# The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention

Nicholas Weaver
ICSI
nweaver@icsi.berkeley.edu

Vern Paxson
ICSI
vern@icir.org

Jose M Gonzalez
ICSI
chema@icsi.berkeley.edu

## ABSTRACT

Today's network intrusion prevention systems (IPSs) must perform increasingly sophisticated analysis—parsing protocols and interpreting application dialogs rather than simply searching for signature strings—for which the necessary algorithms defy full implementation in hardware, being much more readily implemented using general-purpose CPUs. Yet the performance of such CPUs increasingly lags behind that necessary to process today's high-rate traffic streams.

We observe that in many environments much of the traffic comprising a high-volume stream can, after some initial analysis, be qualified as "likely uninteresting." Thus, we would like a means by which we can couple a general-purpose CPU with a specialized hardware element such that only the hardware element processes the bulk of the bytes in a network stream, while the CPU can still inspect those elements of network flows deemed germane for security analysis.

To this end, we have developed an in-line, FPGA-based IPS accelerator, the *Shunt*, using the NetFPGA2 platform. The Shunt maintains several large state tables indexed by packet header fields, including IP/TCP flags, source and destination IP addresses, and connection tuples. The tables yield decision values the element makes on a packet-by-packet basis: forward the packet, drop it, or divert it through the IPS. By manipulating table entries, the IPS can specify the traffic it wishes to examine, directly block malicious traffic, and "cutting through" traffic streams once it has had an opportunity to "vet" them, all on a fine-grained basis. We base our design on a novel series of caches, with a "fail safe" miss policy, coupled to a host PC to handle both cache management and higher level IPS analysis. The design requires only 2 MB of SRAM for its extensive caches, and can support four Gbps Ethernets on a single Virtex 2 Pro 30.

## 1. INTRODUCTION

Stateful, in-depth, in-line traffic analysis for intrusion detection and prevention is growing increasingly more difficult as the data rates of modern networks rise. One point in the design space for high-performance network analysis—pursued by a number of commercial products—is the use of sophisticated custom hardware. For very high-speed processing, such systems often cast the entire analysis process in ASICs.

In this work we pursue a different architectural approach, *Shunting*, which marries a conceptually quite simple hardware device with an Intrusion Prevention System (IPS) running on commodity PC hardware. Our goal is to keep the hardware both cheap and readily scalable to future higher speeds; and also to retain the unparalleled flexibility that running the main IPS analysis in a full general-computing environment provides.

The Shunting architecture uses a simple in-line hardware element that maintains several large state tables indexed by packet header fields, including IP/TCP flags, source and destination IP addresses, and connection tuples. The tables yield decision values the element makes on a packet-by-packet basis: forward the packet, drop it, or divert ("shunt") it *through* the IPS (the default). By manipulating table entries, the IPS can, on a fine-grained basis: *(i)* specify the traffic it wishes to examine, *(ii)* directly block malicious traffic, and *(iii)* "cut through" traffic streams once it has had an opportunity to "vet" them, or *(iv)* skip over large items within a stream before proceeding to further analyze it.

The efficacy of this approach depends on the degree to which the IPS can "shed load" by identifying large-volume subsets of traffic that it can safely skip. Opportunities for these arise, for example, due to encrypted SSH and SSL sessions, for which the IPS can only usefully analyze the initial negotiation process, or HTTP sessions that transfer large items such as images or movies. While such flows make up only a small proportion of the connections seen on a network link, in many environments they make up a large fraction of the bytes, due to the widely documented "heavy-tailed" nature of network traffic [11, 12, 6, 23, 22, 5].

Reference [7] presents the overall architecture and evaluates it in detail. In this paper we focus on our subsequent efforts to design and implement an FPGA-based realization of Shunting. The device can operate in-line on a network link, facilitate switch-based LAN monitoring, or as a load balancer for a clusterized Intrusion Detection System (IDS).

We implemented the Shunt on top of the NetFPGA2 [19] research and education platform. This platform contains four Gbps Ethernets, two 2MB SRAMs, and a Virtex 2 Pro 30 FPGA, all located on a single PCI card which fits inside a standard host. We began by modifying an existing design, a 4-port Ethernet NIC that used only one of the SRAMs as a buffer, to create *RNET*, a framework for in-place packet manipulation and routing. The RNET framework provides a shim between each receiving MAC and the main controller. Each shim buffers one packet at a time, and can manipulate the packet before routing it to any output MAC or to the host.

We then built the Shunt using the RNET infrastructure. The design centers around two primary caches: a connection cache of $2^{16}$ entries and an IP cache of $2^{16}$ entries. The connection cache uses multi-location associativity, a variant of a design by Song et al [16], where two separate hash functions are used to provide two different possible locations for each entry, to allow the host to move entries to free up space. The IP cache is a multilocation *permutation* cache: rather than using a conventional tag/index structure, we use a 32-bit block cypher to encrypt the IP address to create the tag and index, which can result in a 50% savings in memory by allowing part of the tag to be implicitly stored.

For both these caches, we encode an action (shunt, sample, forward, or drop) and a priority. Additional rules also encode actions and priorities based on fixed-header fields. The hardware selects the highest priority match, or, if no match, defaults to shunting the packet to the host. Additionally, the rules for connections can have an optional record that specifies an alternate destination MAC, VLAN, and/or output port to which connections should be forwarded, and an alternate rule that applies if the TCP sequence number is within a specified range (to skip over items within TCP streams).

The resulting design requires 21,200 4-LUTs for logic, 2,770 LUTS for routethrough (87 % of the available resources), and 135 out of 136 available BlockRams. It requires only 41 cycles to make a decision when unloaded (and no more than 101 cycles when fully loaded), running at 62.5 MHz. Packets that pass directly through the hardware path see only 5 $\mu$s of additional network latency.

We begin in Section 2 with a survey of related work and a discussion of the NetFPGA board. Section 3 discusses our *RNET* addition to the NetFPGA firmware, designed as a general platform for network processing. Section 4 discusses our overall hardware architecture and how this architecture realizes our desired tasks. We then in Section 5 present multi-location associativity, which allows us to more efficiently utilize our caches. Section 6 discusses permutation caches, a space-saving technique we employ for the IP cache that doubles the available capacity when associating small values with 32-bit keys.

Section 7 details the actual implementation used for the Shunt's caches and general operation. We evaluate the Shunt in Section 8, with present conclusions in Section 9.

## 2. RELATED WORK

The NetFPGA version 2 [19] was developed by McKeown et al as a platform for both research and network experimentation. It consists of a single Virtex 2 Pro (V2Pro 30, speedgrade 5) FPGA, two 2 MB (512kx36) SRAMs, a quad-port Gigabit Phy, on a PCI card, with the PCI interface implemented in a Spartan II FPGA.

Additionally, the NetFPGA platform has three significant pieces of code associated with it: **UNET**, **CNET**, and assorted software tools. UNET is a generic design for student projects. It consists of a single Ethernet MAC and associated control logic, including memory interfaces. CNET implements a quad port Ethernet NIC, complete with a host DMA interface and an Ethernet driver. The NetFPGA tools include a configuration downloader which allows the NetFPGA to be reconfigured, a driver for the NetFPGA board, and an API to peek and poke both status registers and the two onboard SRAMs.

There has been considerable hardware designed for intrusion detection. Several projects have implemented partial or complete regular-expression based rulesets [17, 18], while a large number of commercial intrusion detection and prevention systems claim to use hardware acceleration [**?**]. In particular, [17] also takes a preprocessor approach, but it only implements the static ruleset to filter out uninteresting communication, without the dynamic, per connection control we provide.

The most closely related work to ours is SPANIDS [15]. SPANIDS is a front-end load balancer for parallel intrusion detection applications, which uses a series of hash functions to determine which analyzer should recieve a packet. The SPANIDS load balancer receives packets on a single Gigabit ethernet, rewrites the destination MAC address based on a series of hash functions, and outputs the packet. SPANIDS uses four small hash tables of 4096 entries to determine where to route the packet, with these tables implemented in on-chip SRAM. Unlike the Shunt, SPANIDS cant precisely route individual connections, only hash-based aggregates of connections to balance flows and prevent hotspots.

## 3. RNET

Although the NetFPGA platform is designed for easy extensibility as part of class projects, the design framework for class projects, **UNET**, was not suitable for our purposes. The UNET design only activates a single ethernet and, more importantly, lacks a DMA interface to the host. Instead, we began with the NetFPGA **CNET** design, which implements a 4 port Ethernet NIC, complete with DMA packet transfer and a Linux driver, as the starting point for our work.

We wished to create a general framework for packet processing, not just an application-specific instance. We observed that many network processing tasks have the following properties: Packets are read in from an Ethernet, may be modified in-place (such as changing MAC addresses and IP TTLs, or decrypting payload data), and then written out to an appropriate MAC or forwarded to the host for further analysis. This analysis may also need some reasonably-sized shared memory, and an easy interface to the host if a packet's operation is more complex than what the hardware can support.

Thus we created a small module, a shim, that fits between each receiving MAC and the memory arbiter which processes packets destined for the host. The purpose of the shim is to read in a packet from the MAC (on a 32 bit, 62.5 MHz bus) into a buffer, process the packet with user-specified logic, and then forward the packet to its appropriate destinations. The shim has to wait for the arbiter to complete the transaction if the packet is forwarded to the host, but once the shim begins writing the packet to the output MACs it begins reading the next packet. This can allow the shim to operate near or at gigabit rates if the packets don't need to be redirected to the host.

In addition to the shim, other portions of the design needed to be modified. In order to prevent contention, each MAC send path was given 4 additional FIFOs, for a total of five. These FIFOs are served in a round-robin fashion. As a result, the RNET framework implements a full 5x5 crossbar, with the 5x4 crossbar to the output MACs having independent buffers for each path.

Finally, the memory controller for the second SRAM was modified to provide 5 read and write ports. Each shim is given a single pipelined read and write port to this shared 256kx36 SRAM. Again, these requests are also serviced in a round-robin fashion, and the memory controller is pipelined for greater throughput.

The resulting framework (Figure 1) can then be used to implement a large class of packet processors. As the packet is read from the MAC, at 2 Gbps, the headers are extracted and the packet written into a BlockRAM buffer. Once the packet is fully read in, any user logic can modify the packet in place and decide where the packet should be routed. The biggest limitation on RNET-based designs is that effectively all BlockRAMs are used in the Virtex 2 Pro 30, mostly because of the 20 BlockRAMs required simply for the output buffers for the MAC output crossbar and the other 4

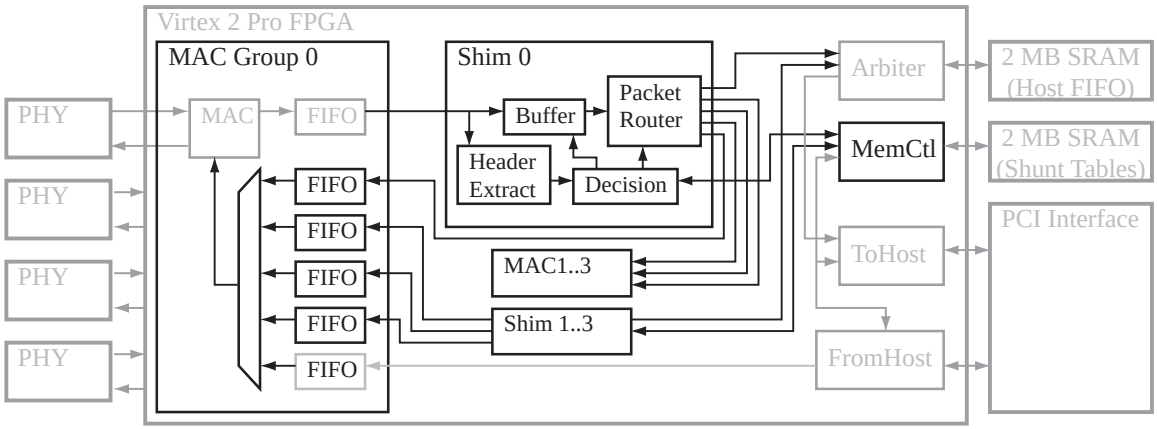


**Figure 1: The RNET structure. Items in gray are carried over from the CNET infrastructure unchanged.**

BlockRAMs for the packet buffer in the shim, which are on top of the already significant buffering used in the base CNET design.

## 3.1 Click Interface

One other addition was an interface to allow the Click [9] router framework to access NetFPGA resources. Click is a C++ framework for writing software routers and software packet processing elements which runs on Linux.

For packets being passed to and from the host, using the CNET driver, Click treats the NetFPGA like any other Ethernet, with each port having a unique Ethernet name. Click can read and write approximately 20k packets per second through this interface.

To access control information, including both NetFPGA status registers and the shared SRAM memory, a peek/poke interface is provided, which can allow Click elements to write and read memory state. In particular, the 2 MB shared SRAM can be both read and written by programs written in Click to facilitate communication with the shims, without needing to modify or add status registers contained in the CNET infrastructure.

# 4. THE SHUNT'S ARCHITECTURE

The Shunt is designed to accelerate three separate-but-related intrusion detection tasks: in-line operation (necessary for intrusion prevention), LAN operation, and IDS load balancing. We have designed the Shunt to perform in all these environments using a single common hardware design.

The key to the Shunt's operation is its ability to act as a programmable, priority based filter. For each packet received, the Shunt examines the layer 3 and layer 4 headers incruding the source IP, the destination IP, and the connection 5-tuple (source IP, destination IP, IP protocol, source port, and destination port), to find the appropriate 2 bit action (forward the packet onward, drop the packet, forward the packet and sample with a specified probability, or Shunt the packet to the host for further examination). Each matching rule also has a 3 bit priority (values 0 to 7), with the highest matching rule being selected, and a 3 bit sample schedule if the selected action is sample.

The header examination uses a set of static rules. Non-IP packets will always be Shunted to the host. Likewise, packets which are IP fragments (which can be used for evasive purposes), contain IP options, or are TCP connection delimiters (SYNs, SYN/ACKs, FINs, or RSTs) are Shunted to the host with priority 3.

In contrast, both the connection rules and IP rules are programmable. The Shunt looks up the source IP, the destination IP, and the connection 5-tuple. The IP lookup just involves finding a matching action and priority. Connection lookup, however, can also involve an optional record stored in a separate table. This optional record can specify a different destination for the packet, both in terms of MAC address and VLAN tag and can also specify an alternate action if the packet's TCP sequence number is within a specified range.

The goal of the alternate record's sequence skipping is to enable the IPS to skip over a predefined "less interesting" range of traffic. For example, in an HTTP stream, a large embedded image is of little interest to most IPSs. By using the header to determine the length of the image, the sequence-skipping can be used to have the Shunt directly forward the image, while ensuring that the subsequent traffic will still be directed to the host for detailed examination.

## 4.1 In-line Operation

For in-line operation, the Shunt (and associated IPS) are protecting an institution from external threats, by filtering all traffic on the wide area network (WAN) link or links. To protect a Gigabit link, the Shunt will need to be placed in-line, with one port for the LAN side and another port for the WAN side. In this mode, a single NetFPGA board can process two Gigabit WAN links.

In this mode, the Shunt's role is to act as a front end filter for an IPS running on the Shunt's host. When the IPS determines that a particular connection doesn't significantly benefit from deeper inspection, it will place a *forward* rule for this connection. Any subsequent traffic will be directly routed from the input Ethernet to the output, without loading the host. Likewise, if the IPS detects that a host is behaving offensive in some way (attacking internal hosts, or attempting to disrupt the IPS itself with offensive traffic), it can institute a high-priority drop rule for traffic coming from this IP.

## 4.2 LAN Operation

For LAN operation, the Shunt's role is to isolate and control traffic passing between a large group of hosts, either for IPS operation [20, 21] or to implement LAN-based policy control [3]. As such, *all* traffic on the local network must pass through the Shunt before proceeding to the destination.

There are two options for LAN traffic management: direct routing and VLAN rewriting. In direct routing, every host or group of hosts is on a separate Shunt port. In this context, the connection's

table's optional record for each destination will specify which output port a connection should be routed to.

For VLAN rewriting, every host is on its own unique VLAN, using untagged switch ports, with the Shunt on one or more VLAN trunks which can read and write every 802.1(q) VLAN on the switch with tagged packets. For VLAN rewriting, the optional record specifies the destination VLAN. Any packet which is forwarded will have its VLAN tag rewritten and then be reinjected back into the same port, where the switch will route the packet to its destination. This, naturally, requires switches which both support VLANs and maintain per-VLAN MAC caches.

One limitation for these LAN operations is that forward operations can only be encoded in the connection table's option field, not the IP table. As such, the IP table is effectively limited to simply blocking offensive sites, not whitelisting good traffic.

## 4.3 IDS Load Balancer

A final deployment we are pursuing is as a load balancer for a large cluster-based IDS deployment. In this role, four 1 Gbps tap ports are fed into a switch, with each data feed on a unique VLAN. The Shunt is placed on four ports on the switch, with each port configured with a VLAN trunk. Each port has access to one of the tap feeds, and can also write to an experimental VLAN which the cluster systems are on.

In this mode, the host only acts as a load balancer and manager, putting in appropriate forward rules for all active connections. This will require work on a per-connection basis, but allows significantly greater flexibility than static rules, as the load balancer can, on request, also drop connections, react to node failures, or redirect connections to different analyzers on demand. Since the architecture does not include the IDS nodes reinjecting traffic, the Shunt does not operate in-line, and thus does not support intrusion prevention; however, the approach could be extended to support such operation.

# 5. MULTILOCATION ASSOCIATIVITY

Traditionally, higher associativity caches will have multiple locations at the same index. Thus if the cache is 2-way associative, and $D_0$, $D_1$, and $D_2$ all hash to the same index, only two entries can actually be stored. In a multilocation associativity cache, multiple hash functions are used rather than one, and the value may be at the index specified by any hash function. This design, because it is a cache rather than a complete hash table (and therefore no chained buckets) is a simplification of the Fast Hash Table proposed by Song et al [16], which was itself based on bloom filters [2].

In a multilocation cache the multiple hash functions are used to specify multiple locations where an element might reside. Thus for a 2-way multilocation cache, two different hash functions are used and the data could be at either location. Unlike a bloom filter, however, the hashed locations are checked to see if the data is actually stored at the location. Otherwise, its a cache miss. This multilocation design allows the cache to be much more fully populated, as if $D_0$, $D_1$ and $D_2$ map to the same location with one hash function, it is highly unlikely that they map to the same location for the second hash function.

Additionally, when the cache evictions are rare and the cache is managed by a sophisticated processor, entries can be moved. By conducting a partial or complete depth-first-search, the cache manager can help ensure that the cache is completely full, a similar but simpler process to the pointer balancing in Song et al's Fast Hash Table design.

Figure 2 shows how location associativity can help better utilize the cache. As can be clearly seen, when the cache is only lightly occupied, the choice of associativity has little effective. But as the number of entries exceeds the size of the cache, the multilocation associativity helps considerably. Adding the search operation improves things even more.

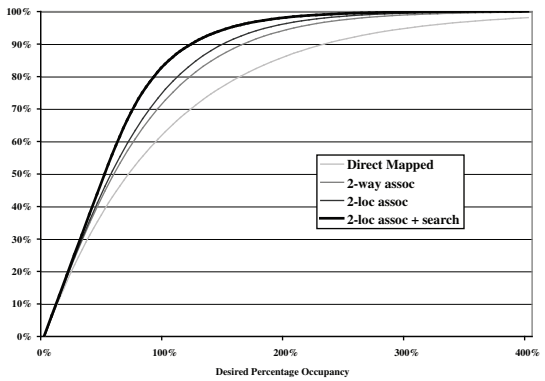


**Figure 2: The fraction of the cache actually filled as a function of the number of inserts attempted for four different strategies: Direct mapped, number of data inserted for 4 different strategies: Direct mapped, 2-way associative, 2-location associative, and 2-location associative with a small search to move entries. These results are for 100 runs each with a 64k entry cache.**

A small simulator was made to model this cache architecture, with 100 runs for each parameter selected. For Figure 2, a 64k entry cache was used. In particular, when 64k elements are hashed and inserted into the direct mapped cache (desired occupation equals 100% of cache size), on average, only 41,000 elements can actually reside in the cache. Changing to a two-way associative design allows 47,600 elements to be actually cached. A two-location associative cache, however, allows 49,800 elements. If a small, depth 5 search is used to find an appropriate entry when there is a conflict, now 54,800 elements can be stored, using the exact same amount of memory. Thus going with multiway associativity and a small search to find valid configurations can result in a 15% increase in cache utilization.

There are three disadvantages to this style of cache compared with a conventional cache. The first is that it requires multiple hash functions instead of a single function. This is simply because N-way location associativity requires N hash functions (or N different keys to the same keyed hash function). In general, this cost is usually low.

The second is that, if a search is employed, it can be costly as the cache fills up. Instead of simply checking $N$ locations to determine where to insert, a small search of depth $K$ will require checking $K * N$ locations. Thus some tradeoff will need to be determined where to halt the search and just evict an old entry instead.

The biggest concern is that an $N$ location cache requires accessing $N$ *different* memory locations. If the cache is stored in SRAM, and the entry size is equal to or greater than the word size, this is not an issue. However, for DRAM-based caches, or any memory system which fetches large groups of words at a time, location associative caches may not be effective.

# 6. PERMUTATION CACHES

One of the keys to the Shunt's design is efficient caches. With only a 2 MB working memory, we needed to develop efficient caches to maximize the hit rate while minimizing the working memory. In particular, for the IP cache, we used a variant on the permu-

tation cache we first described in our AC-TRW paper [10] which allows us to double the capacity of this cache.

A permutation cache is particularly well suited to associating a small amount of data (such as an 8 bit source action and an 8 bit destination action) to 32 bit keys. Rather than splitting the key into an index and tag, a permutation cache first encrypts the key using a block cypher where the block size is the same as the cache's key size. Since a block cypher is really a permutation, this guarantees that each key will map to a unique value. Additionally, by using a cypher with a random cryptographic key, the permutation is randomized and therefore can't be predicted by an attacker, avoiding the attack by Crosby et al [4].

The resulting 32 bit value is then split into an index and tag, with the index used to find the proper location and the tag verified when fetching the associtated value, just like a conventional cache. As a result, an encryption cache for 32 bit keys with $2^{16}$ locations only needs 16 bits of tag per entry, rather than the 32 bits required if a hash was used instead of a permutation.

We extended the permutation cache to support multilocation associativity by using different cryptographic keys. Instead of just storing the tag, an additional ID number is used to specify which cryptographic key was used for this entry. Thus with 2 keys, this becomes a 2-way multilocation associative encryption cache. If two values encrypt to the same location with one encryption key, they will, with very high probability, map to different locations when the other encryption key is used, giving a freedom for cache layout we discussed in Section 5.

## 6.1 Keyed Permutation

Due to the usage model in a permutation cache, we don't need a cryptographically strong block cypher, we only need an efficient block cypher-like keyed permutation, one which requires only a small amount of FPGA resources and which can be computed in one or two clock cycles. Additionally, we need a 32 bit block cypher, while most block cyphers operate on 64 or 128 bit blocks.[1] We also desire a 64 bit key, which allows a large amount of entropy to be injected into the permutation.

Thus we committed the classic cryptographic sin and developed our own 32 bit keyed permutation specifically for use in 32 bit permutation caches. Our goal was to have a single round with a reasonable amount of mixing which can be efficiently implemented on a 4-LUT based fabric as an S/P (Substitution/Permutation) network. Thus the primitives we used are 4-bit S-boxes (from the Serpent [1] block cypher), byte addition, fixed rotation, and 4-input XOR.

The initial input first passes through the initial S-boxes (based on the Serpent [1] 4-bit S-boxes). The resulting output bytes are rotated and are bytewise added to the first 32 bits of the key. The resulting word is then is passed through bytewise rotation and 4-input XORs, with each XOR combining 3 bytes of data with 1 byte of key. Finally, the data passes through one more round of S-boxes, and then a series of 4-input XORs and rotations. Figure 3 shows the complete pseudocode.

Although we never actually need to decrypt data for our application, the decryption process is effectively the opposite of the encryption process, with inverted operations in reverse order. Decryption requires exactly the same resources as encryption, and would be necessary for any application which needs to examine the entire contents of a permutation cache, rather than just looking up a specific entry.

[1]The RC5 [13] and RC6 [14] cyphers can be parameterized down to a 32 bit block, but they are not efficient in this application due to their multiround structure, choice of primitives including 16 bit variable rotations, and complex key schedule.

```
Keyed Permutation Function
Input: Din[31:0], K[63:0]
Output: Dout[31:0]
  B0 <= (SBoxA(Din[7 : 0])>>>2)  + K[ 7: 0]
  B1 <= (SBoxA(Din[15: 8])>>>3)  + K[15: 8]
  B2 <= (SBoxA(Din[23:16])>>>4)  + K[23:16]
  B3 <= (SBoxA(Din[31:24])>>>5)  + K[31:24]
  C0 <= B0 ^ (B1>>>1) ^ (B2>>>2) ^ K[39:32]
  C1 <= B1 ^ (B2>>>4) ^ (B3>>>5) ^ K[47:40]
  C2 <= B2 ^ (B3>>>7) ^ (C0>>>1) ^ K[55:48]
  C3 <= B3 ^ (C0>>>3) ^ (C1>>>4) ^ K[63:56]
  D0 <= SBoxC(C0)
  D1 <= SBoxC(C1)
  D2 <= SBoxC(C2)
  D3 <= SboxC(C3)
  E0 <= D0 ^ (D1>>>1) ^ (D2>>>5) ^ (D3>>>2)
  E1 <= D1 ^ (D2>>>2) ^ (D3>>>6) ^ (E0>>>3)
  E2 <= D2 ^ (D3>>>3) ^ (E0>>>7) ^ (E1>>>4)
  E3 <= D3 ^ (E0>>>4) ^ (E1>>>1) ^ (E2>>>5)
  Dout[31:0] <= {E3, E2, E1, E0}

SBoxA-> Apply Serpent SBox0 to upper 4 bits
        Serpent SBox1 to lower 4 bits
SBoxC-> Apply Serpent SBox2 to upper 4 bits
        Serpent SBox3 to lower 4 bits
```

**Figure 3: The pseudo-code for our keyed permutation (a simplified block cypher).**

| Start Address | End Address | Purpose |
|---|---|---|
| 0x00000 | 0x0FFFF | Staus Registers, Keys, Misc I/O |
| 0x10000 | 0x1FFFF | IP Address Cache, $2^{16}$ entries |
| 0x20000 | 0x3FFFF | Optional Records, $2^{15}$ entries |
| 0x40000 | 0x7FFFF | Connection Cache, $2^{16}$ entries |

**Table 1: The memory allocation used in the NetFPGA Shunt**

This design is very efficient when targeting an FPGA. All steps require only 32 LUTs each. Thus with two S-box steps (64 LUTs), the initial key addition (32 LUTs), the key-dependent mixing (32 LUTs), and the key independent mixing (32 LUTs), the total cypher only requires 160 LUTs. Given a registered input and only a single pipeline register on the output, this cypher runs at the target 62.5 MHz clock cycle on our Virtex 2 Pro FPGA, without needing placement directives.

Additionally we have deliberately designed it for hashing IP addresses. In this case, it is OK if the lower bits of the output are not as high quality as the upper bits, as it is the upper bits which are used as the index for looking up entries. As a result, for both the computation of the C and E words, the feedback loop causes the upper bytes to be more affected by all input data and key bits.

# 7. SHUNT CACHES

For the actual implementation of the Shunt, we needed to fit all the caches into the single 512kx36 (2 MB) second SRAM on the NetFPGA board. Table 1 summarizes our memory allocation. We reserved the locations 0x0000 to 0x0FFFF (the first $2^{16}$ addresses) for miscelanious I/O, including status registers, debugging information, and the two permutation keys which are written by the host.

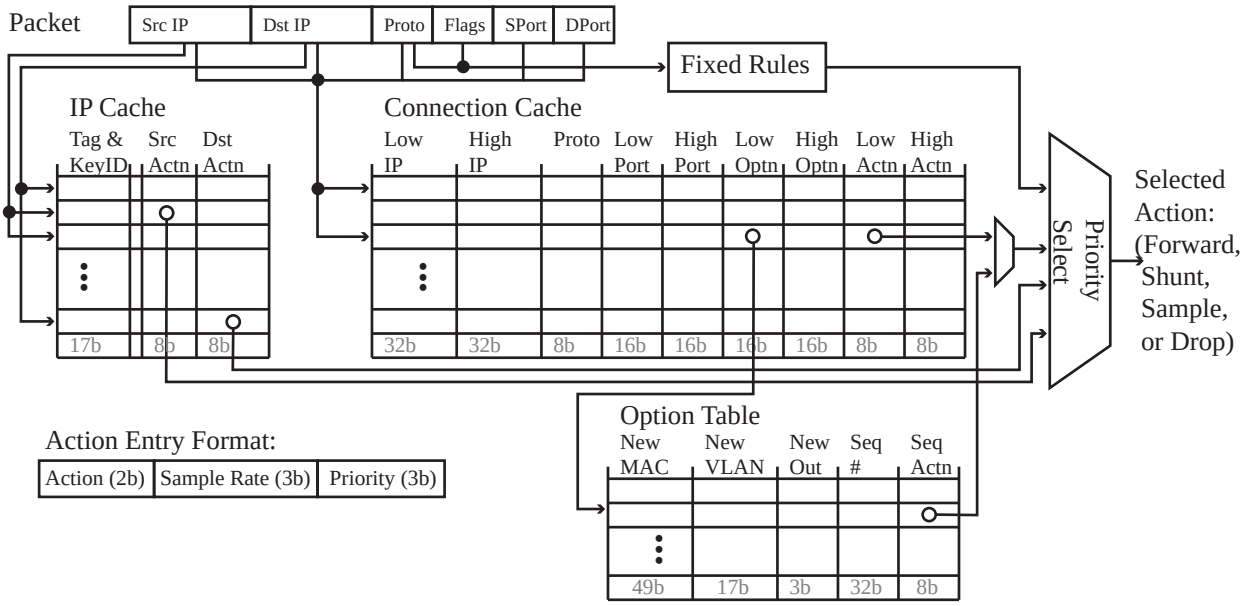


**Figure 4: The packet processing operation used by the Shunt**

The IP address cache uses $2^{16}$ addresses from 0x10000 to 0 0x1FFFF. We implemented the address cache as a 2-location associative permutation cache. With $2^{16}$ addresses and 2 keys, the first 16 bits are used to store the tag, one bit is used for the key ID #, 8 bits are used for the SRC IP record, and 8 bits are used for the DST IP record, with 3 bits of the entry unused. Looking up a packet in the IP cache requires checking four locations, two for the SRC record and two for the DST record.

The connection cache uses $2^{18}$ addresses, from 0x40000 to 0 0x7FFFF. This cache is two location associative, using our keyed permutation as the basis for the hash function and same two keys used for the address cache. Each record in the connection cache is 4 words, thus the cache contains $2^{16}$ entries. The entry contains both IP addresses (64 bits), both port numbers (32 bits), the IP protocol number (8 bits), and an 8 bit action field. Additionally, two 16 bit pointers are included, one for each optional record field, one for each direction of the connection.

Finally, the optional records use $2^{17}$ addresses, from 0x20000 to 0x3FFFF. This optional record contains a 48 bit optional MAC address to overwrite the destination MAC, a 16 bit optional VLAN tag, a 2 bit alternate destination port designation. All these fields also have an associated bit which specifies if the alternate destination (MAC, VLAN, and/or Ethernet port) should be used. Finally, the optional record contains a 32 bit TCP sequence number and an associated 8 bit action. If the packet is a TCP packet, and the packet's sequence number is less than the recorded sequence number, the alternate action field is used instead of the connection cache's action field.

We pipeline the memory access when a packet is received in order to improve memory access time. We first access the 4 words used to specify the two, 64 bit encryption keys used for both the permutation cache for IP lookup and the hash function for connection lookup. Then the two possible connection entry locations (4 words each) are fetched. Then the 4 words forr the IP cache. At this point, the state machine may pause to ensure that the connection entries are properly loaded, before fetching the 4 words pointed to by the optional record. Thus processing a packet requires fetching 20 words from 8 contiguous locations in memory.

## 7.1 The Packet Processing Procedure

When a packet is received, the entire packet is first read into the Shim's BlockRAM buffer. As the packet is received, the appropriate fields (including IP header, TCP header, and Ethernet header) are captured and stored in registers. Once the packet is completely read in, the IP cache, connection cache, and alternate record are looked up. For each cache which matches, the appropriate action field is used, or, if there is no match, the default action of *shunt* with priority 0 is selected.

Additionally, the fixed rules are examined. Non-IP packets are always shunted to the host. IP packets with IP options set are shunted with priority 4, as are TCP SYNs, FINs, and RSTs. Only the highest matching action is selected, with the resulting packet either being shunted to the host, forwarded to the destination, or sampled with a copy going both to the host and destination.

Finally, if the connection cache entry has an alternate record, and the alternate record specifies that the MAC or VLAN tag should be overwritten, this record is overwritten in place before the packet is forwarded. Because this overwriting uses the same memory interface used to write the packet to the BlockRAM, we need to wait for the packet to be completely received before this can occur.

## 7.2 Priority Inversion and Cache Management

The caches are always managed by the host, never the Shunt hardware. The Shunt hardware only reads the caches, to determine the appropriate action. It is up to the host to manage the cache, including both setting entries (when the policy requires them) or evicting entries when space is required.

An important feature is that an evicted cache entry is safe. If there is no entry, the packet is always *shunted* to the host. Thus if only one rule applies, it is always legal for the host to evict that entry if space is needed in the cache.

There is one exception, however. If a high priority entry and a low priority entry exist for the same connection (such as a low priority *drop* associated with an offensive IP but a high priority

*forward* for an allowed connection), and the high priority entry is evicted, the Shunt will compute the wrong action.

We rely on the host never creating the condition where priority inversion occurs. If an evicted entry would create such a situation, the host must either also evict the low-priority rule (to remove the inversion) or select a different entry to evict.

# 8. EVALUATION

We evaluated the Shunt's hardware in several contexts, including the hardware utilization, latency required to process packets both through a hardware only and a hardware/software path, bandwidth testing, and the cycles required to make a decision.

Currently, the complete Shunt implementation requires 21,200 LUTs, or 77 % utilization of the Virtex 2 Pro 30 FPGA's available resources. Another 2770 LUTs are used for routethrough, with a total LUT utilization of 87%. 95% of the slices are occupied. The Shunt also uses 135 out of 136 available BlockRAMs. We believe that we can save 3000 LUTs by removing several redundant BogoCrypt instantiations in each shim and instead multiplex a single implementation. The Shunt meets the target clock rate of 62.5 MHz.

To measure the overall latency incurred by the Shunt, we connected two systems, each to their own Gbps switch, and then bridged the switches either with a cable, with the Shunt set to forward all packets (hardware-only path), or with the Shunt forwarding all packets to the Click test harness, which reinjects the packets (hardware-plus-software-interface path). Using Linux `ping -f -c 10000`, the direct connection showed a average RTT of 176 $\mu$sec, the hardware-only path took 187 $\mu$sec, and the hardware+software path 344 $\mu$sec. Thus, packets forwarded by the Shunt incur only an additional 5 $\mu$sec of latency.

We tested the Shunt's ability to process large data rates using ipperf [8] in the Deter testbed. Using a single sending host and a receiving host on the other side, each Shunt port is capable of receiving and processing data at 480 Mbps, as there is currently a bug in the input FIFO which is causing a lockup condition when higher data rates are attempted. Additionally, one other known bug is causing corrupt MAC and VLAN tags when overwriting packet contents.

The Shim itself is capable of processing packets at full Gigabit line rate, but only for reasonably sized packets which are not directed to the host. It requires 41 cycles from when a packet is completely received in the BlockRAM buffer to when it can be read out, when the board is lightly loaded. During heavy load, memory contention could increase this by, at most, 60 clock cycles, resulting in a maximum decision time of 101 clock cycles.

If the packet is destined for the host, the Shim will have to wait until the arbiter reads the packet into the host packet buffer before receiving the next packet. Additionally, since the host interface is only 32b, 33 MHz, it is obviously insufficient to support full Gigabit line rates. But if the packet is destined solely for another Ethernet, it can begin reading the next packet. Since the interface from the MAC is 2 Gbps, and the interframe gap is 20 bytes, the Shunt can maintain full line rate for forwarded packets if the average packet size is over 80 bytes. Since the minimum Ethernet packet size is 64 bytes, the Shunt can't quite keep up with a full rate stream of minimum sized packets, but can process a stream of slightly over minimum size at full rate. In practice the Shunt's throughput will be limited by the fraction of packets which are shunted or sampled, not by its ability to forward packets which don't involve the host.

# 9. CONCLUSIONS

We have developed the Shunt, an FPGA based accelerator for intrusion prevention systems based on the NetFPGA architecture. The Shunt's design is based on RNET, a modified version of the NetFPGA CNET design which is optimized for developing network processing applications.

The Shunt uses a novel cache structure to track addresses and connections of interest. It uses a 2-location associative cache for connections, and a 2-location associative permutation cache for tracking addresses. The permutation cache allows twice as many IP address entries to be stored in the same memory. We also developed a new block cypher specifically for FPGA-based permutation caches, which can be realized in 160 LUTs, while the multilocation associativity allows the cache to be more effectively utilized by the software host. Additionally, the caches are "safe", with cache misses resulting in packets being shunted to the host.

As a result, the Shunt can utilize a very small amount of memory, a single 2 MB (512kx32) SRAM to maintain its caches, and is implemented on a relatively small (Virtex 2 Pro 30) FPGA. The shunt is also fast, requiring 41 cycles to make a decision when lightly loaded (and a maximum of 101 cycles when fully utilized). For packets handled entirely in hardware, additional latency is is only 5 $\mu$s, which is nearly unmeasurable for network traffic.

# 10. REFERENCES


[1] R. Anderson, E. Biham, and L. Knudsen. Serpent: A proposal for the advanced encryption standard.

[2] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, July 1970.

[3] Martin Casado, Tal Garfinkel, Aditya Akella, Michale Freedman, Dan Boneh, and Nick McKeown. SANE: A protection architecture for enterprise networks. In *Usenix Security*, 2006.

[4] Scott Crosby and Dan Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.

[5] M. Crovella. Performance evaluation with heavy tailed distributions. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–10, London, UK, 2001. Springer-Verlag.

[6] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.*, Philadelphia, Pennsylvania, May 1996. Also, in Performance evaluation review, May 1996, 24(1):160-169.

[7] J.M. Gonzalez. *Efficient Filtering Support for High-Speed Network Intrusion Detection*. PhD thesis, University of California, Berkeley, 2005.

[8] National laboroatory for applied network research, distributed applications support team, iperf, the tcp/udp bandwidth measurement tool. http://dast.nlanr.net/projects/iperf/.

[9] R. Morris, E. Kohler, J. Jannotti, and M. Frans Kaashoek. The click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[10] Nicholas Weaver and Stuart Staniford and Vern Paxson. Very fast containment of scanning worms. In *13th USENIX Security Symposium*. USENIX, August 2004.

[11] V. Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*,

2(4):316–336, 1994.

[12] V. Paxson and S. Floyd. Wide area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[13] Ronald L. Rivest. The RC5 encryption algorithm, from dr. dobb's journal, january, 1995, 1996.

[14] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher.

[15] Lambert Schaelicke, Kyle Wheeler, and Curt Freeland. SPANIDS: A scalable network intrusion detection loadbalancer, 2005.

[16] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processings. In *SIGCOMM*, 2005.

[17] Haoyu Song, Todd Sproull, Mike Attig, and John Lockwood. Snort offloader: A reconfigurable hardware nids filter.

[18] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10 gbps fpga-based network intrusion detection system.

[19] Greg Watson, Nick McKeown, and Martin Casado. Netfpga: A tool for network research and education. In *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, 2006.

[20] Nicholas Weaver, Dan Ellis, Stuart Staniford, and Vern Paxson. Worms verses perimiters: The case for hard lans, in submission.

[21] Nicholas Weaver, Vern Paxson, and Robin Sommer. Work in progress: Bro-LAN pervasive network inspection and contr ol for lan traffic, 2006.

[22] W. Wilinger, V. Paxson, and M. Taqqu. Self-similarity and heavy tails: Structural modeling of network traffic. In R. Adler, R. Feldman, and M. Taqqu, editors, *A Practical Guide To Heavy Tails: Statistical Techniques and Techniques*. Birkhauser, 1998.

[23] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5:71–86, 1997.