

# IP Firewall Proof of Concept

on the Intel IXP 1200 platform

**Overview and  
Micro – Code Description**

18-844 Design and Evaluation

*Prof. Peter Steenkiste*

David Matsumoto  
Hemant Bhanoo  
Stanley Wang

Table of Contents

Table of Contents ..... 3

Introduction ..... 4

Introduction to Firewalls ..... 5

Desired Features..... 7

Functionality Subset ..... 8

    I. Security Functionality ..... 8

    II. Ease of Configuration ..... 9

    III. Reporting..... 10

    IV, V: Documentation and Installation ..... 10

Our Design..... 11

Problems..... 11

Design Details..... 11

Control Plane Design..... 13

Evaluation ..... 16

Code..... 17

    rec\_ipverify.uc ..... 17

    rec\_lmatch.uc ..... 34

    firewall.uc..... 39

    test\_init.cpp ..... 46

Introduction

We start off by giving a brief introduction to Firewalls, and why we chose to implement a firewall on this platform. Next, we describe the desired functionality of an ideal firewall, followed by a discussion of the compromises we made in order to create a working proof-of-concept in a semester.

We then jump right into the details of our design, assuming that the reader is familiar with the IXP 1200 platform<sup>1</sup>. After this, we present a block diagram describing the flow of control through the data plane.

We then explain some of the more interesting blocks in detail. We conclude with a retrospective discussion including shortcomings, evaluation, and possible future work.

The appendices contain code excerpts from the files, which we modified significantly.

<sup>1</sup> If not, Intel provides adequate documentation describing the platform both from a higher level, and in more detail.

---

## Introduction to Firewalls

As the Internet gains both popularity and importance in our lives, the spectrum of applications making use of network connectivity is growing rapidly.

In order to protect both individuals and organizational structures, it is crucial to be able to provide various levels and forms of security to users of such an infrastructure.

Security encompasses a very wide variety of problems, including data integrity, authentication, access control, and privacy.

Firewalls address a particular aspect of security namely, being able to selectively isolate access both to and from a part of a network in the effort to enforce abstract policies.

An example of this may be that we want to limit global access to a company's internal network, while allowing a subset of internal applications or people to access the global Internet.

Since such functionality could easily be implemented into the routing devices that would no doubt be present in such a scenario, it is questionable whether or not we need a separate device to provide such functionality.

There are several justifications for this near-duplication of hardware. Firstly, one of the underlying design paradigms of the modern-day Internet is to isolate functionality into logical blocks so that each block becomes simple, self contained, and easily replaceable.

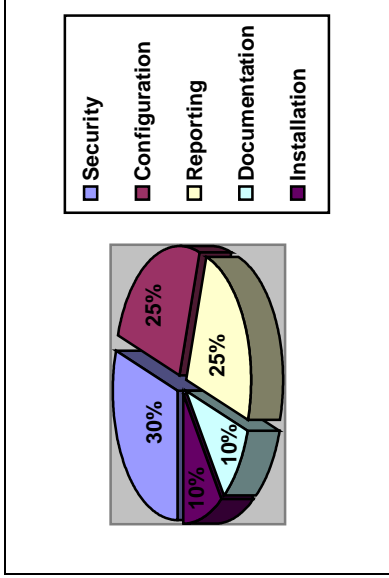
Second, whenever we deal with security, we must assume the worst-case scenario in which all unwanted users are malicious in nature. Under this assumption, we must try to maintain the availability of network resources while blocking out unwanted users. By separating the burden of access control from the more fundamental functionalities of path determination and packet forwarding, we reduce the risk of losing connectivity.

Why did we choose this as a project for the IXP platform? The IXP is a fast and versatile packet processor. One of the driving forces behind using the IXP platform seems to be to foster the development of software configurable network devices.

Since security is such a fast changing field – constantly altered by the emergence of new protocols and operating paradigms – it makes sense to develop a firewall on a platform that facilitates the easy addition of new features.

## Desired Features

Although we had some factors in mind when setting out to design the specifications for a firewall, we referred to a set of product tests carried out by the Network World Fusion magazine. In August of 1999 they compared the performance of several firewalls (<http://www.nwfusion.com/reviews/0799firewall.html>) and their scoring criteria proved useful in deciding what features to concentrate on.



Of course, the security functionality offered by a firewall is most important – and that was no surprise to us. However, there is a large emphasis placed on ease of configuration as well as the ability to report violations of policy. This, on second thought, is not surprising considering that the end user of this product is most likely a network administrator who has to deal with large numbers of hosts and network nodes. The ability to easily setup the firewall, and furthermore to be able to isolate policy violators is key to his effectiveness.

## Functionality Subset

We had several constraints to take into account when deciding what features to implement.

1. *Time.* This was one of the most important factors. We needed to be able to demonstrate a working firewall by the end of the semester so it was necessary to choose a subset of features that could be realistically implemented.
2. *Platform.* We were given a brief introduction to the platform by means of a small project. Having worked on the project, we were introduced to the complexity of developing in a new and consequently relatively undocumented platform. We decided to set various levels of goals so that we could progress from one level to the next if and when we got familiar with the platform.
3. *Experience.* Between the group members, we had limited hands-on experience in network security so it was again important for us to set achievable goals first, but allowing for suitable extensions if time permitted.

Keeping these in mind, let us discuss each of the five design criteria in turn.

### I. Security Functionality

All of the most basic firewalls allow filtering based on information that can be easily extracted from packet headers. The most common is expressed as a *five tuple*: {ip protocol, source address, destination address, source port, destination port}.

This is also very convenient to implement because this *five tuple* accurately represents a logical ‘flow’ of information in a majority of the traffic seen on the Internet.

Exactly specifying the values of each of the fields in this tuple would be an arduous task so there must be some form of aggregation which enables policies to be specified in a sensible manner. The most common form of this is known as *wildcarding*.

Here, each field has an associated *bitmask* that allows you to specify which bits to compare in a field. Thus it is possible to specify “all source addresses in the 128.2 network” using a value of 128.2.0.0 and a bitmask of 255.255.0.0.

This is a very basic form of packet filtering and does not require much intelligence on the part of the firewall. Most commercial products have detailed knowledge of the protocols being carried over IP and allow you to specify rules in much more detail.

One of our goals at the outset was to design our firewall so that we could easily add in more functionality if we had the resources to do so. Our initial plan was to allow some sort of URL based filtering of HTTP requests.

## **II. Ease of Configuration**

Since ease of configuration was an important factor, we decided that it would be nice to allow remote configuration of our firewall, if desired. This would involve an administrator being able to send configuration information to the firewall by using customized application running on a remote machine.

Admittedly, this in itself is a potential security risk for at least two reasons:

1. Unauthorized users may be able to send spurious packets, thus reconfiguring the firewall policies to suit their (malicious) needs.
2. If unauthorized users are able to see the specific policies implemented on the firewall by sniffing packets, it becomes easier to find loopholes in rules.

That said, many network elements support remote configuration and rely on standard encrypted transfers (such as *ssh*) to achieve security.

We decided not to worry about the security aspects, but instead focus on the ability to configure the firewall remotely, adding or removing rules on the fly.

## **III. Reporting**

Network administrators need as much information as possible about the type of traffic traversing their network but, at the same time, it is quite costly to monitor *all* traffic.

There is a feature implemented by Cisco in their routers that allows you to specify a flow in a similar manner to a firewall policy except that the flow is not dropped – you simply log occurrences of that flow.

We decided that we would bundle that functionality into our implementation and allow both violating and non-violating flows to be logged.

Our intention was that these logs would be sent out to the administrator periodically as local memory began to fill up.

Again there are security issues involved with this (similar to those described in the previous section) but we decided to ignore these.

## **IV, V: Documentation and Installation**

Given that this was a proof of concept project on a test platform, we didn’t pay much attention to installation, and this paper serves as the documentation.

---

## Our Design

It may be worthwhile to mention at the outset that we were not able to add in much of the functionality that we had hoped for due to various factors. However we did end up with a better design than we started off with which added to the complexity of the project.

### Problems

Before proceeding, it makes sense to note down the various problems we had – since it provides a good perspective when looking at the actual functionality that the implemented.

- *Administrative Packets*: It wasn't until a week before the project completion deadline that the TA figured out how to send packets to/from the control plane. The changes affected our microcode, so we decided not to risk breaking our existing code and instead to not include that functionality
- *Interplane Communication*: Communication between the control and data planes was untested, so we decided again to use SRAM polling (explained later) instead.

### Design Details

It would be very inefficient to do a rule lookup with the arrival of every packet – especially because, due to the existence of wildcards in the rules, and the fact that they rely on ordering, they cannot be hashed. Instead, we cache each flow in a hash table as it arrives, and do a rule lookup only once per new flow<sup>2</sup>.

Since the flow lookup turned out to be the most time-intensive processing (assuming that the flow is cached), we decided to optimize this as much as possible. To do this, we used a hash table with a linked list at each hashing position to deal with collisions. Dealing with insertion and removal of cache entries turned out to be non-trivial since we need to maintain our own free-memory pool, and do all of the pointer arithmetic in microcode<sup>2</sup>. If we are looking at a flow that we have not seen

---

<sup>2</sup> See flowcharts in appendix

before, we fill in all of the details for the flow except the bit that signifies pass or drop of that flow. The incomplete entry is then passed up to the control plane, and we wait until the results from a full rule lookup are available.

Once we have a valid cache entry, we increment the packet counter that exists as part of the cache entry, and then make a decision as to whether or not to allow the packet through.

If we allow it to pass, a route lookup is performed, and the packet continues in its regular path through the router. If we decide to drop it, we branch to the predefined `packet_late_discard` label, where the packet buffers are freed, and the thread readies itself to receive a new packet.

We believe the provided flowcharts will make it easier to understand the details of the packet flow.

## Control Plane Design

---

Our control plane can be divided into two main functions. The first function deals with the polling, and validation of a flow. The primary goal of this function is to update the cache with the correct “forward or drop” bit setting after validating the cache entry against the policies in the policy table. The second function deals with the addition and removal of new policies. The administrator uses the shell to enter the policies that are to be added or removed from the firewall. The administrator also has the option of flushing the cache and policy table.

Originally, we had more functions in the control plane, however we were limited greatly by the fact that the only communication the control plane could have with the data plane was through polling of memory locations. It also meant that we couldn’t receive or send packets out on the data plane. This prevented us from sending out logs, or receiving control packets. It also cut into our throughput and efficiency because of the constant polling we had to do. However, the two functions described do work on the IXP 1200 as we have demonstrated in our demo.

We define a policy to be made up of several filters. A filter is a restriction on one field on a TCP/IP packet; source address, destination address, source port, destination port, and protocol. A table of all existing filters, where each entry contains a policy id, a filter id, a filter value and a filter mask, is kept on the SRAM. The policy id defines which policy the entry belongs to. The filter id of an entry contains which type the filter is, whether it is a filter on source address, destination address etc. The filter value contains the value of the filter, and the mask allows wild card values to exist. We also keep an array of policies on the SRAM. An entry in this table contains the policy id, the number of filters it contains, and an integer value to tell the firewall whether it should drop or forward the packet. The validation of a flow is done through a linear search of a filter table and then another linear comparison through a policy table.

To further understand this function of the control plane, we will describe an example usage. The control plane first initializes all the SRAM locations that it will be using. This includes the filter

table, policy table, and the cache table. The program then does a round-robin poll of 7-8 memory locations, where each location is specific to a receive thread. The polling location not only tells the control plane when to run, but also tells the control plane which function to run. For example, by setting the polling location to 1, the control plane will know that the data plane wants it to run a cache validation. Next to each polling location, contains a memory address reserved for an argument that is passed to the control plane. We use this argument to tell the control plane the location of the data that we wish to pass as an argument into the function that the control plane is going to run. In our example, we set the polling location to 1 (meaning Cache Validation), and we set the memory location next to the polling location to the address where the cache entry is located. The control plane will then “wake up” when it sees that the polling location’s value is no longer zero.

In order to begin the cache validation, the control plane must first extract the 5-tuple from the cache entry in SRAM. Once we have that information, we do a linear traversal through our filter table in memory, comparing each filter with our information. We keep an array of counters on each filter that passes, using the policy id entry of the filter as the index to our array. Once we finish traversing the filter table, we then traverse our policy table comparing the number of filters each policy contains to our array of counters. If the number of filters match, we then break out of our comparisons, and proceed to update the “allow or disallow bit” of the cache with the matching policy’s integer field containing whether we should drop or forward the flow. After this, we set the polling and the argument memory locations to zero, and we continue polling until the next thread updates the polling memory location.

The second main function of our control plane is the adding and removing of policies. Originally, we designed our administrator to be a remote user that could send control packets to the IXP. We had a great deal of difficulty using the PETH drivers to send control packets to the IXP, so instead we chose to simplify by writing the policies to the file and having the IXP read in the file. Unfortunately, this did not work either. We then had to resort to inputting policies from the shell of the IXP through a text based

interface. It worked quite well functionally. We would call functions in the shell to add a policy, which would prompt us to type in the number of filters, information for each filter, and whether this policy is to add or drop.

The code for the control plane can be found in the file `control_plane.cpp` or in appendix A.

## Evaluation

Due to time constraints with the lab, we were unable to do a proper evaluation of the firewall. However, here are some observations based on hindsight:

- The first packet in a flow experiences extreme delay in our firewall. This is due to the fact that we are using SRAM polling to request a rule lookup from the control plane. In hindsight, using the `scratched` memory may have increased the speed of this.
- We are only using the source and destination address (64 bits total) as a key for the hash function. This was done to simplify the code (since a 64 bit hash function, `hash1_64` is provided to us). This means that our hash table becomes quite unbalanced since it is likely that multiple flows will exist between two hosts. Hashing on the protocol as well (and possible the source and destination ports) would even out the distribution.
- We have not dealt with removal of cache entries. Several algorithms are suggested:
  - *Random*: remove a random cache entry. Not very predictable.
  - *LRU*: remove the least recently used entry. Difficult to implement. Also, it may not give good performance since a connection could be idle for a long time, and then suddenly burst.
  - *TCP-aware removal*: This is probably the best strategy and involves removing a TCP entry from the cache if we predict that a flow has ended. Doing this involves keeping some state within the firewall. This is quite feasible since we have a few unused bits in the cache entry to play with. Combining this with some sort of random removal (to take care of non TCP flows if required) would probably be the best strategy.
- In general, having a more elaborate mechanism for storing connection state information would greatly enhance the flexibility of the firewall since we could offer a wider (and more intelligent) set of filters.



## Code

---

### rec\_ipverify.uc

```
//-----  
//  
//          I N T E L   P R O P R I E T A R Y  
//  
//    COPYRIGHT (c) 1998-99 BY INTEL CORPORATION. ALL RIGHTS  
//    RESERVED. NO PART OF THIS PROGRAM OR PUBLICATION MAY  
//    BE REPRODUCED, TRANSMITTED, TRANSCRIBED, STORED IN A  
//    RETRIEVAL SYSTEM, OR TRANSLATED INTO ANY LANGUAGE OR COMPUTER  
//    LANGUAGE IN ANY FORM OR BY ANY MEANS, ELECTRONIC, MECHANICAL,  
//    MAGNETIC, OPTICAL, CHEMICAL, MANUAL, OR OTHERWISE, WITHOUT  
//    THE PRIOR WRITTEN PERMISSION OF :  
//  
//          INTEL CORPORATION  
//  
//          2200 MISSION COLLEGE BLVD  
//  
//          SANTA CLARA, CALIFORNIA 95052-8119  
//-----  
// rec_ipverify.uc  
// IPV4 verify the ip header prior to routing decision  
//  
//  
// system: SA1200  
// subsystem: IP route microcode  
  
17  
  
// usage: example  
// author: dfh 11/8/97  
//  
// revisions:  
//          dfh          1/12/99          bi-endian. lower level macros perform swaps if #define  
LITTLE_ENDIAN  
//          dfh          3/31/00          use ip.uc macros  
//  
// -----SA1200 microcode-----  
  
// Prerequisite:  
//      rec_nextpac.uc  has executed  
//      first 4 longwords of the packet has been read in from receive FIFO  
//  
//      register usage at this point  
//  
//      $xfer0          MAC DA<31:0>  
//      $xfer1          MAC DA<48:32> and SA<15:0>  
//      $xfer2          MAC SA<31:0>  
//      $xfer3          MAC length and ip version, IHL, TOS  
//      protocol_len    datalink payload length  
//      packet_buf_addr address of buffer in sdram where packet will be stored  
//      buf_handle      address offset of the current descriptor  
//      rec_state       fastport flag, packet sequence number, byte enables, eop,  
sop  
  
//  
#include "ip.uc"                                ; ip macros
```

```

// cause insertion of FIREWALL code
//
//#ifndef FIREWALL
#define FIREWALL
//endif

ip_verify#:
.local sa01 ip_prot ip_sa ip_source_port ip_dest_port arrayAccess
immed[ip_prot, 0] ; this acts as a flag to prevent the real ipverify code being run multiple times

ip_verify2#:

// move second 4 quadwords of receive fifo element to sdram
//
    sdram_r_fifo_rd[packet_buf_addr, rfifo_entry, 4, 4, ASYNC]          ; (see stdmac.uc)

read_rfifo2#:
// get the next 6 longwords of the packet from the receive FIFO
// continuing from where layer 2 left off, and wrapping
//
    r_fifo_rd[$xfer4, rfifo_entry, 2, 2]
#ifdef LITTLE_ENDIAN
    alu[sa01, 0, +16, $xfer1, >>16]
#else
    alu[sa01, 0, +16, $xfer1]
    ; save sa bytes 0-1 for later merge
#endif
    r_fifo_rd[$xfer0, rfifo_entry, 4, 1], ctx_swap

```

19

```

    ip_verify[exception, 14]          ; ip header starts at byte 14
    br>0[packet_discard#]
    ctx_arb[voluntary]
    ip_modify[14, 14]
#ifdef FIREWALL
    alu[--,--,B,ip_prot]
    br!=0[firewall_ok#] ;if we've already run the firewall code, just jump past it.
#endif

ip_ok#:

// *****
// * HBHANO0
// *
// * Now that we're okay, _and_ we have the packet loaded into memory,
// * go ahead and run the firewall code. In general, here's the algorithm:
// *
// *
// * 1) Extract all relevant information
// * 2) Lookup the entry in the policy table
// *    If found -> continue.
// *    If not found, add to the cache and request a policy lookup from the control plane
// * 3) Increment counter for this flow
// * 4) (If this is the last packet in a TCP flow, remove the cache entry)
// * 5) Check the rules
// * 6) Drop/Pass based on the rules.
// *
// *****

#ifdef FIREWALL

```

20

```
ip_da_extract[ip_da,14] ;// ----- B E G I N   F I R E W A L L   C O D E -----
-----
```

```
// insert code here to check if the the packet is for us. (PETH STUFF!)
```

```
ip_prot_extract[ip_prot,14]      //extract ip protocol
ip_sa_extract[ip_sa,14]          //extract ip source address
ip_source_port_extract[ip_source_port,14]  //extract ip source port
ip_dest_port_extract[ip_dest_port,14] //extract dest port; get da. ip header starts at read xfer byte 14
```

```
// * HBHANO0
// * Step two in the firewall: go through the flow cache
// * and see if there are any matches.
// *
// * Here's what the cache looks like:
// *
// * char bitvector; (1 byte)
// * char protocol; //(1 byte)
// * short int counter; //(2 bytes) - packet counter.
// *
// * int src_addr; //(4 bytes)
// * int dst_addr; //(4 bytes)
// *
// * short int src_port; //(2 bytes)
// * short int dst_port; //(2 bytes)
// *
// * int string_ptr; //(4 bytes) - ptr to the string we want to search for in memory
// * int func_ptr; //(4 bytes) - ptr to the function we give the string to, to search
```

21

```
// * //for in the packet
// *
// * used for chaining the hash table:
// * int next; (4 bytes)
// * int prev; (4 bytes)
```

```
.local compare
```

```
// The hash multiplier tells the has unit which bits to use from the input when hashing.
// We want this set to FFFF so that all the bits are used...
```

```
immed[$xfer7, 0x0ffff]
csr[write,$xfer7, hash_multiplier_64_lo], ctx_swap
csr[write,$xfer7, hash_multiplier_64_hi], ctx_swap
```

```
// * in order to use hashing, first create a tuple containing the appropriate
// * information to hash on. After that, traverse through the rest of the list to
// * find the appropriate entry.
// *
// * Note: only the src and dst address are being used to perform the hash. Although
// * this may not be ideal - it means that we can use the hash_64 function trivially,
// * and without complication. Since we're handling hash collisions anyways, it shouldn't
// * affect performance tooooo much.
```

```
alu[$xfer4, --, B, ip_sa]
```

22

```

alu[$xfer5, --, B, ip_da]
hash1_64[$xfer4], ctx_swap

// we should now have hashed values in $xfer0/$xfer1. Let's
// just use the 16 LSBs of $xfer0 to access our array.
.local policyCacheBase
immed[policyCacheBase, 0x2001]

alu[arrayAccess, --, B, $xfer4, <<24]
alu[arrayAccess, --, B, arrayAccess, >>21] ;24 - 21 = 3 => size of each cache entry is 2^3=8 32bit-
words wide.
alu[arrayAccess, arrayAccess, +, policyCacheBase]
.endlocal //policyCacheBase
.local chaining hashTableHead
alu[hashTableHead, --, B, arrayAccess]

cache_lookup_begin#:
// do a read_lock ONLY if we are at the head of the hashtable
// this prevents more than one thread trying to look up the same flow at the
// same time. Not very efficient, but prevents problems during addition of
// new cache entries...

alu[--, hashTableHead, XOR, arrayAccess]
br=0[lookup_lock#];lock if we're at the head
sram[read, $xfer4, arrayAccess, 0x0, 1], ctx_swap ;else just do a regular read
br[lock_done#]

lookup_lock#:
sram[read_lock, $xfer4, arrayAccess, 0x0, 1], ctx_swap ;actual locking

```

```

lock_done#:
alu[controlWord, --, B, $xfer4]

// if nothing exists at this hash position, we're done - no match!
alu_shf[compare, --, B, controlWord, >>31] ; the MSB of the 1st byte is 1 if this entry is in use
br=0[add_entry_to_cache#], defer[1] ; no entries at all, so add entry to cache...
alu[chaining,--,B, 0x0]

alu[chaining,--,B,0x1]

// now try to match the entry, else proceed in the chain (if it exists)
alu_shf[compare, 0x0, +8, controlWord, >>16]; compare = 2nd MSByte = protocol

alu[compare, compare, XOR, ip_prot]; compare protocol fields...
br!=0[cache_lookup_fail#]

sram[read, $xfer5, arrayAccess, 0x1, 3], ctx_swap; load the src/dst address and ports

// compare the source addresses:
alu[compare, ip_sa, XOR, $xfer5]
//if the match was INCORRECT, we'll get a nonzero answer
// so branch
br!=0[cache_lookup_fail#]

// compare the destination addresses:
alu[compare, ip_da, XOR, $xfer6]
//if the match was INCORRECT, we'll get a nonzero answer
// so branch

```

```

br!=0[cache_lookup_fail#]

// port information only makes sense IF
// the packet is either UDP or TCP
// so check what the protocol is:
alu[--, ip_prot, XOR, 0x6]
br=0[check_ports#]
alu[--, ip_prot, XOR, 0x11]
br!=0[skip_ports#]

// create a combination of the port info:
check_ports#:
.local ports
alu_shf[ports, 0x0, +, ip_source_port, <<16]
alu[ports, ip_dest_port, OR, ports]

alu[compare, ports, XOR, $xfer7]
//if the match was INCORRECT, we'll get a nonzero answer
// so branch
br!=0[cache_lookup_fail#]
.endlocal //ports

skip_ports#:
// if we reached this far, then all of our TUPLES matched.
// now we have to deal with extended features, and then
// deal with a rule match.

sram[unlock, --, hashTableHead, 0, 1]
br[rule_match#]

```

25

```

cache_lookup_fail#:
// *****
// come here if any portion of the lookup fails
// ie. advance in the chain until we reach the end.
// *****

sram[read, $xfer5, arrayAccess, 6, 1], ctx_swap ; read in the pointer....

alu[compare, $xfer5, XOR, 0x0]; if the 'next' pointer is NULL,
br=0[add_entry_to_cache#] ; then we need to add this flow to the cache...

alu[arrayAccess, 0x0, B, $xfer5]; else, update the address with the 'next' pointer
br[cache_lookup_begin#]

add_entry_to_cache#:
// *****
// * we reached the end of the cache without a successful match.
// *****
// *
// * Here we have to enter code to chain this entry on to the
// * end of the hash table using our 'freelist'.
// * we fill in all the required parameters for the flow, except

```

26

```

// * for the bitvector conatining the decision of the policy lookup.
// *
// * Then we pass this information up to the control plane. The control
// * plane (StrongARM) then does the lookup of this policy and replies
// * to us when it has filled in the fields. At that point, we should be
// * able to continue by checking the result of the policy.

.local memaddr

addCacheEntry[memaddr, arrayAccess, chaining]
alu[arrayAccess, --, B, memaddr]

.endlocal //memaddr

// * arrayAccess now contains the address of the NEW entry.
// *
// * ask the control plane to look this entry up...

// each context is assigned two addresses:
// one for a 'function code', and one for
// a 'parameter'

.local pollingAddress
.local tempaddition
immed[pollingAddress, 0x1FF, <<8]
immed[tempaddition, 0xE0]
alu[pollingAddress, pollingAddress, +, tempaddition]
.endlocal //tempaddition
#define_eval CONTEXT0_OFFSET (FID*8)
#define_eval CONTEXT1_OFFSET (FID*8+2)
#define_eval CONTEXT2_OFFSET (FID*8+4)

```

27

```

#define_eval CONTEXT3_OFFSET (FID*8+6)
// context
br=ctx[1,poll_loc1#]
br=ctx[2,poll_loc2#]
br=ctx[3,poll_loc3#]

br[start_poll#], defer[1]
alu[pollingAddress, pollingAddress, +, CONTEXT0_OFFSET]

poll_loc1#:
br[start_poll#], defer[1]
alu[pollingAddress, pollingAddress, +, CONTEXT1_OFFSET]

poll_loc2#:
br[start_poll#], defer[1]
alu[pollingAddress, pollingAddress, +, CONTEXT2_OFFSET]

poll_loc3#:
br[start_poll#], defer[1]
alu[pollingAddress, pollingAddress, +, CONTEXT3_OFFSET]

start_poll#:
// at this point, we should know the correct polling address
sram[read, $xfer5, pollingAddress, 0, 1], ctx_swap
// verfiy that it is free:
alu[--,--,B,$xfer5]
br!=0[start_poll#] ; if, for some reason, the address is nonzero, WAIT.
alu[$xfer6, --, B, arrayAccess]; setup the parameter (the location of our flow cache)
alu[$xfer5, --, B, 0x1]; setup the function type
sram[write, $xfer5, pollingAddress, 0, 2], ctx_swap

```

28

```

wait_for_lookup#:
nop; just
nop; adding
nop; some
nop; delay
nop; to
nop; give
nop; the
nop; strongarm
nop; time to do a
nop; policy
nop; lookup.
sram[read, $xfer5, pollingAddress, 0, 2], ctx_swap
alu[--, --, B, $xfer5]
br!=0[wait_for_lookup#], guess_branch

.endlocal //pollingAddress

// so now that they've done the lookup (wohoo!)
// make sure we UNLOCK the hash table head so that some other dudes
// can do a flow lookup....
sram[unlock, --, hashTableHead, 0, 1]
// * now read the first 32bit word into $xfer4 so that
// * we are consistent with the assumptions of the remaining code...
sram[read_lock, $xfer4, arrayAccess, 0x0, 1], ctx_swap
sram[unlock, --, arrayAccess, 0x0, 1]
.endlocal //chaining hashTableHead

rule_match#:
//alu[arrayAccess, arrayAccess, -, 0x1]; ---HACK---

```

29

```

//br[check_permit#] ; ---HACK---
// *
// * So if we made it this far, that means that we passed
// * all of the tests.
// *
// * In this case, we have a match, and need to apply
// * the appropriate rule.
// * Note:
// *   o first update the counter for this flow...
// *   o then remove this flow from the cache if it is
// *     the last packet in a TCP session (approximation)
// *
// * $xfer0 still contains the initial 32bit word, of which
// *   the MSByte looks like this:
// *
// * -----
// * |InUse|   |   |   |   | |Log|Drop|
// * -----
// *   7   6   5   4   3   2   1   0
// *
//
// #define DROP 1

.local counter memlocation controlBits

// update the counters for this flow...
sram[read_lock, $xfer4, arrayAccess, 0x0, 1], ctx_swap
alu_shf[compare, 0x0, B, $xfer4, >>16] //compare contains the upper word
alu[counter, 0x1, +16, $xfer4] // increment the lower word by one
alu_shf[$xfer4, counter, OR, compare, <<16] // put the two words back together.
sram[write_unlock, $xfer4, arrayAccess, 0x0, 1], ctx_swap ; write the updated info

```

30

```
br[check_permit#] ; hack - isolate the CACHE REMOVAL CODE
```

```
// possibly remove this entry from the cache:
alu[compare, ip_prot, -, 0x6] ;if packet isn't TCP
br!=0[check_permit#]; we're done
tcp_control_extract[compare, 14];get control bits // (defined in ip.uc)
alu[compare, 0x11, -, compare] ;see if it's a FIN_ACK
br!=0[check_permit#]; if not, we're fine.

// * else:
// *
// *
// * Cache_entry removal code:
// * P = previous, N = next
// * if(P== NULL) (head)
// *   copyCacheEntry(next, this)
// *   freeCacheEntry(next)
// *   this.previous = NULL
```

31

```
// * else
// *   freeCacheEntry(this)
// *
alu[memlocation, arrayAccess, +, 0x6] ; increment by 6
sram[read, $xfer5, memlocation, 0x0, 2]

// * Now:      $xfer5 = next
// *           $xfer6 = previous
// *
alu[--,--, B, $xfer6]
br!=0[cache_remove_not_head#]
// * If we're here, it means that this *is* the head of the list:
copyCacheEntrytoHead[memlocation, $xfer5]
freeCacheEntry[$xfer5]
br[check_permit#]

cache_remove_not_head#:
freeCacheEntry[memlocation]

.endlocal //counter memlocation controlBits

//immed[exception, FIREWALL_DESTROY]
//immed[exception, 0]
//immed[compare, DROP]
//alu_shf[compare, compare, AND, controlWord, >>24]
//immed[exception, FIREWALL_DESTROY]
//br[packet_discard#] ; defined in rec_nextpac.uc

//mess with packet to corrupt it, then:
//br[packet_discard#] ; defined in rec_nextpac.uc
```

32



```

// This means that we're A-okay. The packet is free to go.

.endlocal ; // compare

#endif // FIREWALL
all_done#:

```

## rec\_lmatch.uc

```

//-----
//
//          I N T E L   P R O P R I E T A R Y
//
//  COPYRIGHT (c) 1998-99 BY INTEL CORPORATION. ALL RIGHTS
//  RESERVED. NO PART OF THIS PROGRAM OR PUBLICATION MAY
//  BE REPRODUCED, TRANSMITTED, TRANSCRIBED, STORED IN A
//  RETRIEVAL SYSTEM, OR TRANSLATED INTO ANY LANGUAGE OR COMPUTER
//  LANGUAGE IN ANY FORM OR BY ANY MEANS, ELECTRONIC, MECHANICAL,
//  MAGNETIC, OPTICAL, CHEMICAL, MANUAL, OR OTHERWISE, WITHOUT
//  THE PRIOR WRITTEN PERMISSION OF :
//
//          INTEL CORPORATION
//
//          2200 MISSION COLLEGE BLVD
//
//          SANTA CLARA, CALIFORNIA 95052-8119
//-----
// rec_lmatch.uc
// IPV4 longest specific address match
//
//
// system: SA1200
// subsystem: IP router microcode
// usage: example
// author: dfh 9/28/97
//
// revisions:

```

```
// dfh      Feb 28, 2000      use ip.uc macros, ip_da_extract, ip_trie5_lookup

// -----SA1200 microcode-----
//
// Prerequisite Register Usage:
//      symbol
//      hi64k_base      - base address of 3 adjacent tables:
//                          1. 64k entry lookup table, each entry 32 bits, total 256KB
//                          2. 256 entry lookup table, each entry 32 bits, total 1KB
//                          3. many 16 entry trie tables, each entry 32 bits
//      route_table_base -      sdram route entries 128K x 8 longwords (1MB)
//                          32 byte stride = offset must be shifted left 5
//      ip_da      - 32 bit IP destination address
//
// description:
//      Get the route entry whose ip address has the most specific match with ip_da.
//      Also known as longest-prefix match.
//      lookup result:
//      $$route_entry0-3 - 4 longwords of route entry info,
//                          including the forwarding interface and MAC DA
//      Algorithm:
//      please refer to IXP1200 Software Reference Manual (SRM)

#include "ip.uc"

.xfer_order $$xfer0 $$xfer1 $$xfer2 $$xfer3 $$xfer4 $$xfer5 $$xfer6 $$xfer7
.operand_synonym $$route_entry0 $$xfer0
.operand_synonym $$route_entry1 $$xfer1
.operand_synonym $$route_entry2 $$xfer2
```

35

```
.operand_synonym $$route_entry3 $$xfer3

.import_var hi64k_base route_table_base

check_permit#:
#ifdef FIREWALL
.local compare
immed[compare, 1]
sram[read, $xfer0, arrayAccess, 0, 1], ctx_swap
alu_shf[--, compare, AND, $xfer0, >>24]
br=0[packet_discard#]
br[ip_verify2#] ;branch back up to reclaim all the register space we messed up.
.endlocal
#endif // FIREWALL
firewall_ok#:

route_lookup#:
      ip_da_extract[ip_da, 14] ; get
da. ip header starts at read xfer byte 14
      ip_trie5_lookup[route_ent_offset, ip_da, hi64k_base, 0] ; perform longest prefix
match
      br!=0[ip_get_route#], guess_branch
      immed[exception, IP_NO_ROUTE]
      br[packet_discard#]

ip_get_route#:
.local temp_route_base
      immed_w0[temp_route_base, route_table_base] ; load shared address value
      immed_w1[temp_route_base, route_table_base>>16]
```

36

```

// the transfer from rfifo was done at the top of ipverify, in order to free the rfifo element earlier
// we should have the signal back well before now
//
    ctx_arb[sdram] ; is transfer from
rfifo done?

// Write thread done as soon as the rfifo element is transferred to sdram
// By the time it turns around and get sot the Receive Scheduler, this thread will be done
//
#ifdef RECEIVE16 ; if no receive scheduler, there are 16
receive threads
    fast_wr[3, THREAD_DONE] ; notify receive scheduler with EOP encode
#endif

//-----
#ifdef LAYER4
// do some busywork to emulate layer 4+
    delay[8]
    sram[read, $xfer2, temp_base3, 2, 1], ctx_swap
    delay[8]
    move[$xfer0, 0]
    move[$xfer1, 1]
    move[$xfer2, 0]
    move[$xfer3, 1]
    hash2_48[$xfer0], ctx_swap ; 2 hash lookups
    delay[8]
    sram[read, $xfer2, temp_base3, 2, 1], ctx_swap
    delay[8]
    sram[read, $xfer2, temp_base3, 2, 1], ctx_swap
#endif

37

//-----
// get route entry
    sram[read, $$route_entry0, temp_route_base, route_ent_offset, 2], optimize_mem, ctx_swap
    alu[output_port, --, b, $$route_entry0] ; save output port for
enqueue

.endlocal // temp_route_base

got_output_port#:
write_layers23#:
// modify layer 2. insert MAC DA from route entry, write first 32 bytes to sdram
    move[$$xfer0, $$route_entry1] ; next hop da bytes 0-
3
#ifdef LITTLE_ENDIAN
    alu[$$xfer1, $$route_entry2, +, sa01, <<16] ; next hop da bytes 4-5: 0
merge with sa bytes 0-1
#else
    alu[$$xfer1, sa01, +, $$route_entry2, <<16] ; next hop da bytes 4-5: 0
merge with sa bytes 0-1
#endif
    move[$$xfer2, $xfer2] ; sa bytes 2-5
    sram[write, $$xfer0, packet_buf_addr, 0, 4] ; write first 32 bytes of modified
mpacket

.endlocal // sa01 RunFirewall ip_prot ip_sa ip_source_port ip_dest_port

; rec_enqueue.uc follows

```

## firewall.uc

```
// *****
// *
// *
// *
// *
// *
// *
// * firewall.uc
// * helper macros for the firewall
//
//
// * system: IXP1200
// * subsystem: IP microcode
// * usage: library macros
// * author: hbhanoo, dec 2000
// *
// * revisions:
// *****
```

```
#ifndef FIREWALL_UC
#define FIREWALL_UC
```

```
// * copyCacheEntry
// *      description: Just copies a fixed number of words from one location to another
//
// *      outputs:
// *      _target
// *      inputs:
```

39

```
// *      _source      (if source = 0, then do nothing!)
// *      example usage:
// *      copyCacheEntry[_source, _target]
// *
```

```
#macro copyCacheEntrytoHead[_target, _source]
// * we're allowed to play with xfer registers 5,6,7 only
// *
```

```
alu[--,--,B,_source]
br=0[finish_copyCacheEntry#]
```

```
sram[read, $xfer5, _source, 0x0, 3], ctx_swap ; read first 3 words
sram[write, $xfer5, _target, 0x0, 3], ctx_swap ; write first 3 words
sram[read, $xfer5, _source, 0x3, 3], ctx_swap ; read next 3 words
sram[write, $xfer5, _target, 0x3, 3], ctx_swap ; write next 3 words
sram[read, $xfer5, _source, 0x6, 1], ctx_swap ; read second last word
sram[write, $xfer5, _target, 0x6, 1], ctx_swap ; write second last word
```

```
; skip the 'previous' pointer - this should still == 0 if we're at the head
finish_copyCacheEntry#:
```

```
#endm
```

```
// * freeCacheEntry
// *      description: Frees an entry from the cache. Preserves the integrity of all
// *                  chains in the cache.
// *                  NOTE: The memory associated with this entry is NOT freed.
// *                  This task is reserved for the control plane, in the
// *                  event that it needs to perform further actions
// *                  (such as logging the end of a flow).
```

40

```

// *      outputs:
// *      inputs:
// *      _addr
// *      example usage:
// *      copyCacheEntry[_source, _target]
// *
#macro freeCacheEntry[_addr]
// * we're allowed to play with xfer registers 5,6,7 only
// *
.local previous next ptr
alu[--, --, B, _addr]
br=0[freeCacheEntryDone#]
alu[next, _addr, +, 0x6]
sram[read, $xfer5, next, 0x0, 2], ctx_swap

alu[next, --, B, $xfer5]
alu[previous, --, B, $xfer6]

alu[$xfer6, --, B, next]
alu[$xfer5, --, B, previous]

alu[--, --, B, next] ; if next
br=0[skip_nextptr#] ; ==0 then skip, else:
alu[ptr, next, +, 0x7]
sram[write, $xfer5, ptr, 0x0, 1], ctx_swap
skip_nextptr#:

alu[--, --, B, previous] ; if previous
br=0[freeCacheEntryDone#] ; ==0 then skip, else:
alu[ptr, previous, +, 0x6]

```

41

```

sram[write, $xfer6, ptr, 0x0, 1], ctx_swap
freeCacheEntryDone#:

.endlocal
#endm

// * addCacheEntry
// *      description: Adds a cache entry by grabbing memory from the free pool
// *                  and chaining it on to the appropriate place.
// *      outputs:
// *      _newEntry          ;address of the new entry we added
// *      inputs:
// *      _thisEntry         ;address of current entry
// *      _chaining          ;=1 -> chain on to the end
// *                        ;=0 -> just replace current entry with
fields
// *                        (if we are at the first element in
the cache)
// *      example usage:
// *      addCacheEntry[newAddr, currAddr, 1]
// *
#macro addCacheEntry[_newEntry, _thisEntry, _chaining]
// * we're allowed to play with xfer registers 5,6,7 only
// *
.local next_free_ptr addrptr

alu[--, --, B, _chaining]
br=0[new_entry_not_required#], defer[1]
alu[_newEntry, --, B, _thisEntry]

```

42

```

immed[addrptr, 0x2000]

sram[read, $xfer5, addrptr, 0x0, 1], ctx_swap
alu[_newEntry, --, B, $xfer5] ;our entry can be stored at the next free location
alu[next_free_ptr, _newEntry, +, 0x6] ;read in the next free entry
sram[read, $xfer5, next_free_ptr, 0x0, 1], ctx_swap ;so we can update the freeptr
alu[next_free_ptr, --, B, $xfer5] ;(just make sure that
alu[$xfer5, --, B, next_free_ptr] ;write bank = read bank)
sram[write, $xfer5, addrptr, 0x0, 1], ctx_swap ;finally update the free pointer for future use
// *
// * do the fwd/reverse chaining
// *
alu[addrptr, _thisEntry, +, 0x6]
alu[$xfer5, --, B, _newEntry]; writebank = nextptr
sram[write, $xfer5, addrptr, 0x0, 1], ctx_swap; write the next ptr

alu[addrptr, _newEntry, +, 0x7]
alu[$xfer5, --, B, _thisEntry]; writebank = prevptr
sram[write, $xfer5, addrptr, 0x0, 1], ctx_swap; write the prev ptr

.endlocal // next_free_ptr addrptr

new_entry_not_required#:

.local addrptr outbyte1
alu[addrptr, --, B, _newEntry]; copy the new entry's address
// *
// * WORD 0:
// * char bitvector; (1 byte)
// * char protocol; //(1 byte)
// * short int counter; //(2 bytes) - packet counter.

```

43

```

// *
immed[outbyte1, 0x0]
alu_shf[outbyte1, outbyte1, +, 0x1, <<31]; set the 'used' bit
alu_shf[outbyte1, outbyte1, OR, ip_prot, <<16]; copy the protocol
alu[$xfer5, --, B, outbyte1]
sram[write, $xfer5, addrptr, 0x0, 1], ctx_swap
// *
// * WORDS 1/2:
// * int src_addr; //(4 bytes)
// * int dst_addr; //(4 bytes)
// *
alu[addrptr, addrptr, +, 1]
alu[$xfer5, --, B, ip_sa]
alu[$xfer6, --, B, ip_da]
sram[write, $xfer5, addrptr, 0x0, 2], ctx_swap
// *
// * WORD 3:
// * short int src_port; //(2 bytes)
// * short int dst_port; //(2 bytes)
// *
alu[addrptr, addrptr, +, 2]
alu_shf[outbyte1, 0x0, +, ip_source_port, <<16]
alu[outbyte1, outbyte1, OR, ip_dest_port]
alu[$xfer5, --, B, outbyte1]
sram[write, $xfer5, addrptr, 0x0, 1], ctx_swap
.endlocal // addrptr outbyte1 outbyte2

#endm

```

44

```
#endif // FIREWALL_UC
```

### **test init.cpp**

```
/*
  to do: v1
  1) how many entries in array (Hemant will let us know)
  2) First_free_cache_ptr = 0x2000
  3) Cache base = 0x2001
  4) initialize double linked list on initialization
  5) each entry in array should have next ptr initialized to NULL
  6) NOTE: cache entry = 8* 32 (bits)
  7) for polling
  - #define 2 32-bit words (1 words is for the function call (i.e. lookup cache entry =1 , remove cache
  entry = 2)
  - we signal to Hemant when we're done.

  to do:v2
  1) finalizing the polling()
  2) Finishing the writing cache

*/

/*****INCLUDE
*****/
#include "c:/tornado/target/h/stdio.h"
#include "c:/ixpl200/include/uclo.h"
#include "c:/ixpl200/include/libd.h"
#include "c:/tornado/target/h/stdlib.h"
#include "c:/ixpl200/include/hal_sram.h"
#include "c:/ixpl200/include/mem_map.h"
#include "c:/tornado/target/h/inetlib.h"
```

```

#include "c:/tornado/target/h/taskLib.h"
#include "unistd.h"
#include "config_1200.h"
#include "net_app.h"
#include "ueng.h"
#include "netinet/in.h"
#include <iostream>
#include <string.h>
/*****INCLUDE*****/
*****/

/*****DEFINE*****/
*****/
#define HASH_ARRAY_SIZE 256
/* Hemant will set this address to contain FREELIST_START */
#define CACHE_FREELIST_PTR 0x2000
/* 256 hash array entries, each entry is 32 bytes (256 bits)
   each number in hex is 32 bits
   each cache entry is 8 numbers in hex
   256 * 8 = 2048
   2048 in hex = 800
   2001 + 800 = 2801
*/

#define SRAM_CACHE_START 0x2001
#define SRAM_CACHE_END 0x2801

/* 6911 possible cache entries */
#define FREELIST_START 0x2802
#define FREELIST_END 0xFFA2

```

47

```

/* each policy takes 4 hex numbers
   We set a max of 50 policies
   --> 50 * 4 = 200 hex numbers
   -->200 dec = C8 hex
*/

#define SRAM_POLICY_START 0x10001
#define SRAM_POLICY_END 0x100C9

/* Each filter takes 4 hex numbers */

#define SRAM_FILTER_START 0x100CA
#define SRAM_FILTER_END 0x15000

#define SRAM_LOG_START 0x15001
#define SRAM_LOG_END 0xFFFF5

/* Used for Polling */
/* Logic: If we have only 2 registers (1 for the func, 1 for the param),
then we run the risk of head of line blocking. Peter suggests to have
> 1 place to poll so Hemant can dump packets (that are the beginning of a flow)
if they happen to come in a burst
- If have chosen 5 such polling places. That can be increased/decreased.
*/

#define ORDER_ADDRESS

#define POLL_ADDRESS 0x1FFE0 /* Using that thread 1 uses Poll Address, and the paramter uses
                           the address next over*/
#define NUM_THREADS 12

```

48



```

/* well-known values that tell the control plane what function to call */
#define DO_CACHE_VALIDATION 1
#define DONE_REQUEST 0
#define DO_CACHE_REMOVAL 2

#define SRC_PORT 1 // (16 bits)
#define DST_PORT 2
#define SRC_ADDR 3 // (32 bits)
#define DST_ADDR 4
#define PROTOCOL 5 // (8 bits)

/*****DEFINE*****/

/*****EXTERN
*****/
//Function declarations
static SramUnit* sram;
int number_policies;
#if defined(__cplusplus)
extern "C"
{
//extern

extern int init();
extern int initialize_poll_sram();

extern int initialize_log_sram();
extern int initialize_policy_table();
extern int initialize_cache_table();
extern int polling();
extern int packet_validation(struct cache_entry&, int *, int);
extern int policy_table_lookup(); // return 0 if allowable; -1 to deny.

extern int policy_decision(char*);
extern struct Admin_struct read_policy_pkt(char*);
extern int add_policy(struct Admin_struct &); //add to policy table; return -1 if error
extern int flush_policy(); //used to flush policy table and all filters
extern int flush_cache(); //flush the cache if new policy is added
extern int write_cache(); //called within policy_table_lookup
extern int write_log(); //called within policy_table_lookup
extern int packetize_log_table(); //called within policy_table_lookup
//if after writing a log, the log table is full,
//packetize the table immediately
extern int send_log(); //called by packetize_log_table
extern int stringToHex(char *number);
extern int add_filter(struct Filter *);
extern int policy_add(int,int,int);
extern int validate_flow(int,int,short,short,int);
extern int Admin_read();
}
#endif
/*****EXTERN
*****/
struct Filter{
    int policy_id; //which policy it belongs to
    int filter_id; //type of filter (e.g src port)
    int filter_value; //either contains the value or the address (pointer) to the value

```

```

    int filter_mask; //applies if masks are necessary (we know if its necessary by looking at the
    //policy_id
};

struct Admin_struct{
    int whattodo;
    int numfilters;
    struct Filter filts[6];
};

/* 12 bytes total
   12 * 8 = 106 bits
*/

/* We don't need a previous pointer in policy because we flush the policy table upon deletion of a
policy
*/

struct policy{
    int policy_id;
    int num_filters;
    int next;
    int forward; // 1 is forward/ 0 is drop
};

//= 8 * 4 bytes
struct cache_entry{
    /* 1st LSB - drop or forward packet
       2nd LSB - is logging required
    */

```

51

```

    1st MSB - null entry or used
    6 bits are still free to be used
*/
char bitvector; //(1 byte)
char protocol; //(1 byte)
short int counter; //(2 bytes) - counts # of packets in flow
int src_addr; //(4 bytes)
int dst_addr; //(4 bytes)
short int src_port; //(2 bytes)
short int dst_port; //(2 bytes)
int string_ptr; //(4 bytes) - ptr to the string we want to search for in memory
int func_ptr; //(4 bytes) - ptr to the function we give the string to, to search for in the packet
int next; //(4 bytes)
int prev; //(4 bytes)
//we will next extra bits to decide what extended functions to call
};

//we log at the end of the flow
struct log{
    short int src_port; //(2 bytes)
    short int dst_port; //(2 bytes)
    int src_addr; //(4 bytes)
    int dst_addr; //(4 bytes)
    char protocol; //(1 byte)
    short int counter;
};

int init(){
    /* Note: srv.sin_addr.s_addr is a 32-bit IPv4 address that is
       network byte ordered
    */

```

52

```

int x=0;
struct sockaddr_in srv;
sram = SRAM_Attach();
initialize_policy_table();
initialize_cache_table();
initialize_log_sram();
initialize_poll_sram();
/* Initialize the Free list head ptr */
sram->write(CACHE_FREELIST_PTR, FREELIST_START);

/* The following is a hack to manual configure pre-hashed values in to
the cache array - note that the base of the cache must be added to these
values
*/
// Hash values:
// 1 0x518

//Do initialization
//1st initialize the filters/policies
//2nd initialize hash array/initialize free cache entries to point to the next guy
//3th initialize log sram

// polling();
}

int initialize_poll_sram(){

```

53

```

    unsigned int i;
    for(i = POLL_ADDRESS; i <= NUM_THREADS*2+POLL_ADDRESS; i++){
        sram->write(i,0);
    }
}

int initialize_log_sram(){
    unsigned int i;
    for(i = SRAM_LOG_START; i < SRAM_LOG_END; i++){
        sram->write(i, 0);
    }
}

int initialize_policy_table(){
    int i;
    unsigned int addr = SRAM_POLICY_START;
    static SramUnit* sram;
    printf("Clearing Policy Table\n");
    sram = SRAM_Attach();

    for(i=addr; i < SRAM_POLICY_END; i++){
        sram->write(i, 0);
    }
    for (i = SRAM_FILTER_START; i < SRAM_FILTER_END; i++){
        sram->write(i,-1);
    }
    number_policies =0;
}

```

54

```

int initialize_cache_table() {
    // static SramUnit* sram;
    int flag = 0;
    unsigned int i;
    unsigned int nextaddr = 0;
    /*Initialize the entire cache structure (including the free list
    and the hash array) to be 0
    */

    for(i =SRAM_CACHE_START;i<SRAM_CACHE_END;i++) {
        sram->write(i,0);
    }
    for(i =FREELIST_START;i<FREELIST_END;i++) {
        sram->write(i,0);
    }

    /* Start 256 entries after the cache_start, and initialize the
    free list such that each entry points to the next entry in
    the free list
    */
    for(i= FREELIST_START;i < FREELIST_END;i = i+8){
        /* Write the previous pointer address into memory */

        if(i != FREELIST_START){
            sram->write(i+7,i-8);
        }
        if(i != FREELIST_END -8){
            /* Write the next pointer address into memory */
            sram->write(i+6,i+8);
        }
    }
}

```

55

```

}

int polling(unsigned int maxTimer){
    unsigned int timer=0;
    unsigned int i;
    unsigned int value;
    unsigned int ptr;
    struct cache_entry entry;
    int int_mask,size;
    unsigned int data;
    unsigned int addr;
    int order[50];
    char all_ones = (char)-1;
    unsigned int forward;
    printf("Polling-----\n");
    size =5;
    for(i=0;i<size;i++){
        order[i]=i;
    }
    while(timer < maxTimer){
        timer++;
        taskDelay(2);
        for(i = POLL_ADDRESS; i <= 2*NUM_THREADS + POLL_ADDRESS; i+=2){
            sram->read(i, &value);
            switch(value){
                case DO_CACHE_VALIDATION:
                    printf("DOING CACHE VALIDATION -----\\n\\tThread: %d",i);
                    addr = i+1;
                    /* Find the cache entry that the addr points to which starts at ptr */
                    sram->read(addr, &ptr);
                    sram->read(ptr, &data);

```

56

```

printf("\n\tLocation: %x\n", ptr);
/* Fill up the cache entry before validating */
int_mask = 0xFF000000;
entry.bitvector = data & int_mask;
data = data << 8;
entry.protocol = data & int_mask;
/* we can ignore reading the counter since we're doing validation*/

sram->read(ptr+1, &data);
entry.src_addr = data;
sram->read(ptr+2, &data);
entry.dst_addr = data;
int_mask = 0xFFFF;
sram->read(ptr+3, &data);
entry.dst_port = data & int_mask;
data = data >> 16;
entry.src_port = data & int_mask;
/* We dont need to read the next or prev ptrs in the validation */
forward = packet_validation(entry,order,size);
sram->read(ptr,&data);
data = data | (forward << 24);
sram->write(ptr,data);
/* Now write the cache_entry back into the SRAM */
// write_cache(&entry, ptr);
/* Let the Data plane know that we're done with this entry */
sram->write(i, 0);
sram->write(addr, 0);

break;
case DO_CACHE_REMOVAL:
/* Remove Entry for cache - then send the log of the entry */

```

57

```

break;
case DONE_REQUEST:
break;
default:
break;
}
}
}
/*
#define SRC_PORT 1 (16 bits)
#define DST_PORT 2
#define SRC_ADDR 3 (32 bits)
#define DST_ADDR 4
#define PROTOCOL 5 (8 bits)
*/
// = 8 * 4 bytes

int does_match(struct cache_entry& entry,unsigned int i) {
unsigned int type;
unsigned int val;
unsigned int mask;
sram->read(i+1,&type);
sram->read(i+2,&val);
sram->read(i+3,&mask);
switch(type){
case SRC_PORT:
cout << "Matching SRC_PORT" << endl;
if(((entry.src_port & (short)mask) ^ ((short)val & (short)mask))!=0) {
return 0; //fails
}
}
}

```

58

```

break;
case DST_PORT:
    cout << "Matching DEST_PORT" << endl;
    if(((entry.dst_port & (short)mask) ^ ((short)val & (short)mask))!=0) {
        return 0; //fails
    }
break;
case SRC_ADDR:
    cout << "Matching SRC_ADDR" << endl;
    if(((entry.src_addr & mask) ^ (val & mask))!=0){
        return 0; //fails
    }
break;
case DST_ADDR:    cout << "Matching DST_ADDR" << endl;

    if(((entry.dst_addr & mask) ^ (val & mask))!=0){
        return 0; // fails
    }
break;
case PROTOCOL:
cout << "Matching Protocol " << entry.protocol << " (entry) with " << val << " (filter)" << endl;
    if(((entry.protocol & (char)mask) ^ (val& mask))!=0){
        return 0; //fails
    }
break;
default:
    printf("We shouldn't be here\n");
    return 0;
break;
}
cout << "matched" << endl;

```

59

```

    return 1;
}
int validate_flow(int src_addr,int dst_addr,short src_port,short dst_port, int size) {
    struct cache_entry entry;
    int order[50];
    int i =0;
    entry.src_addr = src_addr,
    entry.dst_addr = dst_addr;
    entry.src_port = src_port;
    entry.dst_port = dst_port;
    for (i = 0; i<size;i++) {
        order[i]=i;
    }
    printf(" Pass? %d",packet_validation(entry,order,size));
}
int packet_validation(struct cache_entry& entry,int order[50],int size){
    unsigned int addr = SRAM_FILTER_START;
    int type;
    unsigned int policy[50] = {0};
    unsigned int policy_id;
    unsigned int i,j;
    unsigned int policy_addr = SRAM_POLICY_START;
    unsigned int num_filters,forward;
    int match =0;
    order[0]=0;
    for(i = addr; i < SRAM_FILTER_END;i=i+4) {
        sram->read(i,&policy_id);
        if(policy_id != -1) {
            cout << "Matching with "<< policy_id << endl;
            if(does_match(entry,i)){

```

60

```

        policy[policy_id]++;
    } else cout << " DOESN'T MATCH " << endl;
}
}
for(i = order[0],j=0;j<size;j++,i=order[j]) {
    //this needs to be updated
    sram->read(policy_addr+i*4+1,&num_filters);
    cout << i << endl;
    printf("%d\n",policy[i]);

    cout << num_filters << endl;
    if(policy[i]==num_filters){
        sram->read(policy_addr+i*4+3,&forward); //meets a rule
        cout << "Forward ? " << forward << endl;
        return forward;
    }
}
return 0;
}

/*
Take in a control packet that specifies a policy
Validate control packet first.
For each filter in the policy, call add_filter
*/
int add_filter(Filter* filter){
    unsigned int i;
    //static SramUnit* sram;

```

61

```

    if(!filter){
        return ERROR;
    }

    printf("Adding Policy\n");
    // sram = SRAM_Attach();
    unsigned int group_id;
    // start search from addr to SRAM POLICY END, jumping every i
    for(i=SRAM_FILTER_START;i<=SRAM_FILTER_END;i=i+4) {
        sram->read(i,&group_id); // find the empty slot
        if(group_id ==-1) { // write to the empty slot
            sram->write(i,filter->policy_id);
            sram->write(i+1,filter->filter_id);
            sram->write(i+2,filter->filter_value);
            sram->write(i+3,filter->filter_mask);
            break;
        }
    }
    return 0;
}
int add_policy(struct Admin_struct &policyitem){
    int i =0;
    for(i=0;i<policyitem.numfilters;i++){
        add_filter(&(policyitem.filts[i]));
    }
    sram->write(SRAM_POLICY_START+4*number_policies,policyitem.filts[0].policy_id);
    cout << "Number of filters " << policyitem.numfilters << endl;
    sram->write(SRAM_POLICY_START+4*number_policies+1,policyitem.numfilters);
    sram->write(SRAM_POLICY_START+4*number_policies+2,0);
    sram->write(SRAM_POLICY_START+4*number_policies+3,policyitem.whattodo);
    number_policies++;

```

62

```

}

int delete_policy(int policy_id) {
    // first clear the filter table of that policy, then relink the list
    unsigned int addr = SRAM_FILTER_START;
    unsigned int i = 0;
    unsigned int temp = 0;
    for(i=addr; i<SRAM_FILTER_END; i=i+4) {
        sram->read(i, &temp);
        if (temp == policy_id) {
            sram->write(i, 0);
            sram->write(i+1, 0);
            sram->write(i+2, 0);
            sram->write(i+3, 0);
        }
    }
    //flush the policy table?
}

int policy_add(int policy_id, int number, int forward) {
    int value[18] = {0};
    int i = 0;
    struct Admin_struct addingpolicy;
    sram = SRAM_Attach();
    cout << endl << endl << endl << endl;
    cout.flush();
    cout << "Type = #define SRC_PORT 1 // (16 bits)
             #define DST_PORT 2
             #define SRC_ADDR 3 32 bits

```

63

```

             #define DST_ADDR 4
             #define PROTOCOL 5
             " << endl;
    cout << " Policy ID: " << policy_id << endl;
    cout << " Filter Number: " << number << endl;
    cout << " Allow?: " << forward << endl;
    for(i=0; i<number; i++) {
        cout << "Filter type? " << endl;
        cin >> value[3*i];
        cout.flush();
        cout << value[3*i] << endl;
        cout.flush();
        cout << "Filter Value? " << endl;
        cin >> value[3*i+1];
        cout.flush();
        cout << value[3*i+1] << endl;
        cout << "Filter Mask? " << endl;
        cin >> value[3*i+2];
        cout.flush();
        cout << value[3*i+2] << endl;
        addingpolicy.filts[i].policy_id = policy_id;
        addingpolicy.filts[i].filter_id = value[3*i];
        addingpolicy.filts[i].filter_value = value[3*i+1];
        addingpolicy.filts[i].filter_mask = value[3*i+2];
    }
    addingpolicy.numfilters = number;
    addingpolicy.whattodo = forward;
    add_policy(addingpolicy);
}

int flush_cache() {

```

64



```

static SramUnit* sram;
printf("Flushing Cache\n");
// get sram pointers and re-initialize sram memory
sram = SRAM_Attach();

//let X be the number of bytes in each cache entry
for(int i = SRAM_CACHE_START; i < SRAM_CACHE_END; i++){
    sram->write(i, 0);
}

}

int write_log() {
    //called by policy_table_lookup after deciding if there is a violation
    //takes in a packet descriptor
    //need timestamp functionality (ask Hemant)
    //5-tuple
}

int packetize_log_table() {
}

int send_log() {
}

int CaddEntry(
    unsigned int address,
    unsigned int bitvector,
    unsigned int source,
    unsigned int dest,

```

65

```

    unsigned int ports,
    unsigned int param,
    unsigned int func,
    unsigned int next,
    unsigned int prev)
{
    unsigned int aptr;
    static SramUnit* sram;
    sram = SRAM_Attach();

    sram->write(address, bitvector);
    sram->write(address + 1, source );
    sram->write(address + 2, dest);
    sram->write(address + 3, ports);
    sram->write(address + 4, param);
    sram->write(address + 5, func);
    sram->write(address + 6, next);
    sram->write(address + 7, prev);
    return 1;
}

int Admin_read() {
    FILE* fileptr;
    struct Admin_struct new_policy;
    int i = 0;
    char* s = malloc(3*sizeof(struct Admin_struct));
    char* ptr;

    if((fileptr = fopen("packet_output" , "r")) == NULL){
        printf("Cannot open packet_output\n");
    }
}

```

66

```

// char *fgets(char *s, int size, FILE *stream);

if(fgets(s, 3*sizeof(struct Admin_struct), fileptr) == NULL){
    printf("Could not read the packet in\n");
}

//printf("This is the string we just read in: \n %s \n", s);

/* Call Strtok to parse values in the packet */
// char *strtok(char *s, const char *delim);

ptr = strtok(s, " ");
new_policy.whattodo = atoi(ptr);
printf("What to do = %d\n", new_policy.whattodo);

ptr = strtok(NULL, " ");
new_policy.numfilters = atoi(ptr);
printf("Numfilters = %d\n", new_policy.numfilters);

for(i = 0; i < new_policy.numfilters; i++){
    ptr = strtok(NULL, " ");
    new_policy.filts[i].policy_id = atoi(ptr);
    printf("Policy_id = %d\n", new_policy.filts[i].policy_id);
    ptr = strtok(NULL, " ");
    new_policy.filts[i].filter_id = atoi(ptr);
    printf("Filter_id = %d\n", new_policy.filts[i].filter_id);

    if(new_policy.filts[i].filter_id == 3 || new_policy.filts[i].filter_id == 4){
        ptr = strtok(NULL, " ");
        sscanf(ptr, "%x", &new_policy.filts[i].filter_value);
        printf("IP address values is %x \n", new_policy.filts[i].filter_value);
    }
}

```

67

```

ptr = strtok(NULL, " ");
sscanf(ptr, "%x", &new_policy.filts[i].filter_mask);
printf("IP mask values is %x \n", new_policy.filts[i].filter_mask);
}
else{
    ptr = strtok(NULL, " ");
    new_policy.filts[i].filter_value = atoi(ptr);
    printf("IP port/protocol value is %x \n", new_policy.filts[i].filter_value);
    ptr = strtok(NULL, " ");
    new_policy.filts[i].filter_mask = atoi(ptr);
    printf("IP port/protocol mask is %x \n", new_policy.filts[i].filter_value);
}
}
add_policy(new_policy);
free(s);
}

```

68