J Comput Virol
DOI 10.1007/s11416-006-0010-4

ORIGINAL PAPER

A parallel "String Matching Engine" for use in high speed network intrusion detection systems

Gerald Tripp

Received: 13 January 2006 / Accepted: 27 March 2006

Abstract This paper describes a finite state machine approach to string matching for an intrusion detection system. To obtain high performance, we typically need to be able to operate on input data that is several bytes wide. However, finite state machine designs become more complex when operating on large input data words, partly because of needing to match the starts and ends of a string that may occur part way through an input data word. Here we use finite state machines that each operate on only a single byte wide data input. We then provide a separate finite state machine for each byte wide data path from a multi-byte wide input data word. By splitting the search strings into multiple interleaved substrings and by combining the outputs from the individual finite state machines in an appropriate way we can perform string matching in parallel across multiple finite state machines. A hardware design for a parallel string matching engine has been generated, built for implementation in a Xilinx Field Programmable Gate Array and tested by simulation. The design is capable of operating at a search rate of 4.7 Gbps with a 32-bit input word size.

1 Introduction

Network intrusion detection consists of monitoring computer networks for various types of security attack. This can be network wide monitoring (network based) or it can be at each individual host computer in the system (host based). Basic network security is provided by network firewalls, which act as an intermediary between the Internet and a local network — these filter network traffic on the basis of header fields in the packets such as the source and destination IP address and TCP port numbers. This type of filtering is good at blocking a large proportion of unwanted incoming traffic. However, some network attacks may be targeted at machines such as web and mail servers that need to be visible through the firewall. In this case, it may be necessary to look inside each incoming data packet to determine whether it represents a potential threat. We may then wish to block that traffic (intrusion prevention) or be able to generate an alert that potentially malicious traffic is present (intrusion detection). The problem is that we may have no particular field to examine inside the packet, and may need to search the entire packet. This is the standard technique that we use for intrusion detection: we first look at the header fields of the packet to see if the packet is potentially of interest and if so we then search the content of the packet for one or more related intrusion detection "signatures". These signatures are short search strings which are chosen as representing a high probability of an attack occurring when present, whilst having a low probability of occurring otherwise.

A lot of current intrusion detection systems are software based, the most well known example probably being Snort [17]. Software solutions can however have problems when presented with a high network load. One solution can be to use host based intrusion detection and to require each computer to perform its own intrusion detection. This however can be targeted by denial of service attacks to put the intrusion detection software on individual machines under heavy load. Host based solutions are also only possible if we are able to add

G. Tripp (✉)
The Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, UK
e-mail: G.E.W.Tripp@kent.ac.uk

intrusion detection software to each host system, and this may not be the case with some embedded systems.

1.1 Summary of this paper

This paper looks at the string matching part of intrusion detection and describes how it is possible to build a "string matching engine" for implementation in a field programmable gate array (FPGA) that uses fine grained parallelism to improve its search rate. The method used is to operate on a multi byte input data word and to partition the matching operation between a set of finite state machines (FSMs), each of which processes one of the byte streams from a multi-byte wide network input and looks for parts of the search string. The results from these multiple FSMs are then combined in a particular way so as to determine whether a string has been matched across all the FSMs.

The next section describes the background and outlines some of the related work in this field. The following section describes the operation of the parallel string matching system proposed in this paper. The FSM implementation section gives details of an existing scheme for compact FSM implementation and an explanation of our modifications to this scheme. The software section gives the results of processing multiple search strings and the resource requirements for various string set sizes and implementation options. The next section gives details of a hardware design for a string matching engine and its performance and resource requirements. The final section gives conclusions and ideas for further work.

2 Background and related work

A lot of existing intrusion detection systems are software based, the most well known example probably being Snort [17]. Many improvements have been made to Snort by optimising the order in which data is compared. Work by Kruegel and Toth [13] uses rule clustering and is implemented as a modified snort rule engine. This uses decision trees to reduce the number of comparisons made against incoming network data and uses a multiple string matching algorithm based on the work by Fisk and Varghese [8].

A paper by Abbes et al. [1] describes a system using a decision tree in conjunction with protocol analysis. The protocol analysis uses a specification file for the protocol being monitored and performs "Aho–Corasick" [2] string matching on only the appropriate parts of the data stream. This technique reduces the overall workload and also reduces the number of false positives as compared with performing matching on the entire data packet or using simple offset and depth constraints.

Work by Paul [16] looks at distributed firewalls and implements stateful packet classification spread across consecutive firewalls. This helps to spread the workload between separate machines.

It can be difficult to perform intrusion detection in software at high network traffic rates and hardware solutions may be required. Software solutions being essentially sequential also suffer from performance problems as we increase the number of rules; [6] state that a software system with 500 rules may have difficulty in sustaining a throughput of 100 Mbps. Hardware solutions have different limitations; we can often increase the number of rules without affecting throughput because of the use of parallelism — the cost of increasing the number of rules may be an increase in hardware resource utilisation instead.

2.1 Overview of existing solutions

A number of hardware based string matching systems for intrusion detection have been described in the literature; an overview of some of the techniques is given below.

A product called ClassiPi from PMC-Sierra is described by Iyer, et al. [11], this is a classification engine and implemented as an application specific integrated circuit (ASIC). This device allows software-like algorithms to be used for various packet classification and packet inspection operations, including the use of regular expressions to search the contents of packets.

Work by Attig and Lockwood [3] uses Bloom filters to perform string searching. Bloom filters provide an efficient method to perform searching for a large number of strings in parallel, but suffer from the disadvantage of producing false positive matches. Attig and Lockwood show that Bloom filters can be used as a very efficient front end to remove the bulk of the network traffic that is known to be benign before input into a conventional software intrusion detection system.

Cho et al. [6] describe a system that uses multiple matching systems, each of which will search incoming network data for a set of distinct string "prefixes". For each possible string prefix, their system will lookup the remaining part of the string that must be compared sequentially against the incoming data to determine whether that string is actually present. Multiple strings with identical prefixes need to be distributed between different matching systems.

An interesting approach is taken by Baker and Prasanna [4], who have a series of input comparators for each data byte of interest — the output of these comparators each feed into a pipeline of flip-flops. Strings can be identified by the use of an AND function that looks for all the required data bytes for a string in the appropriate positions within the pipeline. They show that this can be extended to operate with multi-byte input data by the use of multiple sets of pipelines and looking for strings across the set of pipelines at all byte alignments.

2.2 Finite state machine approaches

A number of systems have been designed that use FSM to perform the searching — most of these use a deterministic finite automata (DFA) to implement string matching. This type of FSM has sets of states, inputs and outputs; the FSM can be in one of its states and there is a mapping between each pair of current state and input to the next state and output. When used in string matching, we use the FSM state to define how much of a string we have matched so far.

The approaches taken by Sugawara et al. [19] and by Tripp [21] is to first compress multi-byte input data into a number of different patterns that are of interest and then to use DFAs to perform string matching several bytes at a time. Moscola et al. [15] convert regular expressions into DFA that operate one byte at a time and show that this can be used to perform matching for standard spam-assassin rules without creating too many DFA states.

A different approach is taken by Franklin et al. [9], who implement non-deterministic finite automata (NFA) in hardware to perform matching of strings from the Snort rule set, this approach first being proposed by Sidhu and Prasanna [18]. This was extended by Clark and Schimmel [7] to operate with multi byte input data.

The text by Hopcroft et al. [10] gives a comprehensive coverage of Deterministic and Non-deterministic Finite Automata.

2.3 String matching algorithms

There are many string matching algorithms described in the literature, most of which were originally devised for software implementation. A hardware implementation has slightly different requirements than that for a software implementation and may well need to be less complex. For efficiency it is more common to build systems that work on a stream of data, rather than providing random access to the contents of a buffer; ideally we would like the string matching to operate at a deterministic rate to avoid the need for buffering.

The fastest method of matching strings is considered to be the Boyer–Moore algorithm [5] and its successors. This performs string matching on a "right to left" basis and skips forward on a mismatch. This gives an average performance that is usually sub-linear, but a worst case performance that may require us to look at some input bytes many times.

The "Knuth Morris Pratt" (KMP) algorithm [12], performs matching on a left to right basis and on mismatch will use the longest partial match as a starting point for further matching. The algorithm can be adapted to operate at deterministic data rate and not re-examine input data on a mismatch.

The Aho–Corasick algorithm [2] matches several strings at the same time. This works by constructing a trie containing the various strings and this is traversed as the data arrives. As with KMP, this can also be modified to operate at a deterministic rate only looking at each input data item once.

Both KMP and Aho–Corasick can be implemented by creating a FSM that operates at one input data item per clock cycle and are therefore ideal for hardware implementation. A common method of implementation for both these algorithms uses a maximum FSM size of an initial state and one state per search character (in one or all strings). When using Aho–Corasick, we would have fewer states when common prefixes of search strings enable us to share a FSM state. The state transition information in both cases will vary in complexity determined by whether on mismatch of a partly matched string there exists a suffix of the data matched that forms a smaller partial match of that string (or another).

3 Parallel string matching

From the work presented by Sugawara et al. [19] and Tripp [21], we can see that high performance can be obtained by creating a FSM that will match multiple bytes in the same clock cycle. However this has the overhead of compressing the input data so as to present a small input word to the FSM. A second issue is that the start and ends of strings have a high chance of appearing part way through an input data word, so we may need to match parts of the start and end of a string with "wild card" characters.

It is far easier to match data from an 8-bit input bus, but this does not give such good throughput. The solution proposed here is to use multiple finite state machines in parallel to process the input data. Course grained parallel FSM solutions have already been implemented, such as the work described by Moscola et al. [15], where input packets are allocated to a number of content scanners on a round robin basis. We propose a

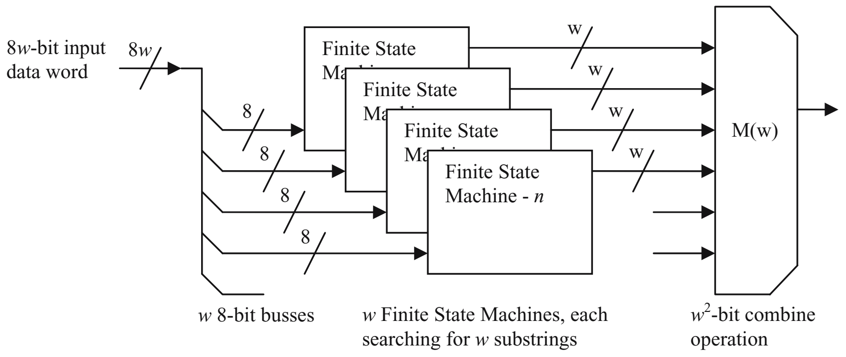


Fig. 1 Matching interleaved substrings

Fig. 2 Interleaved substrings

```
Word size = 4
Search string =      "the-cat-sat-on-the-mat"
Substring 0 =        " e   t   t   -   -   "      = "ett--"
Substring 1 =        "   -   -   -   t   m "      = "---tm"
Substring 2 =        "t   c   s   o   h   a "     = "tcsoha"
Substring 3 =        " h   a   a   n   e   t"     = "haanet"
```

(The substrings are sorted by the order of completion, the reason for which will be explained below.)

fine-grained from of parallelism, where multiple finite state machines process each packet in parallel.

3.1 Parallel finite state machines

The approach we take here is to provide a finite state machine for each byte stream from a multi-byte input data word. If we have a w-byte wide input word, then we can use w separate finite state machines, each of which are looking for all w instances of the "substrings" made up from a w-way interleave from the search string. An example of such a system is shown in Fig. 1. A related, but different, approach is taken by Tan and Sherwood, [20] who use multiple FSMs running in parallel to match a sequence of bits, with each FSM matching a particular bit position from the input data.

All w instances of our FSM are identical, and each will be looking for all w substrings. Each FSM has a w-bit Boolean "match vector" output to specify the substrings matched in any clock cycle. If we find all w substrings appearing in an appropriate order across all w finite state machines at the correct time, then we will have found our search string. We can see an example of a set of substrings of a given search string when $w = 4$ in Fig. 2.

3.2 Combining the output of multiple FSMs

By sorting our substrings on the basis of the order of completion of the match, we have a sequence in byte terms of w consecutive substring matches. However, we are processing our data on the basis of a w-byte input word. The string may be aligned in one of w different ways, with the last w bytes occurring in one or two input data words — the occurrence of each of the last w bytes of the search string relate to the instant when each of the related substring matches will occur. We define here an alignment of c as meaning that of the last w bytes of the search string, c of these will occur in one input word, followed by$(w - c)$ in the following input word, where $0 \leq c < w$.

Byte stream x is being monitored by finite state machine x. Each of the finite state machines is searching for all w substrings, and has a Boolean "match" output for each substring y. Thus we have a group of w^2 FSM outputs: $\mathcal{O}_{x\,y}$ where $0 \leq x < w$ and $0 \leq y < w$, relating to whether FSM x has detected substring y in the current clock cycle. We are also interested in whether string matches occurred in the previous clock cycle, and $\mathcal{O}'_{x\,y}$ is a delayed (pipelined) copy of $\mathcal{O}_{x\,y}$ from the previous clock cycle.

Taking the case where $w = 4$ and $c = 1$ for the string in Fig. 2, we have the alignment shown in Table 1.

Table 1 String match at alignment $c = 1$ ($*_S$ indicates when a match occurs for substring S)

Input byte	Input word					
	$n{-}5$	$n{-}4$	$n{-}3$	$n{-}2$	$n{-}1$	n
0		–	–	–	t	$m\,*_1$
1	t	c	s	o	h	$a\,*_2$
2	h	a	a	n	e	$t\,*_3$
3	e	t	t	–	$-\,*_0$	

We define $M_c(w)$ as being a Boolean operation specifying whether a match occurs at alignment c, in a system with a word size w. In our example above, we have $c = 1$ and $w = 4$; we can see from Table 1, that $M_1(4)$ is as shown in (1).

$$M_1(4) = \mathcal{O}'_{3\,0} \,.\, \mathcal{O}_{0\,1} \,.\, \mathcal{O}_{1\,2} \,.\, \mathcal{O}_{2\,3}. \tag{1}$$

This follows a very simple pattern, and we can produce a general formula for $M_c(w)$. Our complete string match is then defined as $M(w)$ which determines whether the match occurs at any of the w possible alignments. This is shown in (2)[1].

$$\begin{aligned} M_c(w) &= \bigwedge_{i=0}^{w-1} \text{if}(i \geq c) \text{ then } (\mathcal{O}_{(i-c)\,i}) \text{ else } (\mathcal{O}'_{(i+w-c)\,i}), \\ M(w) &= \bigvee_{c=0}^{w-1} M_c(w) \\ &= \bigvee_{c=0}^{w-1} \left(\bigwedge_{i=0}^{w-1} \text{if}(i \geq c) \text{ then } (\mathcal{O}_{(i-c)\,i}) \text{ else } (\mathcal{O}'_{(i+w-c)\,i}) \right). \end{aligned} \tag{2}$$

The combine operation $M(w)$ is independent of the search string and can be implemented as a fixed logic function for a given value of w. We also need $\sum_{x=1}^{w-1} x$ D-type flip-flops to generate the delayed versions of some of the inputs. As an example, the combine operation required for a system with a word size of 4 bytes is shown in (3).

$$\begin{aligned} M(4) = {} & \mathcal{O}_{0\,0} \,.\, \mathcal{O}_{1\,1} \,.\, \mathcal{O}_{2\,2} . \mathcal{O}_{3\,3} + \mathcal{O}'_{3\,0} \,.\, \mathcal{O}_{0\,1} \,.\, \mathcal{O}_{1\,2} \,.\, \mathcal{O}_{2\,3} \\ & + \mathcal{O}'_{2\,0} \,.\, \mathcal{O}'_{3\,1} \,.\, \mathcal{O}_{0\,2} \,.\, \mathcal{O}_{1\,3} + \mathcal{O}'_{1\,0} \,.\, \mathcal{O}'_{2\,1} \,.\, \mathcal{O}'_{3\,2} \,.\, \mathcal{O}_{0\,3}. \end{aligned} \tag{3}$$

This requires four 4-input AND gates, one 4-input OR gate and six D-type flip-flops.

3.3 Summary

In terms of overall complexity, the move from a standard byte-wide Aho–Corasick multi-string matching system to the technique described here requires us to replace a single FSM with w instances of a new FSM for matching sub-strings and one instance of the combine operation described above. The new FSM will have a similar number of states to the original, but will require a factor of w increase in the number of match outputs. Actual resource utilisation will depend on many parameters relating to the FSM implementation, as will be shown later. The resources required for the combine operation are trivial for small values of w — but will grow rapidly in size with w as it implements a w^2 input Boolean function.

4 FSM implementation

Each FSM has to be able to match multiple strings, and this can be done using the Aho–Corasick multiple string matching algorithm. As we are using a multiple string matching algorithm, then this can actually be used to perform substring matching for multiple search strings. There are many ways in which we can implement the FSM; the method chosen here is to use a table to hold the state transition information and then store the current state in a register. This approach has the advantage that we can have a fixed core of logic and memory for any FSM (up to a certain size) and then determine the operation performed by the FSM by specifying the contents of the FSM table. The FSM table can be implemented as a two dimensional array, with the current state and the input as the two index values — as shown in Fig. 3. Each element in the array specifies the next state of the FSM and any output. The FSM in Figure 3 is actually a "Mealy Machine", as it allows the output from the FSM to be dependent on both the state and the input data. The alternative is to use a "Moore machine" as shown in Fig. 4. The Moore machine has an output that is dependent only on the current state and not the input value — this may be simpler in terms of table based implementation as the main FSM table only contains the next state and we only need a one-dimensional state decoder table for the output.

In terms of string matching, the disadvantage of the Moore machine is that we often need more states than for a Mealy machine as we need to have one or more "Terminal states" that the machine can pass through to indicate successful matches. The Moore machine can however be quite good for implementing Aho–Corasick

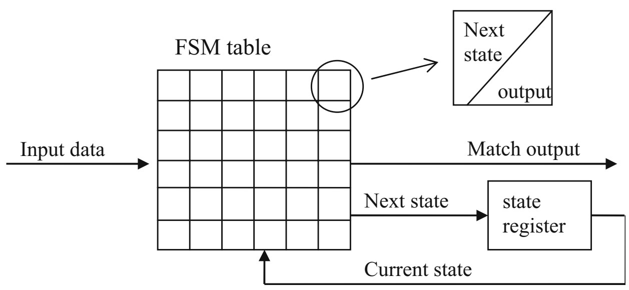


Fig. 3 Table based implementation of a "Mealy Machine"

[1] Note that in (2), we use the signs $\bigvee$ (respectively $\bigwedge$) to represent the Boolean "inclusive-or summation" (respectively the "and product").

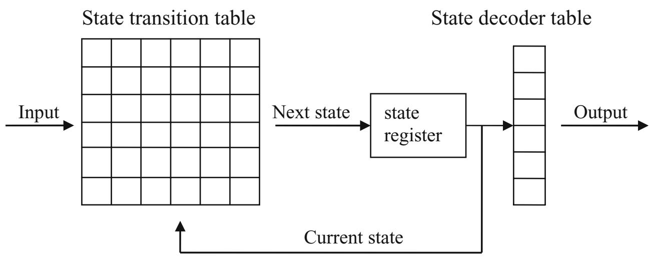


Fig. 4 Table based implementation of a "Moore Machine"

string matching as we potentially require a large number of "match" outputs. The state decoder table can often be replaced by a logic function, but we need to be aware that strings can be identified as matching in more than one state, for example where one string forms part of another — so this may require us to vary the logic function dependent on the search strings used.

4.1 Memory resources

A problem with a simple table based approach is that this can easily require a large amount of memory. For a Mealy machine with s states, i input bits and o output bits, we have a memory requirement M in bits as shown in (4).

$$M = \left(\lceil \log_2 s \rceil + o \right) . 2^{i+\lceil \log_2 s \rceil}. \tag{4}$$

We may find in practice that a string matching FSM presented with a raw data bus as its input, may only be interested in a small number of different input values. We can reduce the complexity of the FSM by first compressing the input data so that we have a compact set of input values representing the values of interest and a single value that represents all other characters. By compressing the input data, it may then be possible to represent this with a smaller bus width as input to the FSM and hence reduce the FSM table sizes.

In practice, even after input compression, the state transition table may be very redundant. We can see this in the example shown in Table 2, which implements KMP based string matching of the string "attack", using a table based Moore machine[2]. This implementation of the KMP algorithm is modified to perform matching at a rate of one character per FSM cycle — and is the same as a similarly modified Aho–Corasick implementation with the same single string.

Table 2 State transition table for a KMP implementation for the string "attack"

Current state	Input				
	0 (z)	1 (a)	2 (c)	3 (k)	4 (t)
0 ()	0	1	0	0	0
1 (a)	0	1	0	0	2
2 (at)	0	1	0	0	3
3 (att)	0	4	0	0	0
4 (atta)	0	1	5	0	2
5 (attac)	0	1	0	6	0
6 (attack)	0	1	0	0	0

Table 3 Default and difference arrays

Current state	Input				
	0(z)	1(a)	2(c)	3(k)	4(t)
Difference array					
0 ()					
1 (a)					2
2 (at)					3
3 (att)		4			
4 (atta)			5		2
5 (attac)				6	
6 (attack)					
Default array					
	0	1	0	0	0

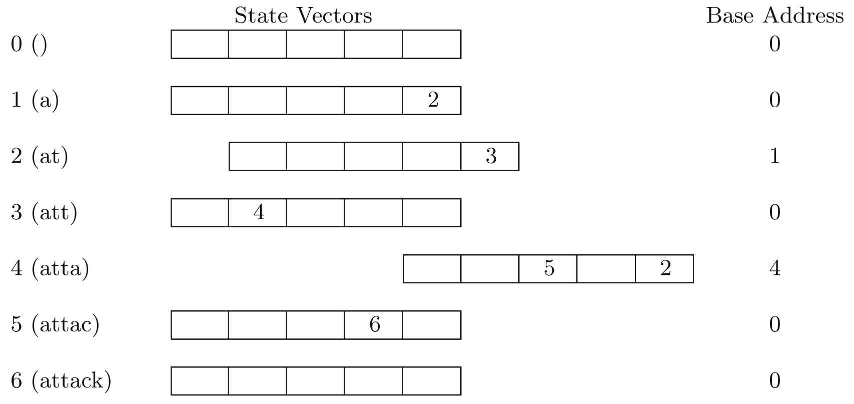
The state transition table in Table 2 contains the next state for the FSM, based on the current state and input. The input value z indicates that the input is a value that is not equal to any of the characters that the FSM is interested in. The FSM in this example has states that relate to the number of characters in the string that have been matched so far; with the string match being indicated when the FSM is in state 6. We can see that in this example, many of the values of next state are 0 which relates to the IDLE (or initial) state, and 1 which relates to the first character having been matched.

4.2 Packed transition tables

A mechanism for compacting the state transition table was described by Sugawara et al. [19]. Their method works on the basis that for a given input value I, a large proportion of transition table entries for current state S

[2] This particular example was chosen as it was found to produce tables that were small enough to include within the body of this paper, both for this subsection and those following.

Table 4 Packed array creation



Packed Array

Next State		4		6	2	3	5		2
Base		4		0	1	0	0		1
Tag	-1	3	-1	5	1	2	4	-1	4

Default Array

Next State	0	1	0	0	0
Base	0	0	0	0	0

will be the same as for that input in the IDLE state. The algorithm uses a default array that contains table entries for all input values of *I* in the IDLE state. All we need in addition to this are the entries from the full state transition table that differ to the entries in the default array — this difference array is typically very sparse. The remainder of this sub-section gives an example to illustrate how the algorithm by Sugawara et al. [19] operates.

Taking the example used in Table 2, we can generate a default array that contains the state transition table entries for all input values in the IDLE state. We then generate a difference array that contains all the table entries that are different to the values for the given input recorded in the default array. These are shown in Table 3. To find the next state for any current state and input, we first look in the difference array for an entry. If there is no entry in the difference array, then we use the value for the current input from the default array instead.

Although this works well, we do not save any space unless we have a more compact way to store the difference array. This difference array is decomposed into a series of state vectors, and these are packed together (overlapping) into a one-dimensional packed array — carefully avoiding any collisions between active entries. Each entry in the packed array is tagged with the current state it belongs to — with, in this example, "−1" being used to flag entries that are unused. To retrieve an entry from the packed array we need to know the base address of the state vector for the current state in the packed array and then use the current input as an offset from that point. If the entry fetched from the packed array has a tag that is equal to the current state, then we have found a valid difference array entry — if not, there is no entry for the current state and input in the packed array, so we use the value from the default array for the current input instead.

We can see an example in Table 4 of how the packed array for our example can be created. To improve performance, each entry (in both arrays) also contains the base address of the state vector in the packed array for the next state. This packed array implementation can be implemented within a FPGA, and will operate at a rate of one input value per clock cycle. A schematic of the design used by Sugawara et al. [19] is given in Fig. 5. The two blocks of RAM are used to hold the default and packed arrays. The current state is held in the register, and multiplexers are used to select between the default and packed arrays depending on whether the tag value from the packed array matches the current state.

This algorithm gives a significant memory saving for large FSMs, as we avoid the use of two dimensional arrays. The creation of this packed array is actually a search operation which requires us to scan through the packed array that we are building trying to find a fit for all the valid entries for each state vector — there are therefore many different packed array contents possible for any particular set of data. It is difficult to give a figure for the resource utilisation as the packed arrays will vary in density, as there are usually many entries in the packed array that are never used.

Fig. 5 Finite state machine implementation from [19]

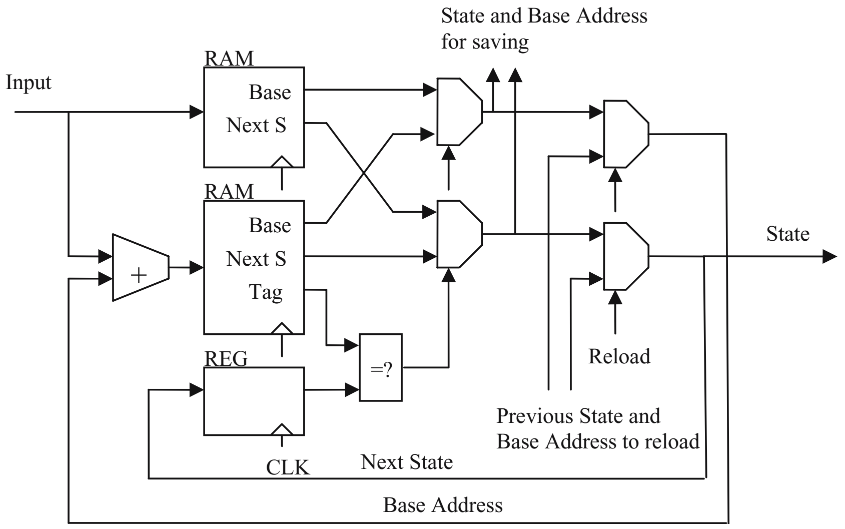


Table 5 Bitwise exclusive-or based packed array creation

```
                 State Vectors                         Base Address
0 ()        |    |    |    |    |    |                      0
              0    1    2    3    4
1 (a)       |    |    |    |    | 2  |                      0
              0    1    2    3    4
2 (at)      |    |    |    |    |     | 3  |                 1
              1    0    3    2         4
3 (att)     |    | 4  |    |    |    |                      0
              0    1    2    3    4
4 (atta)    | 5  |    |    |    |          | 2  |            2
              2    3    0    1              4
5 (attac)   |    |    |    | 6  |    |                      0
              0    1    2    3    4
6 (attack)  |    |    |    |    |    |                      0
              0    1    2    3    4
```

Packed Array

Next State	5	4	-	6	2	3	2
Base	0	2	-	0	1	0	1
Tag	4	3	-1	5	1	2	4

Default Array

Next State	0	1	0	0	0
Base	0	0	0	0	0

4.3 Modifications to the FSM implementation

For our work, the design in Fig. 5 was setup to operate on one packet at a time by providing a restore input corresponding to the IDLE state. After preliminary testing of the FSM design, it was found that performance was limited by the adder carry chain used in the implementation of the "+" operation that provides the index into the packed array. The '+' operator is the obvious way of performing the indexing operation into the packed array — but not the only way. In practice, we are not interested in where the actual entries are located within the packed array, so long as whatever indexing operation we use is consistent and that we never have more that one offset value (for a given base address) that takes us to any particular location in the packed array. On investigation, it was found that it was possible to replace the '+' operator with bitwise exclusive-or (XOR) — this has the advantage of being faster as there is no need for the carry chain that we needed for the '+' operation. The XOR has rather a different effect (than using '+'), as for an offset value represented by an i bit input, the different offsets will all be mapped within a 2^i size block of entries that starts on a 2^i boundary. The locations in the packed array for a group of consecutive offsets will most often no longer be in consecutive locations within the packed array.

If we take the width of the base address in memory as b bits, and the width of the input as being i bits, then assuming that $b \geq i$, then we have a group of $b - i$ bits from the base address that specify a particular 2^i sized "page" in memory, and a group of i bits from the base address that perform a rearrangement of the order of the offset values of the input within that page.

Table 5 shows how a packed array can be built for our difference array from Table 3, this time using XOR based indexing. Again we perform a search operation to pack the difference array into a one-dimensional table. This time, rather than moving down the packed array with a pattern of entries, we look at 2^i different arrangements of the offset values within a particular page. If we have no success at fitting these within one page we move to the next and so on.

The XOR based indexing has some advantages in terms of having different arrangements of the entries within a single page, but a disadvantage that all the entries for a particular state vector need to be in the same page. To show how the offset values in Table 5 are rearranged, these are indicated below each of the state vectors. The results of the XOR operation between the base address and the offset for values of 0–7 are shown in Table 6 to help illustrate how this rearrangement works.

4.4 Summary

Section 4 has shown the general background to table based FSM implementation and the method used by Sugawara et al. [19] to reduce the FSM memory requirements. Our proposal, as explained in Sect. 4.3, is to modify this algorithm to use bitwise exclusive-or in place of the '+' operator used for indexing, as this should enable a faster hardware implementation because we will no longer need the adder carry chain that was previously required.

5 Software

Rather than generating a specific piece of hardware for a given rule set, it was decided that we should identify an efficient size of "string matching engine" and then instantiate a number of these to cover the set of strings. We will not know in advance how many strings will fit into a FSM of any particular size, as this will depend on how compact the packed array can be made. The best size of FSM will depend on a number of factors, but will relate in particular to the memory resources available in the hardware. As we don't know in advance how many strings we can fit into a given FSM, we need to take an iterative approach and try increasing numbers of strings to see how many will actually fit.

Software was written to take a set of strings and to build an Aho–Corasick trie for performing the substring matching for the parallel string matching scheme outlined in Sect. 3. The design was optimised using standard techniques to enable the matching to be performed at a rate of one byte per clock cycle. From this, a state transition table was produced and then compressed using the technique described by Sugawara et al. [19] and outlined in Sect. 4. The FSM was provided with a state decoder table that produces a separate match output for each substring of each search string — this potentially has some redundancy, as there may be substrings that relate to more than one search string, but it enables us to have a system that has a fixed core of logic that is independent of the search strings used. As strings shorter than the word size can generate a match from a subset of the FSMs, we allow one or more of its substrings to be the "null string" which will match in any state, thus we always require a match from all FSMs irrespective of search string length.

The first stage was to choose a sensible size for the FSM. The software was modified so that instead of reading in all the search strings, it stops after a certain limit of search characters had been exceeded and the memory resources required for that amount of search characters reported. This was repeated for a range of maximum numbers of search characters – the search strings being taken from a randomised order set of case dependant rules from the hogwash [14] "insane" rule set. The operator for the packed array index was chosen initially as the ADD operator. It is important to note here that these are experimental results and report the resources required with the above software and the randomised rule set noted above. As previously explained there is quite a lot of variation in the amount of memory required for the packed array, depending on the packing density that is achieved.

The tests were performed for a range of input bus sizes, and the memory requirements for a single 8-bit slice are given in Fig. 6. We see that the amount of memory required increases with the input bus width; a major component of this increase is due to the state decoder, as the memory width of this lookup table is equal to the

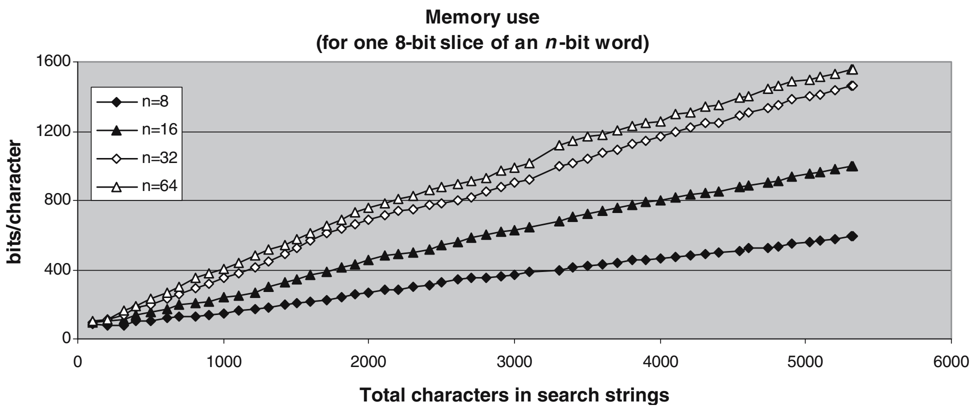


Fig. 6 Memory use

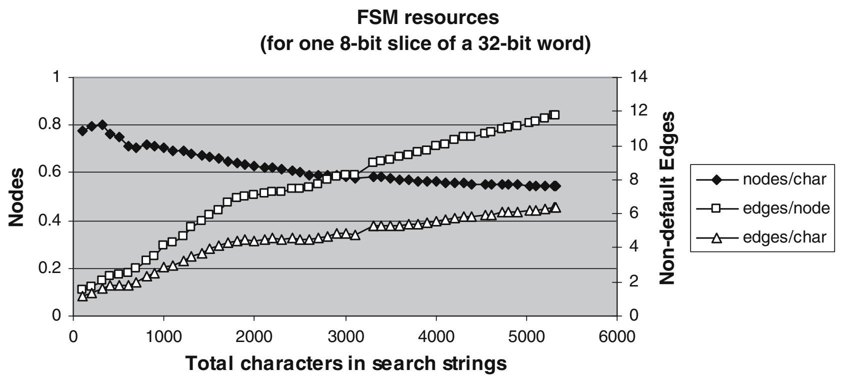


Fig. 7 FSM resources

total number of substrings, and the number of substrings increases with the number of 8-bit slices.

The traces in Figure 6 are roughly linear for a range of total characters from 200 to 2,000. The approximate memory requirements B_w in bits for word size w and character count s in this range are shown in (5). Memory usage will of course vary with the particular set of strings chosen.

$$B_8 = 0.10s + 51, \qquad B_{16} = 0.19s + 53,$$
$$B_{32} = 0.33s + 33, \qquad B_{64} = 0.35s + 54. \tag{5}$$

When we get to a word size of 64-bits, the total amount of memory required does not increase as much as expected, as there are an increasing number of short identical substrings, including null strings. Calculations here show the exact amount of memory required — in practice the memory will only be available in particular sizes, as shown later.

Interestingly, the amount of memory required per search character increases with the total amount of characters in the search strings. This is partly due to the memory requirements of the state decoder, but this effect is present even if we do not take this into account. We would expect to get some gain as we increase the number of search strings as we should have nodes within the trie shared between multiple search strings. We can see this effect in Fig. 7. The number of trie nodes per search character decreases with the total number of search characters as expected; however, this effect is counteracted by the increase in the number of "non-default edges". The "non-default edges" are transitions from one node to another that are recorded in the packed array and this relates to the FSM becoming more complex and there being more interconnectivity between nodes. The overall effect is that the number of non-default edges per search character increases with the number of characters. The total memory requirements for the packed array are roughly proportional to the number of non-default edges, as each "non-default edge" will require its own entry in the packed array.

All three traces in Fig. 7 are approximately linear up to a total of 1,700 characters in the search strings. The approximate results in this range, for a total of s characters are shown in (6).

$$\begin{aligned}\text{nodes/char} &= 0.79 - 9 \times 10^{-5}s,\\ \text{edges/node} &= 3.3 \times 10^{-3}s + 0.82,\\ \text{edges/char} &= 2.0 \times 10^{-3}s + 0.81.\end{aligned} \tag{6}$$

From Fig. 7, it can be seen that when using this style of implementation it is not necessarily the best option to have large Aho–Corasick FSMs. The per search character resources used appear to be lower when only a

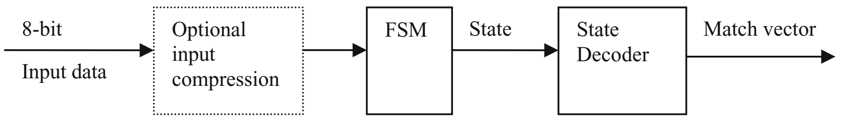


Fig. 8 One 8-bit slice of matching system

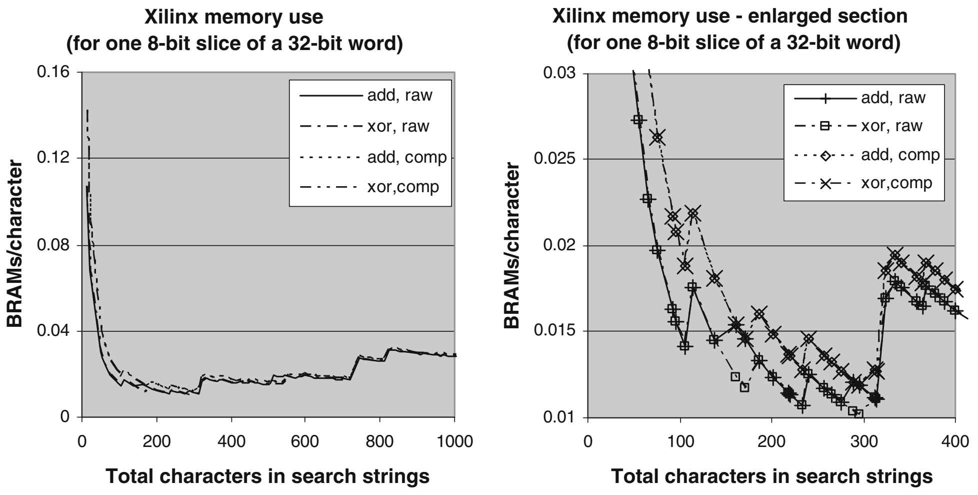


Fig. 9 Xilinx memory use versus the number of search characters

small number of strings are searched for; this however will be dependent on the sizes of memory available for the various FSM tables.

5.1 Determining an optimal FSM size

The software was modified to re-run a number of tests for a fixed input word size of 32-bit, using variations of the algorithms. The tests were run for an increasing number of search strings and the total memory resources required were calculated for implementation within a Xilinx Virtex-II FPGA [22] – this type of FPGA contains 18 Kbit Block RAM primitives (BRAMs) that can be configured in various ways. As an experiment, we also test the effect of preceding each FSM input with a custom built compression table to reduce the redundancy in the input data — as shown in Fig. 8. The tests were run with either raw or compressed input data and with either ADD or XOR used for packed array indexing. The results are shown in the left graph of Fig. 9; the four traces are very close together and an enlarged section, where the resource utilisation is the lowest, is given in the right graph for clarity.

The use of compression did not have much effect when we used a large number of strings, however with a small number of strings the memory used increased because the extra memory needed for input compression was greater than any memory saved within the FSM tables. This is perhaps to be expected when we use the packed array implementation, as we are primarily just concerned about how many entries we need to make in the packed array. The only time when input compression would be particularly important would be for very small FSMs with a small packed array. The choice of ADD or XOR algorithms had very little effect. From the results shown in Fig. 9, the best option appears to be a small FSM implementation dealing with a maximum of 200–300 search characters, and one which uses three BRAM components. The best results for each of the four algorithm combinations are shown in Table 7.

Table 6 Results of the bitwise exclusive-or function applied to input value 0 to 7

Base Address	Offset							
	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 7 Maximum number of search strings for a three BRAM implementation

FSM input	Packed array index	Search strings	Search characters
Raw	ADD	22	275
Raw	XOR	24	295
Compressed	ADD	18	234
Compressed	XOR	18	234

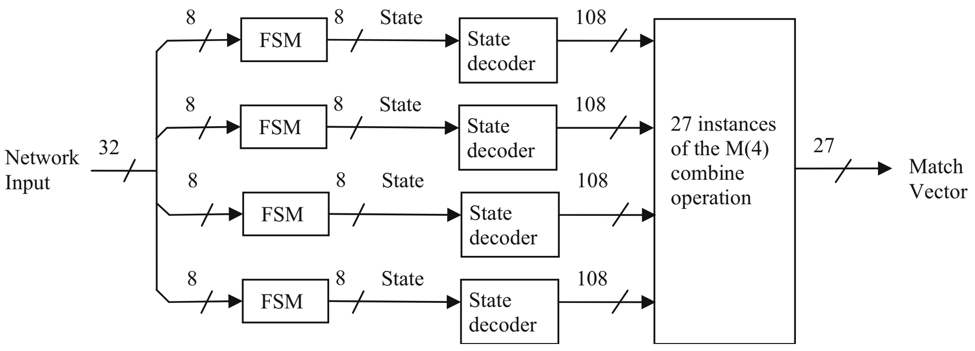


Fig. 10 Matching engine

5.2 Summary

A number of experiments have been performed to determine a suitable size and implementation options for constructing a FSM for a "string matching engine". We found that the choice of using ADD or XOR as the method of indexing into our packed array had little effect in the memory resources used for the FSM implementation – so changing the implementation to use XOR does not appear to be a problem. In many systems, the use of input compression can have a significant impact on the amount of memory resources used, however, in this example we found that we often ended up using more resources because of the overhead of the compression table. In our example, we are dealing with an 8-bit input bus, and this is smaller than the address input to the FSM packed array memory. How far apart the table entries are spread out inside the memory does not appear to have much effect on the memory requirements. In other cases the input compression may still be important, such as when the memory address input would ideally be smaller than the raw data input width.

For this particular implementation, we found that we used fewer memory resources per search character by using a relatively small FSM design. This could be used in a hardware implementation by instantiating a number of small string matching engines, rather than one large one. Rule sets such as those defined by Snort will allow us to have content matching that is case independent. We could deal with this by allocating these strings to separate 'string matching engines' to the ones used for strings that are case dependent and pre-pending an input function that maps all upper case letters to lower case.

6 Hardware implementation

A VHDL model was built of a 32-bit string matching engine that consisted of four 8-bit wide matching "slices" and a unit to combine together the results, as shown in Fig. 10. On the basis of the results above, a decision was made not to use input compression and to use the XOR function for indexing into the packed array for the FSM. The VHDL model was tested by simulation, the design synthesised and built for a Xilinx XC2V250-6 FPGA to determine its performance and resource utilisation. The design was also simulated "post place and route" to test the resulting FPGA design. The parameters of the FSM design were taken from the rule processing results of the previous section. Each FSM has an 8-bit input, an 8-bit state variable and a 108-bit[3] substring match output. Four instances of the FSMs were used with a fixed combine operation to generate a matching engine having a 32-bit data input and a 27-bit match output. This is capable of matching up to 27 search strings in parallel, depending on the length of the strings. The use of the match vector output enables us to indicate matches of multiple search strings occurring at the same time; this match vector output could be used to generate an indication of which strings occurred within a given input data packet (including the detection of multiple matches of different strings).

6.1 Performance and resource utilisation

The VHDL model was first configured for using a bitwise exclusive-or operator for the FSM table index operation and this gave a minimum clock cycle time of 6.7 ns (149 MHz) — given the 32-bit input, this corresponds to a search rate of around 4.7 Gbps.

The resources required for a Xilinx XC2V250-6 FPGA were as follows:

- 12 Block RAM components (out of a total of 24),
- 250 logic slices (out of a total of 1,536).

We can see from the above that the size of any design will be limited by the Block RAM resources. The FPGA

[3] Note: The value of 108 was chosen as it is a multiple of one of the BRAM memory widths, which is 36-bits.

Table 8 A few of the test patterns used and the expected results

Pattern	Byte position that match occurs "cybercop"	 "gOrave"	 "login: root"
- - - -cybercop====================	11		
- - - -ycebcrpo====================			
- - - -ybcecorp====================			
- - - -cybercybercop===============	16		
- - - -gOrave======================		9	
- - - -login: root=================			14
- - - -logOrave====================		11	
- - - -killogin: root==============			17

component used as the target of these experiments is however, by current standards, rather small. Taking the top of the range Virtex4FX FPGA as a comparison, we should be able to fit 46 of these matching engines within the FPGA, using all of the BRAM resources and around 20% of the logic slices. This would enable us to perform a parallel search of a maximum of 1,242 search strings. This will depend however on string length, and a conservative estimate of 20 search strings per matching engine would give us around 900.

For comparison, the VHDL model was rebuilt to use an ADD operator for the packed array indexing and this gave a minimum clock cycle time of 8.4 ns (119 MHz — giving a 3.6 Gbps search rate). This confirms the earlier assertion that using the XOR operator for the packed array indexing would be faster than using ADD.

6.2 Testing

The design has been tested by simulation with a large set of artificial input data containing various combinations of the strings being searched for, including: isolated instances of search strings; combinations of search strings at various spacing and overlap; and some strings rearranged in all 24 variations of the byte ordering in a group of 4 bytes. These tests were repeated for all four byte alignments of the input data — giving a total of 288 test cases. The results of the tests were compared with the expected outcomes to ensure that the search strings only matched as and when was expected. All tests passed correctly both for the original VHDL design and the post place and route simulations. A few examples of patterns used to test matching and the expected results are shown in Table 8.

7 Conclusion

This paper describes the design and simulation of a parallel algorithm for the implementation of high speed string matching; this uses fine-grained parallelism and performs matching of a search string by splitting the string into a set of interleaved substrings and then matching all of the substrings simultaneously.

We show that the FSM implementation technique described by Sugawara et al. [19] can be modified by the use of bitwise exclusive-or in place of ADD for the indexing operation to improve its performance. We also see that this implementation can be optimised in terms of resource utilisation by the choice of FSM size.

A VHDL model of a string matching engine based on the above ideas has been produced, synthesised and built for a Xilinx FPGA and tested via simulation. The results show a search rate of around 4.7 Gbps for a 32-bit input word. The design is table based and changes to the search strings can be made by generating new contents for the tables rather than having to generate a new logic design — this is particularly important for systems being updated in the field.

7.1 Future work

One area where the resources in this design could be reduced is in the state decoder table – which accounts for 50% of the memory resources. This gives a substring match vector for the current state of the FSM, thus showing which substrings match in a given state. This table could be replaced with a piece of logic, but this would need to be rebuilt for every set of strings.

Further work is needed to see if the memory requirements for the state decoder can be decreased, possibly taking advantage of the redundancy that exists within this table. This could, for example, be replaced by a two stage decoder design. Finally, it would also be interesting to see if any parts of the state decoder could be implemented as fixed logic.

References

1. Abbes, T., Bouhoula, A., Rusinowitch, M.: Protocol analysis in intrusion detection using decision tree. In: Proceedings of international conference on information technology: coding and computing (ITCC'04), Volume 1 (pp. 404–408). Las Vegas, Nevada (2004)
2. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun ACM **18**(6), 333–340 (1975)
3. Attig, M., Lockwood, J.W.: SIFT: snort intrusion filter for TCP. In: Proceedings of IEEE symposium on high performance interconnects (Hot Interconnects-13). Stanford, California (2005)

4. Baker, Z.K., Prasanna, V.K.: A methodology for synthesis of efficient intrusion detection systems on FPGAs. In: Proceedings of IEEE symposium on field-programmable custom computing machines FCCM '04. Napa, California (2004)
5. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. commun. assoc. comput. mach. **20**(10), 762–772 (1977)
6. Cho, Y., Mangione-Smith, W.: Deep packet filter with dedicated logic and read only memories. In: Proceedings of IEEE symposium on field-programmable custom computing machines FCCM '04. Napa, California (2004)
7. Clark, C., Schimmel, D.: Scalable multi-pattern matching on high-speed networks. In: Proceedings of IEEE symposium on field-programmable custom computing machines FCCM '04. Napa, California (2004)
8. Fisk, M., Varghese, G.: An analysis of fast string matching applied to content-based forwarding and intrusion detection (2001) (successor to UCSD TR CS2001-0670, UC San Diego, 2001). Retrieved 9 March 2006, from http://public.lanl.gov/mfisk/papers/setmatch-raid.pdf
9. Franklin, R., Carver, D., Hutchings, B.L.: Assisting network intrusion detection with reconfigurable hardware. In: Proceedings of IEEE symposium on field-programmable custom computing machines FCCM '02, pp.111-120. Napa, California, USA (2002)
10. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages and computation, 2nd edition. Addison-Wesley, Reading (2001)
11. Iyer, S., Rao Kompella, R., Shelat, A.: ClassiPi: an architecture for fast and flexible packet classification. IEEE Network, **15**(2), 33–41 (2001)
12. Knuth, D.E., Morris J.H., Pratt, V.B.: Fast pattern matching in strings. SIAM J Comput, **6**(2), 323–350 (1977)
13. Kruegel, C., Toth, T.: Using decision trees to improve signature-based intrusion detection. In: Proceedings of the 6th symposium on recent advances in intrusion detection (RAID2003), Lecture Notes in Computer Science, LNCS 2820, pp. 173–191. Springer Berlin Heidelberg New York (2003)
14. Larsen, J., Haile, J.: Securing an unpatchable webserver … HogWash. Retrieved 9 March 2006, (2001) from http://www.securityfocus.com/infocus/1208
15. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE symposium on field-programmable custom computing machines FCCM '03. Napa, California (2003)
16. Paul, O.: Improving distributed firewalls performance through vertical load balancing. In: Proceedings of third IFIP-TC6 networking conference, NETWORKING 2004, Lecture Notes in Computer Science, LNCS 3042 pp. 25–37. Springer Berlin Heidelberg New York (2004)
17. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of LISA '99: 13th systems administration conference, pp. 229–238. Seattle, WA : USENIX (1999)
18. Sidhu, R. Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proceedings of the 9th international IEEE symposium on FPGAs for custom computing machines, FCCM'01. Rohnert Park, California, USA (2001)
19. Sugawara, Y., Inaba, M., Hiraki, K.: Over 10 Gbps string matching mechanism for multi-stream packet scanning systems. In: Proceedings of field programmable logic and applications, 14th international conference, FPL 2004, pp. 484–493. Springer Berlin Heidelberg New York (2004)
20. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: The proceedings of the 32nd annual international symposium on computer architecture (ISCA 2005). Madison, Wisconsin, USA (2005)
21. Tripp, G.: A finite-state-machine based string matching system for intrusion detection on high-speed networks. In: Paul Turner, Vlasti Broucek (eds) EICAR conference best paper proceedings, pp. 26–40. Saint Julians, Malta (2005)
22. Xilinx Virtex-II Platform FPGAs: complete data sheet – product specification. (2005). Xilinx Inc. Retrieved 9 March 2006 from http://direct.xilinx.com/bvdocs/publications/ds031.pdf