An Intrusion Detection System for Gigabit Networks
(Working paper: describing ongoing work)

Gerald Tripp

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

This draft paper is now superseded by the following technical report:

An intrusion detection system for gigabit networks – architecture and an example system. Gerald Tripp. Technical Report 7-04, Computing Laboratory, University of Kent, April 2004. http://www.cs.kent.ac.uk/pubs/2004/1893/content.pdf

Executive summary

Intrusion detection consists of monitoring network traffic for various kinds of security threat – to do this we need to inspect the contents of packets arriving from the network. This can be very computationally expensive as we may not have any particular location within the packet that we wish to inspect – standard mechanisms therefore tend to include pattern-matching systems that can scan for a string or regular expression anywhere within the packet.

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 are normally required. At these types of speeds, we usually 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 and hence alternative algorithms. Hardware based solutions are probably the only approach currently practical for intrusion detection on high-speed backbone networks running at around 10 Gbps.

We can potentially build very fast intrusion detection systems using custom logic, although this has the problem that if we wish to change the intrusion detection operations performed we may need to generate a new hardware design. Systems can also be built with patterns stored in registers, but here we need to supply copies of incoming data to large numbers of comparators, which does not scale well with the numbers of rules or input word size. Memory based systems are a good way in which to make systems flexible and easy to update – however existing program based systems or large finite state machine based solutions can suffer from lack of memory bandwidth.

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.

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 and to build a demonstrator to show that this can be implemented using current state of the art hardware.

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 [i].

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 UNIX software intrusion detection system called Snort [ii]. 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.

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. [iii] 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 [iv]. 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. [v] 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 [vi]. This paper describes the system called ClassiPi which is a classification engine and is implemented as a custom ASIC. 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. [vii] 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 [viii] 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 utilisation increases exponentially with the input data word size. 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 [ix] string matching algorithm) to work with a multi-character word. 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 [x].

2.2.1 Recursive flow classification

Gupta and McKeown proposed a mechanism for packet classification [xi] 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 classification value for that group of bits. The general idea is that the size of the classification value (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 classified 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.

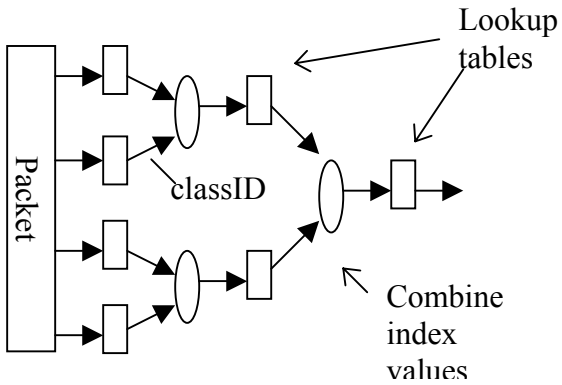


Figure 1 – diagram of RFC, based on [xi]

2.2.2 The 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, 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.

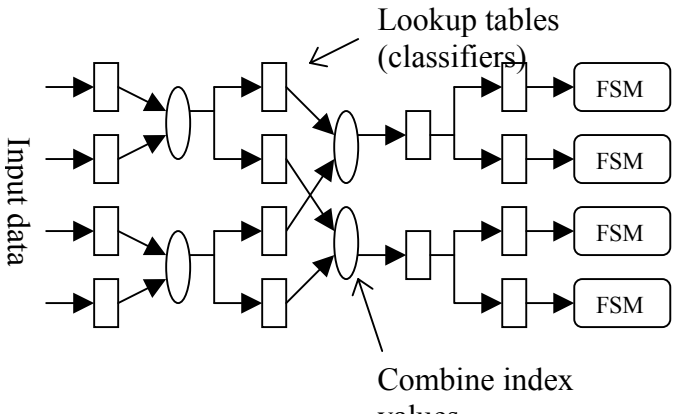


Figure 2 – simple example of proposed solution

The input symbol set for the 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 classifier stage, prior to the input into a FSM.

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.
- 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 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 proposed in this project, 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 course of this project 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. We would aim to update our simulations throughout the project to follow the current state of the art FPGA components.

As this proposal stands, 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 implementing this functionality and the additional use of FPGA resources. There is no plan to include this type of facility in this piece of work – although this could be looked at later as an area for further work outside of this project.

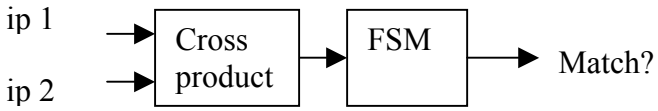
3 Basic Components

Before going into 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. Although not shown here in these high level diagrams, the various stages are pipelined, with one or more pipeline stage per module depending on its complexity. The intention here is to ensure we have a small clock-to-setup delay and hence a good clock rate for the FPGA.

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 with the architecture) and a final value indicating a no-match or full wild card or don't care pattern. This is referred to as the "symbol set" and is specific to each data path. This implies quite a lot of complexity within the compiler – with a symbol table being used for each symbol set. This complexity does not however carry over into the hardware, where mapping between different symbol sets is generally performed using small pieces of RAM as lookup tables. Multipliers and adders may also be used – if available – when generating cross products.

3.1 The Finite State Machine

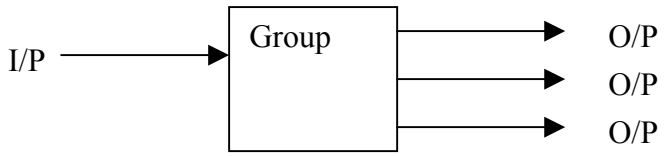
This is implemented as a number of parts, to address the problems related with matching beginning and end of strings using wildcards. A general block diagram is as follows:



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. 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.

3.2 Group

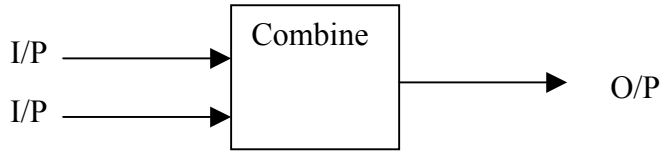
The group is the mechanism that we use to sub-divide a symbol set into multiple smaller symbol sets. This is done in the compiler in reverse by taking the output symbol sets and creating an input symbol set that contains all output symbols. For each output, we determine which input symbols cover each output symbol and build a transformation table for each output. This table will be represented in hardware as simply a piece of RAM.



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.

3.3 Combine

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 performing 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. 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, the compilation is performed 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.

3.4 Input classifiers

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 initialised. 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.

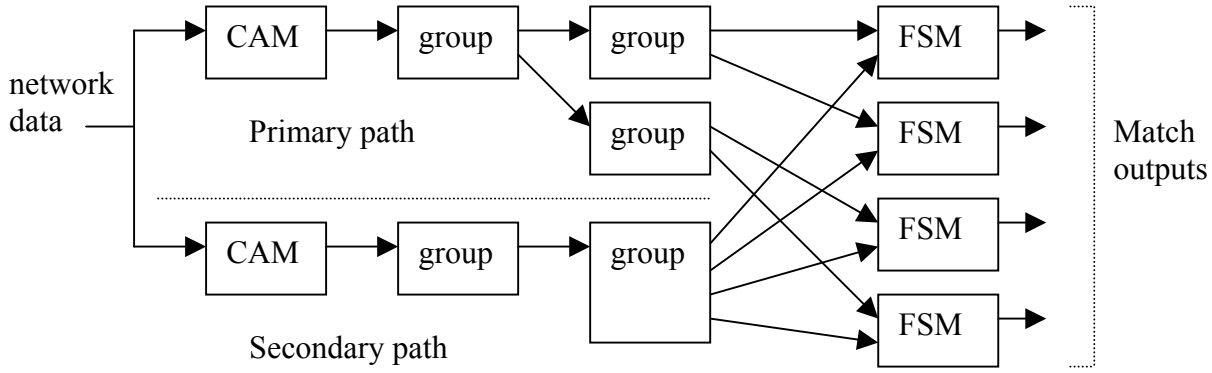
It may therefore be possible to use a single CAM to perform classification of the entire input data word, whereas we are likely to require at least 2 RAM based classifiers and probably more.

4 Compiler

Initial work has looked at the generation of software to perform compilation and simulation of a simple system that uses just a CAM input classifier, two levels of groups and final FSM stages. Primary and secondary paths are used to deal with the main matching and start wild

card matches. The number of groups and FSMs is not fixed by the software, but determined dynamically. The structure of groups at each layer can be specified and the software fits the match patterns to the specified structure.

A simple example of the overall hardware structure is as follows:

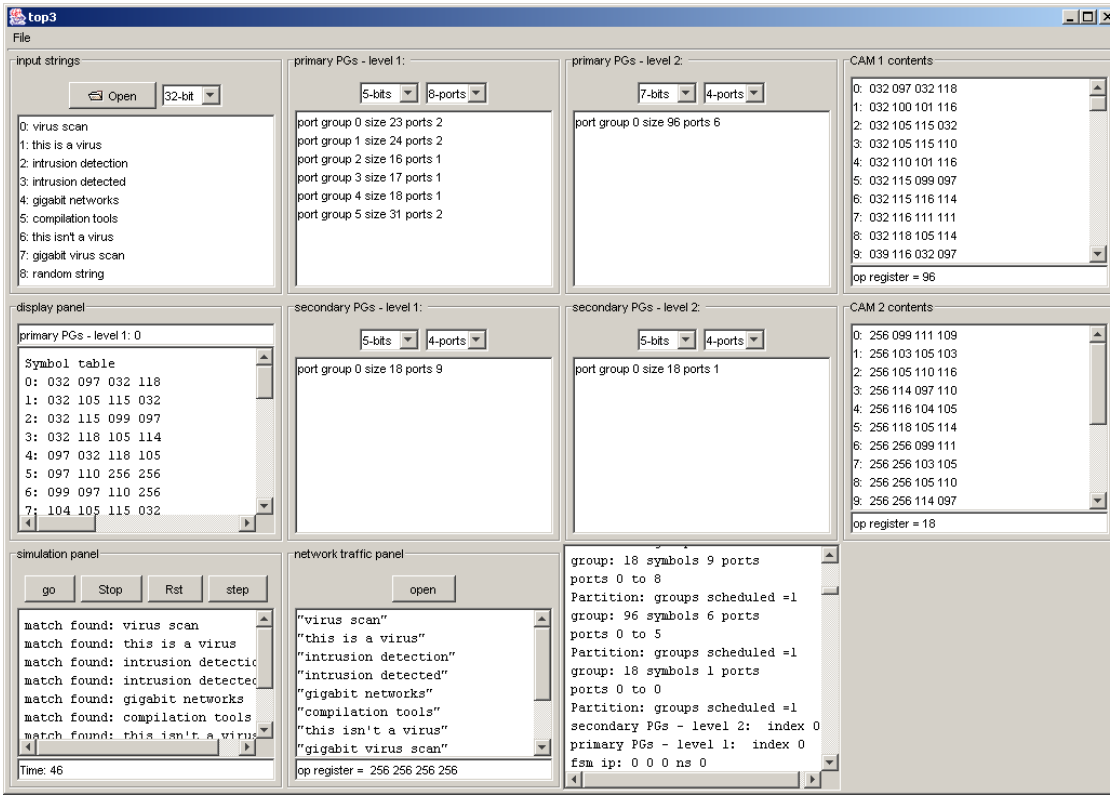


The symbol sets in this example are assumed to be smaller for the secondary path than for the primary path. The first group in the secondary path is therefore not required and hence simply maps input to output.

In a real implementation, the overall structure of the hardware will usually be determined in advance at the time of hardware synthesis and/or FPGA build time. Thus we will have a general structure to which the compiled design will need to be mapped. This mapping is performed to give the best fit, although this may not be an ideal fit and it is likely that some logic blocks may not actually be used. This is an intentional trade-off to avoid the overhead of rebuilding the FPGA design when the intrusion detection signatures change. The time for recompiling the FSMs etc is likely to be a few seconds at most – whereas the time to rebuild the FPGA design may well be tens of minutes at least – also the FPGA place and route is not deterministic and designs with high levels of utilisation of the FPGA may not always build at the first attempt (or at all).

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 being used.

The following diagram shows a screen shot from this compiler that is searching for nine different strings in a file containing the same data.



This software has only just been completed at the time of writing and further testing is still required.

5 Further work

This paper describes ongoing work, so therefore is just a snapshot of the current state. This section describes further work already planned and future ideas.

5.1 Software

5.1.1 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 hardware structure but also the data path widths. This is likely to make the GUI more complex as the screen layout will need to be chosen dynamically.

5.1.2 Compiler input and output

At present the compiler just accepts a set of quoted text strings as input. This needs to be extended to allow arbitrary byte values, such as we may need to search for when looking for a virus signature for example. A parser for the "Snort" rule system has already been written and it is hoped that this can be integrated with the current software.

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.

5.2 Hardware

5.2.1 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 [xii] 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.

5.2.2 Real hardware implementation

The final stage here is to create a test bench system, containing 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.

---

[i] P.Gupta and N.McKeown, “Algorithms for packet classification”, IEEE Network March/April 2001.

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

[iii] 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.

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

[v] 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.

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

[vii] 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.

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

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

[x] 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.

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

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