A Finite-State-Machine based string matching system for Intrusion Detection on High-Speed Networks

Gerald Tripp
University of Kent

About Author

Gerald Tripp is a Lecturer in Computer Science at the University of Kent.
Contact Details: The Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK, phone +44 1227 827566, fax +44 1227 762811, e-mail G.E.W.Tripp@kent.ac.uk

Keywords

Security, intrusion detection, finite state machine, string matching, high speed networks.

A Finite-State-Machine based string matching system for Intrusion Detection on High-Speed Networks

Abstract

This paper describes a finite state machine approach for string matching within high-speed network intrusion detection systems. This method uses a standard table based finite state machine implementation, but this is preceded by a preliminary stage that compresses multi-byte network input data into a series of tokens. Each string is matched using a separate finite state machine, each of which has an individually customised token stream. The finite state machines thus created need only a small amount of hardware resources and the overall system is designed to consume network input data at a rate of one word per clock cycle, independent of the search strings and the input data being searched.

This technique has been tested using a high-level simulation in Java, with an architecture consisting of two Ternary Content Addressable Memory (TCAM) components followed by a tree structure of lookup tables that generate a separate token stream for each finite state machine. The results of the simulation show that the technique is effective for an input word size of up to 64-bits. It is possible to build the lookup tables and the finite state machine tables with relatively small blocks of memory, such as those provided within a Field Programmable Gate Array – the TCAM can be implemented as external components. Most of the complexity in this system is within the software that compiles the string matching rules into the contents for the various lookup tables; the hardware resources required are mainly the various interconnected lookup tables that need initialising each time we change the rule set.

Introduction

Intrusion detection consists of monitoring network data arriving at each host computer (host based) or all of the traffic on a section of a network (network based) for various types of security threat. These security threats could for example be attempts to connect to computers within an intranet to try to take advantage of some form of weakness in the computer applications or operating system.

A simpler form of security is provided by firewalls such as screening routers, in this case the firewall looks at just the packet headers and filters packets on the basis of rules specifying permitted or blocked combinations of source and destination IP addresses, TCP port numbers and various other header fields. The identification of packets based on the header fields is referred to as packet classification and a good overview of the various algorithms is given by (Gupta & McKeown, 2001).

Intrusion detection systems (IDS) go a stage further than packet classification, in that they not only look at the packet headers, they also inspect the packet contents. This normally requires us to look for a particular pattern or string of data inside a packet.

A lot of intrusion detection systems are software based, the most well known example being Snort (Roesch, 1999). Software based systems are fine for medium speed networks, or high speed networks that carry a limited amount of traffic – however when we move to heavily loaded high speed networks, then we typically need some form of hardware assistance. It is possible to get around this by partitioning the workload and requiring each computer system to perform its own host-based intrusion detection; however this adds a significant load to each computer system and also requires us to ensure that the appropriate software is installed on each computer system within our intranet. This may be difficult to enforce in some environments where users have control over

their own computer systems – it also may not actually be possible in some cases such as with embedded or turn key systems.

Summary of this paper

The aim of this paper is to explain the operation of a Finite State Machine (FSM) based pattern matching system, intended for implementation in hardware. A lot of existing work in this area uses comparators to provide true/false inputs for the FSM. This work takes a different approach and uses a FSM with a network data input. This approach has not previously been that well favoured because of the FSM complexity. This work shows how we can reduce this complexity by using a pre-FSM hierarchical compression system and improve the performance by using multi-byte input words.

The next section explains the background to this area of research and outlines some of the related work. The Implementation section explains the mechanisms used and how some of the potential problems are addressed. The Modelling section explains how the system was modelled and the compilation and simulation tools that were produced to test these ideas. The results of this work and details of related work are given in the Results section and the final section gives the conclusions and ideas for further work.

Discussion

At speeds of 1 Gbps and above, it becomes difficult to perform intrusion detection in software and some form of hardware assistance is typically required (Gokhale, Dubois, Dubois, Boorman, Poole & Hogsett, 2002). Custom hardware can be used for this, or we can develop algorithms that can be included within field programmable gate arrays (FPGAs). With the move to hardware based solutions, we have an opportunity to introduce far more parallelism that would be possible in software and hence develop different types of algorithms that would not be practical in software.

Existing work

There is quite a lot of existing work in using hardware to implement string matching for intrusion detection systems, and this sub section provides a sample of some of the techniques used.

A commercial product from PMC-Sierra is described by (Iyer, Kompella & Shelat, 2001). This system called ClassiPi is a classification engine and is implemented as an ASIC (Application Specific Integrated Circuit). This is a flexible programmable device on which software-like algorithms can be implemented. Performance depends on the type of algorithm being implemented, with more complex operations, such as searching for regular expressions, having a lower performance.

An approach by (Gokhale et.al., 2002) compares the contents of packets against a CAM (Contents Addressable Memory). A match-vector is created giving details of the entries matched within the CAM – this is appended to the packet and forwarded to the software for further processing. This approach has the advantage of being dynamically reconfigurable – by changing the contents of the CAM.

A paper by (Cho, Navab & Mangione-Smith, 2002) uses a comparator based system. This operates on 32-bit words and compares input data against parts of the string being matched at each of the possible 4 byte offsets within the word. The match is successful if for a particular byte offset all parts of the string are matched at that offset. Separate logic is used for each of the patterns being

matched and the rules are processed to create the VHDL code required for it implementation. A disadvantage here is that the FPGA needs to be recompiled every time the set of rules are changed.

FSM based approaches

One method of implementing intrusion detection systems is by the use of Finite State Machines. In practice many other designs are actually implementing FSMs within the hardware, although these may not be formally specified as such. A general mechanism can be based on comparing input data with constant match values using comparators and then using these Boolean results as input to the various forms of FSM used. Simple FSMs can be constructed with relatively small amounts of logic within FPGAs – such as the use of 'one-hot-encoding'. A restriction here however is a limitation in FSM complexity – the algorithm we use may require us to compare our input data with many possible values in parallel.

A rather different approach to matching is taken by (Franklin, Carver & Hutchings, 2002), in that they use non-deterministic Finite Automata (NFA) to perform the pattern matching – this approach first being suggested by (Sidhu & Prasanna, 2001). This approach has the advantage of being able to perform matching of regular expressions as well as simple strings. The original work by Sidhu and Prasanna supported the use of dynamic configuration, although this was not used in the implementation by Franklin et.al – thus requiring FPGA's to be rebuilt after a rule change. The work described by Franklin et.al operates on a byte wide data stream, and the authors report clock rates of 30-120 MHz depending on the complexity of the rules being matched.

Table based FSM

A more flexible approach is to use a table based implementation of a FSM. Here we hold the current state in a register, and use tables to provide the mapping from current state and input to next state and output. With this approach, our table input can represent all possible input values – for example being the data input from the network. Holding the tables in memory allows for fast updates to the FSM specification. A disadvantage however is the amount of memory required which increases exponentially with the word size, as shown in Equation 1, where: s is the number of states, i is the number of input bits, o is the number of output bits and M is the memory required in bits.

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

Equation 1 - FSM memory requirements

We have a conflict however, in that given the same clock speed the throughput of the FSM is proportional to the word size. As we will explain later, it is possible to build algorithms that work with a large word size – but given the standard approaches, these may require an amount of memory that is too large or even impossible to provide.

Implementation

The use of a FSM based approach allows us to implement many of the standard string matching algorithms. This section gives a short overview of these techniques, how we can adapt these to work with multi-byte data input, and how compression can be used to reduce the FSM complexity. In hardware implementations, we would typically prefer to operate on a data stream rather than having random access into a packet's content – this restriction affects our choice of algorithm to those that are able to operate on data in this sequential manner. It is also preferable to be able to deal with one data item per clock cycle. This last constraint also has the advantage, given a fast

enough clock speed, of ensuring that the system will always run at network speed irrespective of the data being searched for.

String matching techniques

The fastest way of matching a single string is considered to be the Boyer-Moore algorithm (Boyer & Moore, 1977) and variations on this. The Boyer-Moore algorithm operates by performing the matching of a pattern against input data 'right to left' – starting with the last character of the pattern first. This requires us to scan the input data in both directions and also requires us to skip forward several characters in some situations. This provides excellent throughput, but does not match the hardware requirement of operating on a stream of data.

Another more straight forward approach is the Knuth-Morris-Pratt (Knuth, Morris & Pratt, 1977) (KMP) algorithm. This scans the incoming data in a 'left to right' sequence and thus in order of arrival. On a mismatch, the longest partial match is determined and this used as a starting point for further searching. The KMP algorithm can be implemented easily using a FSM, with the FSM state representing the partial match of the string that has been achieved so far. A basic implementation of the algorithm compares the input data against the value being searched for and will repeat the comparison again (possibly several times) after a mismatch takes us to a lower state. A recent paper by (Baker & Prasanna, 2004) extends this approach to using dual comparators and examines its buffering requirements. The FSM implementation can however be optimised to allow a single transition per input data item, this however requires a more complex FSM and typically requires a table based implementation as described earlier.

We can match multiple strings in parallel by use of an algorithm such as Aho-Corasick (Aho & Corasick, 1975). This builds a tree like structured FSM that relates to all of the strings being matched, and with nodes often being shared by more than one search string for which it is a common prefix. As with KMP, a FSM implementation can be created that allows one transition per input data item, again often creating rather complex FSMs.

Multi-byte input and compressed input symbol stream

We can if we wish use FSM string matching to operate on multiple bytes per clock cycle. A solution to modifying FSMs to operate on more than one input symbol at a time was described by (Hershey, 1994) as a method of creating monitoring systems that would operate at Gigabit speeds.

It is possible to generate FSM implementations of either KMP or Aho-Corasick (for example) that can operate on several bytes at a time. We start with the basic FSM structure that would be used for operating on one byte at a time, but we jump several states in a single clock cycle. Generating the state transitions for this is quite simple, we first create a FSM that operates on one byte per clock cycle and then for each state we can apply all possible input values, one byte at a time to the FSM to determine the next state and any outputs following receipt of that value.

The above method sounds like being difficult to implement as we increase the word size because the number of possible input values increases exponentially. In practice however, for any particular string matching system we are only interested in a subset of the input data values, and any other values will have the same (no match) significance. As an example, we can look at the search string "ABCDEFG". In searching for this in a system with a 4 byte word size, we are only interested in the set of pattern S_{AG} as shown in Equation 2.

$$S_{AG} = (***A, **AB, *ABC, ABCD, BCDE, CDEF, DEFG, EFG*, FG**, G***, ****)$$

Equation 2 – Symbol set for search string "ABCDEFG"

We refer to S_{AG} as the "symbol set" of this system. In this symbol set, the * represents a wild card value, and the **** entry represents all of the input strings that do not match any of the other patterns. These different strings represent all of the possible parts of the string we are searching for at all possible byte offsets within the word. Given that we are only interested in a rather limited sub-set of all possible input values, the input data word can be compressed – or 'classified' – into a relatively small token value (effectively an enumerated type). This therefore gives a very small data input into the FSM, and thus gives vast savings in the memory required for FSM tables for any cases with a large input word size such as above. We therefore reduce the complexity of the FSM, but at the cost of requiring the pre-FSM classifier stage. The classifier partitions all possible values of an input between a smaller set of values on an output – this idea was based on the technique used by (Gupta & McKeown, 1999) for packet classification.

Sharing the pre-FSM classifiers

The pre-FSM 'classifier' stage forms a substantial part of the logic required for this method of implementation. We have many strings that we are trying to match within our IDS – each with its own FSM and pre-FSM input classifier. This approach would be potentially wasteful, as there may be 'symbols' that are shared between multiple rules.

To give a simple example of how the classification can be shared, consider the symbol sets S_{AG} and S_{CJ} that relate to matching strings "ABCDEFG" and "CDEFGHIJ" respectively. If we now combine these two sets, we have the set $S_{AG} \cup S_{CJ}$ which contains all patterns of interest to matching either of the two strings, as shown in Equation 3.

$$S_{AG} \cup S_{CJ} = (***A, ***C, **AB, **CD, *ABC, *CDE, ABCD, BCDE, CDEF, DEFG, EFGH, FGHI, GHIJ, EFG*, HIJ*, FG**, IJ**, G***, J***, ****)$$

Equation 3 – Union of two symbol sets.

We could now form a first stage of classification on the input data to give the combined set, and then two second stages of classification of the combined set to give the set for each of the two FSMs, as shown in Figure 1.

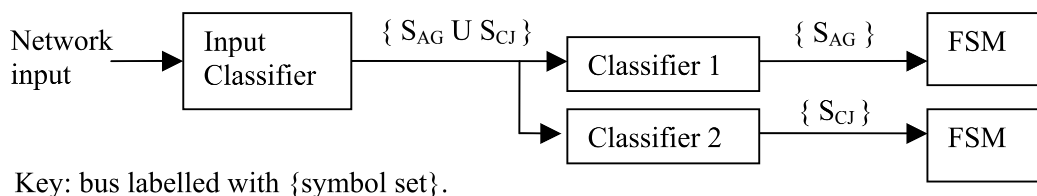

Figure 1 - Sharing the input classifier

In a large system, we can build a tree hierarchy of these classifiers, an example of which is shown in Figure 2. There may be several levels in the tree between the input classifier and the final FSMs. We find in practice that the first stage of classification (the input classifier) is often the most complex, with the subsequent stages requiring relatively little logic – it makes sense therefore to share the input classifier between many rules.

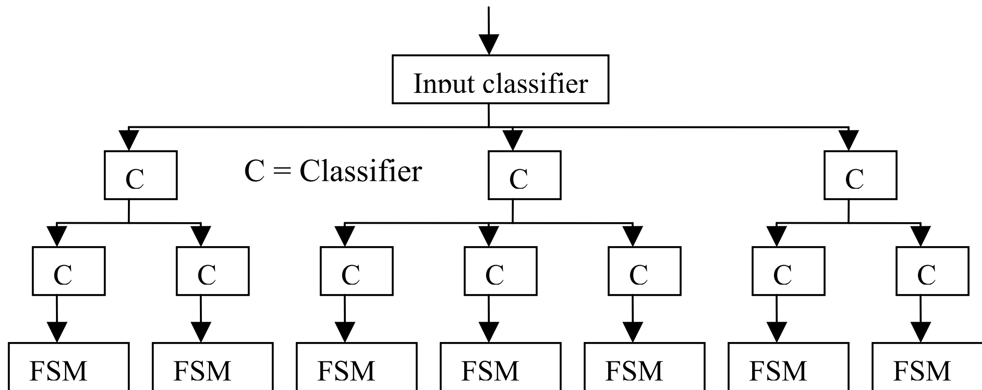

Figure 2 - Example of a three level classifier hierarchy

Dealing with conflicts in priority

One issue that we need to deal with is that some input data may match more than one of our patterns because of the use of wildcards. A piece of input data may match the start of one string and the end of another (or the same) string. We cannot give priority to either of the start or end pattern as this may cause us to miss one of the strings we are searching for.

We can deal with this by partitioning the classification into two parallel systems, and using a 'primary' classifier to detect all strings, apart from those ending with real characters and starting in wildcards – and a secondary classifier to detect strings that start with wildcards. From our example above of matching the string "ABCDEFG", we would have a primary set S_P and a secondary set S_S as shown in Equation 4.

$$S_P = (\mathrm{ABCD, BCDE, CDEF, DEFG, EFG^*, FG**, G***, ****})$$
$$S_S = (\mathrm{*ABC, **AB, ***A, ****})$$

Equation 4 – Primary and secondary symbol sets

The sets S_P and S_S are each sorted in order of priority, with the more wild cards a string contains the lower is its priority. We have no priority conflicts in each of these sets, as input data will be classified with the longest possible match.

When performing any of the intermediate classification stages, we may have several patterns merged together. For example, in Figure 1, both "FGHI" and "FG**" in set $S_{AG} \cup S_{CJ}$ will map into "FG**" in set S_{AG}.

At each of the FSMs, we perform a cross product (or merge) of the primary and secondary inputs – this is shown in Figure 3. The 'cross product' S_M of sets S_P and S_S is shown in Equation 5.

$$S_M = (\mathrm{***A, **AB, *ABC, ABCD, BCDE, CDEF, DEFG, EFGA, EFG^*, FGAB, FG*A,}$$
$$\mathrm{FG**, GABC, G*AB, G**A, G***, ****})$$

Equation 5 – Merge of primary and secondary symbols sets.

It is possible for the input to match both the primary and secondary symbol sets, and we can see in S_M several examples of where this introduces new symbols into the merged set. In some cases we may also have additional patterns that relate to overlaps between start and end symbols.

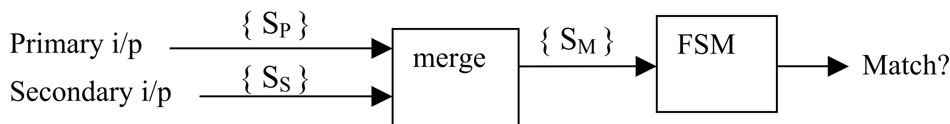

Figure 3 - Finite State Machine and Merge Stage

Summary

We have seen in this section how we can modify the standard string searching mechanisms into a system that operates on a multi-byte input data stream, classifies the input data using a hierarchical tree based compression structure and then searches for each string using a simple FSM that operates on a customised token stream. The problem of conflicts between starts and ends of strings for either the same string or between different strings is dealt with by matching the starts of strings preceded with wildcards separately from other patterns.

Modelling

The intrusion detection system has been modelled in Java to allow the rules to be processed and tested. There are a number of different objects that are modelled here, relating to the corresponding hardware components. The software modelling takes the form of a compiler that reads the various rules from an input file and instantiates the various objects required for the system implementation. The objective is to be able to build data tables for a system that is capable of being implemented within a Xilinx Virtex II FPGA ("Xilinx Virtex-II", 2005).

Compilation

The major role of the compiler is to create the various symbol tables that relate to each of the data paths within the system and to build the contents of the tables for each of the finite state machines and classification stages. The various object modelled are as follows.

The finite state machine

The finite state machine uses a KMP based FSM implementation that operates on tokens that relate to multi-byte input patterns. This is preceded with a merge stage that performs the cross product of the primary and secondary inputs, as explained earlier. The compilation is performed by: generating the primary and secondary symbol sets; building the lookup table for the merge and determining the merged symbol set S_M ; and then finally generating the FSM tables to handle the input relating to symbol set S_M .

The merge stage is quite straightforward as the only complexity occurs with symbols containing wild cards. If a symbol with no wild cards occurs on the primary input, then this must be the valid output – otherwise we use a small lookup table to determine the correct output symbol based on the two inputs. With large word sizes, the lookup table will typically contain a lot of unused entries (relating to impossible combinations of inputs), so it is generally best in this case to build the lookup table dynamically for each string.

The input classifier

This is the first stage of the classification system, which is required to take the raw input data and to generate an output that classifies the data as one of the many patterns that are being searched for.

The input classifier can be created in many different ways. For this work, the input classifier is modelled as Ternary Content Addressable Memory (TCAM). This operates by having a number of words which can contain match patterns – each bit being specified as 0, 1 or don't care. The TCAM is then presented with a data item as a search key, and will search its memory for patterns that matches the key and will identify the highest priority match.

In this system, the TCAM is initialised with the output symbol set in the correct priority, fed with network data as search key input and outputs the address of the pattern with the lowest address that gives a match.

The Classifier Group

Sets of intermediate classifiers with a common address input are formed into a classifier group that takes a data input with an associated input symbol set and generates a number of outputs each with their own smaller symbol sets, as shown in Figure 4.

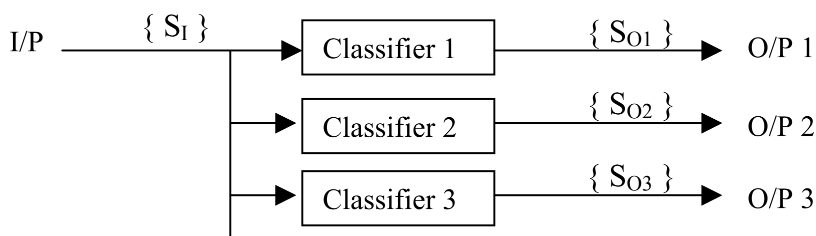

Figure 4 - Classifier group

We build the classifier group by starting with the symbol sets relating to the outputs. We then create an input symbol set from the union of the output symbol sets and then sort this in terms of priority. For each output symbol set, we decide how the input symbol set is partitioned between each of the possible output values and build a corresponding classifier table to perform this function. As the classifier tables share a common address input we can merge these into a single lookup table with a wide output word containing the bit fields for the various outputs.

The classifier groups are modelled as being implemented using memory resources within a Xilinx Virtex II FPGA ("Xilinx Virtex-II", 2005)..

Implementation

The compilation stage above instantiates the various objects required to model the system, and builds the contents for the FSM and classifier tables. The compiler also allows the model that has been built to be simulated, and tested with artificial network data from an input file.

Results

The compiler was built for a four stage architecture as shown in Figure 5. The number of FSMs and classifier groups at each level is decided dynamically. The size of the various objects can be configured via the user interface.

The compiler was tested by using a set of 78 case dependent rules from one of the standard Hogwash (Larsen & Haile, 2001) rule sets. The architecture of the system was set initially to an input word size of 32-bits, a maximum FSM i/p word size of 6-bits and a maximum FSM state variable size of 5-bits. Of the 78 strings, 2 were flagged as being too short for implementation using this system and 2 were flagged as being too long for the FSM size that was selected (the two

strings that were too long were significantly larger than the others). The IDS was simulated in Java with a network data input consisting of instances of strings from the rule set – these strings were identified correctly as expected. The system was also tested by providing pairs of strings from the rule set at various spacings and byte offsets to test the use of the primary and secondary matching – this operated correctly in all cases.

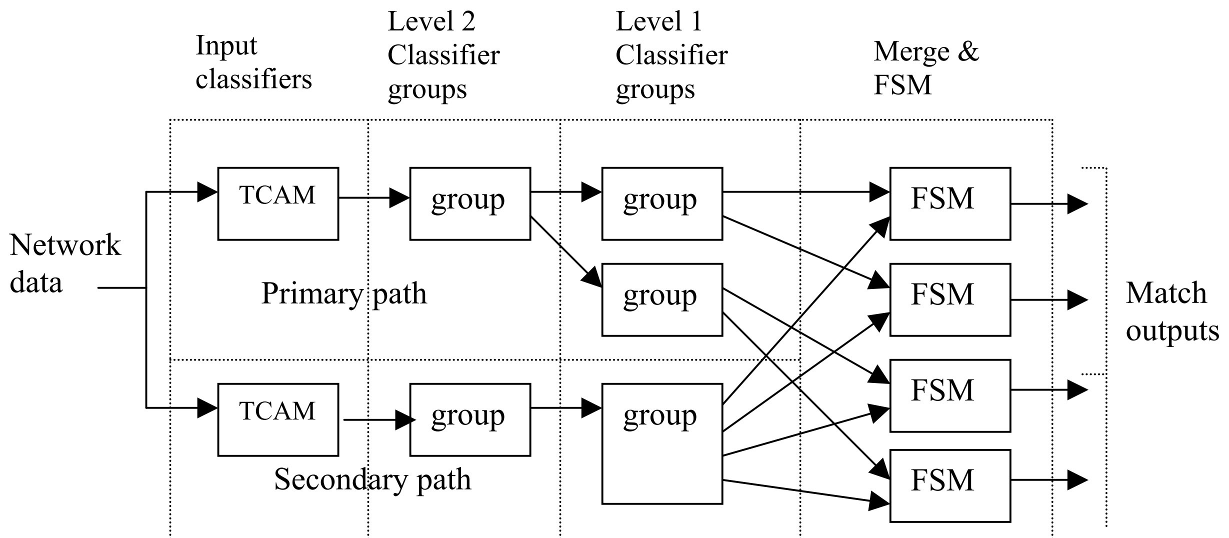

Figure 5 - Simple example of architecture

The same input rule set was compiled for a variety of different word sizes. In each case, the resource utilisation and the number of rules implemented were recorded. The memory requirements were converted by hand to the corresponding amount of Xilinx memory resources utilised, the results of which are given in Table 1.

<table>
<tr><th rowspan="3">Word size (bits)</th><th colspan="3" rowspan="2">Number of search strings which are…</th><th colspan="5">Memory requirements</th></tr>
<tr><th colspan="2">Xilinx Block RAM's</th><th colspan="2">Xilinx slices used for RAM*</th><th rowspan="2">External TCAM</th></tr>
<tr><th>Too short</th><th>Too long</th><th>Ok</th><th>Total</th><th>Per string</th><th>Total</th><th>Per string</th></tr>
<tr><td>8</td><td>0</td><td>2</td><td>76</td><td>45</td><td>0.59</td><td>0</td><td>0.0</td><td>167x8-bit</td></tr>
<tr><td>16</td><td>0</td><td>2</td><td>76</td><td>48</td><td>0.63</td><td>460</td><td>6.1</td><td>590x16-bit & 40x16-bit</td></tr>
<tr><td>32</td><td>2</td><td>2</td><td>74</td><td>88</td><td>1.19</td><td>444</td><td>6.0</td><td>753x32-bit & 165x32-bit</td></tr>
<tr><td>64</td><td>6</td><td>2</td><td>70</td><td>85</td><td>1.21</td><td>840</td><td>12.0</td><td>823x64-bit & 419x64-bit</td></tr>
<tr><td>128</td><td>52</td><td>2</td><td>24</td><td>91</td><td>3.77</td><td>0</td><td>0.0</td><td>424x128-bit & 352x128-bit</td></tr>
</table>

*Note, this relates to memory usage only, NOT logic implementation.

Table 1 - Memory requirement vs. word size

Here, memory items that can be implemented in 16 or less Xilinx logic slices are assumed to be implemented in the small amounts of memory within the slice – memory items larger than this are assumed to be implemented in Xilinx Block RAM components. TCAM resources are assumed to be provided by external components.

From Table 1 we can see that the system works well up to a word size of 64-bits, but beyond this we have a minimum string length that is longer than the majority of the strings in the rule set and a greatly increased Block RAM utilisation. The latter is due to the large increase in the number of additional patterns introduced by the merge stage as explained earlier. In all cases, the amount of external TCAM required is relatively small. The amount of Xilinx slices used for small memory blocks is very small – representing around 2% of the resources of a XC2V8000 in the worst case above.

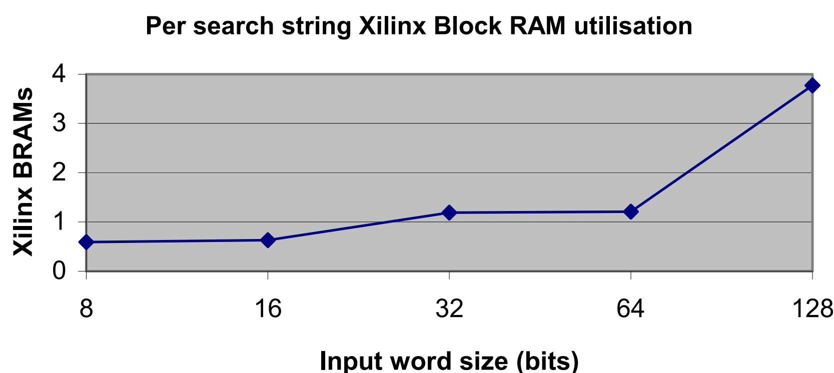

Figure 6 - Per string resource utilisation

The per string Block RAM utilisation is shown in Figure 6 – we can see that up to a word size of 64-bits the resources used grow relatively slowly with the input word size.

Any practical intrusion detection system will require the packet header fields to be examined first as specified by the set of rules used. For each rule that successfully passes when testing the header fields, we will need to search the body of the packet to see if any strings are present that are specified by the rule content options. This paper addresses the problem of string matching and assumes that hardware already exists for testing the header fields. The system described in this paper would search for all content strings specified by all rules and generate a set of Boolean "match" bits that specify for each string whether it appears in the packet. For any rule that successfully passes testing the header fields we would then test the match bits that relate to the strings specified by the rule's content options.

Related Work

A closely related parallel piece of work is described by (Sugawara, Inaba & Hiraki, 2004). The methods used by Sugawara et.al. are similar to the work described in this paper in that they also use the idea of working on a multi-byte data input and classifying this as an 'Identification Number'. There are a number of differences however, in that Sugawara et.al. use Aho-Corasick to implement multiple string matching within (mainly) a large FSM. A significantly different mechanism is also used for dealing with start and end conflicts, and for the initial input classification.

The 'input classification' method used by Sugawara et.al. is similar to the "recursive flow classification" system described by (Gupta et.al. 1999) for packet classification. A major development by Sugawara is a 'sparse array' implementation within a FPGA that enables large

memory savings to be made – this is used for both input classification and also for the large FSMs required for Aho-Corasick.

Related software developments

In recent years the performance of systems such as Snort (Roesch, 1999) have been improved by optimising the order in which data is compared. A paper by (Kruegel & Toth, 2003) uses rule clustering and is implemented as a modification to the snort rule engine. Their work uses decision trees to reduce the number of redundant comparisons made against the content of a packet, with multiple string matching being implemented using an algorithm designed by (Fisk & Varghese, 2001).

A recent paper by (Abbes, Bouhoula & Rusinowitch 2004) uses a decision tree in conjunction with protocol analysis. Their system is guided by a specification file for each the protocols being monitored and is able to perform comparisons and Aho-Corasick multiple pattern matching on only the relevant parts of the data stream. This reduces the workload and targets the pattern matching only on the relevant parts of the data, thus reducing the number of false positives – it also provides a more accurate way of restricting the data searched than offset and depth constraints.

Work by (Paul, 2004) on distributed firewalls, improves their performance by using a stateful packet classification system distributed across several consecutive access control devices, thus reducing the load on any particular system. This serial composition of packet filters is interesting as this technique could also be used for other intrusion prevention systems.

The optimisation of snort to reduce the number of comparisons is particularly important for software based solutions because of the need to perform comparisons sequentially. We have a different set of constraints for hardware solutions, as we may be able to perform multiple comparisons or table lookups in parallel, with the cost being the amount of logic resources used. In comparison with the work described above, this paper takes a very simple approach of using multiple instances of the KMP string matching algorithm. Although many of the searches that take place are not needed this is not a problem as these operate in parallel.

Conclusion

The work described in this paper looks at a method by for performing high speed string matching on a multi-byte data input using a table based finite state machine approach. The problem of the exponential growth of the FSM memory resources with input word size is addressed using a pre-FSM classification stage that removes redundancy in the input data.

The FSM's perform string matching using a KMP based algorithm that is extended to operate on tokens representing multi-byte patterns in the input data stream. The FSMs themselves use only a small amount of hardware resources, with FSM designs for 32 or 64-bit input words that implement a single "standard length" intrusion detection rule being implemented in one Xilinx Virtex II Block RAM component. To avoid the classification stage using large amounts of resources, we share a single tree based classification system between multiple FSM's. As we match the start and ends of strings within words using wild card characters we need to avoid the problem of input data matching patterns for starts and ends of strings at the same time. We do this by splitting the wild card matching of the start of strings into a separate classification system from the rest of the matching and only combine the two streams of classified tokens prior to each individual FSM.

Software has been written to perform the processing required to compile rules into a model of the hardware required, to build the contents for the various lookup tables and to perform a high level

simulation. The system has been tested with a subset of one of the Hogwash rule sets and the resource requirements noted for various input word sizes. The results show that the resource requirements grow slowly with the input word size and that the system is effective with the rules used up to a word size of 64-bits – with coverage of 90% of the rule set.

Although the software compilation is quite complex, the hardware implementation is quite simple, consisting mainly of interconnected lookup tables which are initialised with data from the compiler. The system has been designed to consume input data at a rate of one input word per clock cycle, with pipelining used to improve performance. This deterministic operation is independent of the rules used and the network data being searched. The performance is therefore determined by the size of the input data word and the maximum clock speed possible at this word size. Initial tests of the logic synthesized for the finite state machines give a maximum clock speed of 150 MHz. A conservative estimate for the clock speed of the entire FPGA would be 100 MHz for a system with a 64-bit data input word, thus giving an estimated FPGA performance of around 6.4 Gbps – performance though may be limited by the search rate of the external TCAM. We will only be able to determine an accurate figure for clock speed however after the system has been completely modelled as described in the next subsection.

Further work

The next stage in this work is to build a (register transfer level) VHDL model of an example system and to initialise the various memory blocks with data from the compiler. The VHDL model (along with a behavioural model of a suitable TCAM component) can then be simulated using artificial network data input to provide low level tests of the system – the VHDL model can also be synthesized and a Xilinx design built to determine the overall maximum clock speed.

From the VHDL model, we should be able to go relatively quickly to a design for a real hardware system. During testing we can incorporate data for the lookup tables into VHDL files prior to synthesis – however for a practical implementation we need to be able to provide fast updates to the tables in a working system. To avoid needing to re-synthesize or re-build the Xilinx design, we can build a 'skeletal' design and then modify the contents of the lookup tables at load time by using the Xilinx JBits software ("JBits Tutorial", 2003) to modify the contents of the lookup tables within the Xilinx 'bit' file that is to be loaded into the FPGA. The compiler can be modified to call the appropriate JBits classes itself and thus create a new 'bit' file to load into the FPGA.

To address the issue of rules being too long for the FSM implementation used and to improve the efficiency, we can use FSMs of various sizes and allocate the rules to the appropriate FSM size accordingly. The issue of some rules being case dependent and some not can be addressed by moving to a system with two complete sets of classifiers and partitioning the rules between the two. The distinction between the case of letters for case independent rules can be dropped in the input classifiers dealing with these rules.

The algorithm developed limits the minimum string length to no less than 1 byte less than the word size, as a string shorter than this might start and end with a wild card within the same input word. It would be interesting to investigate if there are any straightforward methods of matching these shorter strings.

References

Abbes, T., Bouhoula, A., & Rusinowitch, M. (2004). 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.

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

Baker Z.K., & Prasanna, V.K. (2004). Time and Area efficient pattern matching on FPGAs. In proceedings of the ACM/SIGDA 12th International symposium of Field programmable gate arrays, FPGA'04 (pp.223-232). Monterey, California, USA.

Boyer, R.S., & Moore, J.S. (1977). A Fast String Searching Algorithm. Communications of the Association for Computing Machinery, 20(10), 762-772.

Cho, Y.H., Navab, S., Mangione-Smith, W.H. (2002). Specialized Hardware for Deep Network Packet Filtering. In proceedings of Field Programmable Logic and Applications, 12th International Conference, FPL2002, Lecture Notes In Computer Science, LNCS 2438 (pp. 452-461). Springer Verlag.

Fisk, M., & Varghese, G. (2001). An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection (successor to UCSD TR CS2001-0670, UC San Diego, 2001). Retrieved 18 March 2005, from http://public.lanl.gov/mfisk/papers/setmatch-raid.pdf

Franklin, R., Carver, D., and Hutchings, B.L. (2002). 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.

Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., & Hogsett, V. (2002). Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In proceedings of Field Programmable Logic and Applications, 12th International Conference, FPL2002, Lecture Notes In Computer Science, LNCS 2438 (pp. 404-413). Springer-Verlag.

Gupta, P., & McKeown, N. (1999). Packet Classification on Multiple Fields. SIGCOMM 99, Computer Communications Review, 29(4), 147-160.

Gupta, P., & McKeown, N. (2001). Algorithms for packet classification. IEEE Network, 15(2), 24-32.

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

Iyer, S., Rao Kompella, R., Shelat, A. (2001). ClassiPi: An Architecture for fast and flexible Packet Classification. IEEE Network, 15(2), 33-41.

JBits Tutorial, (2003), JBits 3.0 software, Xilinx Inc.

Knuth, D.E., Morris J.H., & Pratt, V.B. (1977). Fast pattern matching in strings. SIAM Journal of Computing, 6(2), 323-350.

Kruegel, C., & Toth, T. (2003). 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 Verlag.

Larsen , J., & Haile, J. (2001). Securing an Unpatchable Webserver … HogWash. Retrieved 18 March 2005, from http://www.securityfocus.com/printable/infocus/1208

Paul, O. (2004). 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-Verlag.

Roesch, M. (1999). Snort - Lightweight Intrusion Detection for Networks. In proceedings of LISA '99: 13th Systems Administration Conference (pp. 229-238). Seattle, WA : USENIX.

Sugawara, Y., Inaba, M., & Hiraki, K. (2004). 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-Verlag.

Sidhu, R. & Prasanna, V.K. (2001). 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.

Xilinx Virtex-II Platform FPGAs: Complete Data Sheet – Product Specification. (2005). Xilinx Inc. Retrieved 18 March 2005 from http://direct.xilinx.com/bvdocs/publications/ds031.pdf