. Our implementation experience indicates that strings of lengths up to 16 bytes can be handled efficiently in FPGA hardware. Hence we will use only the 1576 strings of length up to 16 bytes for the purpose of our evaluation which constitute almost 70% of the total number of strings. In the next section we extend our algorithm to handle arbitrarily long strings at the cost of a small amount of extra memory.

We begin by constructing Bloom filters of various configurations for the given set of strings. For this experiment, we used eight or sixteen hash functions per Bloom filter. 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 [11]:

$$m_i = 1.44hn_i \tag{5}$$

We round this number to the next power of 2 since usually the embedded memories are available in the power of 2 sizes. The false positive probability of the Bloom filter constructed in this way is given by the formula [11]:

$$f_i = \left(1 - e^{-n_i h/m_i}\right)^h \tag{6}$$

To store the off-chip table, we will assume the use of a 250MHz , 64bit wide, QDRII-SRAM which is 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. To store up to 16 bytes long strings in one hash table entry we would require two words of this SRAM. With some additional information such as the matching string identifier(s) and the next entry pointer (assuming that hash collisions are resolved by chaining), we would require a few more bytes in the record. We round it up to a total of four words of SRAM (32 bytes) per hash table entry. 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, $\tau = 8ns$ in equation 3. Furthermore, in equation 3, we use $p_1 = 1/100$ i.e. every 100 characters scanned we have a match for one of the shortest strings in our set. This value may seem rather arbitrary, however, it is quite conservative since our experiments with Snort indicate that the string concentration in the network data is as small as 1/20,000.

Using Equation 6, 3 and 4 we can obtain the theoretical pessimistic average system throughput. In order to to see how this system performs practically, we use it to scan a synthetic text composed of random characters interspersed between the strings from the set. We pickup each string from the set and insert random characters between two strings to ensure that there is only one matching string every 100 characters scanned. We term this value as *string concentration*. The results for various configurations are shown in the Table I.

From the table, it is evident that the practical observed throughput is typically more than the corresponding theoretical throughput since the theoretical throughput was computed with some pessimistic assumptions (such as only shortest prefix Bloom filter shows the match). Secondly, it is clear that we can construct a system to scan data at as high as 12 Gbps rate with as small as 220Kbits of on-chip memory. Thirdly, using more hash functions per filter shows diminishing returns. Although the throughput is better with more hash functions, the extra memory needed offsets this gain. For instance, with 220Kbits of memory, we can construct eight engines each having 8 hash functions per filter to give a 12.8 Gbps throughput. With the same memory, we can construct only four engines having 16 hash functions per filter and get the throughput of just 7.8 Gbps. Therefore in this case, with the same amount of on-chip memory, it is more efficient to construct a larger number of engines each of which shows more false positives than

# Engines (r)	# hash functions(h)	Total on-chip memory bits $\sum m_i$	Theoretical Throughput (Gbps)	Observed Throughput (Gbps)
1	8	27648	1.89	1.92
2	8	55296	3.60	3.71
4	8	110592	6.57	6.94
8	8	221184	11.6	12.8
1	16	55296	1.96	1.99
2	16	110592	3.84	3.96
4	16	221184	7.39	7.87
8	16	442368	13.7	15.4

TABLE I

THE EVALUATION OF OUR ALGORITHM 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 1576.

constructing the smaller number of engines showing fewer false positives.

VI. SCALING TO ARBITRARY STRING LENGTHS

A. Basic Ideas

As mentioned earlier, the DetectString$_1$ algorithm works well for short strings and strings with fewer unique lengths. Hash computation over long strings (like 100 characters) consumes more resources and introduces more latency. In this section we extend the algorithm illustrated in the previous section to handle arbitrarily large strings. The basic idea behind this new algorithm is to split up longer strings into multiple fixed length shorter segments and use our first algorithm to match the individual segments. For instance, if we can detect strings of up to length four characters using Bloom filter technique and if we would like to detect a string "technically" then we cut this string into three pieces "tech", "nica" and "lly". These segments can be stitched in a chain and represented as a small state machine with four states q_0, q_1, q_2 and q_3 as shown in Figure 5.

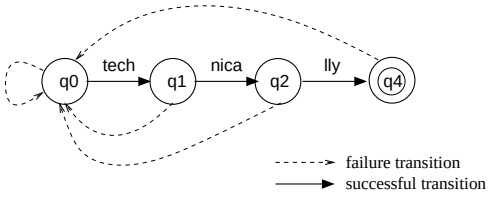

Fig. 5. Illustration of basic technique for handling long strings.

We start from state q_0 and check to see if we get a match for string "tech" in the text stream. Upon a match we go to state q_1. From q_1 we jump to q_2 if the next four text characters match "nica". Otherwise, we jump back to q_0. Similarly, from q_2 go to q_3 if we see a match for "lly" , otherwise we make a failure transition to q_0. Upon reaching q_3 we can declare a complete match for the string "technically". At each state, the matching for the associated segments can be performed using the Bloom filter technique of the previous section.

When this technique is generalized to multiple strings, it essentially results in an Aho-Corasick automaton in which the underlying alphabet consists of symbols formed by group of characters instead of a single character. This can be illustrated with the help of Figure 6. As the figure shows, we first segment each string into k character segments ($k = 4$ for the purpose of illustration) and the left over portion which we call $tails$. We treat each of these k-character segments as a symbol. There are six unique symbols in the given example namely {tech, nica, tele, phon, elep, hant} and the tails associated with these strings are {l, lly, e}.

We then construct the Aho-Corasick Finite Automaton from the strings represented using this *super alphabet*, i.e. the alphabet consisting of k-character symbols. The resulting automaton is shown in the Figure 6(B). We have shown the failure transition to only non-q_0 states for clarity in the figure. All the other states fail to the q_0 state. We now attach the tails of the strings at the appropriate states as shown in the part (C) of the figure. For example, the tails "l" and "lly" are attached to state q_5. However, the tails do not create any state to which the automaton jumps. Tails simply indicate the completion of a string.

To execute pattern matching, we start from state q_0 and look at the k bytes from the text. If this k-character symbol matches with any of the valid symbols associated with the state q_0 then we make a transition to the corresponding state and scan next k characters. At each state, while we look at the k characters, we also consider all the prefixes of these k characters and check to see if any of them matches any of the tails associated with that state. If at any state we get a match for a tail then we report the corresponding string associated with the tail. For instance, from state q_0, if characters "tech" are seen in the text then machine goes to q_1. From q_1 the machine goes to q_2 if the next four characters are "nica". In q_5 we not only try to match all the next four characters but also the prefixes of them. If we get a match for "lly" then we report the string s_2 and s_1 to be matching (since a match for "technically" also implies a match for "technical"). If the machine fails to match a complete k-character segment at any state then it goes to the failure state and retries the same k characters from there. It is possible that the machine follows recursive failure transitions to ultimately reach the state q_0.

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. We leverage the Bloom filter LPM technique described earlier to perform the segment and tail matching. Since this algorithm essentially builds an Aho-Corasick Non-deterministic Finite Automaton (NFA) that *jumps ahead* by k characters at a time, we refer to this machine as Jump-ahead Aho-CorasicK NFA (JACK-NFA).

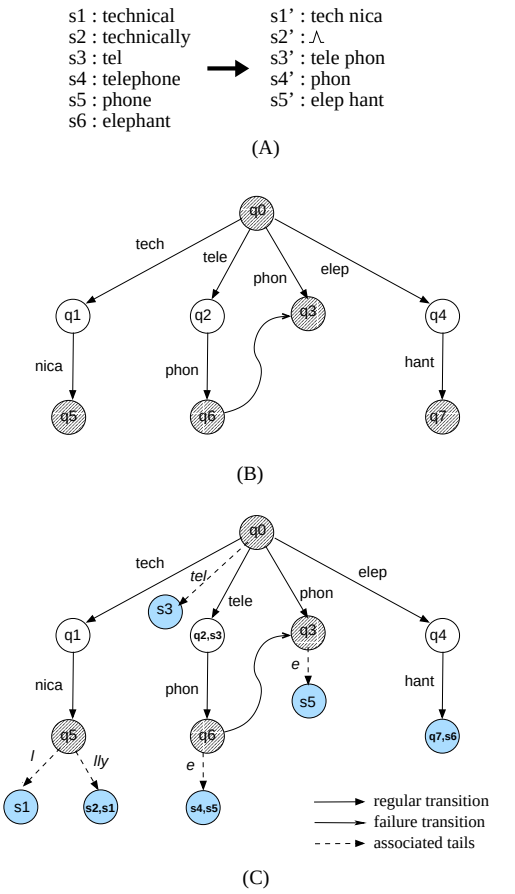

Fig. 6. (A) The original string set and modifi ed string set. The space between the k character boundary is shown as a demarcation (B) Jump-ahead Aho-Corasick (JACK) NFA. The nodes with pattern indicate a state corresponding to a matching substring. Failure transition to only non q_0 states are shown. Failure transitions of rest of the states are to q_0 (C) JACK-NFA with tails associated with states.

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 7. In this figure we use $k = 4$ virtual machines each scans the text by a character offset with respect to the neighbor machines.

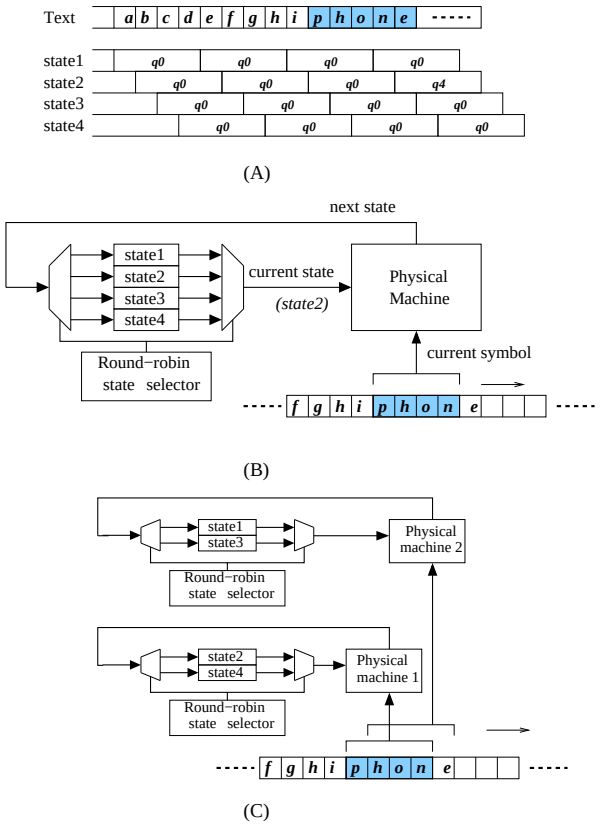

Fig. 7. Illustration of Virtual Machines. (A) Each virtual machine i maintains its $state_i$. This state is updated every k iterations. In each iteration we update k states corresponding to k virtual machines. (B) Machines virtual since only the $state$ variable of each machine is independent. The component that updates the state is the same for all the states. We call it a physical machine. (C) Multiple virtual machines can be implemented using the same physical machine. The fi gure 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 $phone$ since it gets a match for the prefix "e" in the next four characters.

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 the number of virtual machines; they can be less than the number of virtual machines where each physical machine implements multiple virtual machines. To shift the text stream by r characters at a time, we use r physical machines which emulate k virtual machines where $r \leq k$. This can be illustrated with Figure 7(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.

B. Data structures and Algorithm

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 an edge of this NFA. For instance $\langle q_0, tel \rangle$, $\langle q_0, phon \rangle$ etc. are the 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 II. 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 7(B)) with the pseudo-code shown in the Figure 8.

Here, 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 initialized 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

$\langle state, substring \rangle$	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 II

TRANSITION TABLE OF JACK-NFA. THIS TABLE CAN BE IMPLEMENTED IN THE OFF-CHIP MEMORY AS A HASH TABLE.

DetectStrings$_2$

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$_2$($state_j, x$)
6. **report** $strings$
7. $l \leftarrow 0$
8. **while** $(state = NULL)$
9. $\{state, strings\} \leftarrow$ LPM$_2$($state_j.f[l{+}{+}], x$)
10. **report** $strings$
11. $state_j \leftarrow state$
12. $i \leftarrow i+1, j \leftarrow (j+1 \mod k)$

Fig. 8. Algorithm for detecting strings.

the current state of the virtual machine being updated, $state_i$ (line 5). The LPM process (described next) 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 $LPM_2(q, x)$ process to find the longest matching prefix of x which is associated with q. This can be performed in the same same way as LPM_1 algorithm with the exception that we have to check for the prefixes (tails) associated with only the current state of the machine. This can be achieved easily by coupling the segments and tails with the corresponding state as shown in the Table II and storing them in the Bloom filters corresponding to the length of the segment or tail. Thus, the items stored in the Bloom filter are not just the sub-strings but the pairs $\langle state, substring \rangle$. At the most k Bloom filters per engine are required. When we want to probe the filter with a prefix from the text, we concatenate it with the current state before probing as shown in the Figure 9. Hence,

Bloom filter will show a match for a substring at a particular state only if that $\langle state, substring \rangle$ pair is stored in it.

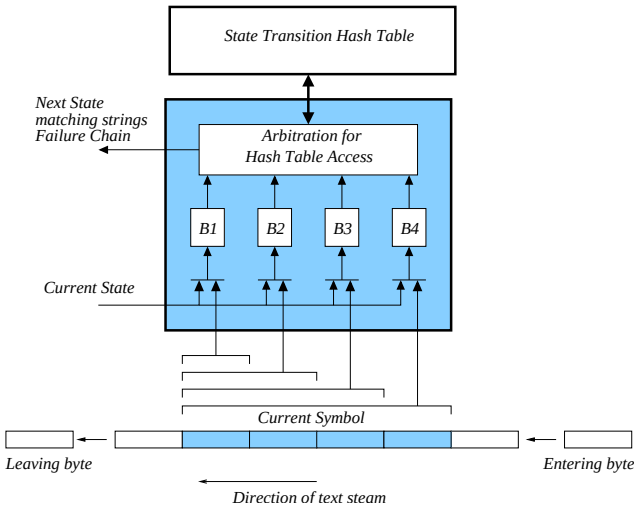

Fig. 9. Implementation of a physical machine. For this figure $k = 4$. The pair $\langle state, prefix \rangle$ is looked up in the associated Bloom filter before off-chip table accesses.

VII. Analysis

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

A. Worst Case Text

We can prove the following theorem regarding the worst case memory access pattern.

Theorem 2: With the algorithm $DetectString_2$ the worst case memory accesses per text character, M_w, are bounded as $M_w \leq 2 + \sum_{i=2}^{k} f_i$
Proof: See Appendix.

By keeping the false positive probabilities f_i moderately low, we can reduce the factor $\sum_{i=2}^{k} f_i$. The worst case memory accesses for the original Aho-Corasick is 2 per character [2]. 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.

B. 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 6, $b_4^{q_0} = 4$, $b_3^{q_0} = 1$. 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, $M_a = T_k$:

$$M_a = \sum_{i=1}^{k} f_i \tag{7}$$

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.

C. Synthetic Text

In subsection VII-A we considered the most pathological text which causes two memory accesses per character and in subsection VII-B, 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 truly only when it appears within a particular context. To quantify the frequency of appearance of the strings in mostly random text, we reuse the parameter *concentration* used earlier in the analysis of DetectStrings$_1$ algorithm which we denote by c. We keep $c = 0.01$ to model our text input

for the analysis of this algorithm as well. 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

$$M_s = M_w c + M_a(1-c) = (2 + \sum_{i=2}^{k} f_i)c + (1-c)\sum_{i=1}^{k} f_i \quad (8)$$

On 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. 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 accesses. Moreover, with r physical engines, we spent t/r clock ticks of the system clock in just shifting the text. If F is the 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

$$R = \frac{8t}{tM_s\tau + \frac{t}{rF}} = \frac{8}{M_s\tau + \frac{1}{rF}} \quad bps \quad (9)$$

Notice the resemblance between Equation 9 and 4. Only the average accesses per character differs in the two cases.

VIII. Evaluation with Snort

We simulated our algorithm with $k = 16$ since the LPM algorithm works well for strings of lengths up to 16 bytes. We used two values of hash functions $h = 8, 16$. 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 100 times the total characters in the string set. The total number of memory accesses and accesses per character were noted.

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, $\tau = 8ns$. First we construct the JACK-NFA and populate the Bloom filters. Then we compute the theoretical false positive probabilities that each Bloom filters should show based on the number items stored in it using Equation 6. Then we use these values to compute the theoretical pessimistic average memory accesses per character using Equation 8. Substituting it in Equation 9 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 III.

Again 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. Like in the previous algorithm, in this case too, 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, it is still feasible to build more but less perfect engines than to build less but near perfect engines (i.e. with very low false positive rate). Finally, we notice that the original algorithm needs around 220 Kbits to achieve a throughput of 12 Gbps for 1576 strings whereas the new algorithm needs around 376 Kbits for a throughput of 10 Gbps for 2259 strings. This indicates that the marginal increase in the memory due to string segmentation is still moderate.

IX. Conclusions

We have presented a high-speed and scalable multi-pattern matching algorithms for Network Intrusion Detection Systems. The algorithms make use of the hardware-based Bloom filters. Our first algorithm essentially performs a Longest Prefix Matching in which we suppress a large fraction of memory accesses using Bloom filters to gain a speed up. We then extend this algorithm to handle arbitrarily long strings by combining it with the Aho-Corasick algorithm. 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. With just 376 Kbits of on-chip memory, our algorithm can scan more than 2200 Snort patterns at more than 10 Gbps data rate.

References

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

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

[3] Z. K. Baker and V. K. Prasanna. Time and area effi cient 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.

# 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 III

THE EVALUATION ALGORITHM DETECTSTRING$_2$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.

[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] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching using Bloom filters. In *Proceedings of ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[10] H. Song, S. Dharmapurikar, J. Turner, and J. W. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filters: An Aid to Network Processing. In *Proceedings of ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[11] 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.

[12] 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.

[13] M. Attig, S. Dharmapurikar, and J. Lockwood. Implementation Results of Bloom Filters for String Matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[14] 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.

[15] 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.

[16] 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.

APPENDIX

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

proof: The proof is by induction on i using Equation 2. For $i = 2$,

$$\begin{aligned} T_2 &= p_2 + (1 - p_2)(f_2 + T_1) \\ &\leq f_2 + T_1 \end{aligned} \tag{10}$$

For $i = 3$,

$$\begin{aligned} T_3 &= p_3 + (1 - p_3)(f_3 + T_2) \\ &\leq p_3 + (1 - p_3)(f_3 + f_2 + T_1) \\ &\leq f_3 + f_2 + T_1 \end{aligned} \tag{11}$$

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

$$T_j \leq \sum_{l=2}^{j} f_l + T_1$$

Hence,

$$\begin{aligned} T_{j+1} &= p_{j+1} + (1 - p_{j+1})(f_{j+1} + T_j) \\ &\leq p_{j+1} + (1 - p_{j+1})(f_{j+1} + (\sum_{l=2}^{j} f_l + T_1)) \\ &\leq (\sum_{l=2}^{j+1} f_l + T_1) \end{aligned}$$

Hence the proof. •

A special case of equation 2 with $p_k = 0$ 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

$$T'_k = f_k + T_{k-1} \tag{12}$$

where T_{k-1} is given by the recursive relation of Equation 2. As a result of Theorem 1, the following holds

$$T'_k \leq T_1 + \sum_{i=2}^{k} f_i \leq 1 + \sum_{i=2}^{k} f_i \tag{13}$$

Theorem 2: With the algorithm $DetectString_2$ the worst case memory accesses per text character, M_w, are bounded as $M_w \leq 2 + \sum_{i=2}^{k} f_i$

Proof: Assume that the machine has processed t characters of the text. There are k virtual machines. 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

$$y_i = \begin{cases} \lceil \frac{t-(i-1)}{k} \rceil & \text{for } t \geq i \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

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

$$w_i = y_i(T'_k + 1) - 1 \tag{15}$$

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 13, 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,

$$M_w = 2 + \sum_{i=2}^{k} f_i \tag{16}$$

•

Sarang Dharmapurikar received his B.E. in Electrical and Electronics Engineering from Birla Institute of Technology and Science (BITS), Pilani, India in 1999. He worked for Wipro Global R&D, Bangalore, India from June '99 to June 2000 as a VLSI Systems designer where he was part of a team which designed Gigabit router chips. He is currently a doctoral student in the Computer Science and Engineering Dept. of Washington University in St. Louis. He is a member of the Applied Research Laboratory (ARL) where he is doing research on different aspects of very high speed networking devices. Particularly, his research interests include algorithm design and hardware implementation of high-speed deep packet processing systems.

John W. Lockwood designs and implements networking systems in reconfi gurable hardware. Lockwood and his research group developed the Field programmable Port Extender (FPX) to enable rapid prototype of extensible network modules in Field Programmable Gate Array (FPGA) technology. He is an assistant professor in the Department of Computer Science and Engineering at Washington University in Saint Louis. He has published over 60 papers in journals and technical conferences that describe technologies for providing extensible network services in wireless LANs and in high-speed networks. Professor Lockwood has served as the principal investigator on grants from the National Science Foundation, Xilinx, Altera, Nortel Networks, Rockwell Collins, and Boeing. He has worked in industry for AT&T Bell Laboratories, IBM, Science Applications International Corporation (SAIC), and the National Center for Supercomputing Applications (NCSA). He served as a co-founder of Global Velocity, a networking startup company focused on high-speed data security. Dr. Lockwood earned his MS, BS, and PhD degrees from the Department of Electrical and Computer Engineering at the University of Illinois. He is a member of IEEE, ACM, Tau Beta Pi, and Eta Kappa Nu.