Rules-based Network Intrusion Detection using a Field Programmable Gate Array

Christopher Hayes and Yatin Singhal
16.671 Advanced Computer Architecture, UMASS Lowell

Introduction

Protecting networks becomes increasingly difficult as their speed and complexity increases. One method of network protection is through intrusion detection – the process of analyzing network traffic to identify unwanted activity. Intrusion detection is generally implemented as a network appliance or software application known as an Intrusion Detection System (IDS). Intrusion Detection Systems use various methods for identifying the unwanted activity – such as pattern matching detection, anomaly-based detection, and protocol analysis.

Pattern matching involves checking specific matching criteria that must be examined in every incoming packet. Patterns may be strings compared against the payload of a packet or values within the header of a packet. Software IDS tools such as *Snort* check packets against checking rules generated from known intrusion techniques. As the rules are matched, alerts or other notifications can be generated to allow a human analyst to determine the next course of action.

The standard rules set for *Snort* contains over 3000 rules that should be matched against incoming packets. However, the most advanced general-purpose processors (at the time of this writing) can only analyze incoming packets against up to 20% of the rules without ignoring incoming packets [2]. Dropped or ignored packets leave holes in the security of a network that intruders can take advantage of.

The drawback to the software-architecture is the serial nature of the processor. A given processor can only analyze a signal packet against a limited set of rules simultaneously. An alternate approach might be to use an architecture that allows rules to be checked in parallel. Such parallel architectures might be realized in hardware elements such as the FPGA (Field Programmable Gate Array).

This paper will detail the design of a pattern matching-based IDS implemented in an FPGA, in an attempt to gain speed advantage over the software-based approach.

Method

The intent of this project is to develop an FPGA-based network intrusion detection system. The system will be rules-based, using rules generated for the software-based *Snort* IDS. String comparisons, based on these *Snort* rules, will be implemented using Bloom Filters or its enhanced derivatives, such as those developed by Song and Lockwood [10].

The block diagram in Figure 1 shows the ideal flow for the hardware platform needed to perform FPGA-based network intrusion detection. Ethernet traffic will enter the platform through an Ethernet MAC/PHY. The output of the MAC/PHY will be sent to the FPGA where the IP packet will be parsed and filtered. Incoming packets will be stored in SDRAM and Bloom Filter state will be maintained in SRAM. Output alerts will be sent out another interface so that they can be viewed by the network security monitor.

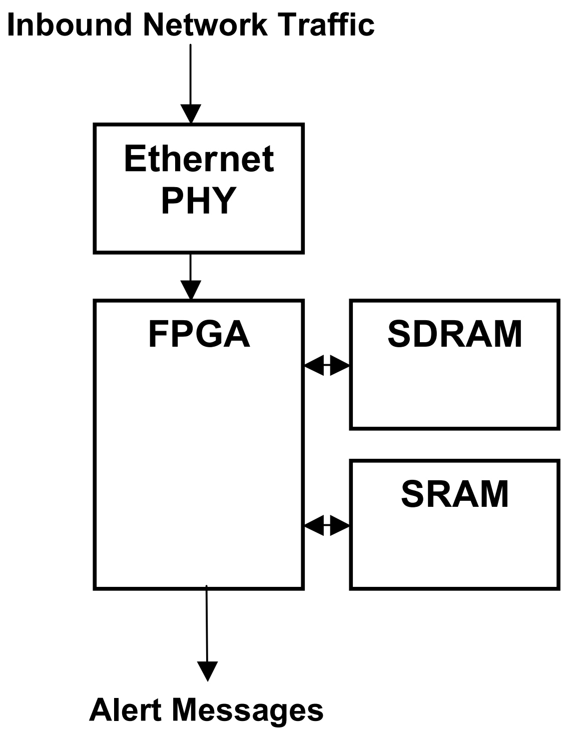

Figure 1. General Block Diagram for a FPGA based IDS.

Some simplifying assumptions have been made, so that focus of the project is kept on optimizing the string matching capabilities.

- In the first iteration, the design will work on a limited set of rules.
- The IDS will only operate on a single packet protocol (TCP) as the development of multiple packet parsers can be handled in future work.
- Design issues related to decoding packets from Ethernet will be avoided by placing test packets into memory attached to the FPGA before processing begins.
- Difficulties related to pasting multiple packets together to search for strings will be avoided in this iteration of design.

The goal of this design is to filter input packets from the output based on defined criteria. This design will perform this function in three stages.

Processing begins by parsing the TCP/IP header. All packets will have their TCP/IP header decoded. The header information is matched against rules in the Snort rule set for matches. If any packet matches any rule, it is sent along to string detection engines for payload inspection and the packet number is passed onto the final processing stage. If there is no match, then the packet is discarded and no further processing is performed on that packet.

The payload inspection block takes each payload that meets the header criteria and compares it against payload strings found within the rule set. If a payload does not match any strings, it is discarded and no further processing is performed on the packet. However, if the payload does contain one or more strings from the rule set, the payload inspection block sends which strings have been matched, along with the packet number to the final processing stage.

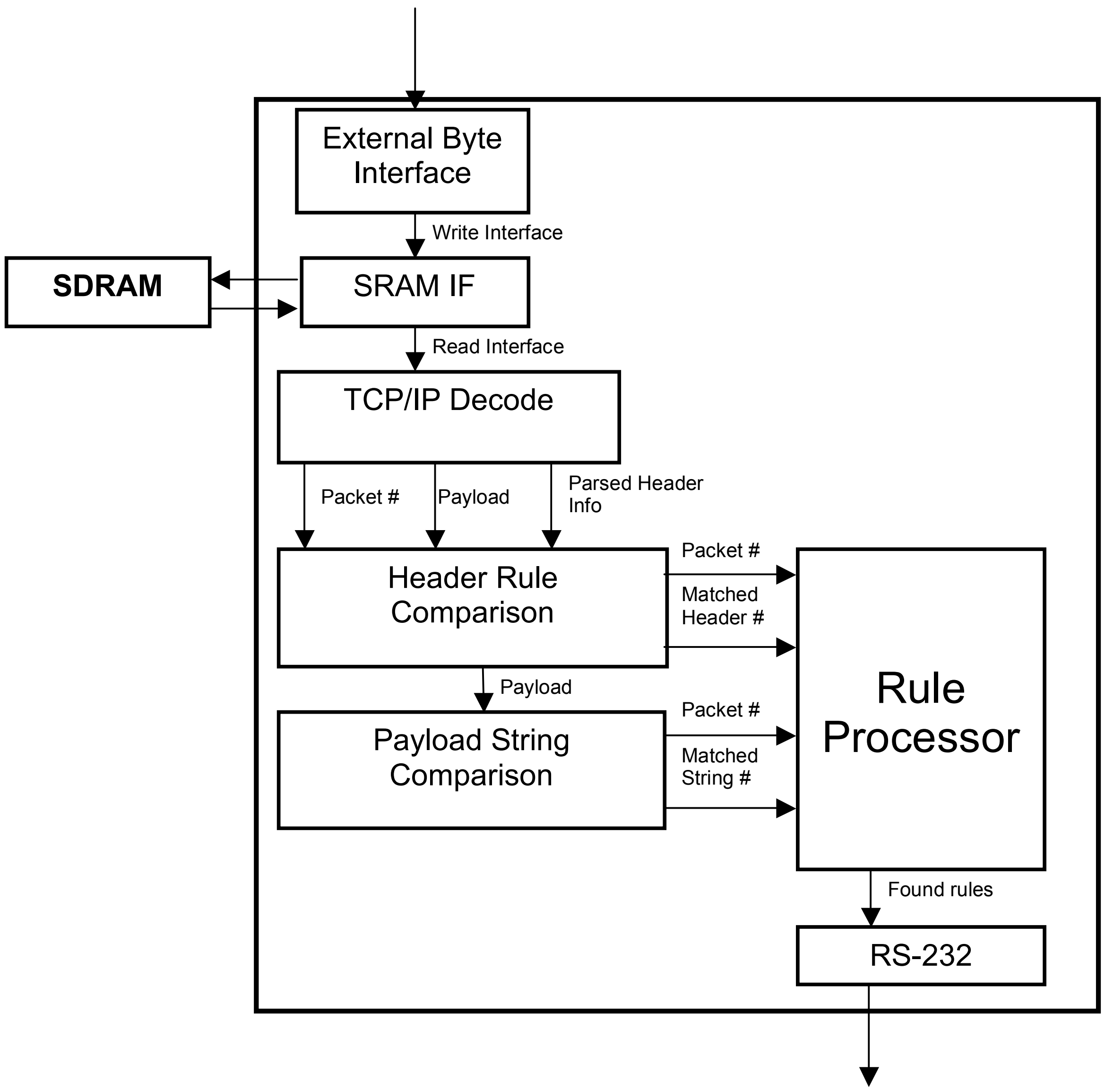

Figure 2. Block Diagram of IDS FPGA Architecture.

In the final processing stage, the matched headers are correlated with the matched payload strings to see if a complete rule has been matched. This block can either check for false positive results from the string comparison stage or it can pass results on for further processing. This block passes its results to the output interface of the FPGA. For demonstration purposes, the output will be a serial RS-232 link to a terminal which will display the results.

The input interface will need to support the aggregate throughput of the network that is to be monitored. This also implies that in the worst case, the SDRAM should be able to handle twice this rate as each packet will need to be written and read. This is, however, a worst case as some packets will only have the header examined. The output message rate will be highly decimated from the input packet rate and will only need to support transmitting an alert message or network command every time a malicious packet is found. At the worst case, every packet is malicious and an alert string will need to be transmitted for each input packet.

Background

Snort Rules

Intrusion Detection Systems enable real-time detection of the network intrusion attempts and open source like snort allows the network administrator to compose and apply intrusion rules that will monitor, detect and protect the network. The intrusion rule can specify the processing of the packets header, processing of the payload (matching patterns of a signature), and the action to be taken when a rue matches.

A typical snort rule is given as follows:

alert tcp any 110 -> any any (msg:"Virus - Possible MyRomeo Worm"; flow:established; content:"I Love You"; classtype:misc-activity; sid:726; rev:6;)

The first portion of the rule signifies that the rule matches any value for the source IP address, destination IP address, and TCP destination port address. However, the source port must be 110. The rule also specifies that the protocol is TCP for which an alert is generated.

- The word "alert" shows that this rule will generate an alert message when the criteria are met for a captured packet. The criteria are defined by the words that follow.
- The "tcp" part shows that this rule will be applied on all TCP packets.
- The first "any" is used for source IP address and shows that the rule will be applied to all packets.
- The "110" is for the TCP source port.
- The "->" shows the direction of the packet.
- The second "any" is used for destination IP address and shows that the rule will be applied to all packets irrespective of destination IP address.
- The third "any" is used for destination port.

The second portion of the rule specifies to search the string “I Love You” over the established TCP/IP connection [2].

Hardware String Comparison

Pattern-matching Intrusion Detection involves searching packet payloads against many string patterns. The current Snort rule set includes over 3000 strings that may be compared against a given payload. Performing a detailed search against this many strings can be prohibitive for a software-based approach. A hardware approach, such as one implemented in a FPGA, can allow for parallelism that can enable intrusion detection to keep up with these line rates.

The simple approach would be to have a string comparison engine for each of the strings. Having over 3000 comparison engines will prove very taxing on FPGA resources. This would require over 3000 string tables stored in independent RAM blocks, with independent sets of logic to control each RAM. It can be easily seen that the simple approach can begin to get out of control as the number of Snort strings increase.

In 1970, Bloom proposed a filtering solution to test whether a candidate is a member of a set of strings. The algorithm substantially reduces the amount of resources necessary to test set membership. This solution, unlike the direct string comparison approach, introduces the probability of false positive string matches (however, there are no false negatives). Another drawback to the Bloom Filter in the intrusion detection application is that it does not identify which member of the set was matched – it merely tests whether the candidate is a member of the set. With some modifications, resulting in an implementation size or false-positive rate increase, the algorithm can be made useful.

The Bloom Filter

A set of hash functions, H, is defined. Each member of the string set, S, is run through each of the hash functions. The result $H_i(S_j)$ is used as an address into a 1 bit table. The bit at the resulting table address is set for every combination of hash function and set member string. At this point, the table contains the bloom filter “coefficients”.

Next the input data stream is run through the same set of hash functions, which addresses the coefficient tables. If all the entries addressed by the hash functions are set to 1, the input string is said to be a member of the set. However, this algorithm can generate false positive results.

In general, a larger set of hash functions decreases the false positive rate. However, the increase of hash functions reaches a point of diminishing returns as the number of hash functions approaches the size of the member string. In general, the false positive rate can be represented as:

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

Where *k* is the number of hash functions, *n* is the number of strings in the string set, and *m* is the number of addresses generated by the hash function [5][8].

Modifications to the Bloom Filter

A drawback to the basic Bloom Filter architecture is in its parallel hardware implementation. If there is a single lookup RAM for bloom filter coefficients, and there are *k* hash functions, the final answer will require up to *k* reads from the RAM. If the ROM is implemented as a single-port RAM, this could reduce the data rate by a factor of *k*.

A multi-ported RAM can help by allowing multiple accesses to the same location. However, most commercial FPGA devices only support up to two ports per internal RAM device. This means that if the number of hash functions is greater than two, the processing rate will decrease by a factor of *k*/2.

However, if the basic algorithm for the Bloom Filter is slightly modified, a speedup for filters with more than two hash functions can be obtained while trading a percentage of RAM resources. This speedup depends on two concepts – selecting perfect hash functions, and organizing the RAM in a different fashion.

A perfect hash function can be defined as a hash function gives a unique result for each member of the string set [9]. The perfect hash function makes it so that all true positives have a unique hash function result and a false positive has equal distribution over the result set. Therefore, if a result is positive, it can be assumed that if it is true, only one string could have created it.

Also, each hash function, unlike the original Bloom Filter, addresses a separate 1 bit table. When the Bloom Filter is modified in this way, the false positive probability is generally lower with the same parameters. However, this improved rate was achieved at the cost of increasing the width of the table by a factor of *k*.

$$P_{fp} = \left(\frac{n}{m}\right)^k$$

In comparison to the standard bloom filter, the false positive rate was traded for memory space complexity. If the modified filter is designed so that it gives the same false positive rate as a standard bloom filter, we can see that the memory has utilization has increased. However, the RAM is organized such that each hash function can access a separate bit in the width of the RAM. This means that the RAM can be organized as *k* single-ported memories or *k*/2 dual-ported memories, depending on the resources available in the FPGA. Now the lookup for all hash functions can happen within a single memory cycle, improving the throughput of the device.

For example, take the instance where a design calls for a Bloom Filter with 4 hash functions and RAM depth that is 8 times the number of strings. The yield is a Bloom Filter with a false positive rate of 2.4%. A modified filter with the same false positive rate can be designed with 4 hash functions and using 28% more RAM. However, the

FPGA implementation of the modified filter can run at twice the rate of the standard bloom filter because all of its lookups happen in parallel.

Table 1. Comparison of Bloom Filter and Modified Filter in performance and RAM Utilization.

		Bloom Filter		**Modified Filter**		**Comparison**	
Number of Hash Functions (k)	**False Positive Rate (P)**	**Ratio of Utilized Memory Depth (n/m)**	**Memory Usage (in bits)**	**Ratio of Utilized Memory Depth (n/m)**	**Memory Usage (in bits)**	**Parallel Speedup**	**RAM Utilization Increase**
2	4.89%	0.125	8n	0.22	9n	1x	12.5%
2	15.5%	0.25	4n	0.39	5.12n	1x	28%
2	40.0%	0.50	2n	0.63	3.18n	1x	59%
4	0.239%	0.0625	16n	0.22	18n	2x	12.5%
4	2.4%	0.125	8n	0.39	10.24n	2x	28%
4	16%	0.25	4n	0.63	6.36n	2x	59%
6	0.0025%	0.03125	32n	0.171	35.1n	3x	9.69%
6	0.0935%	0.0625	16n	0.313	19.2n	3x	20%
6	2.16%	0.125	8n	0.528	11.36n	3x	42%

The String Engine

On a set of rules, with varying string lengths, a method must be developed to give reliable results regardless of the string length. The probabilities of false positive detection shown above make the assumption that each string run through the hash function has the same string length. To maintain false positive probabilities, a different filter must be used for each length string within the set.

There are three basic methods for the implementation of the string engine – a Bloom Filter, a Modified Filter, and a direct string lookup. Based on the system requirements, string engines in the same system may each use a different method. The selection process should begin by analyzing the number of signature lengths available in the rule set. A histogram of signature lengths is a useful tool in characterizing the optimal implementation.

For instance, the histogram for signature lengths of Snort rules related to FTP attacks is shown in Figure 3. This rule set contains 46 distinct signatures that need to be detected. It can be seen that the shortest signature has two characters, the longest signature has 22 characters, and 24 of the signatures have a length of four characters.

Since each signature length will be detected with a different engine, there will be nine engines to process the signatures characterized in the histogram. Shorter signature lengths with a small number of signatures should likely be implemented as a direct string lookup. Larger signature lengths and signature lengths that have a high number of signatures should be considered for either the Bloom Filter or Modified Filter.

The determination of which string filter to select will depend on the desired false positive rate for the system. If the false positive rate drives a design that has a large number of hash functions, a Modified Filter is probably preferred over the Bloom Filter and visa versa.

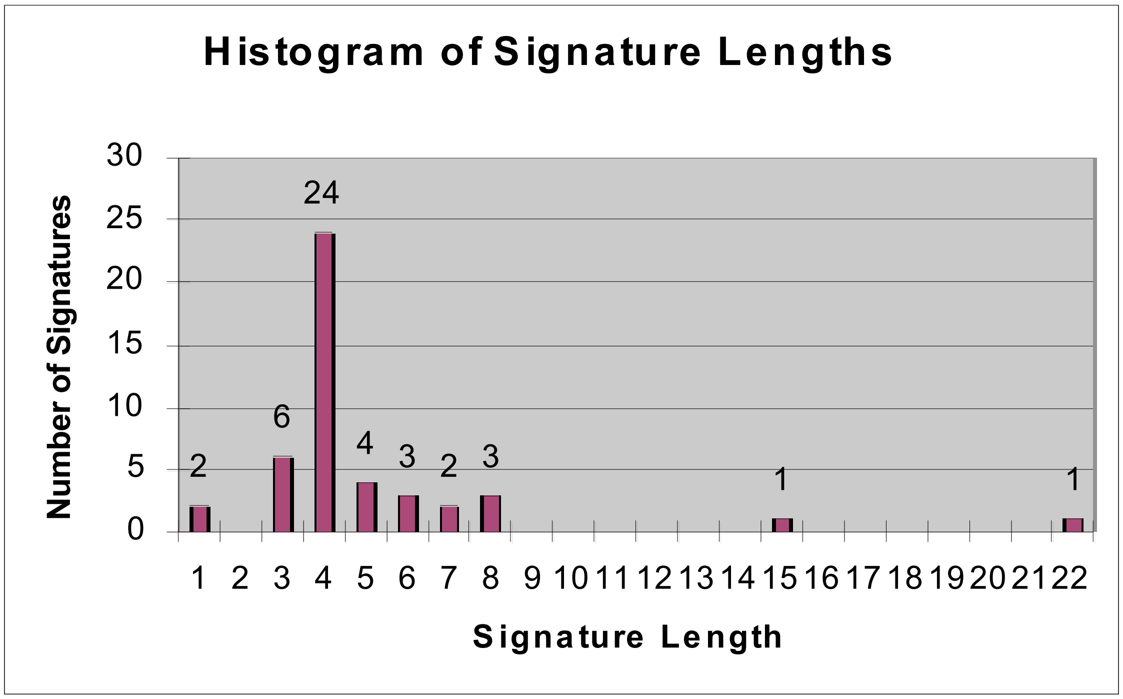

Figure 3. Histogram of Signature Lengths of Snort Rules Related to FTP.

Hardware Design

After analyzing the impact of the modified string filter, a design that implements the modified filter on an FPGA was created. The FPGA board used was an Altera UP3 Educational Board with an Altera EP1C6Q240 FPGA. Unfortunately, the board provided does not have an Ethernet interface, so data is moved to and from the board using the RS-232 serial port.

Since the serial port operates much slower than an Ethernet connection (10baseT, 100baseT, etc…) the packet data is staged into the packet parsing hardware in two steps. First the data is transferred into SRAM adjoining the FPGA at the serial rate. Next the data is processed by packet processing at the processing rate of 48 MHz. The processing clock was selected because it is the fastest oscillator on the board and a RS-232 serial divides evenly into it.

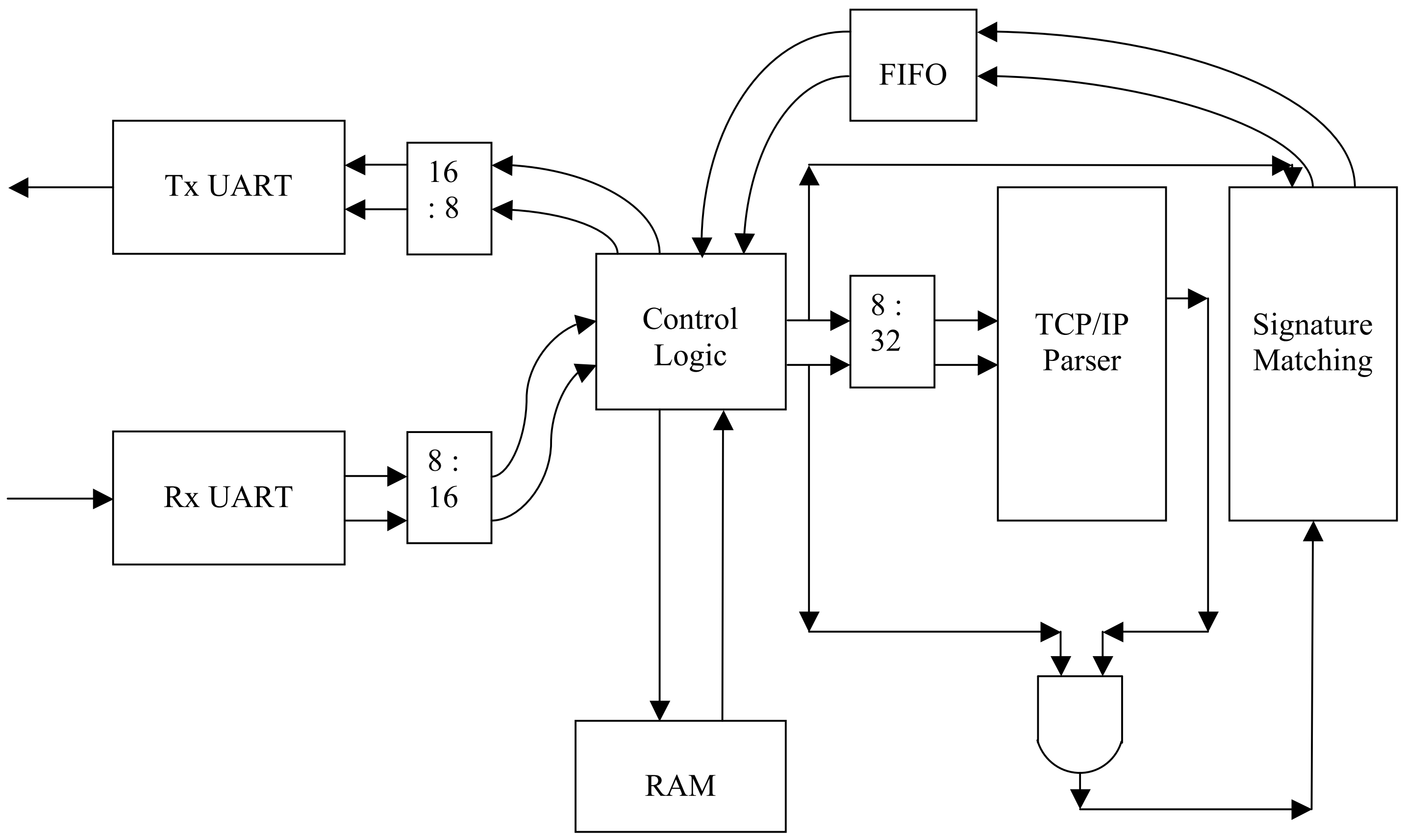

Figure 4. Basic block diagram flow of the IDS.

The intrusion detection system designed here works in two modes. Namely loading the RAM with packets and processing the packet for malicious content. In the write mode the command in given to the control logic to write the packet stream into the RAM. The UART takes input serial bit stream and converts to 8-bit data stream. An additional bit width converter is used to convert the data stream to 16-bit data as an input to the Control Logic. The control logic is configured on write to RAM mode, thus write the packets in to the RAM coming from the software. A valid bit is used for the pipelining of the data stream and stalls the logic if data is not valid on the data bus.

The software then issues a process command to the control logic. The control logic reconfigures itself to pickup the packets from the RAM and convert into a data stream for the packet processing and checking of snort rule set.

The data stream and the valid bit from the control logic are sent to the header stripper and payload rule checker for processing on the packet. The data stream of 8-bit is sent to the payload checker and a converted 32-bit stream to the header stripper. A 32- bit stream for the header is used because the header protocols are 32-bit spanned. The header parser strips of the header from the packet acquiring the required information required for the rule processor and generate a valid bit for the payload parser. This valid bit indicates that the current data stream coming out from the control logic is the payload. Thus the valid bit from the control logic and header parser is ANDed together to produce the valid bit for the payload rule checker.

The payload rule checker upon the activation of the valid bit start to receive the payload stream and bit shifts it and passes through bloom filter for snort ftp rule signature matching. If a signature is found it send a signature ID to the control logic. The control logic redirects this information to the software via UART. The FIFO is used to buffer the data for the UART, as the FPGA operates on 48 MHz while UART operates on 38.4 KHz.

This design mainly does two things. First, the header generates a packet identification number so as to know which packet, the FPGA is currently processing. Secondly, the payload is matched for a signature. This signature matching logic unit generates a signature identification number. The header packet number and the signature packet number are sent to the rule processing, which in our design is the software. The software refigures the rule type from the information received from the FPGA. The software for demonstration purpose prints the packet number and list of signatures found in the payload.

On top of the necessary processing blocks, the following interfacing modules were implemented in order to accomplish the final design:

- SRAM interface
- Serial UART
- Serial Command Processor

TCP/IP Parser

The incoming packets are screened from the TCP/IP parser to extract the information required by the Signature matching processor and Rule processor using a state machine.

The input for the state machine is 32-bits and every clock the state machine changes it state according to the state diagram. Every clock input the next 32-bit input is passed into state machine. The 32-bits are used for this design as the header protocols are 32-bit spanned and the design utilizes this concept efficiently.

The state machine is designed as follows:

1. Check for IP version 4. If the version is incorrect discard the 32-bits and start-over. If the version is correct, Extract the IP header length and total packet length, go to next state (i.e. State 2).
2. Extract the ID (Used to identify the fragments of one datagram from those of another). Go to next state (i.e. State 3).
3. Check for TCP protocol. If not found then skip the 32-bits and go to next state which is State I else go to next state (i.e. State 4).
4. Skip 32-bits. Go to next state (i.e. State 5).
5. Store the Destination IP Address. Go to next state (i.e. State 6).
6. Skip the padding bits (Used as a filter to guarantee that the data starts on a 32 bit boundary.) for the IP header. Go to next state (i.e. State 7).
7. Extract and store the TCP source and Destination Port numbers. Go to next state (i.e. State 8).

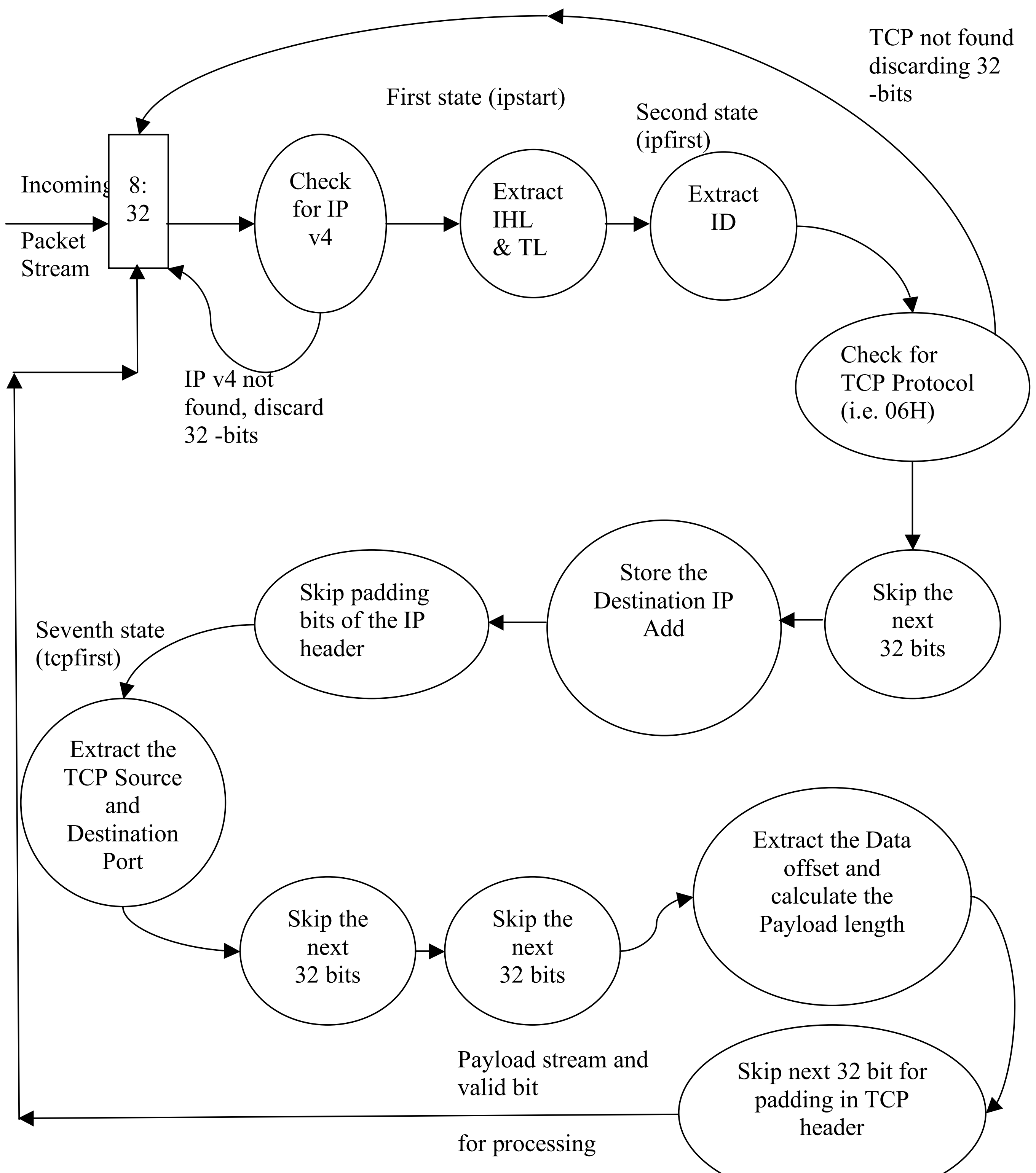
TCP not found discarding 32 -bits
First state (ipstart)
Second state (ipfirst)
Incoming Packet Stream
8: 32
Check for IP v4
Extract IHL & TL
Extract ID
IP v4 not found, discard 32 -bits
Check for TCP Protocol (i.e. 06H)
Skip the next 32 bits
Store the Destination IP Add
Skip padding bits of the IP header
Seventh state (tcpfirst)
Extract the TCP Source and Destination Port
Skip the next 32 bits
Skip the next 32 bits
Extract the Data offset and calculate the Payload length
Payload stream and valid bit
for processing
Skip next 32 bit for padding in TCP header

Figure 5. Histogram of Signature Lengths of Snort Rules Related to FTP.

8. Skip 32-bits. Go to next state (i.e. State 9).
9. Skip 32-bits. Go to next state (i.e. State 10).

10. Extract the Data offset parameter. Calculate the Payload length which is nothing but the Total length –IP header length – TCP header length. Go to next state (i.e. State 11).
11. Skip the next 32-bits for the TCP header padding and go to next state (i.e. State 12).
12. Send a 32- bit stream and the valid bit (indicating valid payload on line). Go to next State (i.e. State I).

Signature Matching

The Signature Matching block scans the packet payload for signature strings. This block contains all of the Bloom Filters, Modified Filters and string matchers necessary to support the string matching for the entire design. For the proof of concept on the Modified Filter, this block will contain only a Modified Filter for signatures with a length of four. It is implemented with four hash functions.

The Modified filter has three key parts – (1) a shift register, (2) hash functions, and (3) lookup tables. The shift register takes the input payload bytes as an input. Each byte is shifted in until the signature length of the filter is in the shift register (in this case, 32 bits).

The 32 bit result of the shift register is sent to each of the four hash functions. The hash function codes the 32-bit input in four different ways. Each of those hash function outputs are then used as lookup addresses in the four lookup tables. The results of the lookup tables are logically "ANDed" together to get the final string matching result.

The hash function selected is a series of shifts of the hash code added to each other with modulo applied in the end. The following function describes the selected function, where *h* is the hash code and *i* is the input 32-bit word.

$$f = ((h << 5) + (h >> 2) + i)\%255$$

Using the equations from the background section, and assuming n=24, m=255 and k=4, we see that the false positive is 0.00785%.

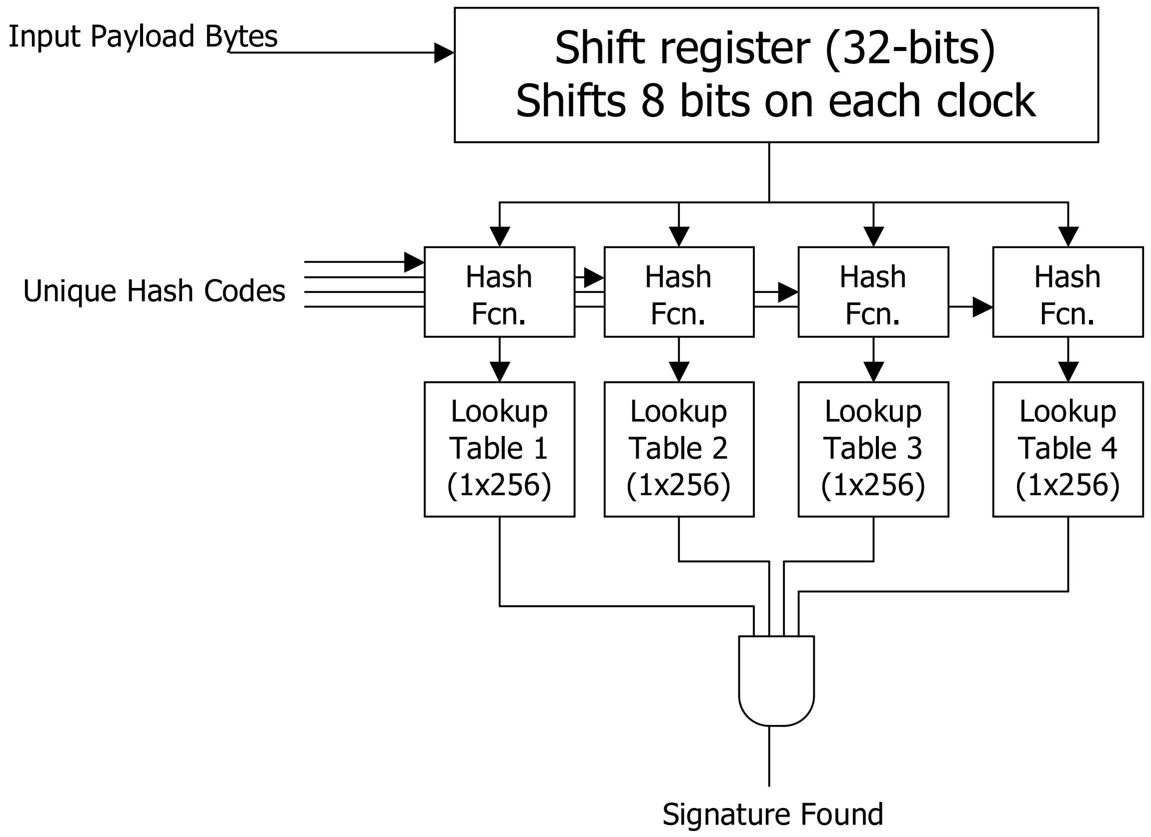

Figure 6. Block Diagram of the Modified Bloom Filter.

Control Processor

The Control Processor accepts commands from the serial software and acts upon them. As mentioned earlier, the Control Processor accepts four commands:

- **Burst Memory Write** – This command writes a burst of 16-bit values directly to the SRAM at a given location.
- **Burst Memory Read** – This commands the FPGA to read a specified number of 16-bit values from the SRAM starting from a specified location.
- **Set Memory Mode** – This command tells the FPGA to switch the data path so that the SRAM can be read and written from the serial port.
- **Set Processing Mode** - This command tells the FPGA to switch the data path such that data from the SRAM can be sent to packet processing on the FPGA.

The format of these commands are detailed in the following table. Length refers to length in 16-bit words. Base address is the SRAM address to begin reading or writing.

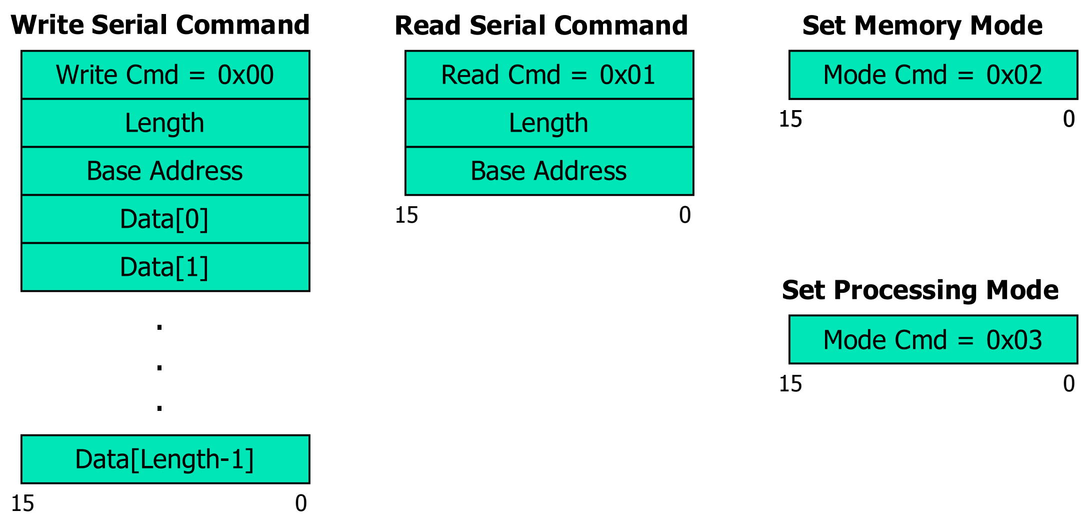

Figure 7. Serial Command Message Formats.

UART

The RS-232 link on the FPGA card was selected as the means to communicate with the FPGA. It is used to send commands and packet data to the FPGA and to receive results from the FPGA. To facilitate this capability, a simple UART (Universal Asynchronous Receiver-Transmitter) was implemented on the FPGA.

The serial link was designed to operate only at 38.4kbps. This was selected because it is evenly divisible from an oscillator frequency on the board (48MHz). The UART was designed to use the "standard" RS-232 protocol which uses one start bit, one stop bit, and eight data bits per byte transfer. There is no parity bit and no handshaking for the link.

When receiving data, the UART works by sampling the incoming data at sixteen times the bit rate, or 614.4kHz. The start bit is always a zero, so the UART searches for the receive serial line to be low for eight consecutive clocks. It assumes that after 8 clocks, it has roughly detected the center of the start bit. From that point, the UART counts another sixteen clocks and samples the line, assuming that this is the center of the first data bit. This process is continued for the other seven data bits and for the stop bit. If the stop bit is detected as a high logic level, the UART assumes the transmission was successful.

When transmitting data, the byte intended for transmission is placed in a shift register. The start bit is put on the TX line for ~26 us (one bit period at 38.4kHz) and then each successive bit of the data is shifted onto the TX line for the same amount of time. After the last bit, the stop bit is transmitted and held until the next word is ready to be transmitted.

The UART was not designed with error detection, error correction, or retransmit capabilities, so incorrect transmissions are unrecoverable in this implementation. It is also assumed that this interface was implemented solely for demonstration purposes and would not be used in an actual intrusion detection system that might use an Ethernet interface to accomplish the same goal.

SRAM Interface

The SRAM was selected as the means for buffering packets for processing. Packets are transmitted over the RS-232 serial link and written to the memory. At a later time, the memory is commanded to burst out the packet information into the packet processing elements within the FPGA. The SRAM on our target board is 16 bits wide and 64k deep.

Because of this, the SRAM Interface needs to support a continuous bursting capability. For this reason the SRAM interface was designed to allow continuous reads or writes to occur on every clock cycle.

In an actual intrusion detection system, this interface would likely be replaced with an SDRAM interface, that connects to a RAM capable of storing many megabytes or even gigabytes of data.

FIFO

A FIFO was used to buffer the results from the processing, which operates at a 48MHz rate, so that they could be read by the UART at the 38.4kbps rate. Initially the FIFO was selected to be 32 bits wide – the width of the output messages – and 128 words deep. This fits neatly into a single block RAM within the Altera device. The FIFO was implemented as a synchronous Altera macro function FIFO, where the read size was only enabled once every 12500 clocks to provide the 3.84kHz word rate needed for a serial link with eight data bits, a start, and a stop bit.

Data Width Conversion

At various points throughout the FPGA, data widths needed to be converted. For example, the output of the SRAM Interface is 16 bits wide, but the header processing operates on 32 bit wide data. Also, the internal UART interfaces provide a byte interface which needs to be converted to 16 bits to interface with SRAM and needs to be converted from 32 bits to interface with the output results from the header and signature processing. This is accomplished using shift registers. Each data width converter is pipelined and enabled with a data valid signal.

Software Design

Hash Code Generation

A program was written to generate the hash codes that would be used within the Modified Bloom Filter. It reads in the signature set from a file and stores it internally. Next, it randomly generates a hash code and applies the hash function using that code to the signature set. If each signature comes out to a unique result, then the hash code is stored and the whole process is repeated until a unique hash code has been generated for each of the *k* hash functions. Finally, the program generates the coefficients for the memory within the Modified Bloom Filter and a text file that indexes the signatures with their hash codes. The flowchart for the operation of the hash code generation software is found in Figure x.

The memory coefficients will be hard coded into the verilog that is compiled for the string engine on the FPGA. The signature index file is read by the serial control software for the FPGA so that it can correlate the matched signature hash results with an actual signature.

The program was written in C++ using the Standard Template Library (STL). It was targeted for Linux and compiled with g++ 3.4.4 under Cygwin on the Intel Pentium 4 platform. The code was written in a portable fashion and should compile and run in most environments with little or no modification.

Serial Control Program

The Serial Control Program runs on an Intel Pentium-based laptop and communicates serially with the FPGA board. At startup it initializes the serial port on the PC and loads the signature index file created by the Hash Code Generation program, and the packet file to be processed by the FPGA.

The program is capable of sending four commands to the FPGA card.

- **Burst Memory Write** – This command writes a burst of 16-bit values directly to the SRAM at a given location.
- **Burst Memory Read** – This commands the FPGA to read a specified number of 16-bit values from the SRAM starting from a specified location.
- **Set Memory Mode** – This command tells the FPGA to switch the data path so that the SRAM can be read and written from the serial port.
- **Set Processing Mode** - This command tells the FPGA to switch the data path such that data from the SRAM can be sent to packet processing on the FPGA.

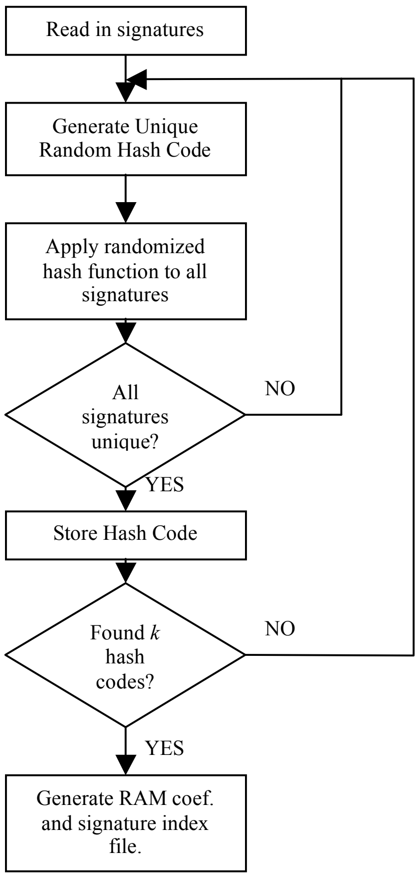
Read in signatures
Generate Unique Random Hash Code
Apply randomized hash function to all signatures
All signatures unique?
NO
YES
Store Hash Code
Found k hash codes?
NO
YES
Generate RAM coef. and signature index file.

Figure 8. Hash Code Generation flow chart.

The program begins operation by putting the FPGA in Memory Mode (Set Memory Mode). Next it burst writes the packet information to the SRAM (Burst Memory Write). Then it reads back the data to verify the contents (Burst Memory Read). At this point packets of data are sitting in the SRAM and ready to be processed by the packet processing functions.

Next the FPGA is placed in operational mode (Set Processing Mode) and the data is read from the SRAM into the packet processing functions (Burst Memory Read). Serial results of the header and string processing are received by this program and the results are decoded and displayed to the screen.

Testing

Testing of the algorithm was performed on an Altera University Education Kit that contains an Altera Cyclone FPGA, SRAM, a serial interface, and an onboard oscillator. It was programmed by a laptop over a Parallel cable connection using the

JTAG protocol. The FPGA card was commanded using the same laptop over a serial RS-232 connection . The laptop ran the Hash Code Generation, Serial Command, and Altera Quartus II FPGA programming environment software.

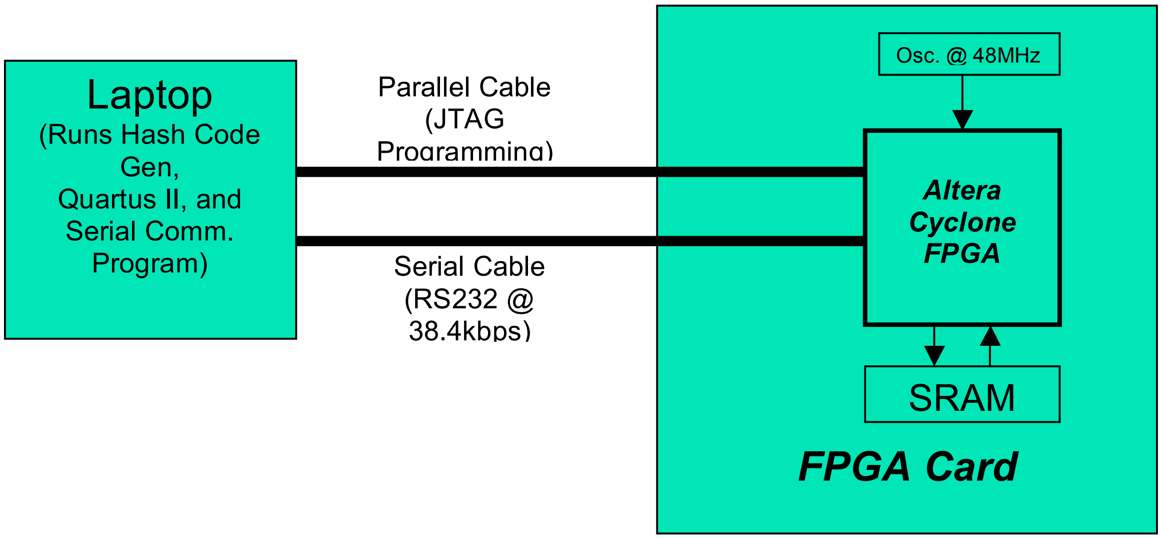

Figure 9. Test Setup Block Diagram.

The FPGA design can be broken down into two processes – Compile-Time Operations and Run-Time Operations. The compile time operations include analyzing the Snort ruleset, generating the hash coefficients for the Bloom Filter, and synthesizing and loading the Verilog design onto FPGA Altera QuartusII board.

Run-Time Operations involve performing operations on the loaded FPGA. The operations work in two separate modes – memory mode and processing mode. As there is no MAC port on the board the design creates a RAM for storing the Packet. In processing mode the hardware is enables to recall the packet from the memory and process the packet for malicious signatures.

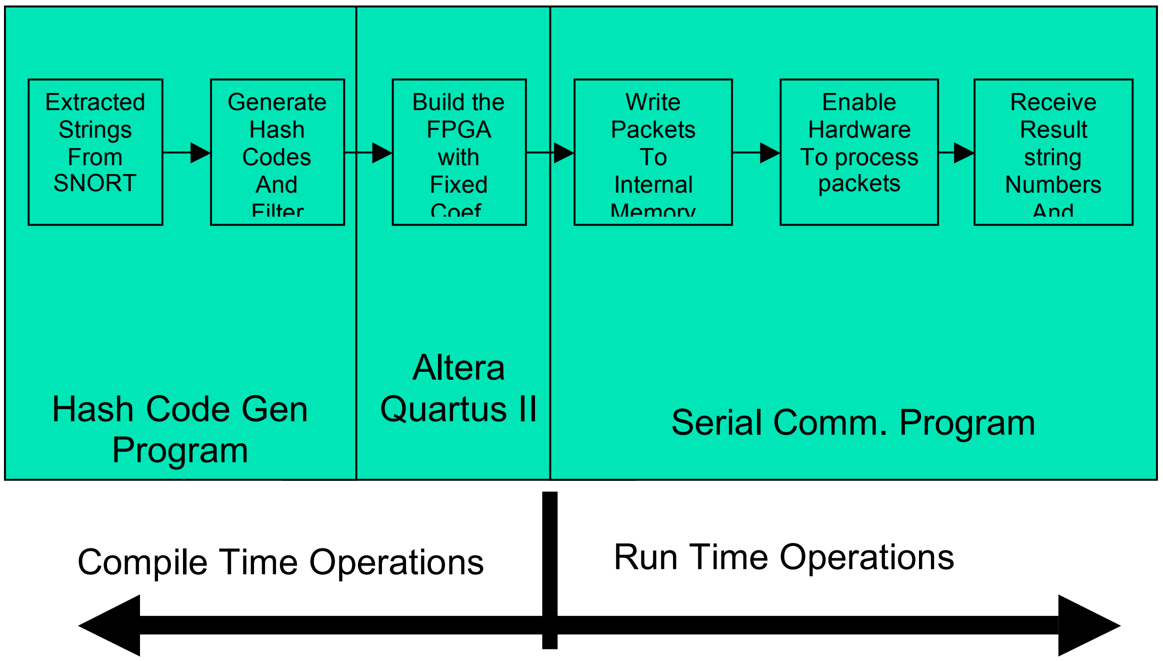
Extracted Strings From SNORT
Generate Hash Codes And Filter
Build the FPGA with Fixed Coef
Write Packets To Internal Memory
Enable Hardware To process packets
Receive Result string Numbers And
Hash Code Gen Program
Altera Quartus II
Serial Comm. Program
Compile Time Operations
Run Time Operations

Figure 10. Compile-Time and Run-Time Operations.

To appropriately test the run-time operation of the intrusion detection system on the FPGA card, a series of test packets were generated that give some basic confidence level in the operation of the algorithm. The packets generated for the purpose of the demonstration were:

- That contains all the signatures.
- Those contain NONE of the signatures (random data).
- Those contain a combination (some signatures, some random data).
- Those contain a signature byte shifted by different amounts (not always 32-bit word aligned).

These types of packet were chosen to fully demonstrate the capabilities of the NIDS. The first type of packet was aimed at displaying any number of signatures can be detected by the signature matching processor. The second one was used during testing for recognizing false positives in the system, thus modifying the number of hash function to reduce the false positives. The third packet was designed to demonstrate that the signature can be placed in between the words or other strings and still the NIDS can find the string. As our system takes advantage of the header rule of placing the bits in 32-bit spacing, fourth packet was generated to defy that rule and check the system to work correctly.

Results

The operational throughput of the SRAM interface is one 16-bit word can be read or written in each clock cycle, the header processing can process one 32-bit word on each clock cycle, and the signature processing can process one 8-bit word on each clock cycle. Clearly, the signature processing, in its current implementation, is the bottleneck of the processing portion of the design.

Table 2. Calculated Throughput of the Individual FPGA Modules.

Module	***Throughput***
SRAM Interface	768 Mbps
Header Processing	1536 Mbps
Signature Processing	384 Mbps

At these rates, the aggregate processing rate is the same as the lowest throughput of all the modules, or 384 Mbps. This means that the processing can keep up with over 3 100baseT lines. If the design was targeted for a different board and FPGA part, it could possibly be run at a higher clock speed. If the design is run at 200MHz, it can keep up with a line rate of 1.6Gbps.

Improvements

- Including all the Snort Rule set and demonstrating its full capability on an actual network.
- Variable length signature match
    - Implementing this part is little tricky but can be done by using multiple Bloom Filters.
    - But this approach utilizes lot of system resources in the process.
    - A modified data structure or design of bloom filter is required for variable length string match.
- Efficient hash function for FPGA
    - The hash function selected in the bloom filter takes many clock cycles.
    - A hash function which is does not use multiplier or divider for finding a unique hash code, is to be selected.
    - This modification will aid in finding a match faster and thus will increase the performance of the system.
- String that span multiple packets
    - Need filtering of header and thus getting the identification number and sequence number of each packet.
    - Need an algorithm to manage state of multiple connections.

    - A state RAM is required to swap the content of Bloom Filter and place the packet payload according to the sequence number extracted from the header.
- Header rule filter
    - Currently header rule filter scans for only ftp snort rule set.
    - The state machine needs to be modified to include every snort rule set.
    - Need Bloom Filter to scan the IP add and Port numbers.
- Rule processor
    - The current design has the rule processing in the software. A logic unit needs to be designed for the rules processor in FPGA.
- Interfaces directly to an Ethernet MAC
    - Requires different hardware.
    - And accordingly the control logic can be reconfigured.
- Design may need to be optimized to run at faster speeds.

References

[1] M. Attig and J. Lockwood. *A Framework for Rule Processing in Reconfigurable Network Systems.* IEEE Symposium on Field-Programmable Custom Computing Machines, April 18-20, 2005.

[2] M. Attig and J. Lockwood. *SIFT: Snort Intrusion Filter for TCP*. IEEE Symposium on High Speed Interconnects, August 17-19, 2005, Stanford, CA.

[3] Z. K. Baker and V. K. Prassanna. *A Computationally Efficient Engine for Flexible Intrusion Detection.* IEEE transactions on very large scale integration (vlsi) systems, vol. 13, no. 10, October 2005.

[4] R. Bejtlich. *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison Wesley. 2006.

[5] B. Bloom. *Space/Time Trade-offs in Hash Coding with Allowable Errors.* Communications of the ACM, July, 1970, pp. 422-426.

[6] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood. *Deep Packet Inspection using Parallel Bloom Filters Bloom Filters : A Tutorial, Analysis, and Survey by James Blustein and Amal El-Maazawi*, Faculty of Computer Science, Dalhousie University, B3H 1W5, Canada.

[7] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. *Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)*. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, Feb. 2001.

[8] M. Mitzenmacher. *Compressed Bloom Filters.* IEEE/ACM transactions on networking, vol. 10, no. 5, October 2002.

[9] M.V. Ramakrishna. A *Simple Perfect Hashing Method for Static Sets*. Proceedings of the Fourth International Conference on Computing and Information, 1992, pp. 401-404.

[10] H. Song and J. Lockwood. *Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems*. IEEE Globecom 2005, St. Louis, MO, Nov. 28, 2005, pp. CN-02-3.