Self-Addressable Memory-Based FSM: A Scalable Intrusion Detection Engine

Benfano Soewito, Lucas Vespa, Atul Mahajan, Ning Weng, and Haibo Wang,
Southern Illinois University

Abstract

One way to detect and thwart a network attack is to compare each incoming packet with predefined patterns, also called an attack pattern database, and raise an alert upon detecting a match. This article presents a novel pattern-matching engine that exploits a memory-based, programmable state machine to achieve deterministic processing rates that are independent of packet and pattern characteristics. Our engine is a self-addressable memory-based finite state machine (SAM-FSM), whose current state coding exhibits all its possible next states. Moreover, it is fully reconfigurable in that new attack patterns can be updated easily. A methodology was developed to program the memory and logic. Specifically, we merge "non-equivalent" states by introducing "super characters" on their inputs to further enhance memory efficiency without adding labels. SAM-FSM is one of the most storage-efficient machines and reduces the memory requirement by 60 times. Experimental results are presented to demonstrate the validity of SAM-FSM.

With an explosive increase of Internet traffic and a more demanding Internet security requirement, designers of future generations of network intrusion detection systems (NIDSs)face more difficult challenges in meeting the ever-stringent design requirements: detecting malicious packets with high fidelity, performing intrusion detection at full wirespeed, and enough flexibility to enable detection rules to be updated easily. NID is classified as either anomaly detection or signature detection. Anomaly detection identifies aberrant events by analyzing network traffic using statistical, machine learning, or data-mining techniques; a signature-based NIDS finds suspicious activities by comparing packets with a pattern database of known attacks. Due to its relatively simple implementation and lower false-positive rate for known attacks, signature-based NID is highly popular.

Essential to signature-based NID is a string-matching algorithm. To determine if an attack exists, the string-matching algorithm compares packets with a rule set comprised of fingerprints of known attacks. There are 3305 such rules defined by Snort 2.4. Each rule consists of two types of strings to be matched: one type comprises header strings with a determined position in the packet header (e.g., source/destination network address and source/destination port number); another type comprises payload strings with a probabilistic position in packet payload (e.g., network worms and computer viruses). A suspicious activity is detected when both header strings and at least one of the payload strings are matched in the packet.

A string-matching algorithm is a computationally intensive task because it checks every byte of a packet (both header and payload) to see if it matches one of a set of thousands of possible attack patterns. Obviously, algorithms [1–3] on a commodity-processor approach do not scale well with increasing line rates of network traffic. To deliver the required high performance, specific techniques such as hashing [4], Bloom filtering [5], and pre-filtering [6] were employed to optimize the average case; however, this average-case optimization might be attacked easily by sending many rare-case packets. A customized hardware accelerator [7] and a rule-based comparator [8] are alternatives to deliver worst-case guaranteed performance. Meanwhile, new attacks are created all of the time. Accordingly, new rules to detect attacks are increasing dramatically in the Snort database [9], and we are expecting this trend to continue. This observation implies that we must design a system that is flexible enough to add new patterns without impacting overall system performance.

The main contribution of this work is a self-addressable memory-based finite state machine (SAM-FSM). It includes a simplified memory structure and a novel address tag-based encoding methodology that results in dramatic memory-size reduction. By taking advantage of the proposed memory-decoding structure, FSM state-collapsing techniques are developed to further reduce memory size. Hardware structures to support FSMs with super states resulting from state collapsing are developed. The design enables the FSM to receive multiple characters as input within one clock cycle when collapsed substrings are detected, which increases system throughput. Finally, pipelined string-matching operations dramatically increase system throughput by enabling a single string-matching engine to process multiple packets simultaneously.

The remainder of the article is organized as follows. The next section briefly discusses related work. We then describe the proposed SAM-FSM engine. Methods to perform state encoding for the SAM-FSM also are discussed in this section. By taking advantage of this proposed hardware architecture, a state-collapsing method is developed in the following section

 0890-8044/09/$25.00 © 2009 IEEE

to further reduce memory size in the implementation. In addition, we discuss techniques to further enhance system programmability and throughput. Experimental results are presented, and the article is concluded in the final section.

Related Work

A flurry of work was proposed to design a high-performance string-matching engine. However, few works consider memory efficiency and the ease of updating new patterns. We present background and then the work that is most related to SAM-FSM.

Deterministic Finite Automata

Deterministic finite automata (DFA) can match multiple strings simultaneously, in *worst-case* time linear to the size of a packet. Figure 1 shows an example DFA used to match "SHE," "HERS," and "HIS." Starting at state *S*0, the state machine is traversed to state *S*1 or *S*4 depending on whether the input character is "S" or "H." When an end state is reached, a string is said to be matched. In the example in Fig. 1, if state *S*7 is reached, string "HERS" is matched.

Each state in the machine has pointers to other states in the machine. If an input character is the next character in a string that is currently being matched, the algorithm moves to the next state in that string; otherwise, the algorithm follows a failure pointer to the first state of another string that begins with that character or to the initial state of the machine if no other strings begin with that character. An example of this can be seen in Fig. 1. If the current state of the machine is *S*5, the last input characters would have been "HE." If the next input character were to be "R," then the next state would be *S*6. If the next input character were not "R," but instead, "S," then the next state would follow a failure pointer to state *S*1, which is the starting point for the string "SHE."

Storage Requirement of Traditional DFA

A DFA can be implemented using hardware or memory. In a memory-based implementation of a DFA, the current state and input character are used as the memory address location of the memory content. The memory content consists of the next state and tag. A memory implementation of a DFA can be reconfigured easily by reprogramming the memory with a new or updated state transition table.

In a memory-based implementation of a DFA, the memory size required to hold the state transition table is based on the number of bits required to represent each state S and the number of bits required to represent each input character (8-bit). For example, Snort Dec. 05 has 2733 patterns that require 27,000 states to represent them so the storage requirement is about 13 megabytes, which is too big to fit onto on-chip memory.

The amount of memory required to store a DFA is large and increases greatly as the number of strings being matched increases. The reason for the large memory requirement in traditional memory-based FSM is that all possible 256 next states of any given state are explicitly stored in the memory array even though many of these next states are the same.

Memory Efficiency Optimization

The Aho-Corasick (AC) algorithm [1] can match multiple strings simultaneously by constructing a state machine. Starting from an empty root state, each string to be matched is represented by a series of states in the machine, along with pointers to the next appropriate state. A pointer is added from each node to the longest prefix of that node that also leads to a valid node in the machine. The major drawback of the AC algorithm is a possible 256 fan-out, which results in low memory efficiency.

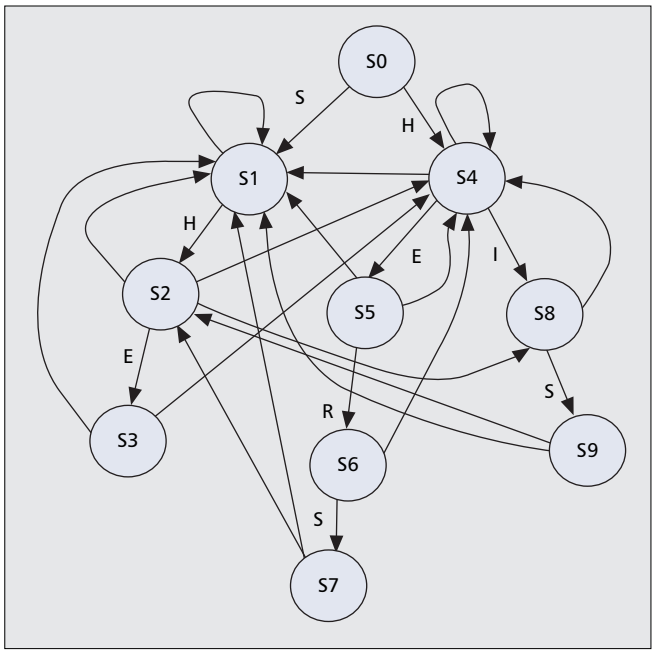


Figure 1. *Finite state machine diagram. This machine is constructed to match patterns: "SHE," "HERS," and "HIS" (failure edge labels are omitted).*

Bitmap and compression [10] were proposed to optimize the AC algorithm data structure to improve the memory efficiency. The problems of bitmap compression require two memory references per character in the worst case and 256 bits per bitmap. A potential problem with path compression is that failure pointers may point to the middle of other path-compressed nodes.

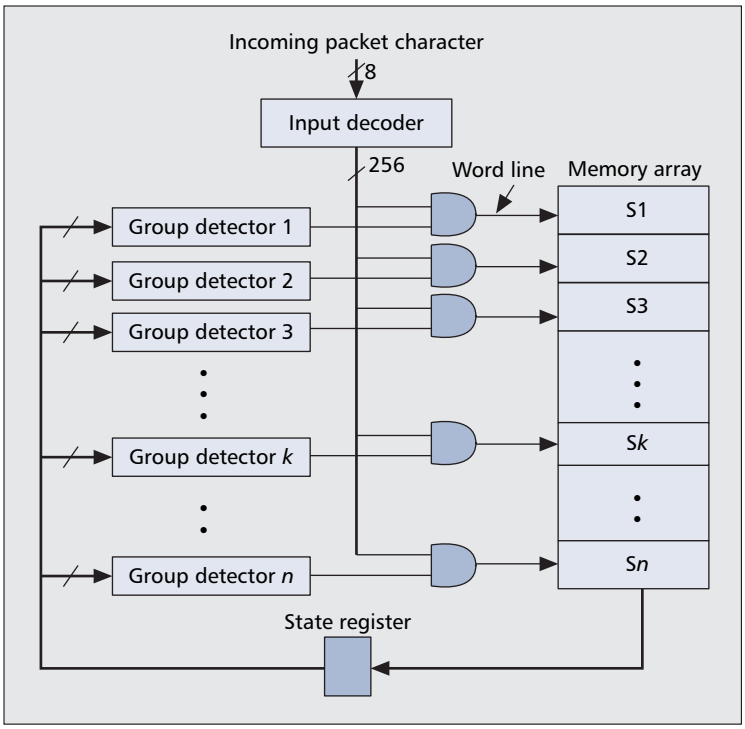
Lunteren [11] introduced a pattern-matching engine, balanced randomized tree (BaRT) FSM pattern matching (BFPM), which utilizes a BaRT-based FSM to provide storage efficiency and a pattern compiler to support dynamic updates. The substantial gain in storage efficiency is due to state transition rules that involve wildcards and priorities; however, simply using wildcards might require more than one memory access to scan the incoming packet character. In contrast, SAM-FSM requires one memory access for each incoming packet character.

Tan and Sherwood [12] proposed a memory-efficient, string-matching engine based on pattern partitioning and the bit-split algorithm. This engine works by converting the large database of strings into many groups called rule modules; inside each module, many tiny state machines are employed to search for a portion of the bits for each pattern. This proposed architecture is capable of high performance; however, the bit-split algorithm requires a hot-coding of a partial match vector in each tile. Also, the patterns are divided into small groups that consist of 16 patterns. This finer granularity of pattern partitioning might increase the complexity of implementation and updating. SAM-FSM does not require duplicating all the characters from the input stream because our decoder (input decoder) automatically selects one of the 256 bits based on the input character.

We further compare SAM-FSM with these optimization techniques later in this article.

SAM-FSM

SAM-FSM hardware architecture is based on the following observation. In a straightforward memory-based FSM implementation for a given state, 256 next-state pointers are stored



■ Figure 2. *SAM-FSM architecture.*

in the memory. Many of them are the same data entries and hence, waste memory resources. If we can design a metapointer for each state that points to all the possible next states, we can store a single pointer for each state and consequently, save a great portion of the memory. Because the metapointer indicates multiple states, the next-state selection can be performed by a special memory decoder according to the metapointer and FSM inputs. The proposed hardware architecture and FSM encoding scheme are discussed as follows.

SAM-FSM Architecture

A metapointer, which we also refer to as a state code, is a sequence of binary numbers that can be partitioned into different segments (also called clusters). For an FSM state, the information contained in its clusters, which is referred to as an address tag in this article, indicates its possible next states. For example, if the code of state S_i is comprised of three clusters c_1, c_2, and c_3, the binary bits contained in c_1, c_2, and c_3 indicate the three possible next states of state S_i. This is why SAM-FSM is referred to as self-addressable memory — the next-state codes will be known based on the current state coding.

The hardware architecture is shown in Fig. 2. It consists of a memory array, a state register, group detectors, and an input decoder. Similar to other memory-based FSMs, the memory array stores state codes (metapointers), and the state register stores the current state. The group detectors detect signatures contained in the code of the current state and identify all the possible next states (by pulling the corresponding decoder output to logic 1). Each group detector circuit detects only one address tag and for a given current state, multiple group detector outputs can switch to logic 1. The input decoder decodes the 8-bit FSM input into a 256-bit one-hot code that controls which logic-1 group detector output can be passed to the memory array (to access the corresponding memory location) according to the FSM input. The outputs of the AND gates directly drive the word lines of the memory array. There is no address decoder inside the memory array. This is a fundamental difference between SAM-FSM and other memory-based FSM implementations. In this work, we modify the memory circuit at low level to improve memory efficiency. For the convenience of discussion, we refer to the combination of group detectors, input decoder, and the AND gates as the memory decoder circuit in the following discussion.

SAM-FSM does not cause any hardware overhead compared to conventional design. The size of the memory array is dramatically reduced by eliminating repeating entries. The group detector circuits are smaller than the conventional memory address decoding circuits because their output logic depends on fewer input variables. The input decoder circuit is the only extra circuit in our design when compared to the conventional memory circuit. However, the overhead introduced by the input decoder is negligible compared to the reduced memory array size. As indicated later in this section, the SAM-FSM potentially leads to very wide memory word, which potentially increases signal propagation delay on memory word lines. This problem can be solved by using pipelined operations, which are discussed later.

SAM-FSM State Encoding

In SAM-FSM, the codes of FSM states are very similar to chromosomes that consist of a DNA sequence. An FSM state code consists of a set of signatures that indicate all possible next states. An FSM state S_i has a corresponding address tag ς_i that should be included in the codes of all the states whose possible next states include S_i. We denote this group of states as G_i. Due to the previous relation, we also refer to ς_i as the address tag of Group G_i. In the following discussion, we frequently use the concept of group to indicate all the FSM states that share a common possible next state. Because a state can have multiple possible next states, a state can belong to multiple groups.

An important question in state encoding is how to place group signatures along the binary bits of the state code. To simplify the decoder design in SAM-FSM, the address tag of a group must be placed on the same positions of all the state codes that contain this address tag. This is the only constraint in the address-tag placement problem. The simplest method is to use one-hot encoding: each group address tag takes a fixed location; if the position contains datum 1, it means the address tag is contained in the code. Otherwise, the position is filled by a value of 0. However, this simple method dramatically increases the memory size.

Some interesting properties regarding group address-tag placement were observed in our study. If two groups have a common state, S_i, the code of S_i must contain the signatures of both groups. Thus, the two signatures must be placed into different positions for all state codes. However, if two groups do not share a common member (we reference the two groups independent of each other), the signatures can take the same position in state codes because there are no states that contain both signatures. This property leads to a state encoding procedure discussed as follows. We apply the algorithm listed in Algorithm 1 to find all mutually independent group sets. For all members of a mutually independent group set, the signatures can be placed in the same location, referred to as a cluster in this article. We can apply binary encoding to generate signatures for the groups within a cluster. The order of clusters can be arbitrary. After the group signatures are generated, the code of each state will be a collection of group signatures following the fixed cluster order.

In line 4 of the pseudo code of the algorithm, a state group procedure is performed as follows. For each FSM state, S_i, place all fan-in states of S_i into a group. After this procedure,

```
   Input: State Trans. Table: T
   Result: Indep. Group Set R
1  Search_Indep_Group( T)
2  begin
3      R = θ ;
4      G = StateGrouping(T) ;
5      Ģ = Create_Graph( G );
6      while |Ģ| ≠ θ do
7          c = MaxClique(Ģ) ;
8          add c to set R ;
9          remove c from Ģ;
10     end
11 end
12 Create_Graph( G)
13 begin
14     while G ≠ θ do
15         remove a group g_i from G ;
16         add a vertex v_i for group g_i on Ģ;
17         for vertex Vv_j on Ģ do
18             if g_j ∩ g_i = θ then
19                 add a edge between v_j and v_i
20             end
21         end
22     end
23 end
```

■ Algorithm 1. *Algorithm for searching mutually independent groups.*

N groups are generated, where N is the number of FSM states. In line 5, a group graph is generated following the procedure described by lines 12–23. In line 7, a well-known Maximum Clique search algorithm is used to recursively search the maximum cliques of the graph. The vertices of the obtained cliques represent the mutually independent groups. The above FSM state encoding procedure is further illustrated by the following example.

The first step is to group all the nodes in the FSM that have the same next state into one group, group G1[S0, S1, S2, S3, S4, S5, S7, S9], G2[S1, S7, S9], G3[S2], G4[S0, S2, S3, S4, S5, S6, S8], G5[S4], G6[S5], G7[S6], G8[S2, S4], and G9[S8]. Now we are ready to build a new graph from these groups that is called a group graph. If there is no same state between two groups Gi and Gj, an edge is created between Gi and Gj. For example, G4 of the group graph has only one edge, the edge to node G2, because there is at least one member of group 4 (node G4) that is also a member of another group, except group 2, the only group that shares no members with group 4.

The second step of our methodology is to cluster the groups from the group graph by taking advantage of the Clique Algorithm, in other words, to find a cluster with maximum clique or groups without repeating states. The clustering result is C1[G1], C2[G2, G3, G5, G6, G7, G9], C3[G4], and C4[G8]. In the first iteration of the algorithm, Cluster C1 has three members: G1, G7, and G9. All three nodes share an edge with each other. The algorithm looks for the maximum clique or groups in a cluster. In this case, G7 and G9 also can belong to cluster C2, which has more members than cluster C1. The clique algorithm moves G7 and G9 to cluster C2, increasing the number of members of cluster C2 and decreasing the number of members of cluster C1 — forming the largest possible clusters — the goal of the Clique Algorithm. The bit length of a cluster is the size in bits required to represent each group in the cluster. For example, Cluster C1 has only one member; therefore we require only one bit to represent its groups. Cluster C2 has six members so we require three bits to accommodate them. The encodings for the groups are G1(1), G2(001), G3(010), G4(1), G5(011), G6(100), G7(101), G8(1), and G9(110). The encoding begins with 1 not 0; for example, group G2 is encoded as 001, not 000. G3 is then encoded as 010 and so on.

The last step is to encode each state by concatenating the group codes for each state. The encoding for each state is shown in Table 1. S2 is contained by groups G1, G3, G4, and G8. The index or encoding for S2 is formed by concatenating the codes for these groups. The codes are: 1, 010, 1, 1, and the final concatenated index for S2 is 101011, as shown in Table 1, column 3.

Memory Size Reduction through State Collapsing

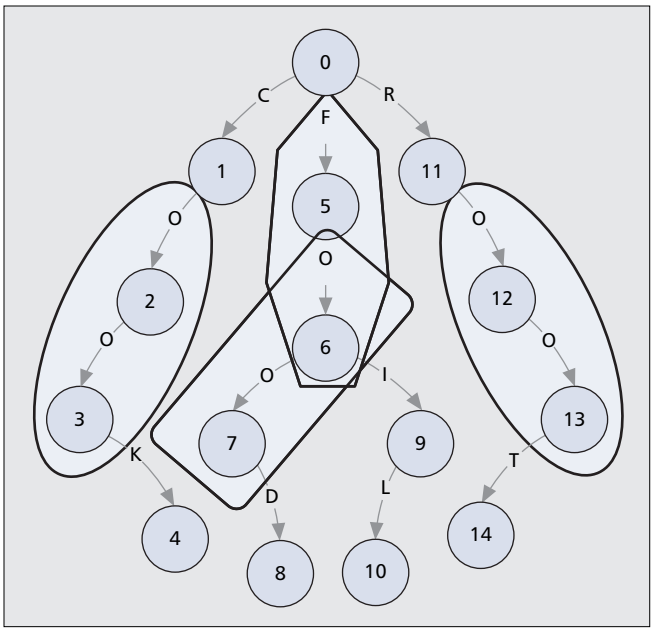
The proposed hardware architecture and state-encoding scheme not only reduce memory size but also simplify the memory-decoder circuits. By taking advantage of the special memory-decoding scheme in SAM-FSM, we can further reduce the memory size through FSM-state collapsing. The basic idea behind state collapsing is to find commonly occurring substrings (referred to as super characters) and collapse the states corresponding to the occurrence of such substrings into a single state. A substring is qualified for state collapsing operation only if its corresponding states have only one fan-out branch, except the state corresponding to the last character. For example, Fig. 3 shows an FSM for patterns *COOK*, *FOOD*, *FOIL*, *ROOT*. The valid super characters are marked by ovals in the figure, whereas the substring surrounded by a rectangle is not a valid super character.

When performing state-collapsing operations, the length and number of occurrences of substrings are important parameters. Such parameters can be selected empirically, based on our experimental results. With the selected parameters, the search of valid super characters can be performed as follows. First, locate all possible valid substrings within the pattern set, up to length N. Second, calculate the occurrence of each substring and sort substrings in descending order by length/occurrence. Finally, replace all states for each valid substring occurrence with one state, referred to as a super state.

To enable the FSM to move from a state to a super state

State	Group	Code
S0	–	0 0 0 0 0 0
S1	G1 G2	1 0 0 1 0 0
S2	G1 G3 G4 G8	1 0 1 0 1 1
S3	G1 G4	1 0 0 0 1 0
S4	G1 G5 G4 G8	1 0 1 1 1 1
S5	G1 G6 G4	1 1 0 0 1 0
S6	G7 G4	0 1 0 1 1 0
S7	G1 G2	1 0 0 1 0 0
S8	G9 G4	0 1 1 0 1 0
S9	G1 G2	1 0 0 1 0 0

■ Table 1. *Concatenated group.*



■ Figure 3. *FSM with circled super-characters.*

within one clock cycle, a look-ahead decoder is required. Figure 4 shows an example of this decoder, which can check for occurrences of up to four character-length super characters. Compared with Fig. 2, the look-ahead decoder requires multiple input decoders, one priority decoder, and one shift register. In addition, the word line *AND* gate takes multiple fan-ins, and its output not only determines the next state but also the stride of shift operation. For example, if $M4_1 = 1$, then the next state is S_7, and shifting stride is 4 ($P_1P_0 = 11$). If no predefined super characters are recognized, only one packet character is processed and shifted along the shift register $D_3D_2D_1D_0$ per clock cycle. In this base mode, $P_1P_0 = 00$. However, if a predefined super character (e.g., ABCD) is recognized, regular shifting operation is broken, and packet character D_4 is forwarded directly to flip-flop 0.

The proposed technique not only reduces memory size but also makes it possible for the FSM to process multiple characters within one clock cycle, which increases the system throughput. However, the selected super character length increases the complexity of the look-ahead decoder. Furthermore, the occurrence of valid super characters in real network traffic affects how effectively the decoder circuit is utilized during FSM operations.

Techniques for Enhancing System Programmability and Throughput

This section presents techniques to further enhance the programmability and throughput of SAM-FSM. We first discuss techniques to improve system programmability by using programmable decoder circuits and an address overlap scheme. We then develop techniques to improve system throughput by enabling a single-string matching engine to process multiple packets simultaneously.

Enhancing System Programmability

An important NIDS design requirement is the ability to update the system easily to include new attack patterns. Because SAM-FSM is a memory-based implementation, new attack patterns can be added to the system by updating memory contents. After the system is deployed in the field, its programmability — measured by the number of new attack patterns that can be added to the system — is affected by the available memory of the system at a given time. Hence, memory resources should be utilized efficiently when adding new attack patterns. This can be accomplished with the help of programmable memory decoder circuits and overlapping decoder address inputs. The need for such techniques is illustrated by the following example.

Initially, Cluster 1 contains 12 groups and hence takes four bits for embedding group signatures. Also, Cluster 2 has seven groups and requires three bits for group signatures. After a series of updates, we assume the numbers of groups belonging to Clusters 1 and 2 become six and 14, respectively. If no programmability is implemented on the memory-decoding circuit, the number of bits used for each cluster must remain the same. Then, one bit in the Cluster 1 code is wasted because only three bits are required to enumerate the six groups in Cluster 1. Meanwhile, the three bits in Cluster 2 can represent only seven of the 14 groups. The other seven groups must be placed into a new cluster. However, if the fan-in of memory-decoder circuits can be reconfigured, the number of bits used for a cluster can be adjusted during update. In the above example, the additional bit in Cluster 1 can be reassigned to Cluster 2. Consequently, no new clusters must be created, and memory efficiency is improved.

Reassigning an address bit from one group (decoder) to another can be accommodated easily by using programmable decoder circuits and an address overlapping scheme. The programmable decoder circuit can be programmed to selectively ignore certain address inputs during decoding operations. The design of such a programmable decoder circuit is not complicated and does not result in significant hardware or performance overhead.

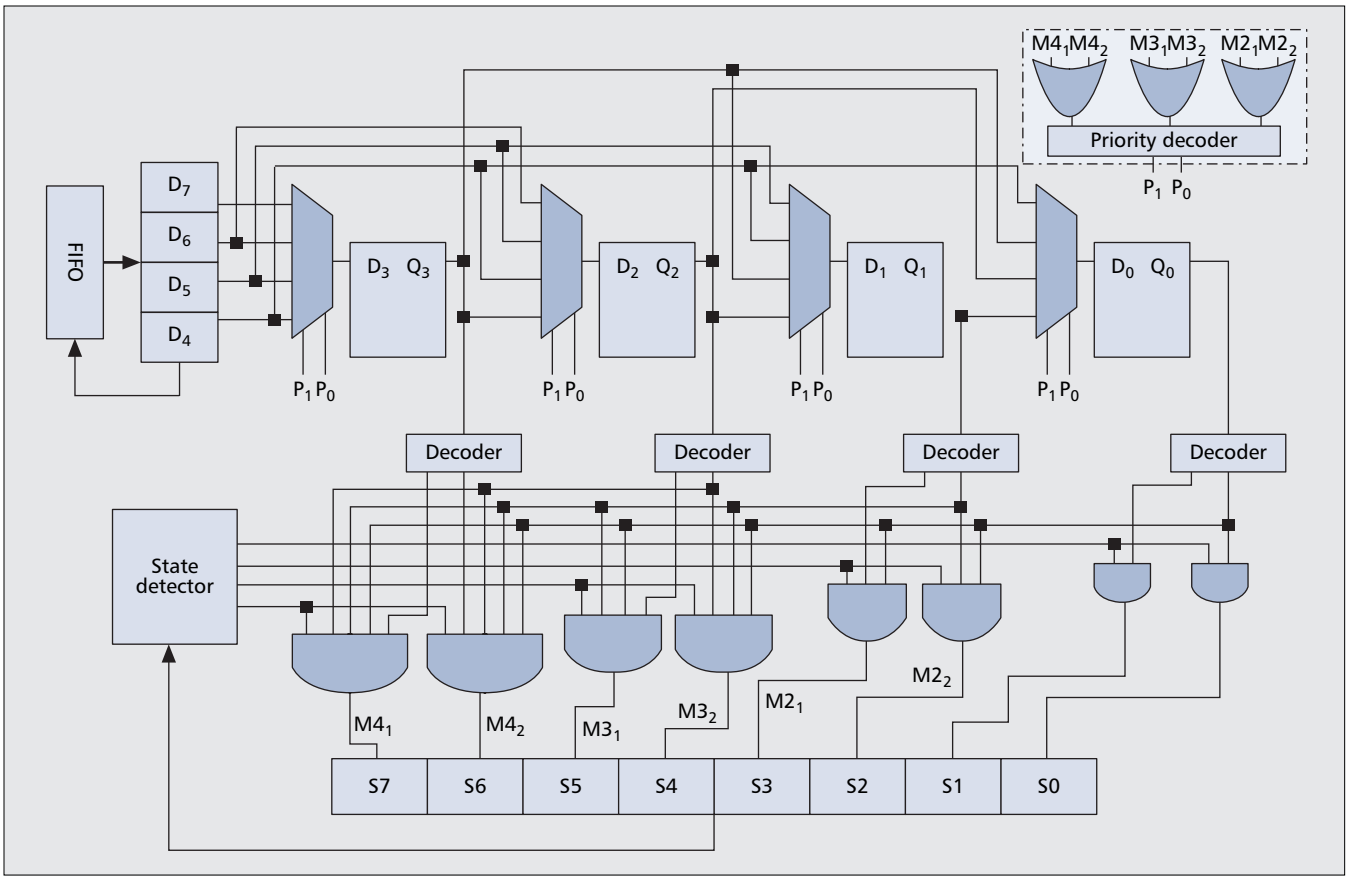
Enhancing System Throughput

In network intrusion detection applications, string matching operations for different packets are independent of each other. By taking advantage of this fact, high-throughput string-matching operations can be achieved by using pipelined operations that enable a single matching engine to simultaneously check multiple packets. This drastically increases system throughput. Although it is a very effective approach to improve the throughput of string matching systems, pipelined operations rarely were exploited in previous NIDS design. To accommodate pipelined operation in SAM-FSM, pipeline registers can be inserted between decoder circuits and memory subarrays.

Figure 5 shows an example pipelined system and a snapshot processing four packets in parallel. In the pipelined system, we use a four-to-one multiplexer to alternately feed data from four packets to the FSM. In the snapshot, we use $P_k[i]$ to represent the i_{th} byte of Packet k. $S_k[i]$ represents the current state of the FSM after the i_{th} byte of packet k is processed. The pipelined implementation does not reduce the processing time (latency) for individual packets but significantly improves system throughput, which is a more important performance parameter in network applications. With this deep pipeline scheme, SAM-FSM can operate with a clock frequency above 1 GHz, which results in a system throughput around 10 Gb/s using only one string-matching machine.

Results

According to the proposed algorithm, a simple FSM compiler for SAM-FSM is developed. It takes the state transition table of a typical DFA as input. Consequently, the compiler groups FSM states, partitions state groups into clusters, generates group signatures and performs state encoding. In our experiments, we randomly take subsets of Snort rules to test the



■ Figure 4. *Look-ahead decoder for super-characters.*

developed compiler. The number of rules contained in the subsets ranges from several to 500 patterns.

With the help of the developed compiler, we further conducted experiments in three directions. First, we implemented SAM-FSM on an FPGA platform and performed circuit simulation to validate the proposed hardware architecture and compiler. Then, we estimated our system throughput on both a field programmable gate array (FPGA) device and customized hardware. Finally, we performed experiments to study the memory resources required by SAM-FSM and compared our method with other approaches.

Architecture Validation

In the circuit implementations, we use FPGA block memory as the memory component of the FSM and use FPGA configurable logic blocks (CLBs) to implement group decoder circuits. In SAM-FSM, the decoder output should directly drive the word lines of the memory array. However, Xilinx block memories do not provide direct access to the memory word lines. Thus, in our implementation, we use an additional encoder circuit to convert the one-hot decoder outputs into a regular binary address that can be provided to the block memory as an address.

Our experiments implemented SAM-FSM on the Xilinx Virtex IV FPGA platform. The implemented FSM is to detect 50 patterns randomly selected from Snort rules. FSMs with and without pipelining techniques are implemented. Circuit simulations show that both unpipelined and pipelined FSMs function properly and are able to detect selected patterns.

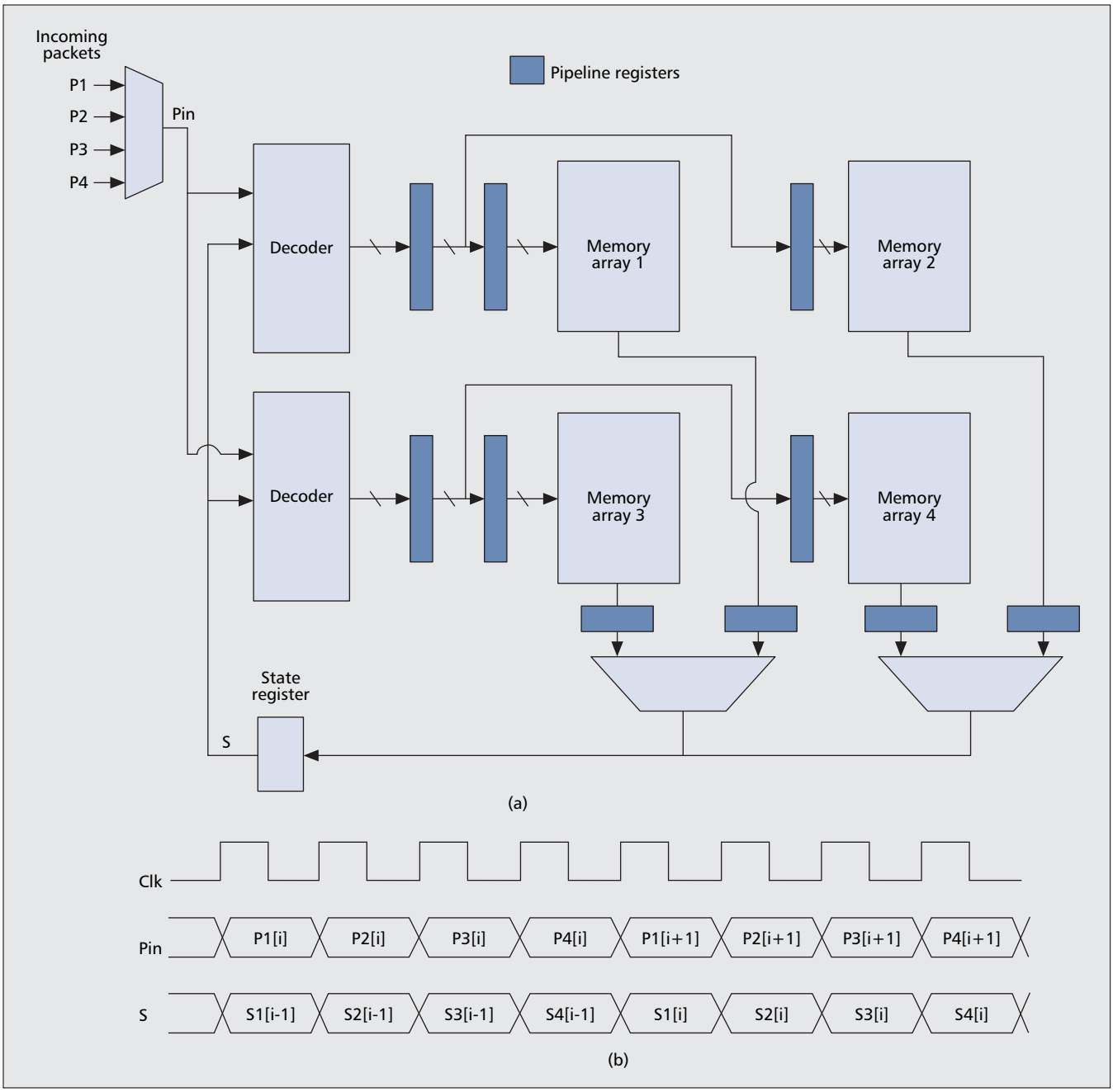
System Throughput

Currently, memory cores, for example, block memories on Virtex devices, which can operate with 500 MHz clocks are widely available. With simple pipelining techniques, SAM-FSM should be able to operate with 500 MHz clocks and thus, achieve 4 Gb/s throughput. If custom-designed hardware is used, deep pipelining techniques can be applied, and the performance should improve significantly. The estimated system performance can be in the range from 10 Gb/s to 18 Gb/s for 128 kilobyte memory size. In the estimation, we are assuming the circuit performance is limited by the partitioned memory array access time, which is estimated using cache access time (CACTI) [13]. Also, SAM-FSM requires dramatically reduced memory size, leading to a small hardware footprint. Thus, multiple copies of the string matching engines can be implemented to further improve the system throughput by parallel computation.

Storage Requirements

Table 2 shows the improvement of SAM-FSM over traditional DFA for different Snort rule sets. For each rule set, we compare traditional DFAs and our DFA in terms of number of states, state-coding length, and memory size. SAM-FSM reduces memory storage requirements by up to 63 times for 500 patterns, as shown in the ninth column. The memory savings are attributed to the reduction in number of states (due to a state-collapsing algorithm) and to storing each state only once in memory. Our state-coding length is much larger than traditional DFA binary coding because we want to simplify the group detector. Regardless, our memory size is much smaller than traditional DFA, which is estimated using the following equations.

The equation used to compute the storage requirement in the straightforward memory-based FSM implementation is $N \cdot \log_2 N \cdot 2^8$, where N is the total number of states of the FSM. This formula assumes that the binary encoding scheme is used in the FSM implementation. Thus, each state code requires $[\log_2 N]$ bits. The memory size required by SAM-FSM is calculated by $N \cdot (L + [\log_2 P])$, where L is the length of state code. P is the number of patterns to be detected. The $[\log_2 P]$ term



■ Figure 5. *An example four-stage pipelined system and its snapshot processing four packets in parallel: a) four-stage pipelined system; b) snapshot of parallel processing.*

in the above expression represents the additional tag bits used to indicate which patterns are matched.

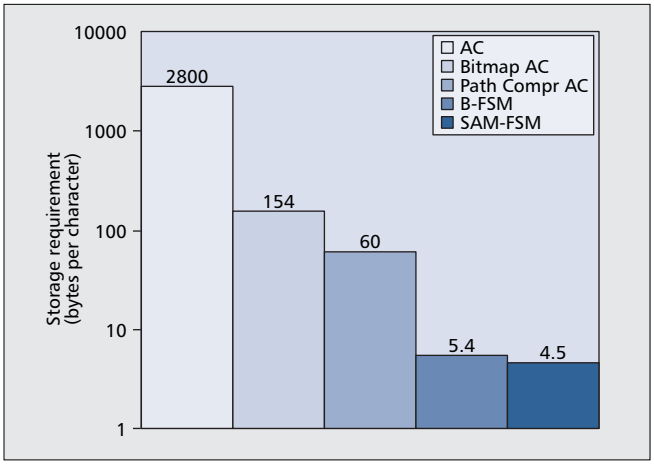
Figure 6 compares storage efficiency in terms of bytes per character with AC [1], Bitmap AC [10], Path compr. AC [10] and B-FSM [11]. Our result is taken from the tenth column of Table 2. Although the B-FSM approach results in memory per character that is almost similar to SAM-FSM, it might require multiple memory accesses to determine the next state, which potentially degrades system throughput. In contrast, SAM-FSM requires only one memory access.

Concluding Remarks

In this work, we present a memory-based, string-matching engine for NID applications called SAM-FSM. It replaces the state transition table used in conventional memory-based FSMs by a much more compact state table, which contains only one data entry for each state of the FSM. Instead of the addresses of the next states, a state table entry contains group signatures of partitioned FSM states. A specially designed memory-decoder circuit conducts next-state selection according to its recognized group signatures and FSM inputs. To facilitate the implementation of SAM-FSM, efficient algorithms were developed to partition FSM states, generate group signatures, and perform state encoding. Experimental results show that SAM-FSM reduces memory size by more than 60 times compared to the straightforward memory-based implementation. Unlike other memory-reduction techniques proposed for string-matching engine design, SAM-FSM does not involves complicated data translation during FSM operations to search next states from compressed memory data, nor does it require additional logic to generate matching output by consolidating many partial matching results.

Snort rules		Tradition DFA			SAM-FSM			Memory size reduction times	Memory size per char (byte)
# of patterns	# of char	# of states	state coding length	mem. size (KB)	# of states	state coding length	mem. size (KB)		
5	98	93	7	20.8	32	15	0.072	288	0.73
20	334	302	9	86.9	97	27	0.388	223	1.16
50	663	568	10	181.7	190	49	1.306	139	1.97
100	1291	1060	11	373.1	366	66	3.339	111	2.59
200	2129	1601	11	563.5	559	91	6.917	81	3.25
300	4313	3098	12	1189.6	1044	115	16.182	73	3.75
400	6722	4837	13	2012.2	1561	139	28.878	69	4.30
500	7637	5267	13	2191.1	1718	151	34.36	63	4.50

■ Table 2. *Memory efficiency compared to a conventional approach.*



■ Figure 6. *Storage requirement comparison with related work.*

In addition to system architecture and algorithm development, low-level hardware design techniques also are discussed. A reconfigurable group-decoder circuit with an address overlapping scheme is proposed to enhance the programmability of the string-matching engine. Pipelined operations for memory-based FSMs also are presented to improve the system throughput. By using these techniques, high-throughput, string-matching engines with easy update capability can be implemented. Our future work includes fabricating a customized memory-based, pattern-matching engine and measuring its performance.

References

[1] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, 1975.

[2] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," *Proc. 6th Int'l. Colloquium Automata, Languages, and Programming*, vol. 71, 1979.

[3] S. Wu and U. Manber, "A Fast Algorithm for Multi-Pattern Searching," *Tech. Rep. TR94-17*, Dept. of Comp. Sci., Univ. of AZ, 1994.

[4] K.-K. Tseng *et al.*, "A Parallel Automaton String Matching with Pre-Hashing and Root-Indexing Techniques for Content Filtering Coprocessor," *Proc. 2005 IEEE Int'l. Conf. App-Specific Sys., Architecture Processors*, 2005, pp. 113–18.

[5] S. Dharmapurikar *et al.*, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, Jan. 2004, pp. 52–61.

[6] I. Sourdis *et al.*, "Packet Pre-Filtering for Network Intrusion Detection," *Proc. 2006 ACM/IEEE Symp. Architecture for Net. Commun. Sys.*, pp. 183–92.

[7] M. Aldwairi, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding Up Intrusion Detection," *SIGARCH Comp. Archit. News*, vol. 33, no. 1, 2005, pp. 99–107.

[8] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *Proc. Reconfigurable Comp. Going Mainstream, 12th Int'l. Conf. Field-Programmable Logic and Applications (FPL'02)*, 2002, pp. 452–61.

[9] Snort, *Snort Rule Database*, 2007; http://www.snort.org/pubbin/downloads.cgi

[10] N. Tuck *et al.*, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *Proc. IEEE INFOCOM*, 2004, pp. 333–40.

[11] J. van Lunteren, "High-Performance Pattern Matching for Intrusion Detection," *Proc. INFOCOM '06*, 2006, pp. 1–13.

[12] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. 32nd Annual Int'l. Symp. Comp. Architecture*, 2005, pp. 112–22.

[13] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model," *WRL Research Report 2001–2002*, Western Research Lab., tech. rep., Palo Alto, CA, Aug. 2001.

Biographies

BENFANO SOEWITO received an M.S. degree in electrical and computer engineering at Southern Illinois University-Carbondale in 2004, where he is currently pursuing a Ph.D. degree in the Department of Electrical and Computer Engineering. His research interests are network processing systems, network intrusion detection, and multicore system programming.

LUCAS VESPA obtained an M.S. in electrical and computer engineering, and is a Ph.D. student and instructor in the Department of Electrical and Computer Engineering at Southern Illinois University-Carbondale. His research interests include networks and security.

ATUL MAHAJAN is a doctoral student at Southern Illinois University-Carbondale. His research interests are in the area of analog and digital VLSI design.

NING WENG (weng@engr.siu.edu) received a Ph.D. degree in electrical and computer engineering from the University of Massachusetts (Amherst) in 2005. He is an assistant professor in the Department of Electrical and Computer Engineering at Southern Illinois University-Carbondale. His research interests are in the areas of computer architecture, computer networks, and embedded systems.

HAIBO WANG (haibo@engr.siu) received his Ph.D. in electrical and computer engineering from the University of Arizona, Tucson, in 2002. Currently, he is an associate professor in the Electrical and Computer Engineering Department of Southern Illinois University-Carbondale. His research interests are in the area of VLSI circuit design.