Shunting: A Hardware/Software Architecture for Flexible, High-Performance Network Intrusion Prevention

Jose M Gonzalez
International Computer Science Institute
1947 Center Street #600
Berkeley, CA, 94720
chema@icsi.berkeley.edu

Vern Paxson
International Computer Science Institute
1947 Center Street #600
Berkeley, CA, 94704
vern@icir.org

Nicholas Weaver
International Computer Science Institute
1947 Center Street #600
Berkeley, CA, 94704
nweaver@icsi.berkeley.edu

ABSTRACT

Stateful, in-depth, inline traffic analysis for intrusion detection and prevention is growing increasingly more difficult as the data rates of modern networks rise. Yet it remains the case that in many environments, much of the traffic comprising a high-volume stream can, after some initial analysis, be qualified as of "likely uninteresting." We present a combined hardware/software architecture, *Shunting*, that provides a lightweight mechanism for an intrusion prevention system (IPS) to take advantage of the "heavy-tailed" nature of network traffic to offload work from software to hardware.

The primary innovation of Shunting is the introduction of a simple in-line hardware element that caches rules for IP addresses and connection 5-tuples, as well as fixed rules for IP/TCP flags. The caches, using a highest-priority match, yield a per-packet decision: *forward* the packet; *drop* it; or *divert* it through the IPS. By manipulating cache entries, the IPS can specify what traffic it no longer wishes to examine, including directly blocking malicious sources or cutting through portions of a single flow once the it has had an opportunity to "vet" them, all on a fine-grained basis.

We have implemented a prototype Shunt hardware design using the NetFPGA 2 platform, capable of Gigabit Ethernet operation. In addition, we have adapted the Bro intrusion detection system to utilize the Shunt framework to offload less-interesting traffic. We evaluate the effectiveness of the resulting system using traces from three sites, finding that the IDS can use this mechanism to offload 55%–90% of the traffic, as well as gaining intrusion prevention functionality.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and protection

General Terms

Security

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

Keywords

Intrusion Detection, Intrusion Prevention, Hardware Acceleration, FPGA, NIDS, NIPS

1. INTRODUCTION

Stateful, in-depth, inline 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" portions or complete traffic streams once it has had an opportunity to "vet" them.

For the Shunting architecture to yield benefits, it needs to operate in an environment for which the monitored network traffic has the property that—after proper vetting—much of it can be safely skipped. This property does *not* universally hold. For example, if a bank needs to examine all Web traffic involving its servers for regulatory compliance, then a monitor in front of one of the bank's server farms cannot safely omit a subset of the traffic from analysis. In this environment, Shunting cannot realize its main performance benefits, and the monitoring task likely calls for using custom hardware instead.

However, in many other environments we expect Shunting to potentially deliver major performance gains. Our basis for this conjecture rests in the widely documented "heavy tail" nature of most forms of network traffic [19, 21, 8, 29, 28, 7], which we might express as "a few of the connections carry just about all the bytes." The key additional insight is "... and very often for these few large

connections, the very *beginning* of the connection contains nearly all the information of interest from a security analysis perspective."

We argue that this second claim holds because it is at the beginning of connections that authentication exchanges occur, data or file names and types are specified, request and reply status codes conveyed, and encryption is negotiated. Once these occur, we have seen most of the interesting facets of the dialog. Certainly the remainder of the connection might also yield some grist for analysis, but this is *generally less likely*, and thus if we want to lower analysis load at as small a loss as possible of information *relevant to security analysis*, we might best do so by skipping the bulk of large connections. In a different context, the "Time Machine" work by Kornexl and colleagues likewise shows that in some environments we can realize major reductions in the volume of network traffic processed, by limiting the processing to the first 10–20 KB of each connection [14].

As a concrete example, consider an IPS that monitors SSH traffic. When a new SSH connection arrives and the Shunt fails to find an entry for it in either of its per-address and per-connection tables, it executes the default action of diverting the connection through the IPS. The IPS analyzes the beginning of the connection in this fashion. As long as it is satisfied with the dialog, it reinjects the packets forwarded to it so that the connection can continue. If the connection successfully negotiates encryption, the IPS can no longer profitably analyze it, so it downloads a per-connection table entry to the Shunt specifying that the action for the connection in the future is "forward."

For heavy-tailed connections, this means a very large majority of the connection's packets will now pass through the Shunt device without burdening the IPS with any further analysis load. On the other hand, if the IPS is dissatisfied with some element of the initial dialog, or with one of the hosts involved, it downloads a "drop" entry to terminate the connection. Note that by providing for reinjection, we can promote an intrusion *detection* system into an intrusion *prevention* system, one that does not merely detect attacks but can block them before they complete. Reinjection also allows the IPS to *normalize* traffic [11] to remove ambiguities that attackers can leverage to evade the IPS [22]. Finally, if the IPS is unable to resolve whether the connection can progress without further analysis, it simply leaves the Shunt's tables unmodified and continues to receive the connection's packets due to the Shunt's default action.

Put more simply, we can frame Shunting as providing a form of filtering that is particularly well suited to preserving as much security-relevant information as possible given the need to discard large volumes of traffic. In this paper we present evidence to back up this assertion, and discuss numerous subtle points that arise when realizing Shunting in practice. We present the Shunting architecture, based on fixed-size table lookups and a shared-memory interface to the IPS that greatly simplifies the hardware implementation because it allows the hardware to make imperfect decisions. Since the Shunt requires only fixed table lookups on header fields, we can implement it readily in a small amount of custom hardware.

We modified the Bro intrusion detection system [20] to take advantage of the Shunt, giving it more direct IPS capabilities than it has had in the past (which involved enabling it to update router ACL entries by logging into the router), and implemented sample modifications to its analysis scripts. Testing this system using full packet traces from a Gbps-connected site with 1000s of hosts shows that Bro can leverage a modest Shunt configuration to offload 55%–90% of the traffic. This in turn suggests that the Shunt architecture should enable Bro to process a Gbps stream with ease when using a Shunting device coupled with a general-purpose, commodity PC platform.

We have implemented the Shunt in hardware on the NF2 [17] FPGA system. While our board does not yet support all of the architecture's features, we can use it to evaluate the main mechanisms, and it includes sufficient functionality to ensure the feasibility of processing data—*including using a general purpose, commodity PC for rich IPS analysis*—at Gbps rates for traffic streams with realistic packet sizes.

We begin in Section 2 with a survey of related work. Section 3 gives an overview of the Shunting architecture and how it lends itself to fast operation in hardware. We then describe in Section 4 a prototype hardware implementation that realizes this promise. In Section 5 we discuss general issues with integrating Bro, and in 6 the decisions we made regarding how to enhance Bro's analysis to leverage the Shunt. We evaluate the effectiveness of Shunting, as well as sensitivities to implementation parameters, in Section 7. In Section 8 we discuss ongoing work, and we conclude in Section 9.

2. RELATED WORK

Intrusion detection systems (IDSs) monitor host or network activity to spot attempted or successful misuse of computers. Such misuses might constitute attacks or simply violations of policy restrictions. While there is a vast literature on IDSs, we touch on it here only in a limited fashion because our Shunting architecture for the most part is indifferent to the particular mechanisms of the IDS it supports. Indeed, we aim for Shunting to provide cheap hardware assistance for a wide range of network-based IDSs.

That said, part of our discussion concerns implementing Shunting in conjunction with a particular IDS, the open-source Bro system [20]. Bro provides an event-oriented framework that couples generic (non-security-specific) analysis of network traffic at layers 3, 4 and 7, with an interpreted, domain-specific "policy script" language used to express higher level analysis triggered by the occurrence of particular events. The ability to script this latter analysis makes it particularly easy to extend Bro to work in conjunction with a shunting device.

When an IDS is capable of not only detecting an attack but also blocking it to prevent it from succeeding, it is termed an intrusion *prevention* system (IPS). Since Shunting directly enables IPS functionality by diverting packets *through* an intrusion analyzer rather than simply giving it a passive copy of the traffic stream, in this paper we will generally use the term IPS to describe the system with which the Shunt interacts, and only use the term IDS when the distinction between detection and prevention is significant.

The prior work most directly related to ours concerns other approaches for using hardware to augment IPS capabilities. Kruegel and colleagues developed an architecture for accelerating signature-based systems using a 4-step process that provides multiple, parallel IPS analyzers each with a subset of the total traffic that conforms to a small superset of the traffic it needs to detect particular attacks [15]. Input traffic flows into a simple hardware device (the "scatterer") that divides the traffic in a round-robin fashion among a group of classifiers (the "slicers"). Each slicer checks every packet to see whether it might match one or more signatures. If so, it forwards the packet to the appropriate "reassembler," which reassembles the packet stream before forwarding the streams to the appropriate IPS engine(s).

Another technique commonly proposed for high-speed processing—generally oriented towards IDS rather than IPS functionality—is "pushing processing into the NIC": using a network interface to offload much of the processing required for passive packet capture and analysis. Shunting resembles this concept, although our processing model is very different and involves explicit inline/diversion decisions. Deri [9] proposes

using a router (Juniper M-series, which allows for traffic filtering based on header fields [16]) as a smart Network Interface Card (*NIC*), performing generic traffic accounting and simple packet filtering and sampling, and sending the filtered/sampled stream to a Linux host. The Intel IXP family of "network processors" provides a framework to perform in-NIC packet-processing [13]. The IXP series is composed of multiple miniature processors that operate in parallel, along with a StrongARM control processor [16]. The IXP has been proposed as a means to accelerate Snort signature matching [2] by implementing portions of the signature matching and other pieces on the Snort stack. Indeed, there is a large literature on implementing signature-matching in custom hardware, but this work is not applicable to accelerating IPSs in general, other than for offloading the signature matching they perform.

Using Endace's DAG 4 cards [3], Iannaconne and colleagues present a network adapter that permits passive monitoring of OC-192 links (10 Gbps) [12]. The authors use the DAG card's FPGA to compress packet headers into flow traces, and send only those flow traces to the PC host. The authors use a hashed, limited-size connection table to store the flow traces, arguing that, with the help of fast PCI buses (64 bits, 66 MHz), it is possible to monitor IP, TCP, and UDP headers on 10 Gbps links, enabling header-only IDS analysis. However, clearly such analysis cannot extend to inspection of application-level semantics, since the available information does not include transport payloads.

The SCAMPI project also proposes using a smart network adapter to limit the amount of traffic that reaches the host in packet capture scenarios [4, 5]. SCAMPI runs on several different architectures, including Intel IXP family of network processors, Endace's DAG cards, and their own network adapter, called "COMBO." COMBO adapters perform systematic (deterministic) and probabilistic 1-in-N sampling, address- and port-based sampling, payload string searching, generic flow-state accounting and reporting, and packet filtering using FPL-2 (an extended, BPF-like language).

In contrast to previous approaches, Shunting is based on coupling an IPS running in a general-purpose computing environment with a separate hardware device, allowing the IPS to control the processing load it sees at the granularity of individual streams. In addition, Shunting achieves this with minimal assumptions about the IPSs overall operation, allowing the specialized hardware to remain *(i)* broadly applicable, and *(ii)* simple and cheap.

3. THE SHUNT ARCHITECTURE

In this section we present the shunting architecture. We begin with an overview of the general architecture and the motivation behind it (§3.1), and then discuss in detail the structure of the Shunt device's tables (§3.2), which act as a cache for the IPS. We finish in §3.4 with an important refinement to the architecture, *forward-N*.

3.1 Overview

Inline traffic processing is a particularly demanding activity, because the speed of the processing directly limits overall network performance. If the inline element cannot keep up with the rate at which new traffic arrives, it eventually will exhaust its buffering capacity and drop some of the traffic, affecting the quality of unreliable connections and imposing a major impairment to reliable traffic due to the transport protocol's congestion response.

At high speeds (Gbps), using a commodity PC for inline packet processing becomes very difficult. Simply monitoring the traffic stream requires 1 Gbps of bandwidth across the I/O bus and 1 Gbps bandwidth to memory. In addition, if the monitor operates at user level, unless we can exploit memory mapping we need an additional 1 Gbps of internal memory bandwidth. If we not only monitor but also forward, then PC inline for a bidirectional Gbps link requires 4 Gbps of I/O bandwidth and 4 Gbps of memory bandwidth (with perhaps another 4 Gbps memory bandwidth if performing user-level analysis), leaving little additional resources for processing. Furthermore, 10 Gbps Ethernet in early deployment, the problem is growing worse.

Figure 1 shows the Shunting architecture we propose for enabling use of inexpensive, highly flexible commodity PCs for inline packet processing. A *Shunt*-based system consists of two elements, a software packet processor (the *Analysis Engine*) and a hardware forwarding element (the *Shunt* itself).

The Analysis Engine, such as an IPS, views the Shunt as a normal Ethernet device, except that the Shunt has a series of tables that act as a cache for rules. The Shunt device treats these tables as read-only; it is the responsibility of the Analysis Engine to both manage the cache and to resolve cache misses by maintaining more comprehensive state.

When a packet arrives, the device chooses from one of three possibilities: (a) forward the packet to the opposite interface (thick, solid line), (b) drop it (thin, dashed line), or (c) divert (*shunt*) it to the analyzer (thin, dotted line) by examining the packet header and selecting the highest priority action. For packets diverted to the Analysis Engine, the analyzer makes another decision regarding the packet's fate: (c.1) inject the packet back to the network interface, or (c.2) drop it. It may optionally at this point also update the Shunt device's tables to offload similar decisions in the future.

In particular, the Analysis Engine must understand that the Shunt hardware is a cache: if a connection is offloaded to the Shunt, the Analysis Engine must still maintain state for the connection, since it may be necessary to flush the cache entry and return to diverting the connection's traffic through the Analysis Engine.

The Shunt architecture aims to achieve several goals. First, we want separation of mechanism and policy, with the Shunt providing only the former. Along these lines, while our implementation couples the system with Bro, we intend the architecture to directly support other types of analyzers, too. Second, we want to keep the Shunt very simple: only examining headers, and with deterministic memory behavior, enables an easy and efficient hardware implementation. Often, packet processing is limited by memory accesses, so we imposed a budget of a limited number of accesses per packet. Related to this, the architecture requires only a minimal amount of buffering, which it achieves by always making immediate decisions regarding the next-hop destination for an arriving packet.

Finally, for the Shunt to realize significant performance gains, the policy used by the IPS must enable the Shunt to forward most packets without involving the analyzer, and at high speed. Thus, for traffic which policy has determined does not require further analysis, the Shunt must impose only a negligible forwarding delay.

3.2 The Shunt's Tables

We accomplished these goals with a simple mechanism: header-based table lookup, where the lookup is "incomplete" in that we implement it quickly in hardware using a cache that may contain only a subset of the table entries. The Shunt's decision making (Figure 1) is conceptually very simple. We use two tables, one indexed by IP address and another indexed by the connection 5-tuple, along with a fixed table (the *static filter*) applied to certain header fields such as TCP SYN/FIN/RST control flags (Figure 3.2).

The device looks up each packet in the tables in parallel. If a lookup finds an entry, the result includes an action (*forward*, *drop*, or *shunt*) and a priority from 0 to 7. A priority encoder then selects

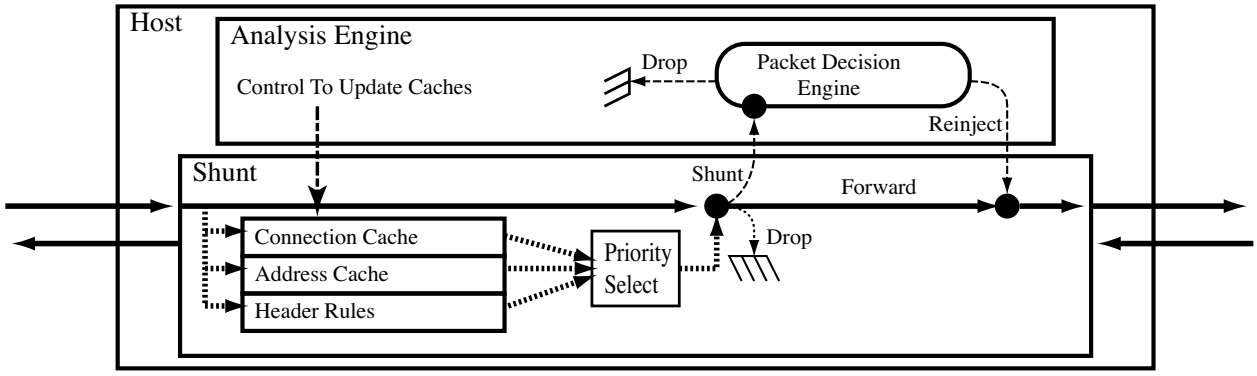

Figure 1: Shunting Main Architecture. The shunt examines the headers of received packets to determine the associated action: *forward*, *drop*, or *shunt* to the Analysis Engine. The Analysis Engine directly updates the Shunt's caches to control future processing, and either drops analyzed packets for immediate intrusion prevention or reinjects them once vetted for safety.

the highest priority entry[1] and performs the corresponding action on the packet. If the device does not find a match in any table, it uses *shunt* as the implicit action.

The connection table has entries indexed with the usual 5-tuple of source and destination IP addresses, source and destination ports, and transport protocol (TCP or UDP). This is the most important table for achieving high performance, as it enables fine-grained, per-connection decision-making on the part of the analyzer. Additionally, the connection table includes an optional record field used to implement *forward-N*, which we discuss later.

The IP address table has actions associated with it for both the source and destination addresses. This table mainly serves to implement static and dynamic firewall rules, such as blocking known and newly detected attackers, or whitelisting high volume servers or authorized vulnerability scanners.

Finally, we also have a fixed header-filter table, which includes default rules (such as diverting fragments and TCP control packets). We compile these static rules into the hardware configuration, with low priorities associated with them to make the entries easy to override.

Other than the static filter, all table entries become populated only upon request by the analyzer (including upon its startup). For example, when the analyzer decides that it is safe to forward the remainder of a connection without further inspection, it instructs the Shunt to add a corresponding entry. This coupling between the Shunt's filtering and the analyzer's decision-making allows the analyzer to vet requests on a connection-by-connection or host-by-host basis, and, once vetted, efficiently skip the subsequent traffic. It similarly becomes easy for the analyzer to summarily block an offending host, which not only blocks all traffic from the offender, but prevents the offender from loading the analyzer with traffic, enabling the IPS to protect itself against overload if it can identify the source of the load.

The default-shunt nature produces a fail-safe device. Only if the IPS instructs the Shunt that it deems a given flow "safe" or "malicious" will the Shunt process the flow in an unconditional manner. In addition, if the IPS cannot keep up with the pace of traffic diverted through it, the traffic does *not* escape analysis, but instead is throttled back to the rate at which the IPS can vet it. While this can have a deleterious effect on network performance, it has the correct safety properties in terms of "better safe than sorry."

An important feature of the architecture is that the Shunt's tables are *caches*: an entry is not guaranteed to be persistent in the Shunt if another entry is inserted. The shunt hashes[2] each potential entry to one or more locations in memory. When adding a new entry, this may evict an old entry. This functionality allows the Shunt to perform a small, bounded number of memory accesses into a fixed-size memory. It is the responsibility of the Analysis Engine to respect that the Shunt device is a cache and not a complete data structure. Thus, packets designated for forwarding or dropping can still be diverted to the Analysis Engine, requiring the Analysis Engine to reinsert the corresponding table entry. Such evictions can however create subtle problems of priority inversion, which we discuss in Section 3.3.

The cache-like nature of these tables enables fast operation. Rather than having to search through a possibly unbounded data structure (e.g., a chain of hash buckets), the packet headers directly index all entries that the Shunt needs to examine.

The IP and connection tables are both directional. Each direction can have a different action and priority associated with it. Thus, for example, the analyzer can monitor the inbound side of a connection (by setting a *shunt* action) while allowing the outbound half unobstructed (with a *forward* action).

Finally, table entries also include a *sample* field. If non-zero, this field specifies an index into a table of probabilities. The device then sends a *copy* of the packet to the analyzer with the given probability. This functionality enables the analyzer to monitor a connection for liveness and volume without having to receive all of its traffic.

3.3 Interfacing to the Shunt

In our design, the Shunt device acts as an Ethernet card to the host, transferring to the kernel any packets directed to the host and processing the remainder according to the device's tables. To control the Shunt, the Analysis Engine directly manipulates the cache entries, which requires knowledge of the specific format and properties of the Shunt's caches. Clearly, we could instead provide a more abstract interface, managing cache deletions and insertions based on higher-level requests. We have not done so yet because so far we have only created a single hardware implementation (Section 4), so in the subsequent discussion we assume that the Analysis Engine manages the caches. Additionally, since we view the Shunt as a device coupled to a stateful IDS system, having the Analysis

[1]Conflicting entries with equal priorities indicate a policy inconsistency. Architecturally, the hardware could signal such conditions. In our implementation, the Shunt uses a fixed set of internal priorities to resolve ties, and it is the responsibility of the Analysis Engine to not create such conflicts.

[2]Using a hash function chosen to resist attacker manipulation [6].

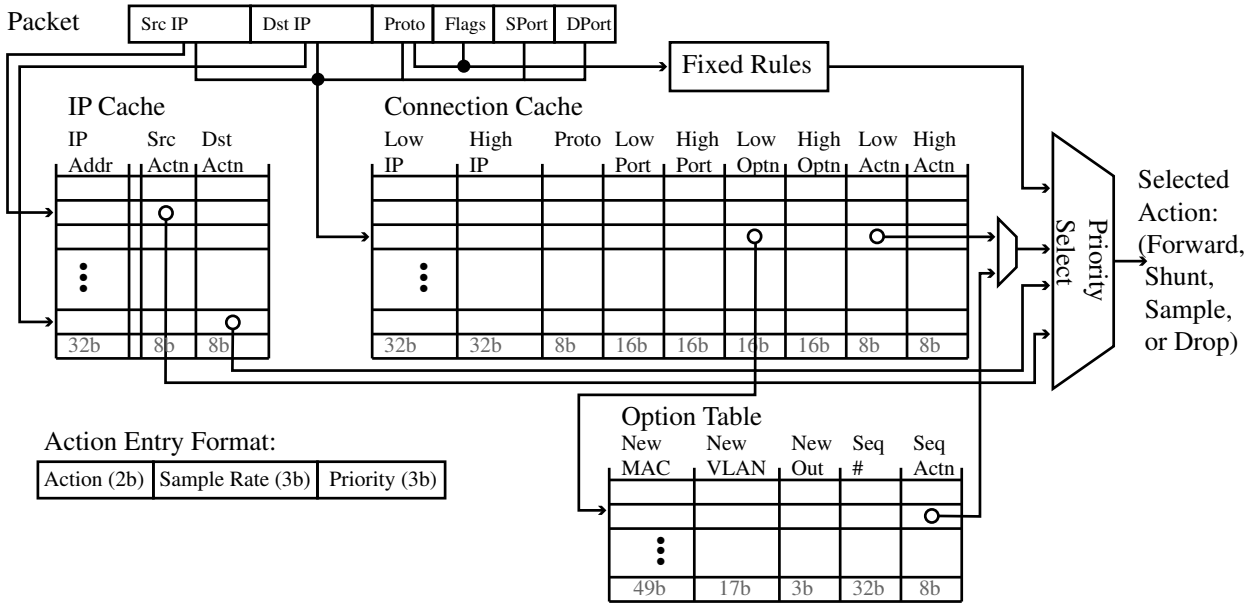

Figure 2: A detailed diagram of the caches used in the decision making process and the bits required for each field. Separate actions can be bound to each direction of a flow; the option field supports *forward-N* (Section 3.4) and destination routing (Section 4).

Engine directly manage the caches eliminates the need for a duplicate state-handling infrastructure to track a complete view of all connections.

An important issue is that the Analysis Engine driver must manage conflicts in the tables to prevent priority inversion. If two dynamic rules apply to the same packet (such as a connection table entry allowing a connection to a blocked known offender) with different priorities and actions, and the higher priority item needs to be evicted, then it is up to the Analysis Engine to ensure that the eviction does not lead to the Shunt now taking an incorrect *forward* or *drop* action. (An incorrect *shunt* action is not a problem, since this allows the Analysis Engine to correct the action.)

Thus, the Analysis Engine must either have direct control over the Shunt's caches (selecting which entry to evict when inserting a new entry), or the Shunt must reliably notify the Analysis Engine of the Shunt's eviction decisions, allowing the Analysis Engine to then also evict any lower priority entries as well. Finally, we note that we can still apply the Shunting approach even if the Analysis Engine does not have such control or notification, providing it constraints itself to never insert lower priority entries that can lead to such conflicts, instead emulating the entries in software.

3.4 Refining the *forward* Action

When we evaluated the architecture as described above, we found a particularly important class of traffic for which the basic architecture lacks sufficient expressive power to effectively offload the IPS. This occurs for protocols that send a series of transactions over a single connection, for which the IPS would like to skip over (potentially large) elements of each transaction, but cannot skip the entire connection because doing so will entail missing control information associated with subsequent transactions.

This arises, for example, with persistent HTTP connections. If the IPS determines that the URL in a given client request is allowed, it would like to skip over having to process the item returned for it by the server; but the *next* client URL might be problematic, in which case at that point the IPS needs to analyze the server's reply.

To enable such offload, we need to extend the basic architecture to offer finer-grained control than per-connection, yet we also need to do so in a manner that remains highly efficient for the shunt device to process and economic in terms of the required table space. Our extension, *forward-N*, is a refinement to the *forward* functionality. The notion behind *forward-N* is "skip the next N bytes" rather than "skip the rest of the connection."

We need to take care, however, in specifying N. If it is simply a byte count, then *(i)* for each new packet we will need to write to the table to update the count by decrementing the number of bytes of payload the packet carries, and, more importantly, *(ii)* the accounting will be incorrect for out-of-sequence packets. These latter can happen due to packet loss or reordering, race conditions in installing the table entry, or deliberate attacker manipulation.

We therefore implement *forward-N* in terms of a TCP connection's sequence space, rather than using a byte count. We augment the per-connection shunt table with 32 bits of sequence number. For packets with shunting decisions of *forward-N*, the device checks whether the upper sequence number of the packet is less than[3] the table entry. If so, it *forward*'s the packet; otherwise, it *shunt*'s it. The Analysis Engine then removes the entry when it determines it no longer serves any offload purpose; for example, when it sees an acknowledgment for a sequence number higher than the cutoff.

To implement *forward-N* we include an additional (optional) field in connection table entries that specifies the sequence number limit. In our hardware implementation, we also use this optional field to specify alternate destinations, enabling the Shunt to act as a packet routing device, not just a packet forwarding device; see §4 below.

For non-TCP traffic, we lack an ordered sequence space to use for a reliable cutoff, so for this functionality we would need to instead use a countdown counter or develop an application protocol

[3] "Less than" in terms of using 32-bit sequence-space arithmetic, i.e., a difference of ≤ 2 GB.

specific rule for (*forward-N*), which would significantly complicate the Shunt.

However, it is not clear that non-TCP protocols transfer sufficiently large, skippable items to merit this addition, rather than benefiting from complete skipping (*forward*) or full analysis. Additionally, a non-TCP *forward-N* would require that the Shunt update its tables on a per-packet basis. In the current design, the Shunt hardware only reads the tables, eliminating a large class of race conditions and other issues that might otherwise arise if it also performs updates.

4. THE SHUNT HARDWARE

We implemented a prototype hardware design for Shunting using the NetFPGA 2 platform [17], using as a starting point the NetFPGA reference implementation for a quad-port Ethernet NIC. The NetFPGA 2 consists of four Gbps Ethernets connected to a Virtex 2 Pro 30 FPGA. Access to the card is via a standard PCI (33 MHz/32-bit) bus. In addition, the platform provides two 2 MB SRAMs, one of which can be used for arbitrary data structures.

Figure 3 shows the block diagram for the NetFPGA-based Shunt. Our design uses a 32K-entry, two-location associative permutation cache for IP addresses, and a 64K-entry, two-location associative cache for connection rules. In an N-location associative cache, the entry can reside in one of N different cache locations, in a manner similar to Bloom filters [1], Bloom-filter based hash tables [25], or skewed association caches [23].

For both the connection table and the address table, we use an 8-bit rule field to specify an action: *forward*, *drop*, *shunt*, or *sample*; a 3-bit priority; and a 3-bit sampling rate. Additionally, we include fixed, low-priority rules for *shunt*'ing TCP SYN/FIN/RST packets as well as IP fragments. As previously discussed, the hardware follows the highest priority match, or, if it does not find a match, *shunt*s the packet to the Analysis Engine for analysis. For the connection table, we canonicalize the 5-tuple and provide a different rule for each direction in the flow.

For connection table entries, our design provides for an additional, optional, record field. (The current hardware supports up to 32K such optional records.) This field can specify a rule that is only valid if the packet's TCP sequence number is less than a prespecified limit, to support sequence skipping (*forward-N*). We can also instead use it to specify an alternate Ethernet interface, MAC address, and VLAN tag, in order to allow the Shunt to reroute packets on a flow-by-flow basis.

Such dynamic rerouting allows the Shunt hardware to act as a load-balancing front-end for a clusterized IDS [26] by dispatching packets via an Ethernet switch to a designated IDS node on a per-connection basis. The VLAN rewriting allows the Shunt hardware to route flows between multiple connections on the same switch for fine-grained isolation.

Apart from its rule caches, the Shunt behaves like another quad-port Ethernet card. Our design provides for access to the caches themselves by reading and writing the Shunt's SRAM, as the SRAM supports direct memory-mapped I/O operations.

For much more extensive discussion of the hardware implementation, including its use of "permutation" and location associative caches, see [27]. Currently, a known bug in the FPGA board's firmware limits the hardware's operation to 480 Mbps, but this problem will be remedied with the next version of the board.

5. INTEGRATING THE SHUNT WITH BRO

To test our architecture in practice, we selected the Bro intrusion detection system [20] as our Analysis Engine, due to its high-level, flexible, and expressive nature, as well as our strong familiarity with its internals. To adapt it, we added an API at the Bro scripting level to support the Shunt's functionality, and modified its analysis policies to then utilize this API. We emphasize, however, that nothing in our Shunting implementation has any particular knowledge of Bro's workings.

By itself, Bro provides only limited intrusion prevention functionality. Its scripts can execute arbitrary programs, which are used operationally to *(i)* terminate misbehaving TCP connections using forged RST packets, and *(ii)* install ACL blocks at a site's border router. However, both these actions occur post facto with respect to the network traffic that led Bro to detect a problem, so for attacks that proceed quickly, the reaction can come too late. In addition, router ACL limitations restrict the use of blocking to 100s or perhaps 1000s of addresses. This might seem like a plenty, but due to the incessant presence of "background radiation" [18], as well as the occasional outbreak of worms or large-scale botnet sweeps, in fact operationally we desire the capacity to block 100,000s of addresses.

With the Shunt, however, Bro can become a high-performance IPS. By vetting each packet before it reaches its destination the combined system can block attacks before they succeed, and proactively block suspect hosts at much larger scale than otherwise.

5.1 Changes to Bro's Internals

Bro's stateful nature already requires that Bro track all active connections and their associated protocol analyzers, as well as all IP addresses of interest. We extended and annotated these data structures to incorporate Shunt-related information.

Bro maintains an internal whitelist of IP addresses, the `PacketFilter` class, which specifies a group of systems that can be safely ignored. We extended this data structure to support blacklisting as well: IP addresses which should always be blocked. This whitelist now allows us to populate the IP table in the Shunt and to update the table on cache misses.

Bro also maintains a record for every established connection. Each connection has associated with it a tree of relevant analyzers, ranging from the TCP stream reassembler and signature matching engines to specific protocol parsers for HTTP, SSH, and other protocols. Bro can apply multiple protocol analyzers to a single connection *concurrently* in order to robustly determine the actual application protocol without relying on the (increasingly untrustworthy) transport port number [10].

We added to this structure notions of "unessential" and "essential" analyzers, as follows. Unessential analyzers will process a connection's packets if present, but the presence of such unessential analysis does not suffice to require the Shunt to divert those packets through the Analysis Engine. However, as long as a connection has associated with it at least one essential analyzer, then it and all other analyzers will receive the connection's packets. The decision regarding whether an analyzer is unessential or essential is made on a per-connection basis, and can change (in particular, essential analyzers becoming unessential) during the connection's lifetime. If every essential analyzer associated with a connection is either removed or demoted to unessential, it is then safe to install a *forward* rule for the connection.

By default, all but the TCP stream reassembler and similar utility analyzers are considered essential. It is up to the analyzer or its associated policy script to either mark the analyzer as unessential (so

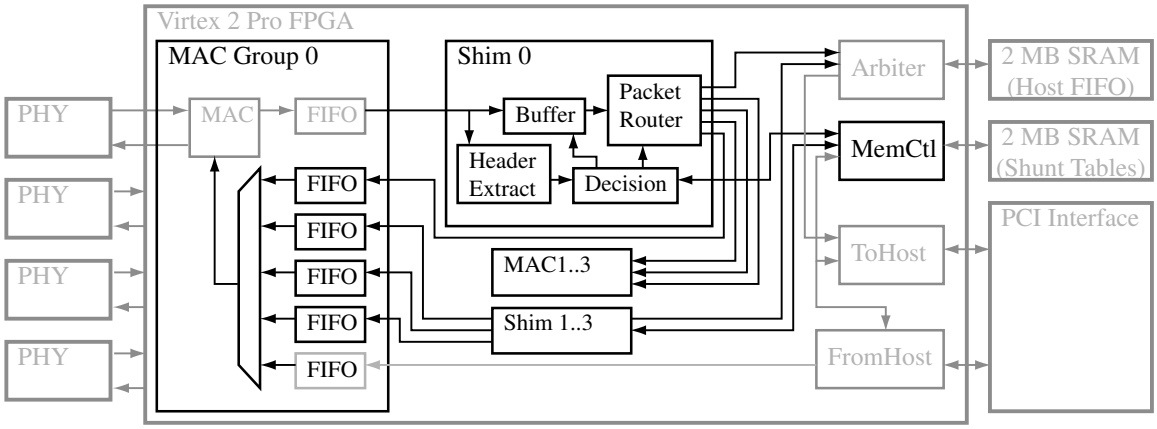

Figure 3: The Shunt Hardware Block Diagram. Items in dark were modified from the NetFPGA reference design.

that it may still receive traffic) or simply remove it from consideration. Even when all analyzers are considered unessential, the *forward* rule uses a lower priority than the default *shunt* rule for TCP SYN/FIN/RST packets, so `connection_finished` and similar end-of-session accounting still operates properly.

The API for *forward-N* functionality takes two parameters: the number of bytes to skip (relative to the point in the stream processed so far), and a smaller value indicating the initial number of these bytes *not* to skip. The function (part of the TCP stream reassembler) converts the byte count to a sequence number to specify in a *forward-N* table entry. However, the stream reassembler does not install this entry until having first processed the given number of initial bytes. We chose this interface to support common functionality in which an analyzer (such as that for HTTP) determines that a large item will soon be transferred and wishes to inspect only the beginning of the item. (If we instead left it up to the analyzer to request *forward-N* after it has received the beginning of the item, often that is difficult for the analyzer to coordinate due to the layering by which it receives aggregated information. For example, one of Bro's natural interfaces for doing so delivers an entire item as a unit to the higher-level analysis, rather than doing so piecemeal.)

Since Bro is not multithreaded, if it determines that a packet should be dropped to block an attack, the drop directive will occur prior to Bro beginning to process the next packet. Thus, at the point where Bro's internal engine requests a new packet, if the analysis of the current packet did not explicitly indicate it should be dropped, we know we can safely go ahead and forward it. This approach limits the latency introduced by the architecture to Bro's per-packet total analysis time, typically well under 1 msec.

6. SHUNT-AWARE POLICIES

In order to effectively leverage Shunting, we must adapt the Analysis Engine system to best employ it for forwarding uninteresting traffic and blocking problematic traffic. In this section, we discuss changes and extensions we introduced to the Bro IDS in this regard. We note that we intend these modifications as *exemplary* rather than complete; we present fairly modest additions (with respect to Bro's full suite of analysis) that nevertheless yield significant performance gains.

When constructing these modified analyses, we have to ensure that they are "safe": that it is acceptable to ignore the *forward*'ed traffic without impairing the security analysis.

6.1 SSH

For SSH analysis, we would like to produce a log of all SSH sessions (including time and volume of data transferred), client and server software versions, and detection of brute-force password guessing. To this end, we modified Bro's SSH analysis script as follows. We first added an event handler for Bro's `connection_closed` event to log the time, source, destination, and volume of the session, where we compute the volume of the session based on the difference in sequence numbers between the connection's SYN and FIN packets. To check for SSH brute-force attacks, we allocate a per-source counter. When a connection begins, we increment the counter and initiate polling of the connection for the next 10 seconds, where every 100 msec we assess the connection's status. As soon as the connection transfers more than 10 KB of data, we assume that the user successfully authenticated, and reset the count to zero. If instead the counter ever reaches a predefined threshold (currently 10), indicating multiple short-lived SSH session, we generate a Bro "notice" reflecting a likely password-guessing attack.

If the polling process determines that the connection appears legitimate and/or inactive, the script demotes the SSH analyzer to *unessential* (as discussed in the previous section). Now the Shunt will forward all subsequent SSH traffic except for the final FIN or RST (unless the user's configuration has incorporated other analyzers still deemed essential).

As a result, we can avoid processing nearly all SSH traffic, while still retaining the ability to (1) detect password guessing, (2) determine the approximate size of file transfers, (3) inspect SSH version strings (present in each connection's initial data exchange), and (4) distinguish between file transfers and interactive sessions in the log (as file transfers sustain much higher data rates that do interactive sessions).

6.2 HTTP

For HTTP, far and away most of the bytes transferred come in server replies. Although some files (e.g., HTML, Java, Javascript, Flash) benefit significantly from IDS analysis, much of the data comes instead in the form of images, video, audio, and binary transfers.

We modified Bro's HTTP reply analysis script to capture the MIME type and expected length of all responses. Then, for any response over a given size (default 10 KB), we examine the MIME type. If the type matches one in a configurable "presumed safe"

MIME Type	Probably Safe	Percentage of payload data	Average Size
`application/`*safe*	Yes	33.7%	1.4 MB
`video/*`	Yes	28.5%	8.9 MB
`application/`*unsafe*	No	14.7%	60 KB
`text/*`	No	8.8%	22 KB
`image/*`	Yes	8.5%	7.8 KB
`audio/*`	Yes	5.4%	2.6 MB
`binary/*`	No	0.6%	218 KB
`multipart/*`	Yes	0.3%	354 KB
other	No	<0.1%	10 KB

Table 1: The different MIME types, whether the type is considered "probably safe", the percentage of the total HTTP replies of each MIME type, and the average payload size for the UNIVERSITY I trace.

whitelist (default: images, video, audio, and some application types), the script instructs the TCP stream reassembler to skip over the payload using *forward-N*. Otherwise, or if the size is unavailable (e.g., due to use of HTTP "chunking"), we perform the full regular analysis.

In general, these "presumed safe" types represent the bulk of the HTTP transfers. Table 1 lists the different MIME types observed in the UNIVERSITY I trace (see Section 7 for trace details); whether we consider items of the given type as likely safe; the fraction of the HTTP responses they represent; and the average item size for all such HTTP responses that specify a payload length. For application data, we currently consider `binary`, `msword`, `octet-stream`, `phdata`, `pdf`, `vnd.ms-powerpoint`, `x-xpinstall`, `x-sh`, `x-pkcs7-crl`, `x-tar`, `x-zip-compressed`, and `zip` as "presumed safe". For some of these, we might want to conduct further analysis, but Bro presently lacks analyzers specific to these item types. If it included these, we suspect that often the analyzer would only need to inspect the beginning of the item transfer (per the next paragraph) to determine whether the item was potentially problematic; if not, then we could still skip the remainder of the item.

Even when skipping the payload, however, we still examine the beginning of each item, regardless of file type. This allows us, for example, to perform signature analysis to verify whether the item's actual type corresponds with its stated type. The savings we present in our evaluation assume we inspect (and thus cannot skip) the default value of the first 5 KB of each item.

6.3 Dynamic Protocol Detection

Bro's Dynamic Protocol Detection (DPD; [10]) initially analyzes *all* traffic in order to determine the protocols (primarily at the application layer) actually embedded in a data stream. Bro's signature engine [24] matches the initial (default 2 KB) data in each connection to find *candidate* protocols that might match the stream. Bro then instantiates instances of these analyzers which concurrently process the stream from the beginning. Whenever an analyzer concludes the stream cannot belong to its protocol, it drops out of further analysis. Otherwise, it continues to process future packets as they are received.

We incorporate DPD into the Shunting framework by initially marking the corresponding signature analyzers as essential. At the 2 KB limit, we demote the signature analyzers to unessential. If no other essential analyzer remains active at that point, then Bro installs a *forward* entry to skip over the remainder of the connection (except for its final FIN/RST control packets, which are matched by the Shunt's higher-priority static filter).

If DPD did identify the flow's protocol, however, then Bro will have classified the corresponding analyzer as essential, and it (and other inessential analyzers) will continue receiving the flow's traffic. Thus, Shunting does not affect DPD's ability to detect the protocol present in a traffic flow.

Trace	Percentage forwarded	
	Bytes	Packets
UNIVERSITY I	54.9%	43.8%
UNIVERSITY II	58.1%	47.0%
UNIVERSITY III	69.9%	52.5%
LAB I	84.5%	75.7%
LAB II	88.2%	79.2%
SC I	91.1%	88.0%

Table 2: Fraction of forwardable (non-analyzed) traffic

7. EVALUATING THE SHUNT

To evaluate the efficacy of the Shunting architecture, we modified Bro's interface for reading trace files to preprocess packets read from traces using the Shunting decision tables. Doing so allowed us to evaluate the tradeoffs for different analysis/forwarding schemes, as developed in the previous section.

We used six traces: three from a large university with several 10s of thousands of users (University I, II and III), two from a research laboratory with 8,000 hosts (Lab I and II), and one trace from a supercomputing center with thousands of users (SC I). We developed our modifications to Bro's processing using only UNIVERSITY I, using the other traces solely for evaluation.

UNIVERSITY I spans one hour and captured 50% of the traffic crossing the border of the university, which employs per-flow load-balancing across two heavily-loaded Gigabit Ethernet links. The trace (captured mid-afternoon on a workday), which includes all packets and their payloads, was constructed from subtraces captured with a cluster of six machines, and totals 222 GB. UNIVERSITY II consists of one hour of traffic, totaling 196 GB, recorded at 4PM on a Friday. We collected UNIVERSITY III at 2–3AM on a Saturday morning, to reflect an off-hours workload. It totals 109 GB.

Due to a node failure undetected during the capture process, UNIVERSITY II and UNIVERSITY III only captured 41% of the traffic rather than 50%. One subtrace on UNIVERSITY III reported a .02% packet drop,[4] while all other traces reported no drops.

LAB I consists of all traffic during an afternoon workday hour, recorded at the Laboratory's 10 Gbps access link, totaling 89 GB of data. The packet recording process reported a measurement drop rate of 0.4% of the packets. LAB II consists of two hours of TCP-only traffic recorded two years earlier at the same facility, also during the afternoon of a workday. The trace totals 117 GB; unfortunately, no measurement drop information is available.

SC I consists of all traffic seen at the border (but inside the firewall, unlike LAB I and LAB II) of the supercomputing center, recorded for 69 minutes during the afternoon of a workday. It totals 73 GB, with a reported measurement drop rate of 0.07%.

For evaluating Shunting, our primary interest is in the proportion of traffic that we can forward without needing further analysis. For an IPS, this represents the fraction of the traffic processed directly

[4] Apparently due to a transient glitch on a collection node.

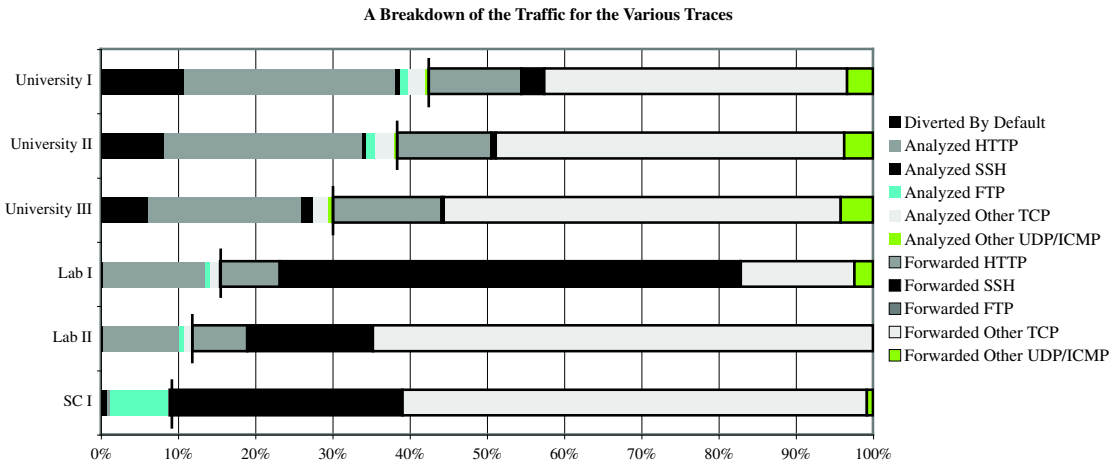

Figure 4: Breakdown of types of traffic that require analysis vs. forwarding.

by the Shunt without involving bus-transfer overhead. Clearly, the crucial question is to what degree we maintain a sound level of security analysis even in the presence of such offload; thus, we strive to formulate algorithms for deciding which traffic to skip that gain the largest offload for the least loss of detection opportunities. We frame our decisions in this regard in the remainder of the section. In addition, we evaluate the behavior of cache sizes using the UNIVERSITY I trace.

7.1 Evaluating the Fraction Forwarded

We processed each trace with Bro running a number of analyzers, including: generic TCP connection analysis; SSH; HTTP requests and replies; dynamic protocol detection; SMTP; IRC (including bot detection); POP; DNS; and scan detection. We also evaluated on a per-connection basis the amount of traffic analyzed versus directly forwarded.

Table 2 summarizes the overall results. For the somewhat less diverse laboratory and supercomputing environments, the offload gain is very large, 75–91% of the packets and bytes. Even for the university environment, we see significant gains along the lines of 50% of the packets and bytes.

Figure 4 breaks down the traffic by bytes analyzed vs. bytes forwarded, for various types of traffic. The Shunt always diverts unclassified traffic (not present in any decision table) to the Analysis Engine, which we show at the lefthand edge of the figure. Following this portion of the traffic we plot the makeup of analyzed traffic (diverted to the Analysis Engine because an analyzer needs to see it) for different application protocols, and then the makeup of forwarded traffic that the Analysis Engine can skip processing due to use of Shunting. (We mark the beginning of this last group with a vertical line to help distinguish it from the preceding group.) Note that the applications presented in the plot reflect not only traffic seen on the application's well-known port, but also traffic identified using dynamic protocol detection.

Indeed, for the university traces we find that the main benefits from Shunting come from the dynamic protocol detection analysis, which often can examine just the beginning of a flow and then forward the remainder if it belongs to an application protocol that the NIDS does not analyze. We also find both the University traces and SC I dominated by large-volume flows.

In contrast, in LAB I's traffic mix, SSH dominates. Such an environment provides a near best-case for Shunting, since SSH gains very large benefit by skipping over large, unanalyzable encrypted transfers. SC I also has a traffic mix dominated by SSH and other large, unanalyzable file transfers. (SC I is also the only environment where the FTP analyzer sees enough traffic to significantly benefit from Shunting.)

The figure demonstrates the central role that traffic types play in the effectiveness of Shunting: SSH can be almost completely forwarded, while even with Shunting HTTP traffic requires significant analysis.

We also see how, even at a single site, the mix of traffic over the course of a day can present significantly different loads to a Shunt-based IDS: comparing UNIVERSITY I (captured during the workday) with UNIVERSITY III (in the middle of the night) we see significant differences, with UNIVERSITY III exhibiting a considerably higher fraction of unanalyzable traffic, and thus deriving greater benefit from Shunting.

Finally, we find that Shunting is somewhat less effective at offloading packets compared to bytes. Since Shunting's benefits are greatest for heavy-tailed flows, it is natural to expect that we can forward a greater fraction of bytes than packets.

7.2 Sizing the Connection Cache

A critical design parameter for the Shunt is sizing the connection cache: it must be large enough to minimize the miss rate, but small enough to limit the hardware cost.

To assess this tradeoff, we analyzed the UNIVERSITY I trace to identify all of the forwarded packets, each of which corresponds to a potential connection table entry. We then fed the resulting access patterns into a custom-written cache simulator to evaluate the miss-rate for different connection table cache sizes. (For this analysis, we did not assume eviction of entries upon observing a TCP FIN or RST control packet, an optimization that could further reduce the miss rate.)

Figure 5 plots the miss rate (Y-axis, log-scaled) as we vary the cache size (X-axis, log scaled) for different cache organizations and eviction policies. We see that the 64K-entry cache used by our hardware implementation provides ample head-room. A direct-mapped cache would experience a 0.41% miss rate, while for a

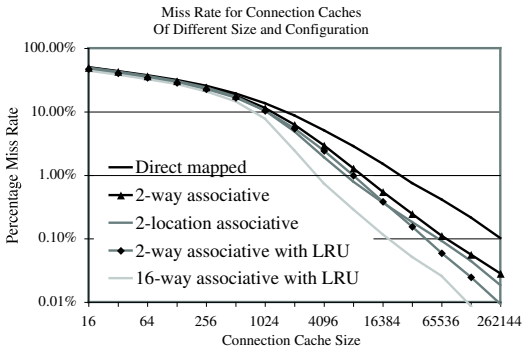

Figure 5: Connection table cache miss rates UNIVERSITY I, with varying cache sizes organizations, and eviction policies (random or LRU).

2-way associative cache this drops to 0.11%. A 2-location associative cache, without any searching, further reduces the miss rate to 0.092%. Finally, a 2-way associative cache with LRU replacement provides a 0.059% miss rate.

Although the associative cache with LRU replacement provides a better miss rate than the 2-location associative cache with random replacement, we prefer the location-associative cache because it is easier to implement. To implement an LRU cache, the Shunt would need to update connection entries upon receipt of packets, as forwarded packets are never sent to the Analysis Engine (so it could not track which entries are least-recently used).

Also of particular note is the relative effectiveness of even small caches. A 2-location associative cache with just 4K entries provides a miss rate of only 1.9%. If entries require 16 bytes, this suggests that a connection cache of just 64 KB would be effective. Thus, a Shunt built as an ASIC or using a programmable-firmware Ethernet card could readily use on-chip memory for its tables.

8. FUTURE WORK

Our primary plans involve porting the Shunt implementation to the 2.1 version of the NetFPGA board and advancing the integration with Bro to a level appropriate (and tested for) 24x7 operational use. The 2.1 NetFPGA board both fixes the input FIFO problem that causes lockup for high-data-rate flows and also includes 64 MB of SDRAM, a larger FPGA, and greater availability in terms of number of units we can obtain.

With the new board we will complete final integration of the Shunt into Bro and operationally deploy it in our network. Since the designers of the NetFPGA 2.1 board plan to also make it commercially available, we hope to deploy at third party sites to increase our operational experience with Shunting, as well as provide enhancements to Bro for intra-enterprise operation.

In addition, since we have validated that small connection caches suffice, we are now investigating whether firmware-programmable Ethernet cards could directly implement a Shunt.

9. CONCLUSIONS

We have developed a new model for packet processing, Shunting, which provides significant benefits for network intrusion prevention in an environments for which an IPS can dynamically designate portions of traffic stream as not requiring further analysis. The architecture splits processing into a relatively simple, table-driven hardware device that processes the entire traffic stream inline, and a flexible analyzer (the IPS proper) that can run separately, communicating with the device either over a local bus or a dedicated Gbps Ethernet link.

We argue that this architecture can realize a number of significant benefits: (1) enabling what previously was a passive intrusion detection system to operate inline, gaining the power of intrusion *prevention*, as well as the opportunity to "normalize" traffic to remove ambiguities that attackers can exploit for evasion [11]; (2) significantly offloading the IPS by providing a mechanism for it to make fine-grained, dynamic decisions regarding which traffic streams it analyzes, and (to a degree) which sub-elements of stream it sees; (3) enabling large-scale, fine-grained (per-address or per-connection) blocking of hostile traffic sources; and (4) providing a mechanism for an IPS to protect itself from overload if it can identify sources that load in excessively.

We have already developed hardware capable of performing the Shunting operations [27], demonstrating that we can keep the specialized cache within the Shunt hardware relatively small, with 64 KB caches producing viably low miss rates. In this work, as well as framing the broader Shunting architecture, we have adapted the Bro intrusion detection system to work with Shunting. We find that with a modest set of additions to its analysis, it can offload 55–90% of its traffic load, as well as gaining the major benefit of enabling fine-grained intrusion *prevention*.

10. ACKNOWLEDGMENTS

The high-level shunting architecture was conceived by Eli Dart and Stephen Lau of the Lawrence Berkeley National Laboratory, and prototyped by them at IEEE Supercomputing. Scott Campbell of LBNL has also been instrumental in exploring ways to realize shunting-like functionality using features offered by commercial high-end routers.

Our thanks to Weidong Cui and Christian Kreibich for volunteering to have their daily network traffic "live behind" our shunting software for testing purposes, and to Robin Sommer for helpful comments on an earlier draft of this paper.

This research was made possible by a grant from the US Department of Energy, Office of Science, and by the National Science Foundation under grants STI-0334088, NSF-0433702 and CNS-0627320, for which we are grateful.

11. REFERENCES

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

[2] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A hardware platform for network intrusion detection and prevention. In *Proceedings of The 3rd Workshop on Network Processors and Applications (NP3)*, 2004.

[3] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of the Passive and Active Measurement Conference*, 2000.

[4] J. Coppens, S. Van den Berghe, H. Bos, E. Markatos, F. De Turck, A. Oslebo, and S. Ubik. Scampi - a scaleable and programmable architecture for monitoring gigabit networks. In *Proceedings of E2EMON Workshop*, September 2003.

[5] J. Coppens, E.P. Markatos, J. Novotny, M. Polychronakis, V. Smotlacha, and S. Ubik. Scampi - a scaleable monitoring platform for the internet. In *Proceedings of the 2nd*

International Workshop on Inter-Domain Performance and Simulation (IPS 2004), March 2004.

[6] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, Aug 2003.

[7] 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.

[8] 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.

[9] L. Deri. Passively monitoring networks at gigabit speeds using commodity hardware and open source software. In *Proceedings of the Passive and Active Measurement Conference*, 2003.

[10] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proceedings of the USENIX Security Symposium*, 2006.

[11] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, end end-to-end protocol semantics. In *Proceedings of the 9th USENIX Security Symposium*, 2001.

[12] G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring very high speed links. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 267–271, 2001.

[13] Intel. Intel(r) network infrastructure processors: Extending intelligence in the network, 2005.

[14] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *Proceedings of the ACM Internet Measurement Conference*, 2005.

[15] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.

[16] E. Markatos. Scampi detailed architecture design. http://www.ist-scampi.org/publications/deliverables/D1.3.pdf, 2005.

[17] N. McKeown and G. Watson. Netfpga 2.0, http://klamath.stanford.edu/nf2/.

[18] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In *Proceedinges of ACM Internet Measurement Conference*, October 2004.

[19] V. Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, 1994.

[20] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

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

[22] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Calgary, Alberta, Canada, 1998.

[23] Andre Seznec. A case for two-way skewed-associative caches. In *ISCA*, pages 169–178, 1993.

[24] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, 2003.

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

[26] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *RAID 2007 (to appear)*.

[27] Nicholas Weaver, Vern Paxson, and José M. González. The shunt: an fpga-based accelerator for network intrusion prevention. In *FPGA*, pages 199–206, 2007.

[28] 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.

[29] 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.