Computer Science at Kent

An Intrusion Detection System for Gigabit Networks – Architecture and an example system

Gerald Tripp

Technical Report No. 7-04
April 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

An Intrusion Detection System for Gigabit Networks
Architecture and an example system

Gerald Tripp

Technical Report No. 7-04
Computing Laboratory, University of Kent. CT2 7NF. UK
E-mail: G.E.W.Tripp@kent.ac.uk

April 2004

Abstract

The aim of this work is to investigate the effectiveness of a finite state machine (FSM) based string-matching scheme for the implementation of high-speed network intrusion detection systems. The work uses standard RAM based techniques for the FSM implementation, but provides a per-FSM input stream consisting of symbols representing multi-byte patterns that appear in the input data. Multiple search strings are processed in parallel using multiple FSMs. This pre-FSM classification stage is used to reduce the redundancy in the input data stream (as seen by an individual FSM) and hence allows a FSM to be implemented with relatively small resources that is able to operate on multiple bytes per clock cycle. The benefit of this approach is that in operating on a relatively large number of input data bits per clock cycle, we are able to cope with an increased network throughput.

An example architecture is described along with an associated compiler. The compiler takes a set of intrusion detection rules, generates the various tables required for system implementation and also provides a high level simulation against some simple synthesised network data. Resource utilisation is presented for a range of input word sizes.

1 Introduction

The aim of this work is to investigate the effectiveness of using a FSM based string-matching system with a pre-FSM "classifier stage" for use in high-speed network intrusion detection systems.

1.1 The problem

Intrusion detection consists of monitoring network traffic either at the host machines themselves (Host-based) or by independently monitoring the network (Network-based) for various kinds of security threat. A general characteristic of intrusion detection is the need to be able to inspect packets arriving from a network.

A sub-set of this problem is packet classification, where we wish to inspect the content of the packet headers. Packet classification's primary role tends to be within routers either to perform the packet routing itself or, more specifically, to identify packets as belonging to for example a particular service class. Many algorithms have been developed for packet classification and a good overview of techniques is given by Gupta and McKeown [1].

Intrusion detection systems are more complex than simple packet classification in that they need to inspect the packet content (a.k.a. deep packet analysis) as well as the packet headers. The general case becomes more complex in that we do not necessarily have a specific

location within the packet that we wish to inspect – the normal mechanisms therefore tend to include pattern matching systems that can scan for a string or regular expression anywhere within the packet.

Until recently, many intrusion detection systems were software based – one well-known example being the software intrusion detection system called Snort [2]. However as we move to higher network speeds, the conventional software solution can have difficultly in keeping up and some form of hardware assist may be required. We can of course partition the load by using host-based rather than network-based detection systems – however, this is only possible in situations where we have the ability to run our own software on the host platform. Host based detection may not be suitable for many simple embedded systems or systems without the performance to carry the additional load of running intrusion detection software.

Once we have identified a potential threat, there are a number of actions that can be taken as defined by the rule that has been matched. This will typically consist of generating a report detailing the perceived threat, but could also require suspect network packets, or a complete network connection, to be dropped.

1.2 Objectives

The aim of this paper is to introduce the ideas behind this work on intrusion detection, look at the details of how an architecture can be modelled and then give a worked example of a sample architecture, a summary of a basic compiler and simulator and results on the resources used by this technique.

1.3 Summary of this paper

The background to this work is outlined in the next section. Section 3 gives the proposed technique that is examined in this work. In section 4, we give details of how string matching can be performed at one word per clock cycle, including how multiple characters can be processed in a single clock cycle. The following section, 5, gives details of the high level components used in this architecture; with the details of how this is modelled and compiled into a hardware solution given in section 6. An basic example architecture and the compilation and simulation tools created for this are explained in section 7. Finally the results of simulating this architecture for a small rule set and some network traffic is given in section 8; with conclusions and ideas for further work in section 9.

2 Background

2.1 Gigabit network intrusion detection systems

At network speeds of 1 Gbps or above, it can be difficult to keep up with intrusion detection in software, and hardware systems or software with hardware assist tends to be required. This is the type of speed targeted by new work in high speed intrusion detection, with network speeds of 1 Gbps (Gigabit Ethernet) and 2.4 Gbps (STM-16) being of particular interest. With these types of speeds, we tend to need customised hardware or custom designs for Field Programmable Gate Arrays (FPGAs). The move to hardware based systems allows the introduction of more parallelism than might be possible in software based systems and hence alternative algorithms. Hardware based solutions are probably the only approach currently practical for intrusion detection on high-speed backbone networks running at speeds of around 10 Gbps.

2.1.1 Existing work in this area

An approach taken by Franklin et.al. [3] has been to build an intrusion detection system using Non-Deterministic Finite Automata (NFA) – this approach to building FPGA based automata first being suggested by Sidhu and Prasanna [4]. The NFA approach has the advantage of being able to match quite complex regular expressions, without the usual stage of needing to convert the NFA into a Deterministic Finite Automata (DFA). Although the work by Sidhu and Prasanna supports the idea of dynamic reconfiguration, this technique was not used by Franklin et.al, so hence the FPGA designs need to be re-compiled following rule changes. The work described by Franklin et.al. operates on 8-bit data items and the authors report FPGA clock rates of 30-120 MHz, dependent on the regular expression complexity.

An approach taken by Gokhale et.al. [5] compares packet content from a data pipeline against entries in a content addressable memory (CAM) – a "match vector" is then produced which gives details of the entries in the CAM that caused a match. The match vector is appended to the data packet and this is forwarded to software for further processing. This approach has the advantage of being dynamically re-configurable.

A commercial product by PMC-Sierra is described by Iyer et.al in [6]. This paper describes the system called ClassiPi which is a classification engine and is implemented as a custom ASIC (Application Specific Integrated Circuit). This classification engine is a flexible programmable device on which more traditional software-type algorithms can be implemented. Performance depends on the algorithm being implemented and the number of rules – with a lower performance being achieved for more complex operations such as matching regular expressions.

Cho et.al. [7] use a comparator based system that operates on 32-bits at a time – this uses four sets of comparison logic to search for a given string at each of the four possible byte offsets within a word. A match is successful if for a particular byte offset, all consecutive parts of a string are found. Separate logic is provided for each string being searched for, such that these can all operate in parallel. Rule sets are processed to generate VHDL code for each rule – a problem with this is that the FPGA needs to be recompiled after a change in the rule set.

2.2 Using finite state machines

One method of building string or pattern matching systems is by the use of finite state machines (FSM) (normally DFA). This technique was explained by Hershey [8] in his PhD thesis, which describes a network monitoring system using FSM based matching, with the current state being held in a register and a separate RAM component being used to generate the output and next state. This approach is useful as it can be implemented in hardware and being RAM based means that the FSM can be changed without modification to the hardware design. A disadvantage is that the memory requirements M (in bits) increases exponentially with the input data word size i.

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

Where: s = number of states
i = number of input bits
o = number of output bits

Unfortunately we need to use a large input word size in order to be able to deal with a high network data rate – this is unfortunate as it is relatively straightforward to modify FSM designs that operate on one character at a time (such as those generated using the Knuth-Morris-Pratt [9] string matching algorithm) to work with a multi-character word (see section 4). In any hardware based design we have rather different criteria for assessing an algorithm's performance – for example we can use multiple FSMs to perform pattern matching for a number of patterns in parallel without the increased FSM complexity that we

might get from algorithms that perform pattern matching on multiple strings, such as Aho-Corasick [10].

2.2.1 Recursive flow classification

Gupta and McKeown proposed a mechanism for packet classification [11] called recursive flow classification (RFC). This system uses a series of lookup tables to classify network traffic on the basis of multiple fields. The mechanism used is that the data from various header fields is broken up into groups of bits, known as 'chunks'. Each chunk is then used as an index into a lookup table that gives a index value for that group of bits.

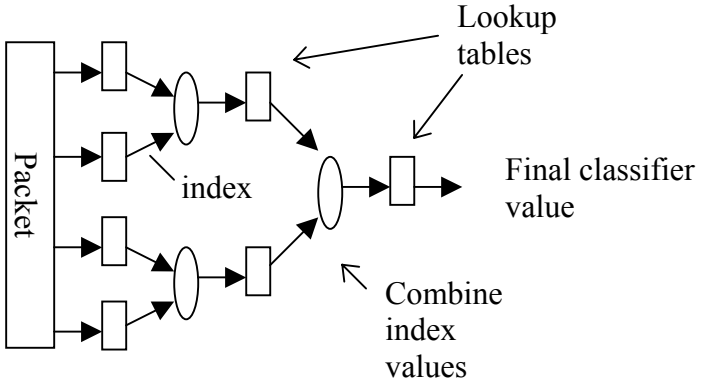

Figure 1 - Diagram of RFC, based on [11]

The general idea is that the size of the index (in bits) should be smaller than the size in bits of the input chunk. Hence an address field can be classified as one of a number of addresses or groups of addresses that are of interest – or it may be shown to be no match. If we take these index values from various 'chunks', then we can combine these together to form a new value for lookup in the second level of the hierarchy. This can be repeated until we have a final classifier value for a whole group of header fields.

3 Proposed technique

Any finite state machine used for pattern matching will typically only be interested in a subset of the possible input values. This effect becomes more pronounced as we increase the word size and try to match several characters at a time. Because of this it is possible to use a form of classification system similar to that developed by Gupta and McKeown and then to generate a limited width data stream for input into a FSM that represents just the input symbol set for that FSM. A benefit of this is that the overall memory requirements for a single FSM that looks at many bytes at time is significantly reduced. Using a state of the art Field Programmable Gate Array, such as the Xilinx Virtex-II series [12], we can implement some of the classification stages and the FSMs themselves using the 18Kbit select Block RAM components – other larger memory requirements, such as for the first classification stage(s) can be satisfied using external RAM chips. The classification system used here is however more complex than that used by Gupta and McKeown in that it does not produce a single classifier value, but one for each FSM. In practice for large numbers of input bits, we may not have any single internal data path that represents all possible input symbols – instead we may start reducing the size of the symbol set as part of an earlier classifier stage and then generate a set of data paths that each carry all possible symbols for a *group* of finite state machines. A simple example of this type of structure is shown in figure 2.

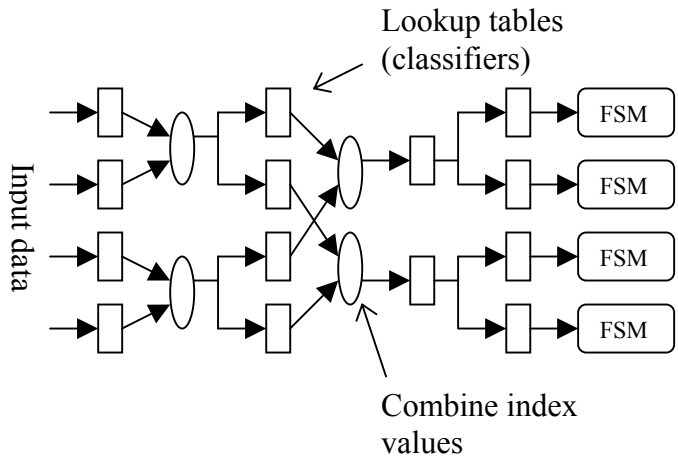

Figure 2 - Simple example of the proposed solution

The input symbol set for a FSM will consist of portions of the string being matched at various byte offsets within the word. At the start and end of the string we will need to perform 'wild-card' matching. Because of the 'wild-card' matching, we will need to prioritise the way the input is classified so as to give the longest possible match. We can however have a conflict in priority between matching the start or the end of strings, particularly between strings for separate FSMs. This problem is resolved by using two parallel classification systems and performing the matching of (say) starts of string patterns separately from all the rest. The two 'classID' values so generated are combined at a final 'merge' classifier stage, prior to the input into a FSM – this technique is explained in more detail in section 4.4.

Benefits

There are several benefits of this approach, which are detailed below:

- As the system is table driven, it should be possible to reconfigure the system for updated rule sets without rebuilding the FPGA design.
- The separation of the classification stages from the FSM itself means that we are able to perform a series of pipelined operations on the data without any impact on the timing of the FSM cycle. We can therefore use a variety of implementation methods for the classification stages including external memory, block memory within the FPGA and even CAM (Content Addressable Memory).
- These techniques expand well with the input word size – and hence data rate. The size of a FSM memory is *mainly* dependent on the string length being searched for – so for most real life intrusion detection rules this remains small. The main memory usage is in the classification, however this is still vastly smaller than that required for a FSM-only solution with a large input word size and is further reduced by the hierarchical division of the symbol set as index values are combined together.
- As each individual FSM will typically have a small sized input word, we can normally implement each FSM almost entirely within one, (or in the case of complex rules, a small number of) Xilinx FPGA (synchronous) block RAM component(s). Initial tests have shown that FSMs based on a single block RAM component are capable of operating a clock periods down to as low as 4.3ns.

Initial work suggests that using state of the art FPGA components and external memory, it should be possible to build a sizeable network intrusion detection system (NIDS) that operates with an input word size of 32-bits and quite reasonable designs with a 64-bit word size. The latter should enable us to build NIDS that operate at data rates of the order of 6.4 Gbps. Further work, as discussed in this paper, is required to investigate the best ways in which to configure such a system and to determine how well this might expand to larger word sizes and hence higher data rates.

Scope and limitations

The parallelism provided by the hardware is at a cost of needing to provide large numbers of relatively small blocks of memory that can be accessed independently. The amount of memory will generally grow with the number and complexity of the rules. FPGAs provide these types of resources, and given trends over the last few years we are probably safe to predict that, over the next few years, new FPGAs are likely to be released with increased amounts of these resources – not least because of the competition between FPGA manufacturers. As the number and size of memory blocks on an FPGA increase so will the potential number of rules that a single FPGA can support.

At present, the plan is to only look for rule matches within individual data packets – this is the approach taken by much of the current work on high-speed intrusion detection systems. An additional feature provided by many of the lower speed NIDS is to track individual connections and to look for pattern matches that may span separate data packets. This would actually be quite straight forward in the type of system proposed here due to the matching being performed as a series of FSMs – the final state (in each FSM) for a particular stream could be saved at the end of a data packet and then restored the next time an in-sequence data packet is seen for that connection. The downside is however the additional work required to implement this functionality and the additional use of FPGA resources. There is no plan to include this type of facility at present – although this could be looked at later as an area for further work.

4 String Matching Techniques

To be able to implement a pattern matching system we need an algorithm that can be easily implemented in hardware. Many of the existing string matching algorithms are designed for software-based implementations, where the algorithm has unrestricted access to both patterns and data stored in random access memory. Although this can also be done in hardware it is more practical if the access to data can be sequential rather than random access. Further more, for a system containing parallel pattern matching it is more effective if all instances of the algorithm have access the same data stream - or at least one of a small number.

Many of the existing algorithms decide their own rate of access to the data being searched - with various advantages and disadvantages of how many comparisons they need to make and how fast they work in various situations. In our case *we* want to decide the rate that algorithm proceeds - say one character (or word) per clock cycle - and hence need to modify the algorithm to work in this fashion.

4.1 Matching a single string

The fastest method of matching a single string is considered to be the Boyer-Moore [13] algorithm (with variations). This operates by scanning a pattern from right to left and skipping forward as many characters as possible on a mismatch. Although this can be implemented in hardware, it is more complex because of the order in which the data is accessed.

A simpler algorithm is the Knuth-Morris-Pratt (KMP) algorithm. This scans the text from left to right and on a mismatch will determine the longest overlap - i.e. the longest partial match. A standard implementation is to use a finite state machine and in each state to make a comparison against a single character from the pattern. If the match succeeds we progress to the next character in the input data and change the state testing for the next character in the pattern. If the test fails we move to the state representing the longest amount of pattern matched and then test the same input character again. This may well happen more than once.

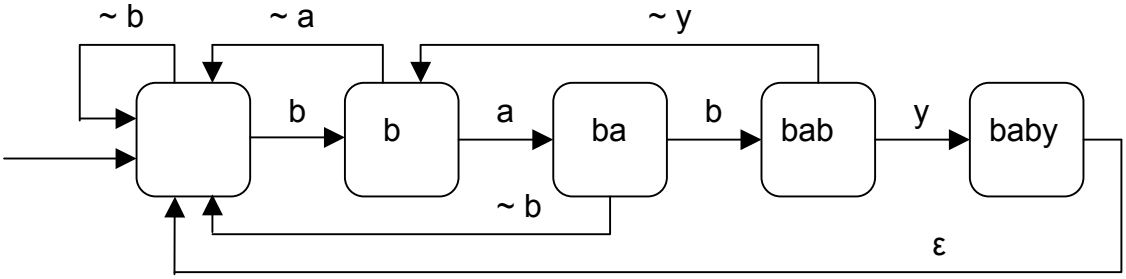

Figure 3 - KMP string matching

Unfortunately, we cannot use this simple implementation because in any mismatch we will have more than one comparison of each input character - thus making the speed of execution dependent on the pattern and the input data. However, given that we have already examined the character that caused a mismatch and that we can determine which state we will go to next (eventually) given that character we can avoid the re-comparison and go directly to the appropriate state and then examine the next character. For example:

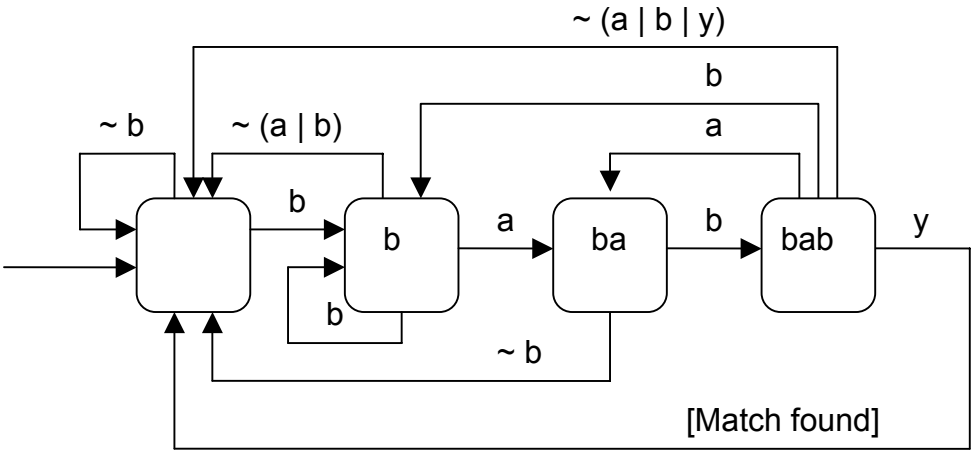

Figure 4 - One character per cycle matching.

In the previous example the mismatch in state "bab" would take us to state "b" to check if the current character is an "a". We can however determine this during state "bab" and go directly to states "", "b" or "ba". (Note that we can also eliminate the final state representing the whole string being matched, thus removing the following ε transition.) Assuming we can handle the multiple transitions from a state, we can now run the above finite state machine at the speed of the incoming data – i.e. one transition per input character. The above example is relatively trivial, however the same mechanism works irrespective of the machine complexity.

4.2 Matching multiple strings

In some cases it may be appropriate to use a single FSM to match multiple string patterns. The classical algorithm for this is Aho-Corasick [10]. This operates by creating a tree-based FSM relating to all of the possible string patterns. This can operate in a similar fashion to KMP, by having a failure condition whereby on a pattern mismatch we move to the state relating to the longest partial match. With Aho-Corasick however we are looking at matching multiple patterns, so the longest partial match may be part of a different string to the one that we are currently trying to match.

For example, taking a simple FSM and the strings: banana, nancy, anna - we get the following:

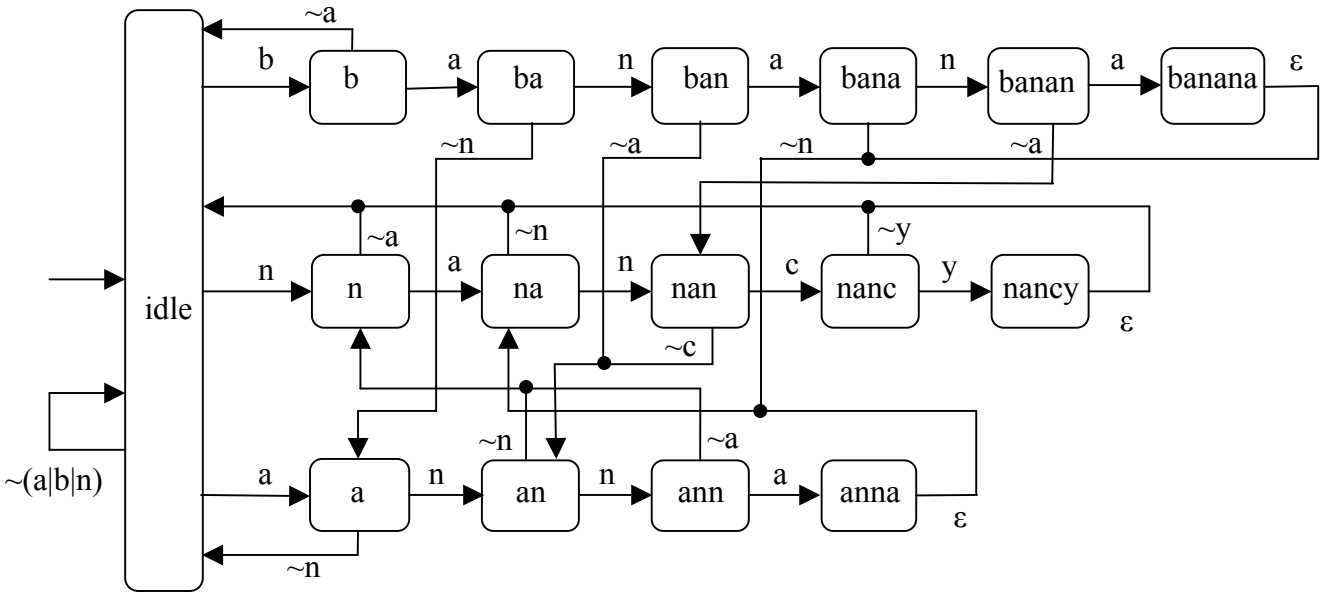

Figure 5 - Multiple string matching using Aho-Corasick

Again this can be redesigned as a FSM that handles one character per clock cycle. This gives a FSM table as follows (* being a wildcard with a lower priority than the other characters):

STATE \ INPUT	A	B	C	N	Y	*
IDLE	A, -	B, -	IDLE, -	N, -	IDLE, -	IDLE, -
B	BA, -	B, -	IDLE, -	N, -	IDLE, -	IDLE, -
BA	A, -	B, -	IDLE, -	BAN, -	IDLE, -	IDLE, -
BAN	BANA, -	B, -	IDLE, -	ANN, -	IDLE, -	IDLE, -
BANA	A, -	B, -	IDLE, -	BANAN, -	IDLE, -	IDLE, -
BANAN	NA, ✓	B, -	NANC, -	ANN, -	IDLE, -	IDLE, -
N	NA, -	B, -	IDLE, -	N, -	IDLE, -	IDLE, -
NA	A, -	B, -	IDLE, -	NAN, -	IDLE, -	IDLE, -
NAN	NA, -	B, -	NANC, -	ANN, -	IDLE, -	IDLE, -
NANC	A, -	B, -	IDLE, -	N, -	IDLE, ✓	IDLE, -
A	A, -	B, -	IDLE, -	AN, -	IDLE, -	IDLE, -
AN	NA, -	B, -	IDLE, -	ANN, -	IDLE, -	IDLE, -
ANN	NA, ✓	B, -	IDLE, -	N, -	IDLE, -	IDLE, -

✓ = match successful

Table 1 – Aho-Corasick FSM operating at one character per clock cycle

4.3 Multi-character input

Many standard string-matching systems operate on only one character at a time. This is simple to implement, but of course as we move to higher data rates it is useful to be able to operate on larger data words sizes so as to increase the throughput. For string matching, this means that we need to be able to operate on multi-character words if we wish to keep up with the data rate from the network. Fortunately, it is possible to modify the existing string matching systems to work with multi-character patterns as will be shown below.

To match multi character patterns, all we need to do is to determine all of the possible combinations of characters we are interested in for our given word size – this will include word length groups of characters from the string we are trying to match on all possible word boundaries. As an example, say we are looking for the string "ABCDEFGH" in a system with a 4-byte word – this will require us to look for the following patterns:

S_{AH} = (***A, **AB, *ABC, ABCD, BCDE, CDEF, DEFG, EFGH, FGH*, GH**, H***, ****)

Where * is the don't-care meta-character and **** is the default no-match symbol. The * meta-character has lower priority than the other characters.

This is a simple implementation in that it can miss the start of new strings following a successful match (so the output is match/no match). It can also only handle one string match per FSM implementation for similar reasons. This technique therefore works ok for a single FSM matching system, but is a problem if we use data paths to carry symbols for more than one FSM – as we plan to do in this work – because of potential conflicts in priority for wild-card matching.

The FSM table for multi-character input can be built easily from a single character input FSM, by simply applying all possible multi-character strings to the FSM in each state and then observing the final state and any output that was generated. This technique will work both for FSM's created using KMP and Aho-Corasick techniques. We need to be careful with the choice of strings used for a Aho-Corasick implementation however as will become evident in the next subsection.

STATE \ INPUT	***A	**AB	*ABC	ABCD	BCDE	CDEF	DEFG	EFGH	FGH*	GH**	H***	****
IDLE	A	AB	A..C	A..D	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
A	A	AB	A..C	A..D	A..E	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
AB	A	AB	A..C	A..D	IDLE	A..F	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
A..C	A	AB	A..C	A..D	IDLE	IDLE	A..G	IDLE	IDLE	IDLE	IDLE	IDLE
A..D	A	AB	A..C	A..D	IDLE	IDLE	IDLE	IDLE, ✓	IDLE	IDLE	IDLE	IDLE
A..E	A	AB	A..C	A..D	IDLE	IDLE	IDLE	IDLE	IDLE, ✓	IDLE	IDLE	IDLE
A..F	A	AB	A..C	A..D	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE, ✓	IDLE	IDLE
A..G	A	AB	A..C	A..D	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE, ✓	IDLE

✓ = match successful

Table 2 - Multi-character input FSM

Some data will match more than one input pattern, so the input data needs to be classified against the possible input patterns in decreasing priority order.

4.4 Avoiding "Dead-periods"

There could be a problem in a real system whereby network intruders may try to use artificially generated string patterns to cause the matching system to develop dead periods during which a real threat begins. To see evidence of this problem, we need to look at how the classifiers may work. Take for an example, the string match that we looked at above. In this case we also have a second string match for STUVWXYZ, giving us the following sets of string patterns:

S_{AH} = (***A, **AB, *ABC, ABCD, BCDE, CDEF, DEFG, EFGH, FGH*, GH**, H***, ****)

S_{SZ} = (***S, **ST, *STU, STUV, TUVW, UVWX, VWXY, WXYZ, XYZ*, YZ**, Z***, ****)

These combine together to give the following:

$S_{AH} \cup S_{SZ}$ = (***A, **AB, *ABC, ABCD, BCDE, CDEF, DEFG, EFGH, FGH*, GH**, H***, ***S, **ST, *STU, STUV, TUVW, UVWX, VWXY, WXYZ, XYZ*, YZ**, Z***, ****)

If we are coming to the end of matching string ABCDEFGH, we may be looking for the string: H***. However, if the input string is HSTU, H*ST or H**S then we could either match the end of the first string, or the start of the second string – but not both. The string matched will depend on how the various strings are prioritised - we would assume that strings with wild-card characters would have a lower priority that those with exact matches, however there is a conflict between the wild-cards at the start or the end of the string. In practice, we can resolve this by using multiple classifiers. These would identify the following strings within the input stream:

Primary set = (ABCD, BCDE, CDEF, DEFG, EFGH, FGH*, GH**, H***, STUV, TUVW, UVWX, VWXY, WXYZ, XYZ*, YZ**, Z***, ****)

Secondary set = (*ABC, **AB, ***A, *STU, **ST, ***S, ****)

For each FSM, we can then generate a classified set of symbols that is significant to that machine. For strings with wild cards we will need to identify strings of a higher priority in the input set that are a subset of that string but which are not used in that FSM. These will need to be folded back into the correct output string. We will then have a pair of input strings for each FSM which can be fed through a final "merge" component, in order to generate the required input. The normal case will be that FSM only needs to be able to identify a single instance of a string in the input packet - this will generate a smaller output set than if the FSM needs to be able to detect multiple instances of the same string.

e.g for the A-H string above, the first classifier will give:

Primary set = (ABCD, BCDE, CDEF, DEFG, EFGH, FGH*, GH**, H***, ****)

Secondary set = (*ABC, **AB, ***A, ****)

Secondary Input \ Primary Input	ABCD	BCDE	CDEF	DEFG	EFGH	FGH*	GH**	H***	****
*ABC	–	–	–	–	–	–	–	H***	*ABC
**AB	–	–	–	–	–	–	GH**	H***	**AB
***A	–	–	–	–	–	FGH*	GH**	H***	***A
****	ABCD	BCDE	CDEF	DEFG	EFGH	FGH*	GH**	H***	****

Table 3 - Merge, possibly ignoring subsequent matches of the same string.

Secondary Input \ Primary Input	ABCD	BCDE	CDEF	DEFG	EFGH	FGH*	GH**	H***	****
*ABC	–	–	–	–	–	–	–	*HABC*	*ABC
**AB	–	–	–	–	–	–	*GHAB*	*H*AB*	**AB
***A	–	–	–	–	–	*FGHA*	*GH*A*	*H**A*	***A
****	ABCD	BCDE	CDEF	DEFG	EFGH	FGH*	GH**	H***	****

Table 4 - Merge, identifying multiple matches of the same string.

Note: strings with internal wild-card characters or strings that are less than the word size will need further study.

In either case, we will have the primary and secondary input symbol sets, S_P and S_S respectively. Given a word size w and a match string of length s, we will have input symbol set sizes as follows:

$$|S_P| \le s+1 \qquad \text{- as there may be repeated patterns}$$
$$|S_S| = w$$

If we take the route of potentially ignoring near subsequent matches of the same string, then we will have an output symbol set S_O as follows:

$$S_O = S_P \bigcup S_S$$
$$|S_O| = |S_P| + |S_S| - |S_P \bigcap S_S| = |S_P| + |S_S| - 1 \qquad \text{- as } S_P \bigcap S_S = (****)$$
$$\le \mathrm{s} + \mathrm{w}$$

For the merge preserving the ability to match close instances of the same string, we need to add the back-to-back strings separated by 0 to w-2 '*' meta-characters. This adds $\sum_{i=1}^{w-1} i$ additional strings, which gives the following size of the output set:

$$|S'_O| \le s + w + \sum_{i=1}^{w-1} i$$
$$|S'_O| \le s + \frac{w^2}{2} + \frac{w}{2}$$

4.5 Summary

We have seen in this section how it is possible to build FSMs that will operate at one symbol match per clock tick and also how this can be extended to work with multi-character words. The use of joint data paths that carry data for multiple FSMs can lead to the problem of dead

periods, whilst matching the end of a string, during which a further match may be missed. Section 4.4 explained how it is possible to avoid this problem by the use of two separate data paths, one of which carries all of the patterns relating to starts of strings and then to form a merge (or cross-product) of these two data paths prior to input into the each FSM.

5 Architecture and High-Level Components

Before going into any specifics of implementation, we shall first describe the basic high-level components that we are trying to realise. When used in any real design, these will of course need to be chosen carefully to match the abilities of the technology used for their implementation – the solution to this problem is addressed during compilation and will be discussed later.

The values of the data paths are an enumerated type that gives a numeric value for groups of bytes that are of interest (length depends on word size used, and position within the architecture) and a final value indicating a no-match or 'don't care' pattern. This is referred to as the "symbol set" and is specific to each data path.

The high level components described here are constructed out of a number of lower level *primitives* that will be described in the next section.

5.1 The "Finite State Machine" component

This is implemented as a number of parts, to address the problems related with matching beginning and end of strings using wildcards. A general block diagram is as follows:

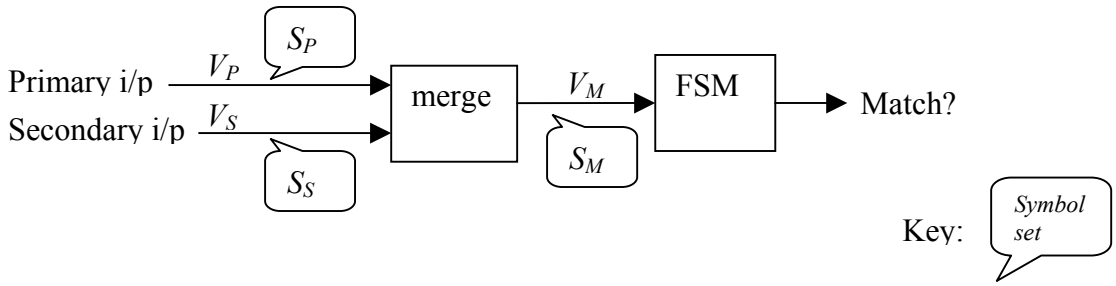

Figure 6 - Schematic of Finite State Machine component.

The two inputs relate to the way in which strings are matched, with the string "starts" with wildcards being matched separately to the rest of the string. Each of the data paths here carries an index value, which relates to the symbol set represented on that particular data path. So we have input values V_P and V_S each referring to a member of the ordered symbols sets (S_P and S_S) for the primary and secondary inputs respectively. These two inputs are merged to give new set S_M, and value V_M, where:

$S_M =$ order($S_P \cup S_S$)

if $S_P[V_P]$ = = don't care, then:

$V_M = x$ such that $S_M[x] = S_S[V_S]$

else

$V_M = x$ such that $S_M[x] = S_P[V_P]$

The merge stage shown above, is the version that may cause subsequent matches of the same string to be missed, but will give priority to completing a string match over starting a new one. The 'order' function sorts the strings into alphabetical order, treating any * meta-

character as having a lower priority than a real byte or character value. In practice, the merge stage for either merge methods can be implemented using the same hardware as used for the "combine" component described in section 5.3 below.

At present the FSM is created from a modified KMP string-matching algorithm that will operate on a representation of multi-byte input and will also match at a rate of 1 word per clock cycle independent of whether any pattern miss-matches occur. Depending on the amount of memory available, the FSM may also be created using a modified Aho-Corasick algorithm to allow each FSM to match multiple patterns.

5.2 The "Group" component

The group is the mechanism that we use to sub-divide a symbol set into multiple smaller symbol sets. This can be generated in reverse by taking the output symbol sets and creating an input symbol set that contains all output symbols.

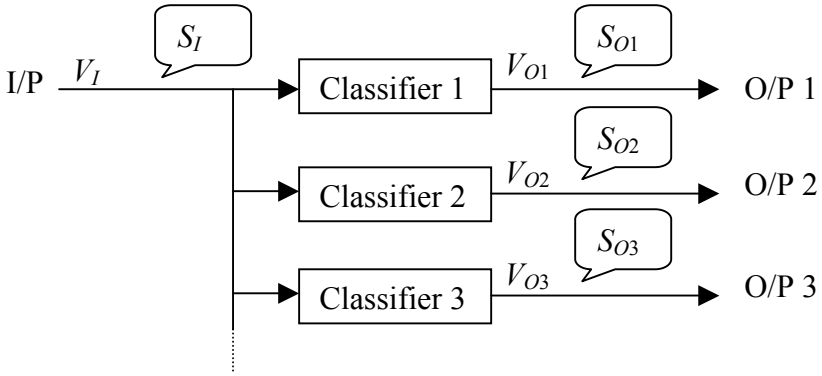

Figure 7 - Group Schematic

So we have S_I = order ($S_{O1} \cup S_{O2} \cup S_{O3}$ ..). For each output, we determine which input symbols cover each output symbol and build a classifier as a transformation table for each output. In most cases, the output data paths from the Group will be represented by less bits than the input, although this may not always be the case. The actual size of the symbol set is also important here as reducing this can also help to reduce the amount of logic used further on.

5.3 The "Combine" component

This provides a reverse operation to the group in that it takes two (or more) input paths and generates a single output path. Assuming the two inputs represent the same "length" patterns, then the output will therefore represent patterns of twice this length. This operation is performed by generating a cross-product of the two input symbol sets, so initially the symbol set size will be the product of the symbol set sizes of the two inputs.

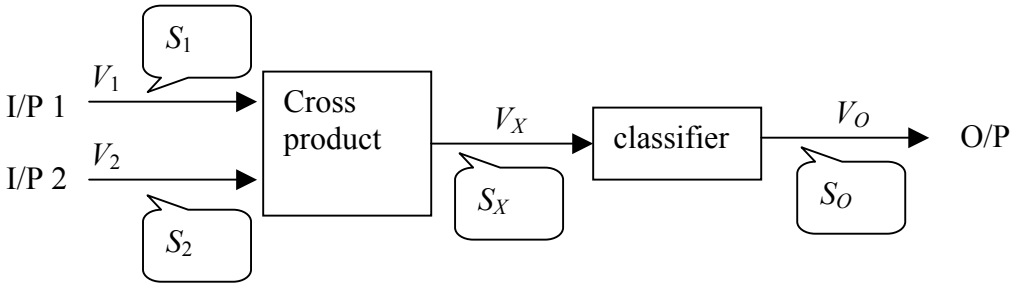

Figure 8 - Schematic of combine component

This cross-product symbol set will however have a very high redundancy, as many of the combinations will not be of interest. To map the initial cross product into the required output symbol set, we again look to see how the output symbol set is covered by the symbols in the cross product and perform a mapping. As with the group, particular note needs to be made here of the priority of the output symbol set when performing this mapping. Again following the example of the group, this is generated in reverse by taking the output symbol set, and using that to determine the symbol sets required for the two input symbol sets.

Various mechanisms can be used for implementation, however an efficient method is to perform the cross-product using a "multiply and add" operation and then to follow this with a RAM based table for removing the redundancy. Particular care is needed with choice of symbol sets here as the table used for removing redundancy can potentially use large amounts of memory.

5.4 The "Input classifier" component

These are the first stage of the classifier network, in that they take raw network data and classify this to generate an output symbol set. A RAM based input classifier might take up to 16-bits of data – and use 64K entries to generate a classified output.

Larger RAM classifiers could be used, however they are not likely to map to a useful word size and would also require increasingly unmanageable amounts of memory to be provided and initialised.

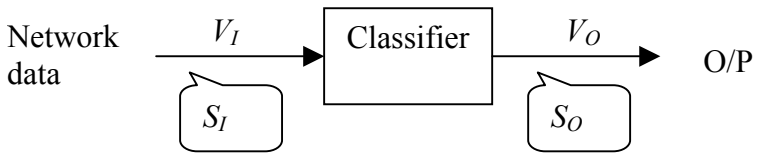

Figure 9 - Schematic of Input Classifier Component

Where: $S_I = (0 .. 2^W-1)$, for an input word size w, and $| S_I | = 2^W$

As an alternative, a classifier might be built from Content Addressable Memory (CAM). A ternary-CAM based classifier could, for example, take 128-bits of input data; here we need roughly the same order of magnitude of CAM words as output symbols, although we may sometimes need more than one word of ternary CAM per output symbol, as a word of ternary CAM can only match groups of inputs values that can be specified using a word containing 0, 1 or don't care bits.

It is therefore possible to use a single CAM component to perform classification of the entire input data word, whereas we are likely to require at least 2 RAM based classifiers and probably more.

6 Modelling and Implementation

From the previous section, we can see that there is quite a bit of complexity in dealing with the data in this way. Fortunately, much of this work can be done in a pre-processing stage, so will only need to be performed each time a design is changed. A complete system would therefore have the following structure.

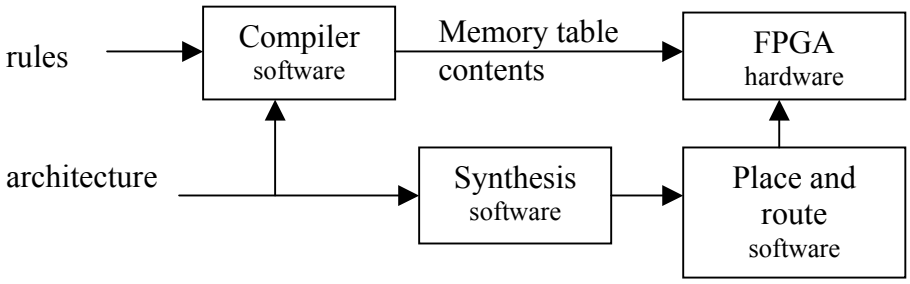

Figure 10 - System structure

Changes to the system architecture require the FPGA design to be rebuilt: this requires logic to be synthesised from the high level design and then an FPGA design created from this logic using place and route tools.

Subsequent changes to the rules used for the intrusion detection system would only require the new set of rules to be compiled. The output of the compilation stage is a set of tables that will need to be loaded into memory within the FPGA or associated external components. This would be done dynamically whilst the system was running, or by use of the Xilinx JBits [14] software to modify the contents of the FPGA load file.

The compiler ends up being quite complex, as it needs to deal with the symbol sets used in different parts of the system – these are implemented using a symbol table to represent each of the different symbol sets. The compiler also needs to create the contents for each of the classifiers. Hence much of the work performed by the compiler consists of manipulating sets of strings and combining and dividing these sets. This information does not need to be carried over explicitly into the hardware as it will be captured by the data loaded into the pieces of memory for the various classifier tables. The hardware within the FPGA is constructed primarily of a set of memory elements, linked in a predefined way as defined by the architecture that is specified.

6.1 Implementation

Out of the various components used in this architecture, there are only a few different primitives used, as shown next.

6.1.1 Classifier

This takes a bit-vector representation of a member of a symbol set and removes the redundancy to give only the values that are of interest. This could be implemented in various ways, but will typically be a piece of RAM.

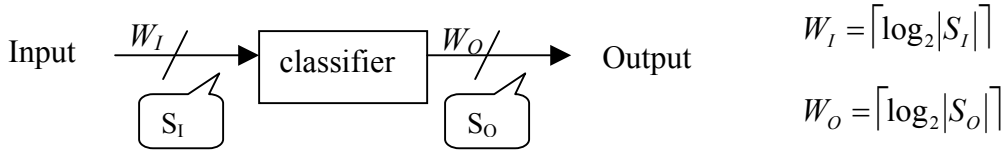

Figure 11 - Classifier

In a RAM based implementation, this corresponds to a piece of memory with W_I address inputs and W_O data outputs. The memory size therefore being $W_O.2^{W_I}$ bits. This corresponds directly to the 'memory lookup' stage described by Gupta and McKeown. An alternative implementation could be to use ternary content addressable memory TCAM – in

this case the memory would require a data search key width of W_I bits and an address output (or equivalent) of W_O bits wide – the memory requirements here being $W_I.2^{W_O}$ bits. The TCAM implementation is particularly useful when the input is heavily redundant – such as within an input classifier stage – as we would expect W_I to be significantly bigger than W_O. The use of TCAM also makes it possible to have a single input classifier that uses a large input word size – whereas the amount of memory required for a RAM version may make it impossible to build.

6.1.2 Cross-product

Here we wish to take two symbol sets and combine these together to give all combinations of the values from the two sets. This corresponds directly to the 'combine stage' described by Gupta and McKeown [11] – in **both** versions shown below. A simple implementation of this can be just to concatenate the two input bit vectors.

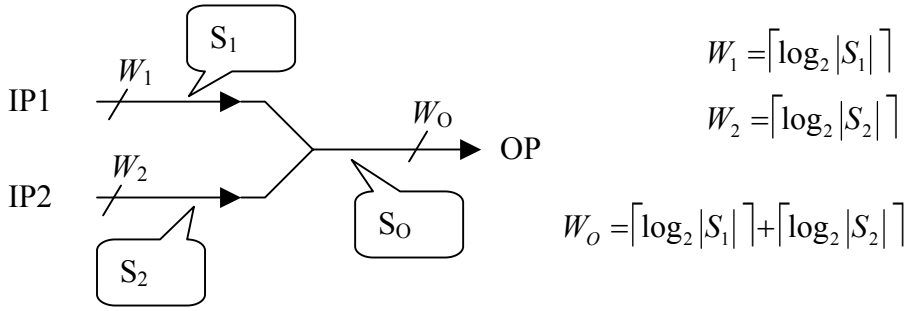

Figure 12 - Simple cross product

However, it may be that the number of elements in a symbol does not completely use all combinations of the input bit vector. For example the number of members of the set being represented on input 1 is $|S_1|$, where:

$$2^{W_1-1} < |S_1| \leq 2^{W_1}$$

An optimal cross product could sometimes be represented using $W_1 + W_2 - 1$ bits. This may not appear at first to be very significant, but could mean that a subsequent classifier stage might require half the amount of memory.

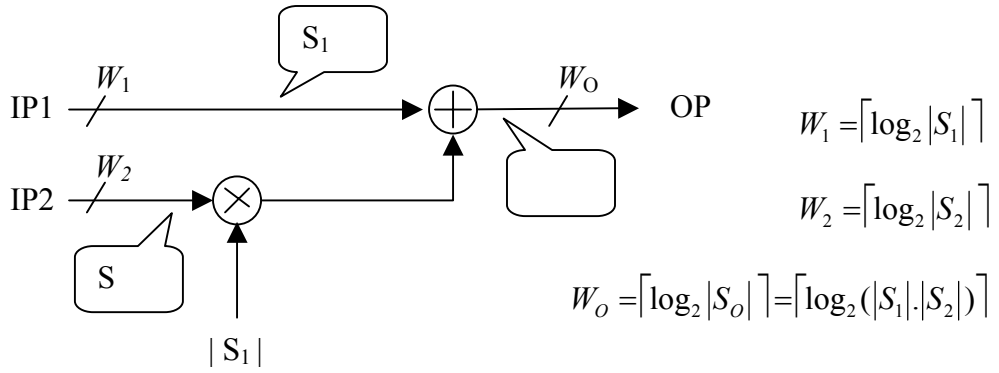

Figure 13 - More efficient cross product

We can generate a smaller cross product as follows: $IP1 + |S_1| * IP2$. If required, this could be implemented using standard multiplier components and adder chains within an FPGA.

6.2 Compilation

A major part of the processing is performed at compile time. This pre-processing will only need to be performed if we change the rules used in the intrusion detection system, or if we change the architecture. The speed of this stage has not yet been measured, but it is unlikely for the whole compilation and loading stages to take more than a few seconds. This compares very favourably with the time taken to rebuild FPGA designs, which can easily be measured in tens of minutes or possibly much more. The rebuild of an FPGA design may also not work first time and may need to be iterated, or will not complete at all if timing constraints cannot be met or if not enough resources are available. Fortunately, the FPGA will only need to be rebuilt if we change the architecture.

6.2.1 Defining the architecture

One particular issue that we need to consider is that we are likely to require one of a number of fixed hardware architectures for the FPGA and then to map the various IDS rules onto this fixed structure. This will mean that we cannot have a free hand in how we compile the rules into a hardware implementation. The advantage of this is that we avoid the need to rebuild the FPGA design for each new rule set, but a disadvantage is that we will have a less than optimal implementation – as we may leave areas of logic unused and may allocate some blocks of memory as being larger than they need to be.

Given this approach, we need to be able to define the overall architecture of the system we are building and also minor details such as sizes of data paths and how components link together – for example defining the maximum size allowed for a port group, or the maximum size for a finite state machine. There are various ways in which we could define the architecture, from building a compiler specifically for a fixed design – through to a compiler that can be dynamically configured at run time from setup files or user input.

6.2.2 Building a model

One way of writing the compiler is to implement the various architectural components as objects within the programme. In this way we can develop a complete model of the hardware architecture within the programme. Given the rule set being compiled, the compiler can generate the symbol sets for each data path and create the classifier tables with which to load the components. The data for the classifier tables can of course be output from the programme for loading into the hardware.

A benefit of this approach is that, given a suitable GUI, the user can see how each stage has been implemented for a particular set of rules. Also as the compiler will contain a complete model of the system being designed, it is very simple to extend this to provide a high level simulation of the IDS using the same data tables that would be used within the hardware.

7 An example architecture and its compiler

Initial work has looked at the generation of software to perform compilation and simulation of a system with a fixed overall structure. This uses CAM input classifiers, two levels of groups and final FSM stages. Primary and secondary paths are used to deal with the main matching and string start wild card matches.

The overall size of the FSMs, the size of the data paths and the maximum size of each of the four types of groups can be specified within a GUI and can also be loaded-from/saved-to a

configuration file. The number of groups and FSMs is not fixed by the software, but determined dynamically.

The software takes as its input the set of required IDS rules and compiles these to fit into the given architecture. If particular rules cannot be implemented in the specified architecture then this is indicated. A simple example of the overall hardware structure is shown figure 14.

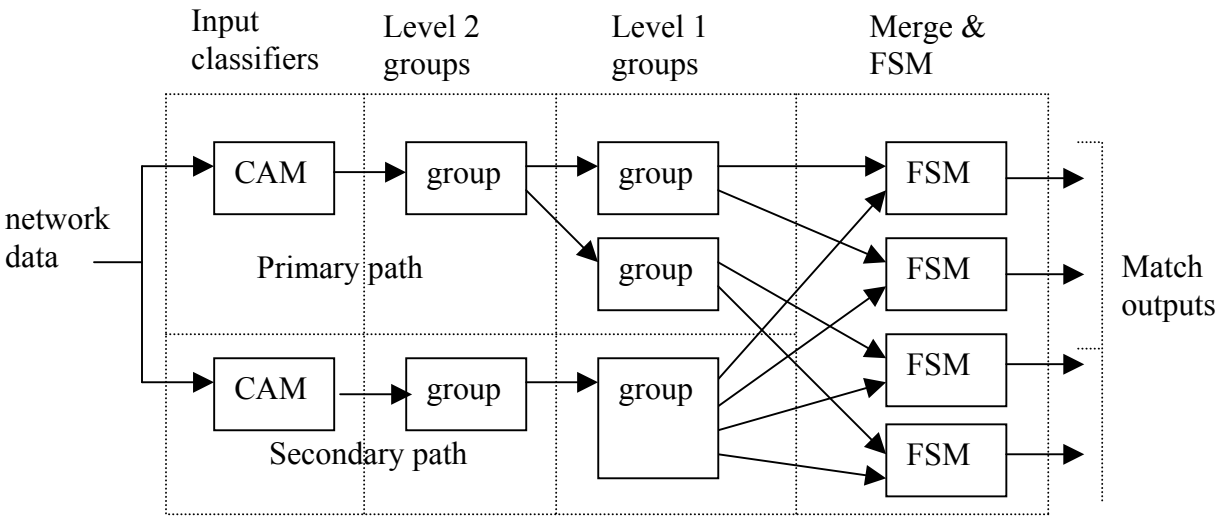

Figure 14 – Simple CAM based architecture

The symbol sets in this example are assumed to be smaller for the secondary path than for the primary path. The first group in the secondary path is therefore not required and hence simply maps input to output. Real rule sets will typically provide far larger structures. The input word size is specified, as are the bus sizes, and maximum numbers of output ports for each of the blocks of groups. The search strings are read in from an input file, these are compiled with the given parameters to create the various groups, FSMs and CAMs (content) that are used. The design can be tested by performing a high-level simulation of the design against network data from a file. The simulation allows the design to be run to completion or single stepped. For debugging and testing ideas, any of the FSMs or groups can be examined to see the symbol sets being used and the contents of the various lookup tables.

Figure 15 shows a screen shot from this compiler. Each stage allows the various word and group sizes to be specified. From the various parameters set, each of the FSM and port group stages will indicate the amount of memory required for their implementation. The level-one port groups also have the option of allowing the user to "fold" the port group into the previous FSM stage. This causes the lookup tables for that stage to be incorporated into the lookup tables for the FSM merge stage. Whether this is a useful thing to do will depend on the various word sizes being used. Any folded port group would be replaced with registers to provide the equivalent pipeline delay.

The combine component is already implemented as an object within the compiler, but was not used for this particular architecture, as the CAM input classifier was large enough to cope with the complete input word size.

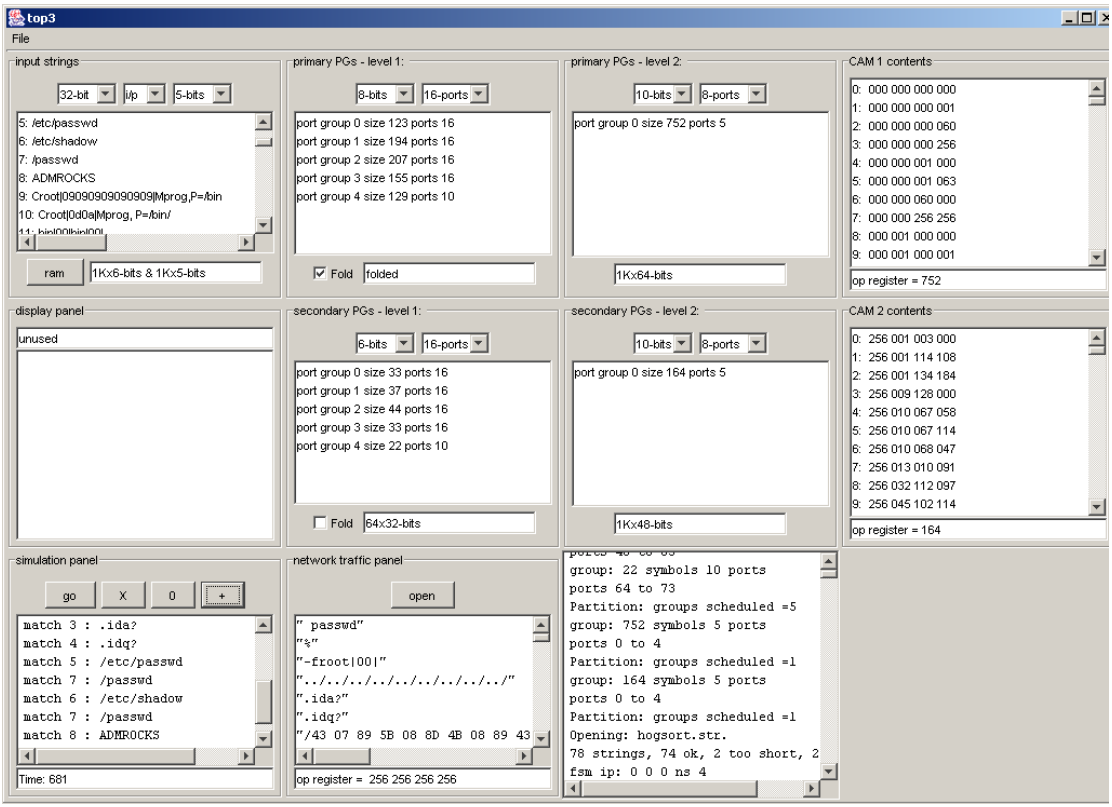

Figure 15 - Screen shot from compiler

8 Results

The architecture described in the previous section and its associated compiler are now tested with some sample rules. The rules used here are taken from a standard hogwash[1] [15] rule set. Of these rules, only the case dependent rules are used and these are sorted to give a set of (78) unique search strings – which are held in an input rule file.

8.1 Detailed results using a 32-bit input word size

The compiler is set to an input word size of 32-bits, with a maximum FSM i/p word size of 5-bits. We now read these 78 search strings into the compiler. Of these strings, two are flagged as being too short for implementation with the given word size and two are flagged as being too long for the amount of FSM memory that we have allowed for (1 Kbyte). In this example, the primary level one port group has been folded. This will therefore increase the size of the memory needed for the FSM merge stage (which otherwise may have been quite small).

The architecture so generated is simulated (within the compiler) using the input search string file as the network data – this generates the expected matches against the rules that consist only of ascii characters and no escaped control codes. To ensure that the technique of using primary and secondary paths will avoid "dead spots", a simple test is also performed using a pair of test strings at various spacings and starting at different byte offsets within a word.

This uses the two strings from the rule file "/passwd" and "ADMROCKS". First we try these at different spacings between the two strings:

[1] Hogwash is an in-line "packet scrubber" based on the snort packet scanning 'engine' that can be used to identify and drop suspicious looking network packets.

Non matching characters between the two strings

Byte offset within word of first string	0 ADMROCKS/passwd	1 ADMROCKS /passwd	2 ADMROCKS /passwd	3 ADMROCKS /passwd
0	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS"	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓
1	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓
2	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓
3	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ✓

✓ = match successful

Table 5 - Test strings at various spacings

Now we repeat the experiment, with the second string "overlapping" the first string – i.e. starting before the first string is finished.

Overlap between strings

Byte offset within word of first string	0 ADMROCKS/passwd	1 ADMROCK/passwd	2 ADMROC/passwd	3 ADMRO/passwd
0	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***
1	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***
2	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***
3	"/passwd" ✓ "ADMROCKS" ✓	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***	"/passwd" ✓ "ADMROCKS" ***X***

✓ = match successful

X = match unsuccessful

Table 6 - Second test string starting before end of first

In both cases, the string matching system performed exactly as was expected.

Next we look at the resources used by any real implementation. The compiler gives details of the number and size of memory components required for the given set of rules – these were converted by hand into counts of the numbers of blocks of RAM that would be required for implementation within a Xilinx Virtex II FPGA and detailed in Table 7.

Stage	Instances per stage	Size of each instance	Xilinx block RAMs per instance	Xilinx block RAMs per stage
FSM	74	1k x 6-bits	0.5	37
FSM merge	74	1k x 5-bits	0.5	37
Primary PG level 1	5	(folded)	0	0
Primary PG level 2	1	1k x 64-bits	4	4
Secondary PG level 1	5	64 x 32-bits	0.5	2.5
Secondary PG level 2	1	1k x 48-bits	3	3
Total number of Xilinx Block RAM components:				83.5
Primary i/p classifier	1	752 x 32-bit	10 bit address o/p	CAM
Secondary i/p classifier	1	164 x 32-bit	10 bit address o/p	CAM

Table 7 - Memory requirements

In practice, we might use slightly more memory than this by generating a more generic structure into which the data could be loaded. Space for additional rules would be created and this space left unused in case it is required later.

8.2 Comparison of results using a variety of input word sizes

The work in section 8.1 was repeated for a variety of input word sizes, and the overall results of the compilation in terms of resource requirements and the number of rules it was able to implement is given below.

Word size (bits)	Number of rules which are …			Xilinx Block RAM's		External CAM requirements
	Too short	Too long	ok	Total	Per rule	
8	0	2	76	49.5	0.65	116x8-bit
16	0	2	76	84.5	1.11	589x16-bit & 39x16-bit
32	2	2	74	83.5	1.13	752x32-bit & 164x32-bit
64	6	4	68	81.5	1.2	785x64-bit & 405x64-bit
128	52	15	11	20	1.82	128x128-bit & 163x128-bit

Table 8 – Comparison of memory requirements vs. input word size

From the results above, we can see that for this particular set of rules, the system works well up to and including an input word size of 64-bits. At an input word size of 128-bits, then we have a problem that most of the strings we are searching for are shorter than the word size, and at present our algorithm does not support this – further work is needed here to see if there are any modifications to algorithm that would enable us to improve this. There is also a marked increase in the memory requirements at a 128-bit input word size – although this would still allow a practical implementation. The CAM utilisation is acceptable in all 5 cases.

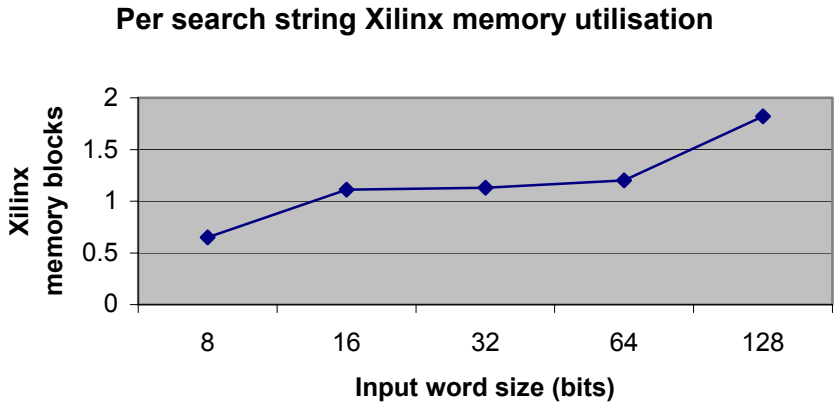

Figure 16 - Memory block use

With the strings that are too long, there are a few cases that are marginal and which are affected by the slight increase in the number of strings in the symbol sets as we increase the word size. Two of the strings however are a lot longer than the others. It probably does not make sense to increase the size of all of the FSM memory to allow these to be supported. Instead it is more practical to allow us to have an architecture that consists of a variety of different FSM memory sizes, with a few large FSMs reserved specifically for long strings.

9 Conclusions and Further work

This paper describes an architecture for building FSM based hardware string matching systems that are able to use large input word sizes. The benefit of this technique for intrusion detection is that we are able to operate directly on the large data word sizes that we will typically receive from the physical layer of the network and hence enable intrusion detection systems to be built that will operate at line speed on high speed networks. The problem of the exponential increase in FSM table size with input word size is addressed by removing the redundancy in the input data. The system is made more efficient by allowing multiple FSMs to share various data paths within the system – so some of the larger resources are amortised across the multiple FSMs.

The problem of conflicts in priority between matching the start and end of a string at arbitrary byte offsets within a word is explained, along with a solution that matches starts and ends of strings separately and only combines the two symbol sets immediately prior to input into the FSM.

A series of high-level components are described from which we can construct our architecture, along with explanations of how these operate.

Finally an example architecture based on the use of CAM input classifiers is presented, along with a compiler than can process the rule sets, create the various symbol sets and the contents for the input classifier tables. The compiler operates by building a model of the architecture being constructed, and this is initialised with the contents of the various classifier tables to allow a high level simulation to be performed against a file of synthetic network data. Results of compiling a standard rule set are given, with resource utilisation being shown for a range of input word sizes.

The results show this technique to be effective in the given example for a word size of up to 64-bits – this being limited by the architecture imposing a minimum length for search strings. The amount of memory required per search string increases slowly with the input word size, and in particular no longer shows the exponential growth that would be seen in a system that did *not* remove input redundancy. The results section performs only a high level simulation, so no details are yet available for absolute performance – this will only be known after a low

level design for the hardware has been built. The FSMs themselves have been shown to operate with a minimum clock period of as low as 4.3ns in some cases. Other components are based almost exclusively on pipelined synchronous memory, so should not create any major problems with the clock period. Our target clock rate for any design would be around 100 MHz – it is hoped that this should not be a problem so long as we do not have any major delays caused by signal fan-out with the FPGA.

9.1 Further work

9.1.1 RAM input classifiers and Combine stages

Although the software is already written to implement the *combine* stages, these have not been tested in a real configuration. The next stage in the design of the compiler should be to allow various possible hardware configurations, including the use of RAM for the initial classification stages and the *combine* stages that will then be required. These are probably best specified in a separate configuration file that specifies not only the data path widths but also the hardware structure. This is likely to make the GUI more complex as the screen layout will need to be chosen dynamically.

9.1.2 Compiler input and output

At present the compiler just accepts a set of quoted text strings as input, using the "Snort" format to permit raw data to also be specified using hex numbers. A parser for the "Snort" rule system has already been written and it is hoped that this can be integrated with the current software, so as to work with the snort style rule files directly and thus obtain other information as well as just the strings being matched.

The compiler currently produces no output apart from displaying data within the GUI. To enable this to operate with any hardware system we will need to output the contents of the various lookup tables to enable this data to be loaded into FPGA memory.

9.1.3 Hardware simulation

Tests have already been carried out on a VHDL simulation of a single input FSM for a Xilinix Virtex FPGA, and this has been successfully loaded and tested [16] with FSM designs created using a modified Aho-Corasick algorithm. Future work here will require the design of sample architectural structures into which designs can be loaded and simulated. As with previous work this would use VHDL and be targeted at state of the art Xilinx FPGAs.

9.1.4 Real hardware implementation

The final stage in any project of this kind is to create a hardware test bench system. This would need to contain a large FPGA, large memory components and a fast path to a high-speed network interface. This would be used on a real network to test out the algorithms against real network traffic, including traffic created to look like possible intrusion attacks and also attempts for attacks to be hidden or disguised.

---

[1] P.Gupta and N.McKeown, "Algorithms for packet classification", IEEE Network March/April 2001.

[2] M.Roesch, "Snort - Lightweight Intrusion Detection for Networks", USENIX Technical Program - 13th Systems Administration Conference - LISA '99, 1999.

[3] R.Franklin, D.Carver and B.L.Hutchings. "Assisting Network Intrusion Detection with Reconfigurable Hardware", Proceedings: IEEE Symposium and Field-Programmable Custom Computing Machines FCCM '02, April 2002.

[4] R.Sidhu and V.K.Prasanna. Fast Regular Expression Matching using FPGAs. Proceedings of IEEE workshop on FPGAs for Custom Computing Machines, April 2001.

[5] M.Gokhale, D.Dubois, A.Dubois, M.Boorman, S.Poole and V.Hogsett. Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. FPL 2002, Lecture Notes in Computer Science 2438, pp. 4004-413, Springer-Verlag 2002.

[6] S.Iyer, R.R.Kompella, A.Shelat, PMC-Sierra Inc. "ClassiPi: An Archictecture for fast and flexible Packet Classification.". IEEE Network March/April 2001.

[7] Y.H.Cho, S.Navab, W.H.Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering", In proceedings of FPL2002: 12th International Conference on Field Programmable Logic and Applications, Montpellier, France September 2-4, 2002.

[8] P.C.Hershey. Information collection architecture for performance measurement of computer networks. Ph.D. Dissertation. University of Maryland College Park, 1994.

[9] D.E. Knuth, J.H. Morris and V.B. Pratt, Fast pattern matching in strings, SIAM J. Computing 6 (2) 1977, pp.323-350.

[10] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18(6) 1975, pp.333-340.

[11] Packet Classification on Multiple Fields. P.Gupta and N.McKeown, SIGCOMM 99, Computer Communications Review, Vol. 29, No. 4, Sept 1999, pp.147-160.

[12] Xilinx Virtex™-II Platform FPGAs: Complete Data Sheet – Product Specification. Xilinx Inc., 14 October 2003.

[13] A Fast String Searching Algorithm, R.S. Boyer and J.S. Moore. *Communications of the Association for Computing Machinery*, **20**(10), 1977, pp. 762-772.

[14] JBits Tutorial. Xilinx Inc., 9 July 2003.

[15] Hogwash – packet scrubber. Principle authors Jason Larsen and Jed Haile. http://hogwash.sourceforge.net

[16] "Real Time Virus Scanning using the Aho-Corasick Matching Algorithm", G.Gao, MSc Dissertation, University of Kent, 2003.