High-Performance Pattern-Matching for Intrusion Detection

Jan van Lunteren
IBM Research, Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
Email: jvl@zurich.ibm.com

***Abstract*— New generations of network intrusion detection systems create the need for advanced pattern-matching engines. This paper presents a novel scheme for pattern-matching, called BFPM, that exploits a hardware-based programmable state-machine technology to achieve deterministic processing rates that are independent of input and pattern characteristics on the order of 10 Gb/s for FPGA and at least 20 Gb/s for ASIC implementations. BFPM supports dynamic updates and is one of the most storage-efficient schemes in the industry, supporting two thousand patterns extracted from Snort with a total of 32 K characters in only 128 KB of memory.**

I. Introduction

A clear trend that can be observed in the Internet is the increasing amount of packet data that is being inspected before a packet is delivered to its destination. In the early days, packets were solely routed based on their destination address. Later, firewall and quality-of-service (QoS) applications emerged that examined multiple fields in the packet header, for example, the popular 5-tuple consisting of addresses, port numbers and protocol byte [1]. More recently, network intrusion detection systems (NIDS), virus scanners, spam filters and other "content-aware" applications go one step further by also performing scans on the packet payload. Although the latter type of applications tend to reside closer to the end user, thus involving link speeds that are only a fraction of the speeds in the backbone, the ongoing performance improvements throughout the Internet make it very challenging to perform the required packet processing at full wirespeed.

Popular signature-based NIDSs, such as Snort [2], identify intrusions by testing packets against collections of rules that specify conditions for the packet header and payload. The header conditions are usually similar to those used in firewall rules, and can therefore be evaluated using the same type of algorithms [3]. The payload conditions typically involve strings and regular expressions that have to be matched against the entire payload or sections of it [4]. Although a substantial amount of work has been performed in the area of pattern-matching in the past thirty years, most of the existing algorithms are not suitable for new generations of NIDSs that require simultaneous matching of hundreds or thousands of patterns at processing rates of multiple gigabits per second.

The key contribution of this paper consists of a novel approach for pattern-matching that is able to meet the requirements of state-of-the-art and future NIDSs and other content-inspecting applications. This approach is based on a new programmable state-machine technology, called B-FSM, that is optimized for hardware implementation. The B-FSM-based pattern-matching (BFPM) scheme combines three algorithms to achieve a substantial improvement in storage efficiency over state-of-the-art algorithms, in combination with a deterministic pattern-matching performance, fast dynamic updates, and several other features that are important to NIDSs.

The remainder of this paper is organized in the following way. Section II provides an overview of the pattern-matching engine for which the BFPM scheme has been designed, illustrating several motivations behind the design of the algorithms. Section III discusses related work on pattern-matching. Section IV introduces the B-FSM technology that forms the core of the BFPM scheme, which is discussed in Section V. The performance of the BFPM scheme is evaluated in Section VI. Section VII presents experimental results which are then compared with the performance of existing schemes in Section VIII. Section IX concludes the paper. This paper will, for the most part, focus on string-matching. Although regular-expression support and several other features will also be touched upon, space constraints force a detailed discussion of these topics to be postponed to a later paper [5].

II. Pattern-Matching Engine

This section will provide an overview of the design objectives and target architecture of a pattern-matching engine for which the BFPM scheme has been developed.

A. Design Objectives

1) *Deterministic performance*: The engine will detect all patterns, including multiple occurrences and overlaps, by processing the input stream on-the-fly, in a single pass, at a fixed deterministic processing rate that is independent of the input stream and pattern characteristics. The processing rate being independent of the input stream makes the engine itself less vulnerable for attacks, while the deterministic nature simplifies system integration, in particular with respect to the dimensioning of resources such as bus bandwidth and buffer sizes.
2) *Storage requirements*: A main focus in the design of the engine has been on achieving a high storage-efficiency, which allows the performance to be increased by using fast but more expensive on-chip memory for storing and searching large numbers of patterns.

1-4244-0222-0/06/$20.00 (c)2006 IEEE

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.

3) *Fast dynamic updates*: Patterns can be added and removed dynamically from the engine's memory without interrupting the pattern-matching operation. The updates are performed through incremental modification of a data structure stored in random access memory.
4) *Patterns*: The pattern-matching engine supports strings and regular expressions. It allows several conditions to be specified for each individual pattern, e.g., case sensitivity, whether the pattern should be searched for in a certain segment or throughout the entire input stream, and whether an event should be generated if a pattern is detected or if it is not. Conditions can also be specified for multiple patterns, e.g., the order in which these patterns should occur, and the distance between them.
5) *Hardware implementation*: Because the patterns are programmed through the modification of memory contents, the pattern-matching engine is suitable for implementation using FPGA and ASIC technologies.
6) *Scalability*: A single-chip FPGA implementation of the pattern-matching engine will support tens of thousands of patterns. Multi-chip FPGA and single-chip ASIC implementations will scale to hundreds of thousands of patterns. At the same time, there is no basic limit on the maximum pattern length (except for memory capacity).
7) *Additional functions*: (1) Flexible allocation of pattern-matching resources to input channels, allowing the number of input channels and the number of patterns for each individual channel to be programmed. (2) Multi-session support, enabling the processing of different streams in an interleaved fashion, by storing and retrieving the process state for each session. (3) Pattern subset selection, enabling input streams to be matched against the entire pattern collection or against subsets that can be selected, for example, based on protocol information.

B. Concept

Fig. 1 illustrates a high-level block diagram of the pattern-matching engine. The core component is the pattern scanner that performs the string and regular expression matching on the input stream(s). The output of the pattern scanner consists of the pattern identifiers together with the locations (offsets) at which the patterns have been detected. This output is then processed by the result processor, which will check single- and multi-pattern conditions that have been specified for these patterns (see above), and generates the output in a format required by the pattern-matching application. The scanner control component performs session management, pattern subset selection, and manages the scanner resource allocation to the various input streams. The pattern compiler, implemented in software, will generate and dynamically update the data structures for the above hardware components, based on the patterns and associated conditions that are provided by the pattern-matching application. The remainder of this paper will present the BFPM scheme that forms the base of the pattern scanner, whereas several other features of the pattern-matching engine will be discussed in [5] as mentioned above.

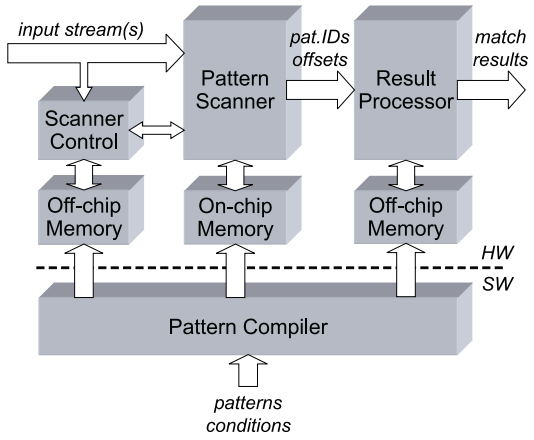


Fig. 1. Pattern-matching engine.

III. Related Work

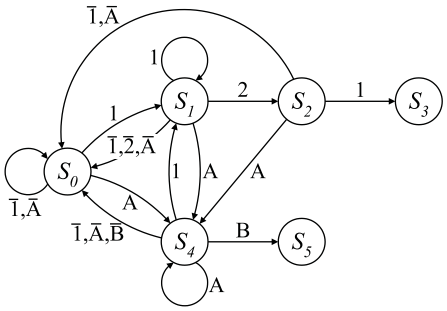
Pattern-matching has been a topic of intensive research that has resulted in a substantial number of publications in the past several decades. Two of the most popular algorithms are those published by Aho and Corasick [6] and Boyer and Moore [7] almost 30 years ago. The Aho-Corasick algorithm constructs a finite state machine (FSM) for detecting all occurrences of a given set of patterns by processing the input in a single pass, performing a state transition for each input character. The Boyer-Moore algorithm, which is basically a single-pattern matching solution, exploits two heuristics (named "bad character" and "good suffix") to skip portions of the input stream in order to improve the average performance. Concepts similar to Aho-Corasick and Boyer-Moore can be found in many other pattern-matching algorithms, such as the algorithms by Commentz-Walter [8], Wu and Manber [9], and more recently, the Aho-Corasick-Boyer-Moore (AC_BM) algorithm by Coit et al. [10] and the Setwise Boyer-Moore-Horspool (SBMH) scheme by Fisk and Varghese [11].

The heuristics applied by Boyer-Moore, Wu-Manber and other schemes for skipping portions of the input enable an excellent average-case performance for a variety of applications, but the resulting dependency of the processing rate on input (and pattern) characteristics is an important disadvantage for intrusion detection, as it makes the NIDS vulnerable to attacks that try to overload the pattern-matching operation by generating worst-case input scenarios [12][13]. The Aho-Corasick algorithm has a deterministic performance and, therefore, does not have this drawback. Instead, its main disadvantage is the large storage requirements needed to implement the FSM. Tuck et al. proposed two optimized versions of Aho-Corasick to address this issue, and were able to achieve a significant reduction in storage requirements [14].

Driven by performance and cost improvements in FPGA technology, several approaches have been developed for the automatic generation of dedicated match functions in hardware for given sets of patterns [15]-[19]. Many of these approaches exploit the parallelism available in hardware implementations,

by instantiating a large number of processing units (e.g., FSMs), that each take care of one or a small number of patterns, thus simplifying the design of each individual unit. Although these schemes have been able to realize impressive processing rates of up to 10 Gb/s and even beyond based on state-of-the-art FPGA technology, their inherent shortcoming for intrusion detection is that upon each update involving the addition or removal of a pattern, the hardware (i.e., the FPGA configuration data) has to be regenerated, making it impossible to support fast dynamic updates without interrupting the pattern-matching operation. These schemes are also restricted to reconfigurable hardware implementations, ruling out the possibility of a higher-performance version based on ASIC technology. The approaches that include a large number of processing units, e.g., on the order of hundreds or more, have an additional drawback that the state of a single pattern-matching session, which comprises the states of all processing units, can become extremely large, making it impossible to provide multi-session support as desired by several NIDSs.

When comparing it with state-of-the-art pattern-matching schemes, the BFPM scheme that will be presented in the next sections, is most closely related to Aho-Corasick: Both construct a FSM to implement the pattern-matching function and achieve a deterministic processing rate. A key difference is that BFPM applies a novel programmable state-machine technology, B-FSM, to implement the FSM, which allows the pattern-matching function to be programmed dynamically in a straightforward and flexible manner using so-called transition rules involving wildcards and priorities. This enables optimizations at various levels in the process that are not possible with Aho-Corasick, resulting in a substantial gain in storage efficiency. The BFPM approach also exploits hardware parallelism in a similar way as several of the FPGA-based schemes discussed above do. However, it does so to a much smaller extent, e.g., typically using between 4 and 32 processing units per session, to achieve a smaller session state enabling efficient multi-session support.

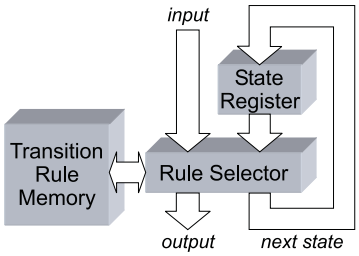


(a) State-transition diagram.

rule	current state	input	→	next state	priority
R_1	S_2	1h	→	S_3	2
R_2	*	1h	→	S_1	1
R_3	S_1	2h	→	S_2	1
R_4	S_4	Bh	→	S_5	1
R_5	*	Ah	→	S_4	1
R_6	*	*	→	S_0	0

(b) Transition rules.

Fig. 2. Example.



(a) Block diagram.

(b) Transition-rule vector.

Fig. 3. B-FSM.

IV. B-FSM

This section will introduce the B-FSM technology that forms the base of the BFPM scheme.

A. Transition Rules

The B-FSM technology is based on the concept of so-called state-transition rules, which specify match operators for the current state and input symbol values, in combination with a next state. The transition rules are assigned priorities in order to resolve situations in which multiple transition rules match simultaneously. This is illustrated using the example of the state-transition diagram shown in Fig. 2(a), which detects the first occurrence of one of two patterns "121h" and "ABh" ('h' means hexadecimal notation) in a stream of 4-bit input symbols. States S_3 and S_5 are end states that correspond to the detection of the two patterns.

This state-transition diagram can be described using the state-transition rules shown in Fig. 2(b), with the wildcard symbol '*' representing a "don't care" condition. Transition rules R_1 and R_2 specify that if the input symbol equals 1h, a transition will be made to state S_3 if the current state is S_2 and that a transition will be made to state S_1 if the current state is any state other than S_2. This is achieved by assigning a higher priority to rule R_1 than to rule R_2. Transition rule R_6 can be regarded as the default transition rule that is applied if no other matching rule has been found. As can be verified, the entire state-transition diagram in Fig. 2 is described by the six transition rules R_1 to R_6.

Fig. 3(a) shows a block diagram of the B-FSM. The transition rules are stored in a transition-rule memory, encoded as shown in Fig. 3(b). The *test part* of each transition rule contains a state, an input and a conditions field. The latter field includes flags indicating whether the state and input are "don't care". The *result part* contains a next state, a table

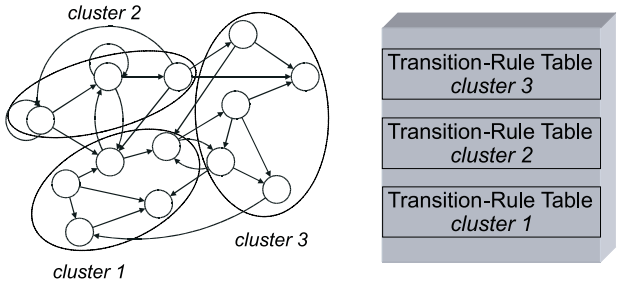
rule	current state	input	→	next state	priority
R_1	S_2	0001b	→	S_3	2
R_2	*	0001b	→	S_1	1
R_3	S_1	0010b	→	S_2	1
R_4	S_4	1011b	→	S_5	1
R_5	*	1010b	→	S_4	1
R_6	*	xxxxb	→	S_0	0

(a) Transition rules.

1	rule R_4	rule R_5	rule R_6	
0	rule R_1	rule R_2	rule R_3	rule R_6

(b) BART-compressed transition-rule table.

Fig. 4. Transition-rule selection.



(a) State-transition diagram. (b) Transition-rule memory.

Fig. 5. State clusters.

address, and a mask field (the latter two fields will be explained below). In each cycle, the rule selector will select the highest-priority transition rule that matches the actual values of the state register and input symbol, and will then update the state register based on the next state field of that rule. If the B-FSM is operated as a Moore machine, then an output value is derived from the state vector. If it is operated as a Mealy machine, then the output value is obtained from an additional output field contained in the result part of the transition-rule vector (not shown in Fig. 3(b)).

B. Rule Selection

The rule selector is based on an optimized version of the Balanced Routing Table (BART) search algorithm [20], hence the name BART-based FSM (B-FSM). BART is a scheme for exact-, prefix- and range-match searches, that is based on a novel hash function with the special property that the maximum number of collisions for any hash index can be limited by a configurable bound P. The hash index is extracted from bit positions within the search key that are selected by a special update function in order to realize the above property. The value of P is typically based on the memory access granularity to ensure that all collisions for a given hash index can be resolved by a single memory access and by at most P parallel comparisons. For a detailed description of BART, the reader is referred to [20] and [21].

The application of the BART scheme for transition-rule selection will now be illustrated for the example above. Fig. 4(a) shows the same transition rules with the input values in binary notation. The underscored bit of the input symbol is an example of a hash index for which the maximum number of collisions is limited to $P = 4$. This can easily be verified: If the most significant input bit equalled 0b, then only rules R_1, R_2, R_3, and R_6 could be matching. If this bit were '1', then only rules R_4, R_5 and R_6 could match. In both cases, the number of collisions per hash index value does not exceed $P = 4$ rules. Note that for larger transition-rule sets, the hash index will be extracted from multiple bit positions within the current state and input symbol vectors that are not necessarily adjacent. These bit positions will be optimally selected by an update function, such that a minimum table size is obtained [20]. Fig. 4(b) shows the corresponding hash table, which will be called transition-rule table, that is indexed by the most significant input symbol bit. Each transition-rule table entry contains at most P transition-rule vectors corresponding to the transition rules mapped on the hash index value, that are ordered by decreasing priority within each table entry. After reading the selected table entry from memory, the state and input conditions of these P transition rules are compared in parallel with the actual values of the state register and input symbol. The first, and therefore highest-priority, transition rule that matches the current state and input symbol, will be used to update the state register.

C. State Clusters

For scalability reasons, the states will be partitioned into multiple disjoint clusters, and for each of these state clusters, a separate BART-compressed transition-rule table is created that stores the transition rules that "start" in that cluster. The transition-rule tables are compressed independently of each other, based on different hash indices that will optimally compress the transition rules of the corresponding clusters. This concept is depicted in Fig. 5.

States within a state cluster are regarded as *local* states and are assigned state vectors that are only unique within the same cluster. Transition rules will only contain local current and next state vectors. The result part of each transition rule will provide the address of the transition-rule table that "implements" the state cluster in which the local next state is located, together with the mask that defines the state and input symbol bits that form the hash index for indexing the transition-rule table. In this way, the actual processing is restricted to the local state vectors only; namely, bit extraction to form a hash index and parallel testing against the current state fields of at most P transition-rule vectors. As a result the B-FSM concept can be scaled to very large state-transition diagrams without impacting the processing complexity, by simply enlarging the width of the table address vector which allows to address a larger number of transition-rule tables. A second advantage of the state-cluster concept is that both the storage efficiency and update performance are improved because the BART algorithm is applied on smaller portions of the data structure.

D. Optimizations

While the original BART scheme provides excellent performance for a wide variety of applications, it can be further optimized for pattern-matching applications as will now be described.

Don't-care Rules

The BFPM scheme uses three types of transition rules as will be discussed in the Section V: (1) A default rule having a wildcard condition for the current state and input, and priority 0, (2) transition rules having a wildcard condition for the current state, an exact-match condition for the input, and priority 1, and (3) transition rules having exact-match conditions for the current state and input, and priority 2. Because the first two types of transition rules do not depend on the current state, the highest-priority matching rule can be determined by a simple table lookup on the input value. This is shown in Fig. 6 for the example above, in which the result parts of rules R_2, R_5, and R_6 are stored in the 16-entry lookup table, that is indexed by the 4-bit input value. The remaining transition rules of the third type (R_1, R_3 and R_4 in the example above), are stored in a BART-compressed transition-rule table as described above. Because these rules have equal priorities and do not include wildcard conditions, there can be at most one matching rule in each table entry, rendering priority resolution unnecessary, thus simplifying the rule-selection logic. If none of the (at most) P rules in the selected transition-rule table entry is found to be matching, then the highest-priority matching "don't-care"-rule, obtained by the table lookup on the input symbol, will be selected.

This optimization can significantly improve the storage efficiency for pattern-matching, because "don't-care"-rules will not occur multiple times in the transition-rule tables (such as rule R_6 in Fig. 4(b)). It also improves the B-FSM processing rate by simplifying the rule-selection logic.

State Encoding and Index Calculation

The original BART scheme involves a flexible hash index calculation, in which the hash index can be extracted from any combination of state and input bit positions. Because these bit positions are not necessarily adjacent, the index bits have to be aligned to form the index value [20]. By optimizing the encoding of the local state vectors within each state cluster, the hash index calculation can be simplified to the following bit-wise operations, which do not involve any bit-alignment:

$$index = (state' \textbf{ and not } mask) \textbf{ or } (input' \textbf{ and } mask) \quad (1)$$

where **and**, **or**, and **not** are bit-wise operators, $state'$ and $input'$ are fixed selected subsets (e.g., the least significant part) of the (local) current state and input symbol values, and $mask$ determines the hash index for a given cluster as described in Section IV-C. In this case, the $mask$ vector determines for each index bit if it is extracted from either the current state value (mask bit is zero) or from the input symbol (mask bit is one). This will now be explained in more detail.

(input)	(result part)
Fh .. Bh	rule R_6
Ah	rule R_5
9h .. 2h	rule R_6
1h	rule R_2
0h	rule R_6

Fig. 6. Lookup table.

If there exist at most P transition rules from a given state, then these transition rules can be stored in the same transition-rule table entry. If the $mask$ vector would only contain zero bits, then the hash index is solely determined by the "$state'$-portion" of the local state vector as can be seen from (1), and as a result, the encoding of the local state encoding provides full control on mapping these transition rules on any desired (i.e., free) entries within a transition-rule table. Because $state'$ in (1) comprises a subset of the local state vector, the remaining state bits allow to map transition rules from different states with less than P transition rules, on the same table entry. The rule selector will then select the correct transition rule after testing the current state against the entire current state fields of the P transition rules in the selected table entry.

If there are more than P transition rules from a single state, then a $mask$ vector is created using the BART-update function, that selects a minimum number of input bit positions for distributing these rules over multiple table entries with at most P rules per entry. The $mask$ vector will include set bits at these bit positions. The remaining index bits are then extracted from the (local) state vector, and, consequently, the encoding of the state vector can be used to control the location of the multiple table entries that will store these transition rules.

The B-FSM compiler will construct the transition-rule tables for any given set of transition rules, by applying the above concepts in the following way. Although the states can be processed in random order, it is typically more efficient to start with the states that have the most transition rules. The first state will be placed in a new cluster and will be assigned a local state vector equal to zero. If the number of transition rules is larger than P, then an optimal index mask is calculated for distributing these transition rules over multiple table entries as described above, otherwise an index mask equal to zero is used. Next it is checked which of the remaining states and corresponding transition rules, can be mapped on the empty entries in the transition-rule table based on the above index mask, by varying the encoding of the local state vector (which has to be unique). All states for which the corresponding transition rules can be mapped on the transition-rule table will be added to this state cluster. After all states have been processed, then the same process will be iterated for the

remaining states and transition rules, by creating a new cluster and transition-rule table.

This approach achieved an optimal filling of all the hash tables using a value of $P = 4$ for all pattern collections analyzed, including the NIDS patterns evaluated in Section VII, which corresponds to a linear growth of the storage requirements with the number of transition rules. As can be understood from the above description, this linear growth could be impacted if the percentage of states that involve more than P transition rules were to grow substantially larger. However, several mechanisms can be applied to prevent this situation or to limit its impact, and thus preserve an approximately linear growth for a wide range of pattern collections, including future ones that might have different characteristics. One such mechanism involves increasing the value of P. Another approach comprises an extension of the pattern distributor component of the pattern compiler (see Section V-C) that will consider the percentage of states with more than P transitions in each pattern collection (a detailed discussion of this topic falls outside the scope of this paper). As indicated above, these mechanisms were not required for any of the pattern collections that were analyzed to obtain an optimal table filling.

The optimizations described above, improve performance and storage efficiency. The hash index calculation according to (1) only involves bitwise operations, which enables a very fast implementation involving only two or three layers of logic. The B-FSM compiler exploits the state encoding to achieve an optimum fill rate of the hash tables that are based on the simple hash index calculation, to significantly increase the storage efficiency. Another advantage of (1) is the fixed width of the hash index, resulting in identical hash-table sizes being a power of 2. By aligning these tables in memory, the table address field in the transition-rule vector, only has to store the most significant portion of the memory address that is common to all table entries, while the hash index comprises the least significant address bits of the selected table entry.

V. The BFPM Scheme

This section will present the BFPM scheme, which exploits the B-FSM technology to realize a fast and storage-efficient pattern-matching solution.

A. A Distributed B-FSM Approach

The small size of the B-FSM logic makes it possible to implement the pattern scanner as an array of multiple B-FSMs, each of which is attached to a portion of the total memory. This concept, which is shown in Fig. 7, has several advantages over a single B-FSM connected to one large memory:

- Improved *performance* because of the tighter integration of the B-FSM logic into the memory: smaller memories are faster.
- Increased *storage efficiency* through selective distribution of the patterns over the B-FSMs, as will be discussed in the next two subsections.

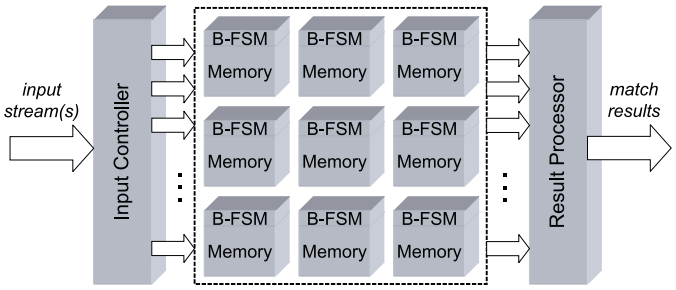


Fig. 7. Pattern scanner.

- Increased *flexibility* by allowing a programmable allocation of B-FSMs to input streams (see Section VII-B).

The actual number of B-FSMs depends on the implementation technology and pattern-matching requirements, but will typically be a value between 4 and 32.

B. String Matching

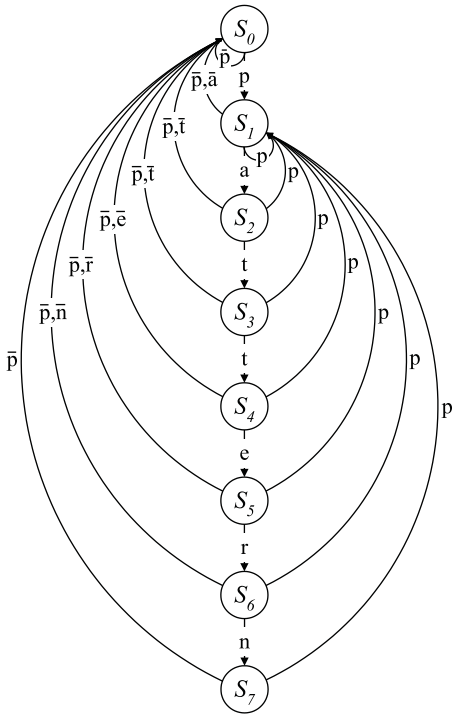
The BFPM scheme will now be described using three examples, which involve detecting all occurrences of the following strings anywhere in the input stream: (1) *"pattern"*, (2) *"testing"* and *"pattern"*, and (3) *"testing"* and *"testcase"*. Figs. 8(a), 9(a), and 10(a) show the state-transition diagrams that can be constructed for these pattern-matching problems using existing algorithms (see Section III), or as in these simple cases, manually. Figs. 8(b), 9(b), and 10(b) show the transition rules generated for these patterns using the approach described in this section. Note that the state-transition diagrams are shown for illustrative purposes only: The transition rules will be derived directly from the patterns. Before discussing the transition-rule generation algorithm in detail, first some observations will be made and some terminology introduced based on the three examples.

Example 1: "pattern"

The transition rules shown in Fig. 8(b) include three types of transition rules with different priorities:

- A *default* transition rule R_0 to a next state S_0, having wildcard conditions for the current state and input symbol, and priority 0.
- A transition rule R_1 for the first pattern character, *'p'*, to a next state S_1, having a wildcard condition for the current state, and priority 1.
- A transition rule R_i, with $2 \leq i \leq 7$, for each of the remaining pattern characters, having a current state equal to the next state S_{i-1} of the transition rule corresponding to the preceding character, and a next state S_i, with priority 2.

This results in a total of eight transition rules that describe the entire pattern-matching operation as can be verified using Fig. 8(a). In this example, there is exactly one transition rule per character (not counting the default transition rule).

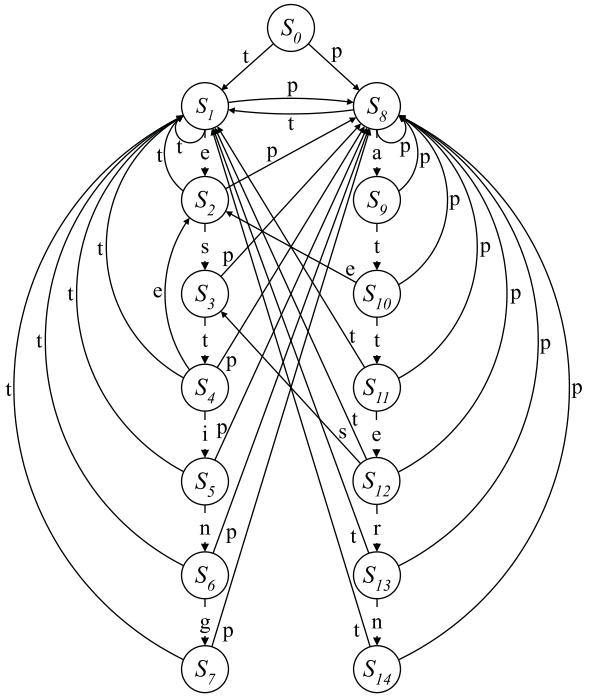


(a) State-transition diagram (prior-art) for example 1.

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	p	→	S_1	1
R_2	S_1	a	→	S_2	2
R_3	S_2	t	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	e	→	S_5	2
R_6	S_5	r	→	S_6	2
R_7	S_6	n	→	S_7	2

(b) Transition rules for example 1.

Fig. 8.



(a) State-transition diagram (prior-art) for example 2 ("default" transitions to state S0 are not shown).

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	t	→	S_1	1
R_2	S_1	e	→	S_2	2
R_3	S_2	s	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	i	→	S_5	2
R_6	S_5	n	→	S_6	2
R_7	S_6	g	→	S_7	2
R_8	*	p	→	S_8	1
R_9	S_8	a	→	S_9	2
R_{10}	S_9	t	→	S_{10}	2
R_{11}	S_{10}	t	→	S_{11}	2
R_{12}	S_{11}	e	→	S_{12}	2
R_{13}	S_{12}	r	→	S_{13}	2
R_{14}	S_{13}	n	→	S_{14}	2
R_{15}	S_4	e	→	S_2	2
R_{16}	S_{10}	e	→	S_2	2
R_{17}	S_{12}	s	→	S_3	2

(b) State-transition rules for example 2.

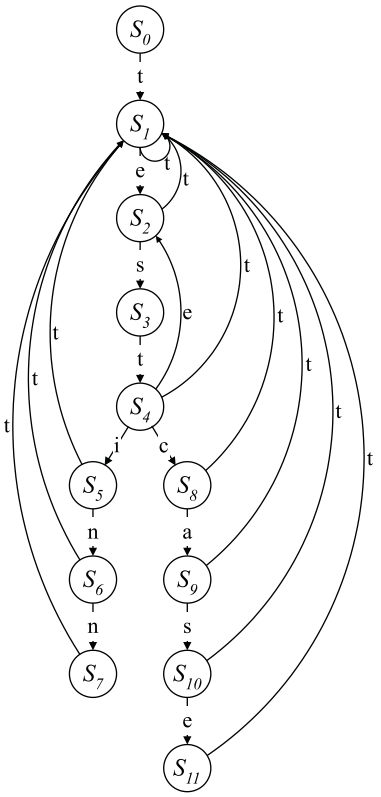
Fig. 9.

Example 2: "testing" and *"pattern"*

The transition rules shown in Fig. 9(b) include a default rule R_0, whereas rules R_1 to R_7 and R_8 to R_{14} correspond to the patterns *"testing"* and *"pattern"* in a similar way as described above for the first example. In this case, however, transition rules R_0 to R_{14} are not sufficient to describe the entire pattern-matching operation, but three additional transition rules, R_{15} to R_{17}, are required as can be verified using Fig. 9(a). This is due to the occurrence of prefixes of the pattern *"testing"* within each of the two patterns: a prefix *'t'* occurs at the fourth character position in *"testing"* and at the third position within *"pattern"*, while a prefix *"te"* can be found at the fourth position within *"pattern"*. As a result, state transitions are needed from state S_4 to S_2, from S_{10} to S_2 and from S_{12} to S_3 to handle input streams containing patterns such as *"testesting"*, *"patesting"*, and *"pattesting"*. The occurrence of a prefix of a pattern within the pattern itself, or within another pattern (not as prefix) will be referred to as intra- or inter-pattern conflict, respectively. Transition rules R_{15} to R_{17} will be called *conflict resolution* rules. In this example, there are on average about 1.2 transition rules per character (i.e., the total number of transition rules divided by the accumulated pattern length, not counting the default rule). This average is larger than one rule per character because of the pattern conflicts.

Example 3: "testing" and *"testcase"*

The patterns involved in the third example share a common prefix *"test"*. The transition rules corresponding to this common prefix need to be created only once. As a result, transition rules R_1 to R_7 correspond to *"testing"*, whereas transition rules R_1 to R_4 and R_8 to R_{11} correspond to *"testcase"*. Transition rule R_{12} is a conflict resolution rule needed to



(a) State-transition diagram (prior-art) for example 3 ("default" transitions to state S0 are not shown).

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	t	→	S_1	1
R_2	S_1	e	→	S_2	2
R_3	S_2	s	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	i	→	S_5	2
R_6	S_5	n	→	S_6	2
R_7	S_6	g	→	S_7	2
R_8	S_4	c	→	S_8	2
R_9	S_8	a	→	S_9	2
R_{10}	S_9	s	→	S_{10}	2
R_{11}	S_{10}	e	→	S_{11}	2
R_{12}	S_4	e	→	S_2	2

(b) State-transition rules for example 3.

Fig. 10.

handle input streams containing patterns such as *"testest"*. In this case, the average number of transition rules per character equals 0.8, which is less than one rule per character owing to the existence of a common prefix.

Transition-Rule Generation

The three examples have shown that the transitions rules can be generated directly from the patterns in a relatively straightforward manner. The basic generation process involves the creation of a transition rule for each pattern character, with the first pattern character handled slightly differently than the other characters, as described above. This process is only impacted by the existence of common prefixes between patterns, which results in transition rules being shared among these patterns, and by pattern conflicts that require the addition of special conflict resolution rules. The pattern generation, in particular the common-prefix and pattern-conflict detection, can be performed in several ways, including processing of (1) the original patterns, (2) the transition rules created by the basic generation process, or (3) the states needed to implement the pattern-matching problem. This section will discuss the third variant, which comprises the following steps.

1) For each pattern, a total of k states is created, with k being the pattern length, corresponding to all possible prefixes of that pattern with lengths varying between 1 and k characters.
2) All duplicate states, i.e., states corresponding to identical pattern prefixes, are filtered.
3) A default transition rule is created to state S_0 with priority 0.
4) For each state that is associated with a single-character pattern prefix, a transition rule is created having a wild-card condition for the current state, the single-character prefix as input symbol, and priority 1.
5) "Between" any two states for which it holds that a suffix of the pattern prefix associated with the first state equals the pattern prefix associated with the second state after removing its last character, a transition rule will be created having that last character as input symbol, and priority 2. If this results in multiple transition rules from the same state having the same input character, then only one transition rule will be created that corresponds to the longest prefix/suffix combination.

The various steps of the transition-rule generation process will now be illustrated using the third example discussed above, involving the two patterns *"testing"* and *"testcase"*. The first step results in the creation of a total of seven states for *"testing"* and a total of eight states for *"testcase"*, which will be denoted by S_0 to S_7 and by S'_0 to S'_8, respectively. These states together with the associated pattern prefixes are shown in Fig. 11(a). In the second step, the states with duplicate pattern prefixes will be filtered. In this case, states S'_1 to S'_4 have the same pattern prefixes as states S_1 to S_4 do and will therefore be removed. This results in the states shown in Fig. 11(b) (with states S'_5 to S'_8 renamed to S_8 to S_{11}, to be consistent with Fig. 10). The third step involves the creation of a default rule to state S_0. The fourth step involves the creation of a transition rule for each state associated with a pattern prefix consisting of a single character. In this example, there is only one such state, S_1. As a result of steps three and four, transition rules R_0 and R_1 are created, which are listed in Fig. 10(b).

The fifth step searches for all pairs of states for which a suffix of the pattern prefix associated with the first state equals the pattern prefix associated with the second state after removal of its last character. Two examples of such pairs of states are (S_1, S_2) and (S_4, S_2). The pattern prefix associated with state S_2, becomes *'t'* after removal of its last character, which equals the pattern prefix associated with state S_1, *'t'*, and the last

state	prefix
S_1	*t*
S_2	*te*
S_3	*tes*
S_4	*test*
S_5	*testi*
S_6	*testin*
S_7	*testing*

state	prefix
S'_1	*t*
S'_2	*te*
S'_3	*tes*
S'_4	*test*
S'_5	*testc*
S'_6	*testca*
S'_7	*testcas*
S'_8	*testcase*

(a) States (step 1).

state	prefix
S_1	*t*
S_2	*te*
S_3	*tes*
S_4	*test*
S_5	*testi*
S_6	*testin*
S_7	*testing*

state	prefix
S_8	*testc*
S_9	*testca*
S_{10}	*testcas*
S_{11}	*testcase*

(b) Filtered states (step 2).

state	prefix	state	prefix	rule
S_1	*t*	S_2	*te*	R_2
S_2	*te*	S_3	*tes*	R_3
S_3	*tes*	S_4	*test*	R_4
S_4	*test*	S_2	*te*	R_{12}
S_4	*test*	S_5	*testi*	R_5
S_4	*test*	S_8	*testc*	R_8
S_5	*testi*	S_6	*testin*	R_6
S_6	*testin*	S_7	*testing*	R_7
S_8	*testc*	S_9	*testca*	R_9
S_9	*testca*	S_{10}	*testcas*	R_{10}
S_{10}	*testcas*	S_{11}	*testcase*	R_{11}

(c) State combinations (step 5).

Fig. 11. Transition-rule generation.

character of the pattern prefix associated with state S_4, *"test"*. As a result, transition rules will be created from state S_1 to S_2 and from state S_4 to S_2 (rules R_2 and R_{12} in Fig. 10(b)). The complete list of pairs of states with matching suffix and prefix portions (underscored) and corresponding transition rules is shown in Fig. 11(c).

Pattern Identifiers

Upon detection of a pattern, an identifier is provided to the result processor for subsequent processing as was discussed in Section II. These pattern identifiers are derived directly from the state vectors using a variable-sized BART-compressed table (based on the original algorithm [20]) for each state cluster, which will provide the user-defined pattern identifier corresponding to any local state vector that relates to the detection of a pattern (e.g., state S_7 in Fig. 8). The pattern identifiers could also be generated by operating the B-FSMs as Mealy machines, which would involve the addition of a pattern identifier field to the result part of each transition rule. However, the Moore-machine based approach described above, will typically result in a higher storage efficiency.

C. Pattern Compiler

The pattern compiler is the function that creates and dynamically updates the data structures for the B-FSM array. It consist of three main components as shown in Fig. 12. The first component is a pattern distributor, which distributes a given set of patterns into N disjoint pattern sets, with N being the number of B-FSMs that will be used to detect these patterns. The second component is the transition-rule generator, which will generate the transition rules for each of the N pattern sets as described above. Finally, the B-FSM compiler will create the data structures for each of the N B-FSMs as described in Section IV.

Pattern Distributor

A key objective of the pattern compiler is to minimize the storage requirements, i.e., the total size of the B-FSM data structures. For the pattern distributor and transition-rule generator components, this translates into minimizing the number of transition rules generated for a given set of patterns. Whereas the operation of the transition-rule generator is basically fixed and cannot be optimized for this purpose, the pattern distributor is able to do so through selective distribution of the patterns over the B-FSMs based on the pattern characteristics.

The three examples discussed in Section V-B revealed that the existence of common prefixes between patterns will decrease the number of transition rules, whereas pattern conflicts will increase this number. Based on this observation, the pattern distributor will try to allocate patterns with common prefixes to the same B-FSM and conflicting patterns to different B-FSMs, in order to minimize the total number of transition rules and improve the storage efficiency. A larger number of B-FSMs provides the pattern distributor with more possibilities to distribute the patterns, and, therefore, increases the potential for improving the storage efficiency.

The current implementation of the pattern distributor processes the patterns in the order in which these occur in the (base) rule set, whereas new patterns are processed as soon as these are added. Each pattern is processed by comparing it with the N collections of previously processed patterns, and estimating the number of additional transition rules that would result from adding it to each of these collections, based on the numbers of common prefixes and pattern conflicts. The pattern will then be added to the collection that results in a minimum number of additional transition rules. To achieve an approximately even distribution of the patterns over the N collections, the actual number of transition rules in each collection is also taken into account. Given the simplicity of the approach described above, it is expected that ample room for improvement exists by applying a more advanced pattern distributor, which is a topic of ongoing research.

Dynamic Incremental Updates

All three algorithms that comprise the pattern compiler, support incremental updates, enabling the dynamic addition and removal of patterns. Each update will result in the modification of one B-FSM data structure, which can be performed in the following two ways without interrupting the pattern-matching operation: (1) By creating copies of the modified transition-rule tables which are then linked into the data structure using atomic write operations, similar as described in [20], and (2) by writing the entire updated B-FSM data structure into the

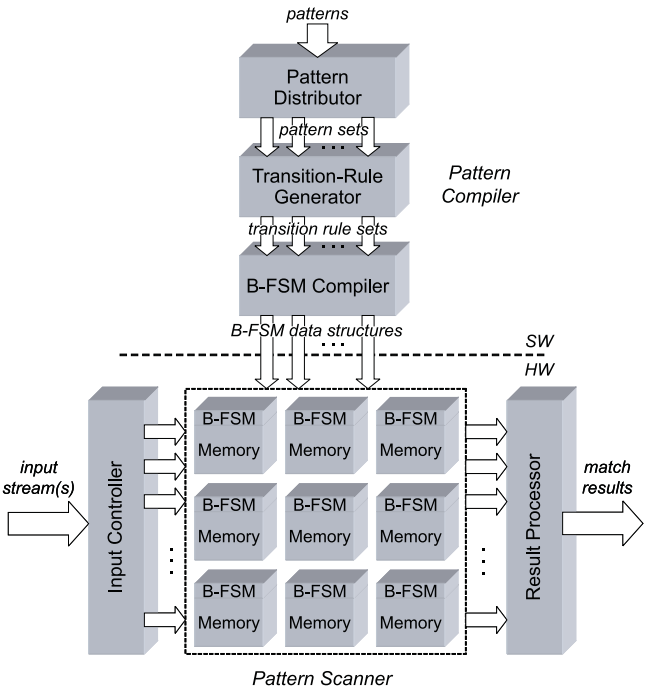
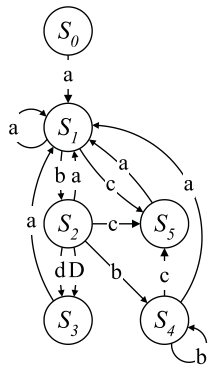


Fig. 12. Pattern scanner and compiler.



(a) State-transition diagram
("default" transitions to state S0 are not shown).

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	a	→	S_1	1
R_2	S_1	b	→	S_2	2
R_3	S_1	c	→	S_5	2
R_4	S_2	b	→	S_4	2
R_5	S_2	c	→	S_5	2
R_6	S_2	d	→	S_3	2
R_7	S_2	D	→	S_3	2
R_8	S_4	b	→	S_4	2
R_9	S_4	c	→	S_5	2

(b) State-transition rules

Fig. 13. Case-sensitivity and regular-expression support.

transition-rule memory of an additional non-active B-FSM, which is then activated, i.e., starts matching patterns in the input stream, followed by the deactivation of the original B-FSM when it reaches state S_0 (which makes sure that no patterns are missed during the "switch").

D. Case Sensitivity and Regular Expressions

Case sensitivity is supported in multiple ways. The basic approach, which exploits the property that most NIDSs specify case-sensitivity constraints at the granularity of entire patterns, involves the processing of case-sensitive and case-insensitive patterns by different sets of B-FSMs. The case-sensitive patterns are handled as described above. The case-insensitive patterns are first converted to the same case (either lower or upper) before being processed by the pattern compiler to generate the B-FSM data structures. The input stream to these B-FSMs is also converted to the same case, thus realizing a case-insensitive pattern-matching operation.

The BFPM scheme provides more advanced support for case sensitivity and regular expressions, beyond what has been presented in this paper, based on enhancements of the B-FSM functionality and the pattern compiler. These enhancements include, for example, an input classifier to provide direct support for character classes such as "whitespace" and "digit", and a counter array to efficiently handle several types of quantifiers, that are commonly used in regular expressions [22]. As indicated in the introduction, a detailed discussion of these topics has to be deferred to [5]. Instead the following example is provided, which involves the detecting of all matches of two regular expressions, $ab[d|D]$ and $ab{*}c$, anywhere in the input stream, with the first regular expression including one case-insensitive character. Fig. 13(a) shows the corresponding state-transition diagram which is only shown for illustrative purposes. Fig. 13(b) shows the transition rules that have been derived from these regular expressions by the (enhanced) pattern compiler that will be presented in [5].

VI. Performance Evaluation

A. Processing Rate

The BFPM scheme has a fixed deterministic processing rate that is independent of the input and pattern characteristics; it only depends on the speed of the B-FSM implementation in a given FPGA or ASIC technology. Actual performance numbers will be presented in Section VII.

B. Storage Requirements

This section will analyze the storage complexity for a random set of n patterns, comprising a total of m characters, and an average pattern length of $k = \frac{m}{n}$ characters. These patterns will be divided by the pattern distributor over N collections, with N being the number of B-FSMs, resulting in an average of $\frac{n}{N}$ patterns and $\frac{m}{N}$ characters in each pattern collection. A worst-case assumption is made that no common prefixes exist (which would reduce the number of transition rules), but only the effect of pattern conflicts will be regarded.

There are two types of conflicts, as described in Section V-B. Intra-pattern conflicts exist when a prefix of a pattern occurs within the pattern itself. For a given random pattern,

the average number of intra-pattern conflicts will increase with the pattern length according to $O(k)$. Accordingly, the total number of intra-pattern conflicts for a collection with $\frac{n}{N}$ patterns will grow according to $O(\frac{n}{N}k) = O(\frac{m}{N})$, and for all N collections according to $O(m)$.

Inter-pattern conflicts exist when a prefix of a pattern occurs within another pattern (but not as prefix). For a given random pattern with an average length of k characters that is part of a collection with $\frac{n}{N}$ patterns, the average number of inter-pattern conflicts will increase with the number of patterns and its pattern length according to $O(\frac{n}{N} + ck)$, with c being some constant. The total number of inter-pattern conflicts for all $\frac{n}{N}$ patterns in the collection will then increase according to $O\left(\left(\frac{n}{N}\right)^2 + c\frac{n}{N}k\right) = O\left(\left(\frac{n}{N}\right)^2 + c\frac{m}{N}\right)$. The total number of inter-pattern conflicts for all N collections will grow according to $O(\frac{n^2}{N} + cm)$.

As can be understood from Section V-B, the number of transition rules will equal the total number of characters plus the number of intra- and inter-pattern conflicts, in case there are no common prefixes. Consequently, the total number of transition rules will grow according to

$$O\left(m + c'\frac{n^2}{N}\right). \tag{2}$$

Under the assumption that the B-FSM data structures grow linear with the number of transitions rules, (2) also represents a (worst-case) estimate of the storage complexity of the BFPM scheme, that does not take into account the positive effect of common prefixes on the storage efficiency. Eq. (2) shows that if the number of B-FSMs scaled linearly with the number of patterns, the total storage requirements would grow linearly with the total pattern and character count according to $O(m + c''n)$. This could be applied to a product roadmap, comprising multiple versions of the pattern-matching engine intended for rule sets involving different numbers of patterns. Assuming this linear relation, the total memory then has to increase linearly with the number of patterns that have to be supported by the various versions of the pattern-matching engine. By implementing the B-FSMs in all products with the same amount of local transition rule memory, the number of B-FSMs will also increase linearly with both the total memory size as well as the number of patterns, rendering the above linear relation valid.

C. Update Performance

The main focus of the design of the BFPM has been on the processing rate and storage efficiency. Although there exists enough potential to improve and optimize the update speed of the pattern compiler, for example, by replacing the linear searches in the current implementation by efficient prefix search algorithms [23], this has not yet been done. Therefore, the current implementation of the pattern compiler has a time complexity of at least $O(n^2)$, as can be understood from the description in Section V.

VII. Experimental Results

The BFPM pattern-matching operation has been validated using software and VHDL-based simulation models, as well as a hardware prototype implemented in an FPGA. The B-FSM data structures were generated by a prototype of the pattern compiler, implemented in C. Experiments have been performed using rule sets from commercial NIDSs as well as from the open-source NIDS Snort. Because of confidentially reasons, only results can be presented for the Snort rules.

A total of 2.8 K strings were extracted from the Snort rule set (end of 2004), having a total of 36.7 K characters. After filtering duplicate patterns, about 2 K unique patterns remained that consisted of 31.6 K characters. About 1.5 K patterns were case-insensitive with a total of 25.2 K characters and an average pattern length of 16.7 characters, whereas 0.48 K patterns were case-sensitive with a total of 6.4 K characters and an average pattern length of 13.2 characters.

A. Pattern Compiler

The case-insensitive and case-sensitive patterns were processed separately by the pattern compiler as described at the beginning of Section V-D. Table I shows the resulting numbers of transition rules and storage requirements for various numbers of B-FSMs. The table shows that for the case-insensitive and the case-sensitive patterns an average of about one transition rule per character can already be achieved with only 8 and 12 B-FSMs, respectively. Increasing the number of B-FSMs will even further reduce this.

The B-FSM data structures are comprised of BART-compressed transition-rule tables involving $P = 4$ rules per table entry, and a total table size of about 1 KB. For each B-FSM, all transition-rule tables were entirely filled, except for the last one, which was partially filled and contained space to store additional rules. Table I lists the memory that is effectively used to store the transition rules as well as the memory that is "allocated" to the transition-rule tables. The table shows that on average 3 to 5 bytes of memory are needed for each pattern character, if there are 12 or more B-FSMs. It also shows that all 2 K string patterns that were extracted from the Snort rule set, with a total of 31.6 K characters, can be stored in about 128 KB of memory.

The pattern compiler used an average processing time on the order of a second per pattern, using an Intel Pentium M processor running at 1.7 GHz and 1 GB of memory. A slightly optimized implementation involving a simplified pattern distributor, resulted in an average processing time on the order of a millisecond per pattern, at the cost of an approximate 15% increase in storage requirements.

B. B-FSM implementation

The B-FSMs have been implemented in various Xilinx FPGA technologies, the fastest being Virtex-4 [24]. Each transition-rule memory has been implemented using dual-port block RAM (Xilinx on-chip memory technology) and is shared between two B-FSMs that are connected to the independent access ports. The two B-FSMs can process two input streams

TABLE I
PATTERN COMPILER PERFORMANCE.

	1.5 K case-insensitive patterns, 25.2 K characters					0.48 K case-sensitive patterns, 6.4 K characters				
B-FSMs	rules	rules/char	memory	(allocated)	mem/char	rules	rules/char	memory	(allocated)	mem/char
4	39.5 K	1.57	183 KB	(188 KB)	7.4 B	11.1 K	1.75	49 KB	(52 KB)	7.9 B
6	32.1 K	1.27	149 KB	(152 KB)	6.0 B	8.6 K	1.36	38 KB	(39 KB)	6.1 B
8	25.8 K	1.02	117 KB	(120 KB)	4.7 B	7.7 K	1.21	34 KB	(37 KB)	5.4 B
12	22.0 K	0.87	99 KB	(104 KB)	4.0 B	6.6 K	1.03	29 KB	(36 KB)	4.6 B
16	18.9 K	0.75	83 KB	(92 KB)	3.4 B	6.2 K	0.97	27 KB	(36 KB)	4.3 B
20	18.8 K	0.74	82 KB	(93 KB)	3.3 B	5.8 K	0.92	26 KB	(35 KB)	4.1 B

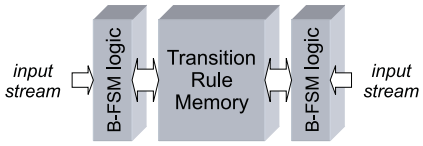


Fig. 14. Dual-port transition-rule memory.

TABLE II
STORAGE-EFFICIENCY COMPARISON (BASED ON [14]).

method	memory	mem/char
Aho-Corasick [6]	53.1 MB	2.8 KB
Wu-Manber [9]	29.1 MB	1.6 KB
Bitmap compr. Aho-Corasick [14]	2.8 MB	154 B
Path compr. Aho-Corasick [14]	1.1 MB	60 B

based on a single set of patterns "stored" in one transition-rule memory, thus realizing two independent pattern-matching channels connected to each transition-rule memory. This is illustrated in Fig. 14.

Because we did not have actual Virtex-4 hardware available at the time of writing, evaluating the B-FSM implementation for this technology was restricted to synthesis experiments using the Xilinx ISE Foundation tools (version 7.1i). These synthesis experiments have demonstrated the feasibility of running the B-FSMs at clock frequencies ranging from 100 to 125 MHz (for FPGAs of various speed grades), while performing one state transition per clock cycle, which includes address generation, memory access and rule selection as described in Section IV. With each state transition processing one input byte, this results in a processing rate ranging from 0.8 to 1 Gb/s for a single B-FSM (depending on the FPGA speed grade), and a total processing rate of 1.6 to 2 Gb/s for the two pattern-matching channels connected to each transition-rule memory. Based on similar experiments, processing rates of at least 2 Gb/s per channel have been estimated for ASIC implementations.

The small size of the B-FSM logic, which is negligible compared with the area consumed by the transition-rule memory, in combination with the high storage efficiency of the BFPM scheme enables an efficient implementation of the distributed B-FSM concept described in Section V-A. This allows the B-FSMs to be allocated in various ways to the input streams, making it possible to balance the total pattern-matching throughput with the total number of patterns that are supported. This is explained by the following example.

All 2 K string patterns extracted from Snort fit into 128 KB (as discussed above). The FPGA implementation described above provides two independent channels for matching two streams against these 2 K patterns. An FPGA with at least 512 KB of block RAM used as transition-rule memory could hold the following configurations:

- Four copies of the above 128 KB B-FSM structure, allowing 8 independent input streams to be matched against the set of 2 K patterns.
- Two copies of a 256 KB B-FSM structure, allowing 4 independent input streams to be matched against a set of about 4 K patterns (assuming a linear scaling of the storage requirements), or
- A single 512 KB B-FSM data structure, allowing 2 independent input streams to be matched against a set of about 8 K patterns (assuming a linear scaling of the storage requirements).

Higher aggregate processing rates can be realized and larger pattern collections can be supported by increasing the amount of block RAM used as transition-rule memory. Given that the largest Virtex-4 devices contain about 1 MB of block RAM, single-chip FPGA implementations will be able to support aggregate processing rates on the order of 2 to 10 Gb/s, for NIDS rule sets involving about 2 K patterns, whereas ASIC implementations are expected to achieve at least 20 Gb/s.

VIII. COMPARISON

This section provides a comparison of the storage efficiency and performance of the BFPM scheme with the Aho-Corasick and derived algorithms, to which BFPM is the most closely related as described in Section III. The comparison is based on the experimental results published by Tuck et al. [14] for 1.5 K unique patterns with a total of 19.1 K characters that were extracted from the Snort rule set in June 2003. These results include the storage requirements listed in Table II. Although a comparison based on the same patterns is preferable (but has not yet been performed), this table, in particular the average memory per character, provides a clear indication of the gain

in storage efficiency that is obtained by the BFPM scheme, which exceeds factors of 10 to 500 over the schemes listed in Table II.

By exploiting its high storage efficiency in combination with its small logic costs, BFPM is also able to achieve high aggregate processing rates by implementing multiple instances of the pattern-matching data structures and B-FSMs on the same chip. As a result, single-chip implementations are estimated to achieve processing rates of up to 10 Gb/s for FPGA technology and of at least 20 Gb/s for ASIC technology. Tuck indicates processing rates of up to 8 Gb/s for ASIC implementations of the compressed Aho-Corasick algorithms. Although it is difficult to make a fair performance comparison that takes all algorithmic and hardware-implementation aspects into account, the difference clearly shows the performance potential of the BFPM scheme.

IX. Conclusion

This paper has presented a new hardware-based scheme for pattern-matching, called BFPM, that meets the performance and functional requirements of new generations of NIDSs. BFPM builds on a novel programmable state-machine technology, called B-FSM, which is based on the concept of transition-rules that involve wildcard conditions and priorities. It appeared that the problem of simultaneously detecting a large number of patterns in an input stream can be very efficiently mapped on transition rules. Based on this observation, the BFPM employs two algorithms to convert a pattern collection into an optimized intermediate structure comprised of transition rules, which is then converted by the B-FSM compiler into a collection of hash tables that are directly processed by an array of B-FSMs in hardware.

The BFPM scheme achieves a high deterministic performance that is independent of input and pattern characteristics, in combination with high storage efficiency, dynamic updates, and several other features needed by NIDSs. BFPM achieves aggregate processing rates of up to 10 Gb/s using state-of-the-art FPGA technology, whereas ASIC implementations are expected to achieve at least 20 Gb/s. Experiments with actual NIDS rule sets have shown that BFPM is able to support all 2 K string patterns extracted from the Snort rule set, comprising a total of 31.6 K characters, in only 128 KB of memory, which is a significant improvement in storage efficiency over the state-of-the-art algorithms.

Ongoing research aims at increasing the B-FSM performance by processing multiple characters per state transition, while maintaining the processing rate of one transition per clock cycle. A related project investigates the use of the B-FSM technology as core component for a new kind of fast programmable accelerator engines, supporting a range of applications including pattern-matching and XML processing.

Acknowledgment

The author would like to thank Charlotte Bolliger for her excellent help with the preparation of this manuscript.

References

[1] W.R. Cheswick and S.M. Bellovin, "Firewalls and Internet security: repelling the wily hacker," Addison-Wesley, Reading, Mass., 1994.

[2] M. Roesch, "SNORT - Lightweight intrusion detection for networks," *Proc. LISA 99: USENIX 13th Systems Administration Conference*, pp. 229-238, November 1999.

[3] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24-32, March/April 2001.

[4] R. Sommer and V. Paxson, "Intrusion detection: Enhancing byte-level network intrusion detection signatures with context," *Proc. of the 10th ACM Conference on Computer and Communications Security*, pp. 262-271, October 2003.

[5] J. van Lunteren, "High-performance regular-expression matching," in preparation.

[6] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.

[7] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, Oct. 1977.

[8] B. Commentz-Walter, "A string matching algorithm fast on the average," *Proc. of the 6th Colloquium, on Automata, Languages and Programming*, pp. 118-132, July 1979.

[9] S. Wu and U. Manber,"A fast algorithm for multi-pattern searching," *Technical report TR-94-17*, Department of Computer Science, University of Arizona, May 1994.

[10] C. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection," *Proc. of the DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.

[11] M. Fisk and G. Varghese, "Applying fast string matching to intrusion detection" [Online]. Available: `http://woozle.org/~mfisk/papers/`.

[12] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435-2463, December 1999.

[13] S. Antonatos, K.G. Anagnostakis, and E.P. Markatos, "Generating realistic workloads for intrusion detection systems," *Proc. of the 4th ACM SIGSOFT/SIGMETRICS Workshop on Software and Performance*, pp. 207-215, January 2004.

[14] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *Proc. IEEE Infocom*, vol. 4, pp. 2628-2639, March 2004.

[15] R. Sidhu and V.K. Prasanna, "Fast regular expression matching using FPGAs," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 227-238, 2001.

[16] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111-120, April 2002.

[17] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31-38, April 2003.

[18] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 249-257, April 2004.

[19] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 21-23, April 2004.

[20] J. van Lunteren, "Searching very large routing tables in wide embedded memory," *Proc. IEEE Globecom*, vol. 3, pp. 1615-1619, November 2001.

[21] J. van Lunteren and A.P.J. Engbersen, "Fast and scalable packet classification," *IEEE Journal of Selected Areas in Communications*, vol. 21, no. 4, pp. 560-571, May 2003.

[22] J. Friedl, "Mastering regular expressions," second edition, O'Reilly, 2002.

[23] M.A. Ruiz-Sánchez, E.W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8-23, March/April 2001.

[24] G. Lara, "Virtex-4: breakthrough performance at the lowest cost," *Xilinx Xcell Journal*, issue 51, pp. 33-37, Winter 2004.