EE533 learning

Version 0.1

Date: May 14, 2023

Table of Contents

[Table of Figures 3](#_Toc134977863)

[Salient Features 4](#_Toc134977864)

[Limitations and Future Work 4](#_Toc134977865)

[Architecture 5](#_Toc134977866)

[Dual-core interface with Upstream/Downstream 6](#_Toc134977867)

[Upstream/Downstream interface 6](#_Toc134977868)

[Handshake between CPU (Core) and State Machine 7](#_Toc134977869)

[Dual-core interface with Accelerator 11](#_Toc134977870)

[Deconstructing the incoming packet into composable format 12](#_Toc134977871)

[Tool chain 15](#_Toc134977872)

[Hand-built Logic Analyzer 15](#_Toc134977873)

[Instructions for making Logic analyzer work 16](#_Toc134977874)

[Custom Compiler 17](#_Toc134977875)

[Subtle Bugs discovered along the way 17](#_Toc134977876)

[SNORT 18](#_Toc134977877)

[Rules tree 18](#_Toc134977878)

[Configuration options 18](#_Toc134977879)

[Configuring SNORT 1st time (after every login) 20](#_Toc134977880)

[Measuring performance (bandwidth) 21](#_Toc134977881)

[Tools and Skillset Summary 22](#_Toc134977882)

[Appendix 22](#_Toc134977883)

[Appendix A: Supported Instruction Set 22](#_Toc134977884)

[Appendix B: Offset Calculations for scriptimp.sh 22](#_Toc134977885)

[Glimpse of Logic Analyzer Output 23](#_Toc134977886)

[Appendix C: Width of Logic Analyzer 24](#_Toc134977887)

[Appendix D: Cost Advantage of FPGA-based solution 24](#_Toc134977888)

[Appendix E: Assembly programming for traversing packets 24](#_Toc134977889)

[Understanding protocols from point of view of manipulation 26](#_Toc134977890)

[Appendix F: First and Last Word (64 bits width) of a new packet 27](#_Toc134977891)

[Appendix G: Useful files generated in the process of bitfile 29](#_Toc134977892)

[Directory Structure (git) 30](#_Toc134977893)

[Knowledge Bank 30](#_Toc134977894)

[SNORT Rule option: flow explanation 30](#_Toc134977895)

[SNORT manual 32](#_Toc134977896)

[CIDR 32](#_Toc134977897)

[TCP indicating end of current conversation 33](#_Toc134977898)

[References 36](#_Toc134977899)

# Table of Figures

[Figure 1 Custom Design placement within the NetFPGA 5](#_Toc134977900)

[Figure 2 overall architecture of dual core and accelerator 6](#_Toc134977901)

[Figure 3 handshake of CPU with State Machine 7](#_Toc134977902)

[Figure 4 state diagram for handshake between upstream/downstream and cpu 8](#_Toc134977903)

[Figure 5 subtle bug in the design 9](#_Toc134977904)

[Figure 6 subtle bug in design continued 10](#_Toc134977905)

[Figure 7 Dual Core arbitration with NetFPGA enabled by Smart Arbiter 11](#_Toc134977906)

[Figure 8 Dual core handshake with Accelerator 12](#_Toc134977907)

[Figure 9 Depicting that header transformed into indexes and payload are merged together 13](#_Toc134977908)

[Figure 10 stride of 1 to move from 7-byte payload pattern to another 13](#_Toc134977909)

[Figure 11 continued till the last 7-byte1 payload 14](#_Toc134977910)

[Figure 12 header transformed into pseudo-index and payload fed together to bloom filter 15](#_Toc134977911)

[Figure 13 elaborate output from compiler 17](#_Toc134977912)

[Figure 14 subtle bug in compiler 17](#_Toc134977913)

[Figure 15 SNORT rules high-level options 18](#_Toc134977914)

[Figure 16 logic analyzer output segregated into various fields 23](#_Toc134977915)

[Figure 17 description of src and dest IP offsets as observed in logic analyzer output 25](#_Toc134977916)

[Figure 18 use of online wireshark equivalent to analyze the packet and write appropriate assembly program 25](#_Toc134977917)

[Figure 19 use of online wireshark equivalent to analyze the packet and write appropriate assembly program 26](#_Toc134977918)

[Figure 20 First 64-bit Word description of NetFPGA 27](#_Toc134977919)

[Figure 21 First 64-bit words after passing through the Output Port Lookup 28](#_Toc134977920)

[Figure 22 example of the frist word (64-bits) decoded 28](#_Toc134977921)

[Figure 23 example continued 29](#_Toc134977922)

# Salient Features

1. Deep packet Inspection (Payload as well)
2. Scalable to 11,500 rules (there are 11,500 rules out of 45k rules that have single content field[[1]](#footnote-1))
3. We selected 7 bytes as the pattern for matching as our design has some basic limitations like, an instance (verilog instantiation) of hash function took a clock cycle to calculate the hash (for a N-byte input, and N in our case was chosen to be 7). Therefore, in order to speed-up calculations of hashes, we had to employ 10 different hash functions. (We decided to have only packets that have 16 bytes of payload[[2]](#footnote-2), since, pattern could be present anywhere in these 16 bytes, therefore we needed 10 [16-7+1] different instances of set of hash function and an instance of same bloom filter.) By set of hash function we mean that for instance in order to achieve 11% false positive rate we had to use 10 hash functions,) thus, for 16 byte payload with 7-byte pattern, we ended up having 10x10 hash functions and 10 bloom filters to make the entire logic work in 1 click cycle

# Limitations and Future Work

1. Throughput: A core after accepting the data cannot accept a new packet, if the accelerator for instance is busy. Only after accelerator has worked-upon the packet, can the core go to READ\_READY state and read/send the data downstream and then accept new packet
2. Hash Function
   1. With a stride of 1 i.e. in a 16-byte payload with pattern matching of 7 bytes, and pattern may be present at any offset (for instance at locations Byte0~6 or Bytes 3~9) it would worst-case take N-M+1 clock cycles to traverse the packet, calculate hash values and matching with bloom output. (where N is the payload size, M is the pattern length)
   2. Currently, if we have to extend the capability to variable length patterns, then we need to have customized/appropriate hash functions specific to each unique length and appropriate bloom filter

# Architecture

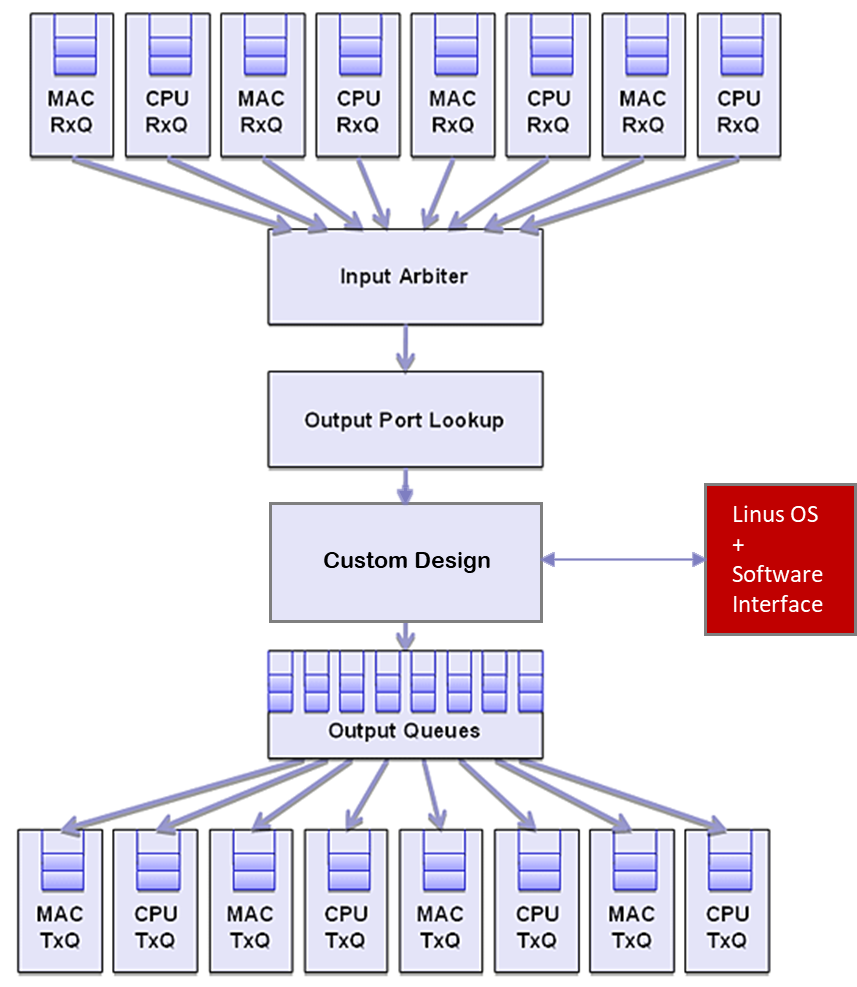


Figure 1 Custom Design placement within the NetFPGA

## Dual-core interface with Upstream/Downstream

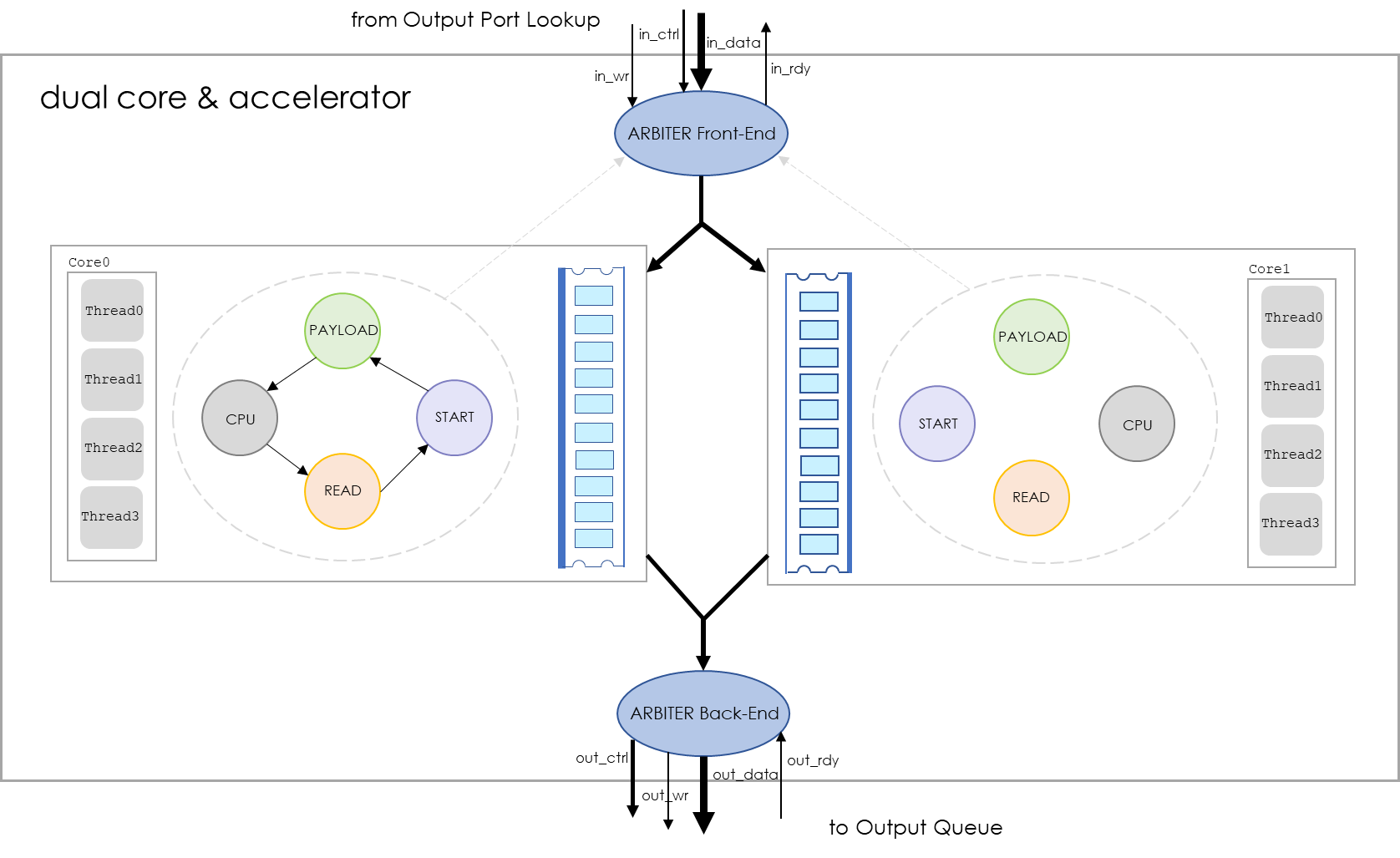


Figure 2 overall architecture of dual core and accelerator

### Upstream/Downstream interface

**// Upstream**

* **in\_rdy**: Output from ids module to the upstream module (output\_lookup\_queue) indicating that the ids is ready to accept inputs
* **in\_wr**: Signal from output\_lookup\_queue (upstream module) to the ids that upstream module wants to write to the downstream module
* Both signals should be HIGH to ensure handshake

**// Downstream**

* **out\_rdy**: Input Signal from output\_queue to ids that output\_queue is ready to accept the inputs
* **out\_wr**: Output from ids to output\_queue (downstream module) that ids is writing the valid data from out\_data and out\_ctrl
* Both signals should be HIGH to ensure handshake

Note: for our processor we continue to use the top Verilog module name as ids

### Handshake between CPU (Core) and State Machine

Through use of special registers which to CPU appear as just another locations in memory, we executed assembly instructions that in turn interacted with the outside world (aka toy implementation of Memory-mapped I/O)

A diagram of a state machine

Description automatically generated with medium confidence

Figure 3 handshake of CPU with State Machine

A picture containing text, diagram, font, handwriting

Description automatically generated

Figure 4 state diagram for handshake between upstream/downstream and cpu

#### Description of State Machine

1. **START**: Wait for a new packet, as soon as new packet comes (in\_ctrl != 0,) move to the **HEADER** State followed by the **PAYLOAD** State
2. **PAYLOAD**: keep writing to the FIFO till the time end of packet doesn’t comes (Indicated by in\_ctrl != 0.)
   * Move to the **CPU\_PROCESS** state when in\_ctrl != 0, and write the current state (CPU State, 3’b111) to a special memory location [’h200 = ’d512], indicating to the CPU to start processing the packet
3. **CPU\_PROCESS**: CPU from the beginning continues to do load from this special memory location and checks whether the state is changed to CPU State (3’b111,) if yes, then start the processing
   * Read the **readptr** (stored at a special location [16’h400 = 16’d1024]), add the **offset** (5 discovered during packet inspection, hard-bound in the assembly program) and load the lower 16 bits of the target IP address, add one to it and store it back
   * Indicate to the State Machine that CPU has completed its job by writing to a special memory location ([’h100 = ’d256])
   * State machine as soon as sees this flag as HIGH, moves to the READ\_READY state
4. **READ\_READY**: Reading commences and once complete go back to START State

Subtle bug in the design

A screenshot of a computer program

Description automatically generated with low confidence

Figure 5 subtle bug in the design

A screenshot of a computer program

Description automatically generated with low confidence

Figure 6 subtle bug in design continued

A picture containing text, diagram, plan, parallel

Description automatically generated

Figure 7 Dual Core arbitration with NetFPGA enabled by Smart Arbiter

## Dual-core interface with Accelerator

Modus-operandi is that CPU (after moving to CPU State) would first check for protocol field (at offset 3 and the last byte) and if the rules list has any rules for this protocol, the continue to extract the other 4 fields (Source IP, Destination IP, Source Port and Destination Port,) else indicate to the state machine to change the READ READY state.

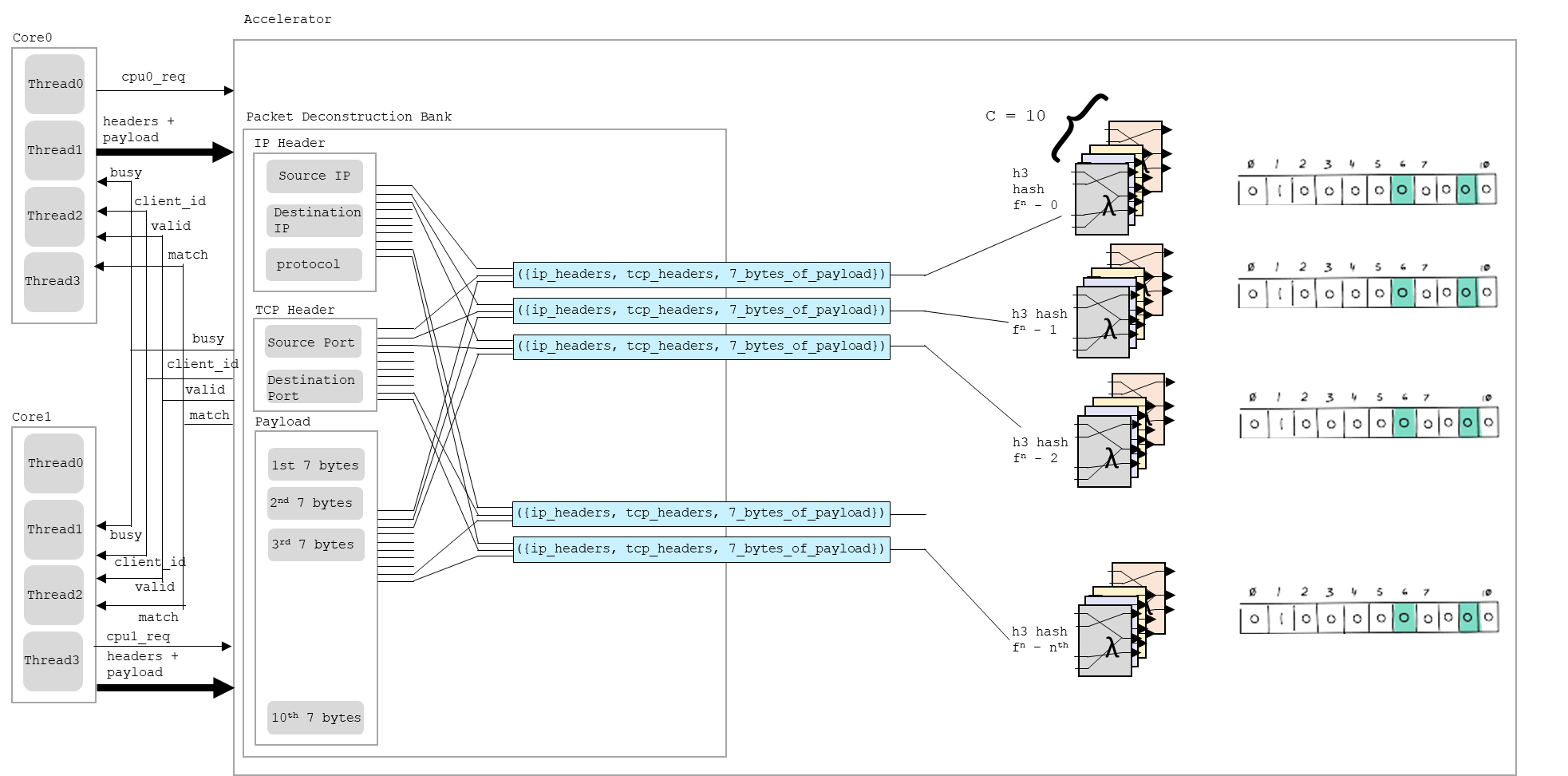


Figure 8 Dual core handshake with Accelerator

### Deconstructing the incoming packet into composable format

Why we input transformed-header and payload into the hash function? Specifically why transformed header?

Because SNORT rules are not payload specific and have conditions like if the Source IP is $HOME\_NET and if the Source Port is $HTTP\_PORTS, then check for this specific content, therefore it is essential that match be found such that it satisfied all the conditions.

Example rule

alert **tcp** **$EXTERNAL\_NET** **$HTTP\_PORTS** -> **$HOME\_NET any** (msg:"EXPLOIT-KIT Redkit exploit kit landing page Received - applet and flowbit"; flow:to\_client,established; flowbits:isset,kit.redkit; file\_data; content:"**<applet**"; metadata:policy balanced-ips alert, policy max-detect-ips drop, policy security-ips alert, service http; reference:cve,2012-0188; reference:cve,2012-0507; reference:cve,2012-4681; reference:cve,2013-2423; reference:url,blog.spiderlabs.com/2012/05/a-wild-exploit-kit-appears.html; classtype:trojan-activity; sid:23225; rev:7;)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 9 Depicting that header transformed into indexes and payload are merged together

A screenshot of a computer

Description automatically generated with medium confidence

Figure 10 stride of 1 to move from 7-byte payload[[3]](#footnote-3) pattern to another

A screenshot of a computer

Description automatically generated with medium confidence

Figure 11 continued till the last 7-byte1 payload

A screenshot of a diagram

Description automatically generated with low confidence

Figure 12 header transformed into pseudo-index and payload fed together to bloom filter

# Tool chain

## Hand-built Logic Analyzer

* Principle
  + Records the data written to the fifo/sram at the same time that data memory is written into (using same write enable)
* Implementation
  + internal write pointer is **reset** using software register. reset acknowledgement is checked by reading upper 24 bits [95:72] and checking if they are 0xFACADE
  + Stars recording when user (through software registers) sends the Command
  + Record till internal write pointer (internal to logic thief) saturates (reaches 255 from 0) (logic thief block memory is 256 deep, 72 bits wide)
  + When write pointer saturates, the upper 24 bits are **set** to 0xDECADE indicating to the user (through hardware register) that logic thioef block ram is full
* Validation
  + Read the logic thief block memory using a bash script and store in a local file for further analysis

### Instructions for making Logic analyzer work

$> sudo cp ~/scriptimp.sh .

$> check for addresses generated in your fedora

$ vim ../lib/C/reg\_defines\_<lab\_name>\*.h

$> cd /usr/local/netfpga/lib/C/reg\_access

$> modify the addresses in the scriptimp.sh with above address

$> ./regread <software register> # to see if 0xFACADE is observed. if unsuccessful, then 0xDEADCAFE # is observed

$> ./regwrite <try various address of your software register as available in /root/netfpga/projects/lab10/include/reg\_defines\_lab10.h>

$> ./regread <software register> # to see if 0xFACADE is observed. if unsuccessful, then 0xDEACDE is observed

$> ./regwrite 0x2000304 0xDEADDEAD # resets the counter

$> ./regread 0x2000310 # should give 0xFACADE is reset was successful

$> ./regwrite 0x2000300 0xDEADCAFE # allows internal logic\_thief write pointer to increment

$> ./regread 0x2000310 # should read 0xFACADE

$> ping n3 (from n0)

$> ./regread 0x2000310 # should output 0xDECADE, indicating that counter is saturated

$> ./scriptimp.sh # to capture the data stored in logic thief block ram, testingreadfile.txt should get generated

## Custom Compiler

Glimpse of output of compiler elaborate file

A screenshot of a computer

Description automatically generated with medium confidence

Figure 13 elaborate output from compiler

### Subtle Bugs discovered along the way

RV64I has 6-bit shift value width (shaft) whereas RV32I has 5 bit shift value width (shaft)



Figure 14 subtle bug in compiler

## SNORT

### Rules tree

Figure 15 SNORT rules high-level options

### Configuration options

#### Configuring $HOME\_NET in SNORT

In Snort, `$HOME\_NET` is a variable that represents the IP addresses or networks that you consider to be part of your "home" or protected network. To designate an IP address or a range of IP addresses as `$HOME\_NET` in Snort, you need to modify the Snort configuration file (usually named `snort.conf`).

Follow these steps to designate an IP address or a range of IP addresses as `$HOME\_NET`:

1. Locate the Snort configuration file: Find the `snort.conf` file in your Snort installation directory. The exact location may vary depending on your system and installation method. Common locations include `/etc/snort/snort.conf` on Linux systems or `C:\Snort\etc\snort.conf` on Windows systems.

2. Open the configuration file: Open the `snort.conf` file using a text editor, such as Notepad, Vim, or Nano.

3. Find the `$HOME\_NET` variable: Look for the line that defines the `$HOME\_NET` variable. It usually looks like this:

```

ipvar HOME\_NET any

```

By default, `$HOME\_NET` is set to `any`, which means that it includes all IP addresses.

4. Modify the `$HOME\_NET` variable: Replace the `any` keyword with the IP address, network range, or CIDR notation that you want to designate as your home network. For example:

- To designate a single IP address (e.g., 192.168.1.1) as `$HOME\_NET`:

```

ipvar HOME\_NET 192.168.1.1

```

- To designate a network range (e.g., 192.168.1.0 to 192.168.1.255) as `$HOME\_NET`:

```

ipvar HOME\_NET 192.168.1.0/24

```

- To designate multiple networks or IP addresses as `$HOME\_NET`, separate them with commas:

```

ipvar HOME\_NET [192.168.1.0/24,10.0.0.0/8,172.16.0.0/12]

```

5. Save and close the configuration file: After modifying the `$HOME\_NET` variable, save the changes to the `snort.conf` file and close the text editor.

6. Restart Snort: To apply the changes, restart the Snort service or process.

By following these steps, you can designate an IP address or a range of IP addresses as `$HOME\_NET` in Snort, allowing you to tailor your intrusion detection system to your specific network environment.

#### Configuring port numbers

You can define the $FILE\_DATA\_PORTS variable in your Snort configuration file (snort.conf) based on your specific needs. For example:

var FILE\_DATA\_PORTS [21,80,443,25,110,143]

This example includes the ports listed above, but you can customize the list as needed for your environment.

By default, $HTTP\_PORTS includes port 80, which is the standard port for HTTP. However, you can also add the default HTTPS port (port 443) to the $HTTP\_PORTS variable if you want to include both HTTP and HTTPS ports in your Snort rules.

To do this, you can modify the snort.conf configuration file and update the $HTTP\_PORTS variable as follows:

var HTTP\_PORTS [80,443]

This will include both port 80 and port 443 in the $HTTP\_PORTS variable, allowing you to use it for both HTTP and HTTPS traffic in your Snort rules.

### Configuring SNORT 1st time (after every login)

Install snort

$ yum install \*.rpm --nogpgcheck

View the current configuration file

$ sudo gedit /etc/snort/snort.conf

Copy the custom rule file to the $RULE\_PATH (for instance /etc/snort/rules/<file\_name>.rules)

$> sudo cp <source\_file\_path> /etc/snort/rules/

Check file type

$> file <file\_name>

.rules files are of type 'ASCII English txt, with very long lines, with CRLF line terminators' whereas files in windows are of type 'ASCII English txt, with very long lines, with CRLF line terminators'

Convert CRLF file to unix using

$> dos2unix <source\_file> <destination\_file>

Path to the rules

$ var RULE\_PATH /etc/snort/rules

$ include $RULE\_PATH/<rule\_file\_name>.rule

Check if configurations are proper

$ sudo snort -T -c /etc/snort/snort.conf -i <interface like eth4>

$ sudo snort -A console -u snort -g snort -c /etc/snort/snort.conf -i <interface like eth4>

## Measuring performance (bandwidth)

* Check Lab11 PDF for instructions, Specifically

$> ssh control.<EXP-NAME>.USCEE533.isi.deterlab.net

$> sudo /usr/local/netfpga/lib/scripts/cpci\_reprogram/cpci\_reprogram.pl –all

$> sudo /sbin/ifconfig nf2c3 10.1.0.2/24 up # repeat for other 3 ports

**following commands to download design and routing table:**

$> nf\_download <location of your bitfile>.bit

$> /usr/local/netfpga/projects/router\_kit/sw/rkd &

**Location from where bash script to be executed**

$> cd /usr/local/netfpga/lib/C/reg\_access

$> sudo killall rkd

**following commands on the server node:**

$> /usr/local/etc/emulab/emulab-iperf –s –p 3004 &

$> sudo tcpdump -i <interface> -w <location to save the .pcap file>

**following command on the client node:**

$> /usr/local/etc/emulab/emulab-iperf –c n1 –p 3004 –F message[[4]](#footnote-4)

## Tools and Skillset Summary

|  |  |  |
| --- | --- | --- |
| Task | Tools/Skillset | Logic |
| Logic Analyzer output extraction | Bash | Same interface to extract the data instead of transporting the data to user environment (windows) and running python |
| Compiler | Python | Ease of Use |
| Simulation and generating .xco files | Xilinx ISE 10.1 | Very old version (free) |
| Bitfile generation | Xilinx tool |  |
| Analyze Packet to find offsets for assembly programming | Wireshark equivalent |  |
| Assembly programming | Work with custom RISC-V inspired design |  |
| Design | Verilog |  |

# Appendix

## Appendix A: Supported Instruction Set



## Appendix B: Offset Calculations for scriptimp.sh

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| scriptimp.sh | | | | |
| Word Order | Bit Width Offset | Field Name | Actual Bit Width |  |
| word\_0 |  | wr\_data\_lo | 32 |  |
| word\_1 | 32 | wr\_data\_hi | 32 |  |
| word\_2 | [7:0] | wr\_data\_ctrl | 8 |  |
| word\_2 | [15:8] | wr\_addr | 8 |  |
| word\_2 | [16] | wr\_en | 1 |  |
| word\_2 | [31:17] | rd\_data\_lo\_lo | 15 |  |
| word\_3 | [16:0] | rd\_data\_lo\_up | 17 | 32 |
| word\_3 | [31:17] | rd\_data\_hi\_lo | 15 |  |
| word\_4 | [16:0] | rd\_data\_hi\_up | 17 | 32 |
| word\_4 | [24:17] | rd\_data\_ctrl | 8 |  |
| word\_4 | [31:25] | rd\_addr | 7 |  |
| word\_5 | [0] | rd\_addr | 1 |  |
| word\_5 | [8:1] | pc | 8 |  |
| word\_5 | [10:9] | state | 2 |  |
| word\_5 | [12:11] | mem\_thread\_id | 2 |  |
| word\_5 | [13:12] | current\_writer | 2 |  |
| word\_5 | [14:14] | change\_writer | 1 |  |
| word\_5 | [15:15] | fsmgenwren | 1 |  |
| word\_5 | [16:16] | ftsfrden | 1 |  |
| word\_5 | [17:17] | CORE0\_ftsfrden | 1 |  |
| word\_5 | [19:18] | current\_reader | 2 |  |
| word\_5 | [20:20] | change\_reader | 1 |  |
| word\_5 | [21:21] | outwr | 1 |  |

### Glimpse of Logic Analyzer Output

A screenshot of a computer

Description automatically generated

Figure 16 logic analyzer output segregated into various fields

## Appendix C: Width of Logic Analyzer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Width of Logic Analyzer Memory | | | | |
|  | CORE1 | CORE0 | COMMON |  |
| OUT\_WR |  |  | 1 | 1 |
| CHANGE\_READER |  |  | 1 | 1 |
| CURRENT\_READER |  |  | 2 | 2 |
| FTSFRDEN | 1 | 1 | 1 | 3 |
| FSMGENWREN |  |  | 1 | 1 |
| CHANGE\_WRITER |  |  | 1 | 1 |
| CURRENT\_WRITER |  |  | 2 | 2 |
| THREAD\_ID | 2 | 2 |  | 4 |
| STATE | 2 | 2 |  | 4 |
| PC | 8 | 8 |  | 16 |
| RDADDR | 8 | 8 |  | 16 |
| DOUT | 72 | 72 |  | 144 |
| WREN | 1 | 1 |  | 2 |
| WRADDR | 8 | 8 |  | 16 |
| DIN | 72 | 72 |  | 144 |
|  |  |  |  | 183 |

## Appendix D: Cost Advantage of FPGA-based solution

|  |  |  |  |
| --- | --- | --- | --- |
| Field | NetFPGA-SUME | NetFPGA-1G | SNORT-based Firepower\* |
| IPS Bandwidth (Mbps) | 22600 | 2260 | 21000 |
| Device Cost (USD) | 6995 | 1599 | 30000 |
| Mbps/Dollar | 3.2 | 1.4 | 0.7 |
|  | | | |
| Cisco Firepower 3120 claimed IPS bandwidth is 21Gbps[[5]](#footnote-5) | | | |

## Appendix E: Assembly programming for traversing packets

|  |  |  |
| --- | --- | --- |
| Offset | Location | Field |
| Protocol | 3rd | Bits 7~0 |
| Source IP | 4th | Bits 47~16 |
| Destination IP | 4th and 5th | 4th: MSBs in Bits 15~0 and LSBs in Bits 63~48 |
| Source Port | 5th | Bits 47~32 |
| Destination Port | 5th | Bits 31~16 |

A screenshot of a computer

Description automatically generated with medium confidence

Figure 17 description of src and dest IP offsets as observed in logic analyzer output

A screenshot of a computer

Description automatically generated with medium confidence

Figure 18 use of online wireshark equivalent to analyze the packet and write appropriate assembly program

A screenshot of a computer

Description automatically generated

Figure 19 use of online wireshark equivalent to analyze the packet and write appropriate assembly program

### Understanding protocols from point of view of manipulation

* IP (layer-3): checksum calculations only include the 20 bytes header
* TCP (layer-4): checksum includes header + data + IP header (some fields)
* ICMP: Checksum doesn’t includes any pseudo header like TCP

## Appendix F: First and Last Word (64 bits width) of a new packet

A screenshot of a computer

Description automatically generated with medium confidence

Figure 20 First 64-bit Word description of NetFPGA[[6]](#footnote-6)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 21 First 64-bit words after passing through the Output Port Lookup

A screenshot of a computer

Description automatically generated with medium confidence

Figure 22 example of the frist word (64-bits) decoded

A screenshot of a computer

Description automatically generated

Figure 23 example continued

## Appendix G: Useful files generated in the process of bitfile

The $NF\_DESIGN\_DIR/synth directory of your project is filled with compiler-generated text files containing important project information

* **nf2\_top.map and nf2\_top.mrp** - Output files for the Xilinx Map tool (which converts your logic into actual FPGA devices such as LUTs). These reports summarize your design and shows you how many FPGA resources (like Block RAMs, slices, etc.) are occupied and how many are still available for your design
* **nf2\_top.srp** - Output file from the HDL synthesis tool. This usually has "helpful" descriptions of any syntax errors in your HDL, as well as warnings for any latches that you might have accidentally created
* **nf2\_top\_par.twr** - Output file from the timing analysis program, showing all your clock domains and counting the number of signals in each domain that failed timing. The ideal phrase to have in this file is the magic line "All constraints were met.“

Note: You should ignore the Timing Summary section at the bottom of this file. The "Maximum frequency" statistic is not useful for this design because it has multiple clock domains. You should only be concerned with the number of timing errors found for each individual timing constraint.

* **nf2\_top\_par.par** - Outfile file from the place&route program, showing how long it took for the route to be calculated, and how many iterations it took to meet timing. The more iterations, the harder it is working to fit your design.
* **synth.txt** - Outfile file containing the entire command-line output of the compilation process (in case you missed it as it was scrolling past)

## Directory Structure (git)

|  |  |  |  |
| --- | --- | --- | --- |
| Dir. Structure (git) | Use-Case | File Names | Remarks |
| netlist | Bitfile | nf2\_top\_par\_10detect7b.bit |  |
| rules | SNORT rules configured | rules\_v2\_9\_7bytes.rules | 5 rules configured for detection |
| util | Fetch the results from Logic Analyzer and processes them to various fields | scriptimp.sh | In DETER |
| util | Fetch the results from Logic Analyzer | script\_raw.sh.sh | In DETER |
| util | Compiler for custom processor design | riscv\_compilerimp.py |  |
| util | rule-breaker.py | Analyzes various rule options to find simplest set of rules that are large enough |  |
| util | Mainstatus.xlsx | Instructions summary |  |
| docs/labinstructions | Important lab instructions |  |  |
| docs/riscv\_instruction\_set |  |  |  |
| docs/gen | Generated in the Fedora |  |  |
| include | Configuring make environment and h/w, s/w registers | ids.xml, project.xml |  |
| initialization | \_elaborate and \_.coe files kept here |  |  |
| lib/C/reg\_defines\_lab10.h | register addresses generated during the bitfile process |  |  |
| logs/logicanalyzer | Logic analyzer output files |  |  |
| payload | Used for testing the bandwidth |  |  |
| pcaps | Generated for making assembly programs |  |  |
| src | Source Verilog files |  |  |
| src/sim | Simulation related source and test bench files | ids\_sim.v, ids\_tb.tbw and ids\_tb.xwv |  |
| sw | Assembly programs |  |  |

# Knowledge Bank

### SNORT Rule option: flow explanation

<https://blog.snort.org/2011/09/flow-matters.html>

Thursday, September 8, 2011

Flow matters

Recently on one of the Snort lists, there was a thread that argued that the "flow" statement in rules didn't matter if you had your variables set correctly. This is a common misconception, so I thought I'd write a post about it and explain why flow, and its use in rules is important.

First let's talk about what flow is. The Snort reference manual says:

The flow keyword is used in conjunction with TCP stream reassembly. It allows rules to only apply to certain directions of the traffic flow.

You can click on the link above to read all about the different flow operators there are, I'm not going to regurgitate them here. But one thing to keep in mind, is that to\_client and from\_server are the same. As are to\_server and from\_client.

Let's take four rules for example.

alert tcp $HOME\_NET any -> $EXTERNAL\_NET $HTTP\_PORTS (flow:to\_server,established;)

Simple. Someone on your HOME\_NET going to EXTERNAL\_NET on a HTTP\_PORTS specified port, and the client initiated the conversation.

Example: Someone going to "www.google.com" or any Web site from your network outbound.

What about the opposite of that?

alert tcp $EXTERNAL\_NET $HTTP\_PORTS -> $HOME\_NET any (flow:to\_client,established;)

Again, pretty simple, someone went to a webpage, and now your rule is looking for the result of that action. The webpage (or file, or whatever) is coming down to the browser from the server.

Example: Someone went to "www.maliciouspdf.com" and downloaded a malicious PDF. You are not attempting to see if someone requested a ".pdf" as a download, (although you may do that to set a flowbit, see my post on resolving flowbit dependancies here), but you are looking for the actual PDF coming back down from the server.

So, let's take our first example and flip the flow. Remember, this is all assuming that you have HOME\_NET and EXTERNAL\_NET set, just like the email thread was implying.

alert tcp $HOME\_NET any -> $EXTERNAL\_NET $HTTP\_PORTS (flow:to\_client,established;)

So, traffic headed outbound from my network over HTTP\_PORTS, but HOME\_NET did not initiate the conversation.

Example: Let's say someone sets their source port as 80, and then infiltrates your network through a connection that starts outside (the initial SYN packet was sent inbound to HOME\_NET), you are looking for the traffic leaving your network headed back to that attacker who set his source port as 80. Maybe they are exfiltrating the DOC files off of your network.

Finally, let's look at the opposite of example two above.

alert tcp $EXTERNAL\_NET $HTTP\_PORTS -> $HOME\_NET any (flow:to\_server,established;)

Traffic headed inbound to the network over $HTTP\_PORTS but HOME\_NET again did not initiate the conversation.

Example: Again, let's say someone sets their source port as 80, and then infiltrates your network, this rule will look for the connections inbound to your network. Let's say they are requesting a DOC file from your network. Whereas the previous rule would look for the DOC file leaving, this rule will look for the initial request.

These are very simplistic examples, hopefully this post will help explain why flow is so important. Variables tell Snort which direction the traffic is going (inbound or outbound of your network in the simplest of terms), flow tells Snort who is responsible for which aspect of the conversation (are you a server, or a client?).

### SNORT manual

<http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node33.html#SECTION00469000000000000000>

### CIDR

CIDR (Classless Inter-Domain Routing) mask is a compact representation of an IP address and its associated network mask. CIDR notation is used to specify an IP address range and the number of significant bits (subnet mask) in the address. The notation is written as an IP address followed by a slash ("/") and the number of significant bits in the subnet mask (also known as prefix length).

CIDR was introduced to improve the scalability of IP addressing and routing, allowing for more efficient allocation of IP addresses and reducing the size of routing tables.

Here's an example to illustrate the CIDR mask:

Consider the IP address range: **192.168.0.0** to **192.168.0.255**. In CIDR notation, this range can be represented as **192.168.0.0/24**.

In this example, **192.168.0.0** is the network address, and **24** is the prefix length, which indicates that the first 24 bits of the IP address are significant and belong to the network part of the address. The remaining bits (32 - 24 = 8 bits) represent the host part of the address.

To understand this better, let's convert the IP address and subnet mask to binary:

IP address: **192.168.0.0** in binary is **11000000.10101000.00000000.00000000** Subnet mask: **255.255.255.0** in binary is **11111111.11111111.11111111.00000000**

In the binary representation of the subnet mask, there are 24 ones, which corresponds to the **/24** prefix length in the CIDR notation. The subnet mask **255.255.255.0** can be represented as **/24** in CIDR notation.

So, the CIDR mask **192.168.0.0/24** represents the IP address range **192.168.0.0** to **192.168.0.255** with the subnet mask **255.255.255.0**.

### TCP indicating end of current conversation

A TCP connection is terminated through a process called the TCP teardown or connection termination. This process involves the exchange of FIN (Finish) and ACK (Acknowledgment) packets between the two communicating devices. The FIN flag in the TCP header indicates that a device has finished sending data and wants to close the connection.

Here's a step-by-step breakdown of the TCP connection termination process:

1. The client (or the device initiating the termination) sends a TCP packet with the FIN flag set to the server (or the other device involved in the connection). This indicates that the client has finished sending data and wants to close the connection.

2. The server sends an ACK packet back to the client, acknowledging the receipt of the FIN packet. This indicates that the server has received the client's request to close the connection.

3. The server sends a FIN packet to the client, indicating that it has also finished sending data and is ready to close the connection.

4. The client sends an ACK packet back to the server, acknowledging the receipt of the server's FIN packet.

After this four-step process, the TCP connection is considered closed. Note that the connection termination process is independent for each direction of the data flow. So, one device can close its sending side of the connection, while the other device can still send data until it also closes its sending side.

To summarize, the FIN flag in the TCP header is used to indicate that a device wants to close a TCP connection. The exchange of FIN and ACK packets between the devices involved in the connection is what leads to the termination of the connection.

Meaning of ‘established’ in SNORT flow field:

source: <https://suricata.readthedocs.io/en/suricata-6.0.0/rules/flow-keywords.html>

The determination of established depends on the protocol:

* For TCP a connection will be established after a three way handshake.

Chart

Description automatically generated

* For other protocols (for example UDP), the connection will be considered established after seeing traffic from both sides of the connection.

Diagram

Description automatically generated

source: chatgpt

once a TCP connection is established after completing the TCP three-way handshake (SYN, SYN-ACK, and ACK), all subsequent packets exchanged between the client and server, including the data packets and the packets involved in the termination process (FIN and ACK), are considered part of the established connection.

When a Snort rule has flow:to\_server,established or flow:to\_client,established, it applies to packets that are part of this established connection. So, it will consider not only the data packets but also the FIN and ACK packets involved in the connection termination process.

# References

1. Snort 2.9 Rules Download: <https://snort.org/downloads#rules>
2. Snort Rules description: <http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node29.html#SECTION00421000000000000000>
3. Sample PCAP files repository: <https://wiki.wireshark.org/SampleCaptures#hypertext-transport-protocol-http>
4. Deter LAB: <https://www.isi.deterlab.net/showuser.php?user=14200>
5. PacketSafari: Web based viewer for PCAPs: <https://app.packetsafari.com/analyze/l/cn2wtYcB565GwqqYEw4U>
6. Description of NetFPGA Headers: <https://ecs-network.serv.pacific.edu/past-courses/2011-spring-ecpe-293b/tutorials/ethernet-hub-tutorial-implementation>
7. Online RISC-V Binary to Text Rule Description: <https://luplab.gitlab.io/rvcodecjs/#q=01000000000000000110000110010011&abi=true&isa=RV64I>

1. We have a python script named rule\_breaker.py through which we analyzed all the rule options and tried to find the simplest set of rules that still has reasonably large set of rules. We essentially traverse through each option of SNORT rule based and counted if we include/exclude this field, then how many rules are left [↑](#footnote-ref-1)
2. Any data (bytes) after layer-4 are considered payload [↑](#footnote-ref-2)
3. Header is added to each 7-byte payload [↑](#footnote-ref-3)
4. note, here message has to be ASCII text because iperf treats this as ASCII file, so if you want to send a string that has ‘qi’, ‘nu’, ‘jo’ and ‘Ai’, then writeas, DON’T convert to hex [↑](#footnote-ref-4)
5. <https://www.cisco.com/c/en/us/products/collateral/security/firewalls/secure-firewall-3100-series-ds.html> [↑](#footnote-ref-5)
6. Source: <https://ecs-network.serv.pacific.edu/past-courses/2011-spring-ecpe-293b/tutorials/ethernet-hub-tutorial-implementation> [↑](#footnote-ref-6)