

1-14-2007

Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface

Derek L. Schuff
Purdue University

Vijay S. Pai
Purdue University

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

Schuff, Derek L. and Pai, Vijay S., "Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface" (2007).
ECE Technical Reports. Paper 342.
<http://docs.lib.purdue.edu/ecetr/342>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface *

Derek L. Schuff and Vijay S. Pai
Purdue University
West Lafayette, IN 47907
{dschuff, vpai}@purdue.edu

Abstract

This paper presents and evaluates a strategy for integrating the Snort network intrusion detection system into a high-performance programmable Ethernet network interface card (NIC), considering the impact of several possible hardware and software design choices. While currently proposed ASIC, FPGA, and TCAM systems can match incoming string content in real-time, the system proposed also supports the stream reassembly and HTTP content transformation capabilities of Snort. This system, called LineSnort, parallelizes Snort using concurrency across TCP sessions and executes those parallel tasks on multiple low-frequency pipelined RISC processors embedded in the NIC. LineSnort additionally exploits opportunities for intra-session concurrency. The system also includes dedicated hardware for high-bandwidth data transfers and for high-performance string matching.

Detailed results obtained by simulating various software and hardware configurations show that the proposed system can achieve intrusion detection throughputs in excess of 1 Gigabit per second for fairly large rule sets. Such performance requires the system to use hardware-assisted string matching and a small shared data cache. The system can extract performance through increases in processor clock frequency or parallelism, allowing additional flexibility for designers to achieve performance within specified area or power budgets. By efficiently offloading the computationally difficult task of intrusion detection to the network interface, LineSnort enables intrusion detection to run directly on PC-based network servers rather than just at powerful edge-based appliances. As a result, LineSnort has the potential to protect servers against the growing menace of LAN-based attacks, whereas traditional edge-based intrusion detection deployments can only protect against external attacks.

*This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448 and CNS-0532452.

1 Introduction

Edge-based firewalls form a traditional approach to network security, based on the assumption that malicious attacks are sent to a target system across the global Internet. However, firewalls do not protect applications running on externally exposed ports, nor do they protect against attacks originating inside the local-area network. Such attacks may arise once a machine inside the network has been compromised by an otherwise undetected attack, virus, or user error (such as opening an email attachment). LAN-based attacks are potentially even more dangerous than external ones as they may propagate at LAN speeds and attack internal services such as NFS.

Network intrusion detection systems (NIDSes), such as Snort, may run on individual host machines on a LAN [25]. However, the overheads of running such systems are quite high, as they must reassemble network packets into streams, preprocess the data, and scan the streams for matches against specified string content and regular expressions. These overheads limit Snort to a traffic rate of less than 500 Mbps on a modern host machine (2.0 GHz AMD Opteron processor system), even with full access to the CPU. Profiling shows that no single portion of Snort is the sole bottleneck: although string content matching is the most important subsystem, the other portions of Snort consume nearly half the required cycles. Further, the CPU power on a running server machine would also need to be shared with the server applications and the operating system. Consequently, it is not feasible to deploy Snort directly on high-end network servers that must serve data at Gigabit rates or higher. These observations tend to limit the deployment of NIDSes to high-end edge appliances.

To allow for the reliable detection and logging of attacks within the LAN, Ganger et al. propose self-securing network interfaces, by which the network interface card executes the NIDS on data as it streams into and out of the host [11]. Their prototype used a PC placed between the switch and the target PC as the self-securing network interface. The newly inserted PC ran a limited operating system and several custom-written applications that scanned traffic looking for suspicious behaviors. Although valuable as a proof-of-concept, a more usable self-securing network in-

terface would need to be implemented as an Ethernet network interface card and execute a more standard NIDS.

This paper has two key contributions. First, this paper presents a strategy for integrating the Snort network intrusion detection system into a high-performance programmable Ethernet network interface. The architecture used in this paper draws from previous work in programmable Ethernet controllers, combining multiple low-frequency pipelined RISC processors, nonprogrammable hardware assists for high-bandwidth memory movement, and an explicitly-managed partitioned memory system [4, 34]. The firmware of the resulting Ethernet controller parallelizes Snort at the granularity of TCP sessions, and also exploits opportunities for intra-session concurrency. Previous academic and industrial work on special-purpose architectures for high-speed NIDS have generally only dealt with content matching [5, 9, 24, 29, 30, 36]. General Snort-style intrusion detection allows stronger security than content-matching alone by reassembling TCP streams to detect attacks that span multiple packets, by transforming HTTP URLs to canonical formats, and by supporting other types of Snort tests.

This paper's second contribution is to explore and analyze the performance impact of various hardware and software design alternatives for the resulting self-securing network interface, which is called LineSnort. The paper considers the impact of the Snort rulesets used and scalability with regard to processors and frequency, policies for assigning flows to processors, the benefits of hardware-assisted (as opposed to pure software) string content matching, and the importance of data caches. The results show that throughputs in excess of 1 Gigabit per second can be achieved for fairly large rule sets using hardware-assisted string matching and a small shared data cache. LineSnort can extract performance through increases in processor clock frequency or parallelism, allowing an additional choice for designers to achieve performance within specified area or power budgets.

2 Background

2.1 Intrusion Detection

Snort is the most popular intrusion-detection system available. The system and its intrusion-detection rule set are freely available, and both are regularly updated to account for the latest threats [25]. Snort rules detect attacks based on traffic characteristics such as the protocol type (TCP, UDP, ICMP, or general IP), the port number, the size of the packets, the packet contents, and the position of the suspicious content. Packet contents can be examined for exact string matches and regular-expression matches. Snort can perform thousands of exact string matches in parallel using one of several different multi-string pattern matching algorithms [2, 8, 14, 35]. Snort maintains separate string-matching state machines for each possible target port, allowing each state machine to scan for only the traffic signatures relevant to a specific service.

Beyond content signature matching alone, Snort includes preprocessors that perform certain operations on the data stream. Some important preprocessors include **stream4** and **HTTP Inspect**. The stream4 preprocessor takes multiple packets from a given direction of a TCP flow and builds a single conceptual packet by concatenating their payloads, allowing rules to match patterns that span packet boundaries. It accomplishes this by keeping a descriptor for each active TCP session and tracking the state of the session. It also keeps copies of the packet data and periodically "flushes" the stream by reassembling the contents and passing a pseudo-packet containing the reassembled data to the detection engine. The HTTP preprocessor converts URLs to normalized canonical form so that rules can specifically match URLs rather than merely strings or regular expressions. For example, it decodes URLs containing percentage escape symbols (e.g., "%7e" instead of the tilde symbol) to their normal forms [6]. It also generates absolute directories instead of the ". . /" forms sometimes used to hide directory traversals. After this decoding, the same pattern matching algorithm that Snort uses for packet data is used on the normalized URL as specified by "URI-content" rules.

The following rules illustrate how Snort uses network data characteristics to detect attacks:

- Buffer overflow in SMTP Content-Type: TCP traffic to SMTP server set, established connection to port 25, string "Content-Type:", regular expression "Content-Type:[^\r\n]300," (i.e., 300 or more characters after the colon besides carriage return or newline)
- Cross-site scripting in PHP Wiki: TCP traffic to HTTP server set, established connection to HTTP port set, URI contains string "/modules.php?", URI contains string "name=Wiki", URI contains string "<script"
- Distributed Denial-of-Service (DDOS) by Trin00 Attacker communicating to Master: TCP traffic to home network, established connection to port 27665, string "betaalmostdone"

The above rules test multiple conditions and use the logical AND of those conditions to confirm an attack. The Snort ruleset language includes 15 tests based on packet payload and 20 based on headers. Over 95% of the rules in recent Snort rulesets specify string content to match, but no rule specifies *only* string content matching. Some rules specify multiple string matches, and all are augmented with other tests, such as specific ports, IP addresses, or URLs. About 30% of the rules use regular expressions, but nearly all of these check for an exact string match as a first-order filter before the more time-consuming regular expression match.

Figure 1 depicts the Snort packet processing loop. Snort first reads a packet from the operating system using the pcap library (also used by tcpdump). The decode stage translates the network packet's tightly-encoded protocol headers and associated information into Snort's loosely-encoded packet data structure. Snort then invokes preprocessors that use and manipulate packet data in various ways. The rule-tree lookup and pattern matching stage determines which rules are relevant for the packet at hand and checks the packet content for the attack signatures de-

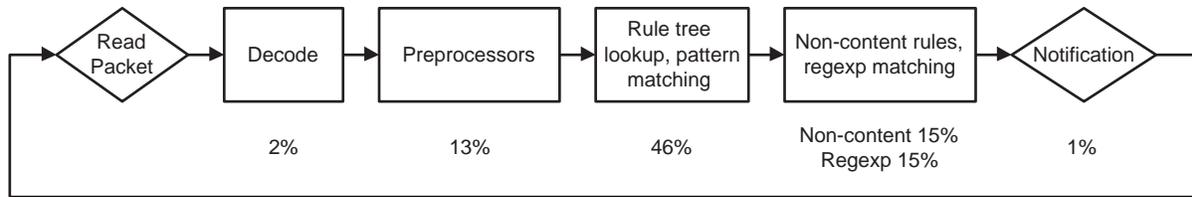


Figure 1. The Snort packet processing loop with percentage of time spent in each phase

fined in the string rules using the multi-string matching algorithms. Packets may match one or more strings in the multi-string match stage. Each match is associated with a different rule, and for each of those rules, all the remaining conditions are checked, including strings, non-content conditions and regular expressions. Because each match from the multi-pattern algorithm may or may not result in a match for its rule as a whole, this stage may be thought of as a “verification” stage. The last stage notifies the system owner through alerts related to the specific rule matches.

Below each stage in Figure 1 is the percentage of execution time spent in that phase for a representative network test pattern drawn from the 1998-1999 DARPA intrusion detection evaluation [13]. The profile shown here was gathered using the `oprofile` full-system profiling utility running on a Sun Fire v20z with two 2.0 GHz dual-core Opterons (4 processors total). Snort only runs on one processor. The system has 4 GB of DRAM and uses Linux version 2.6.8. The Snort configuration used a full recent rule-set and included the most important preprocessors: stream4 and HTTP Inspect as described above. The Snort code profiled here is modified to read all of its packets sequentially from an in-memory buffer to allow the playback of a large network trace, so packet reading consumes no time. The system achieves 463 Mbps inspection throughput for this trace. The stages shown in Figure 1 account for 92% of execution time, with an additional 8% going into miscellaneous library calls, operating system activity, and other applications running on the system. This additional component is not relevant in the context studied in this paper and is thus not considered further.

As Figure 1 shows, string content matching is a major component of intrusion detection, constituting 46% of execution time. Similar observations by others have led some researchers to propose custom hardware support for this stage, based on ASICs [3, 30], TCAMs [36], or FPGAs [5, 24, 29]. Some have also proposed support for regular expressions [9, 24]. Such hardware engines increase performance both by eliminating instruction processing overheads and by exploiting concurrency in the computation. Although some of these hardware systems have been tested using strings extracted from Snort rules, all have used unified state machines for all string matching rather than per-port state machines as in the Snort software.

Although Figure 1 shows the importance of string content matching, it also shows that no single component of intrusion detection makes up the majority of execution time. This is not unexpected since the rules invariably include multiple types of tests, not just string content matching.

Thus, any performance optimization strategy should target the full intrusion detection system.

2.2 Programmable Ethernet Controllers

The first widely-used Gigabit Ethernet network interface cards were based on programmable Ethernet controllers such as the Alteon Tigon [4]. Although more recent Gigabit Ethernet controllers have generally abandoned programmability, programmable controllers are again arising for 10 Gigabit Ethernet and for extended services such as TCP/IP offloading, iSCSI, message passing, or network interface data caching [1, 15, 18, 22, 27]. Although integrating processing and memory on a network interface card may add additional cost to the NIC, this cost will still likely be a small portion of the overall cost of a high-end PC-based network server. Consequently, such costs are reasonable if programmability can be used to effectively offload, streamline, or secure important portions of the data transfer.

Network interface cards must generally complete a series of steps to send and receive Ethernet frames, which may involve multiple communications with the host using programmed I/O, DMA, and interrupts. Programmable network interfaces implement these steps with a combination of programmable processors and nonprogrammable hardware assists. The assists are used for high-speed data transfers between the NIC and the host or Ethernet, as efficient hardware mechanisms can be used to sequentially transfer data between the NIC local memory and the system I/O bus or network. Willmann et al. studied the hardware and software requirements of a programmable 10 Gigabit Ethernet controller [34]. They found that Ethernet processing firmware does not have sufficient instruction-level parallelism or data reuse for efficient use of multiple-issue, out-of-order processors with large caches; however, a combination of frame-level and task-level concurrency allows the use of parallel low-frequency RISC cores. Additionally, the specific data access characteristics of the network interface firmware allow the use of a software-managed, explicitly-partitioned memory system that includes on-chip SRAM scratchpads for low-latency access to firmware metadata and off-chip graphics DDR (GDDR) DRAM for high-bandwidth access to high-capacity frame contents.

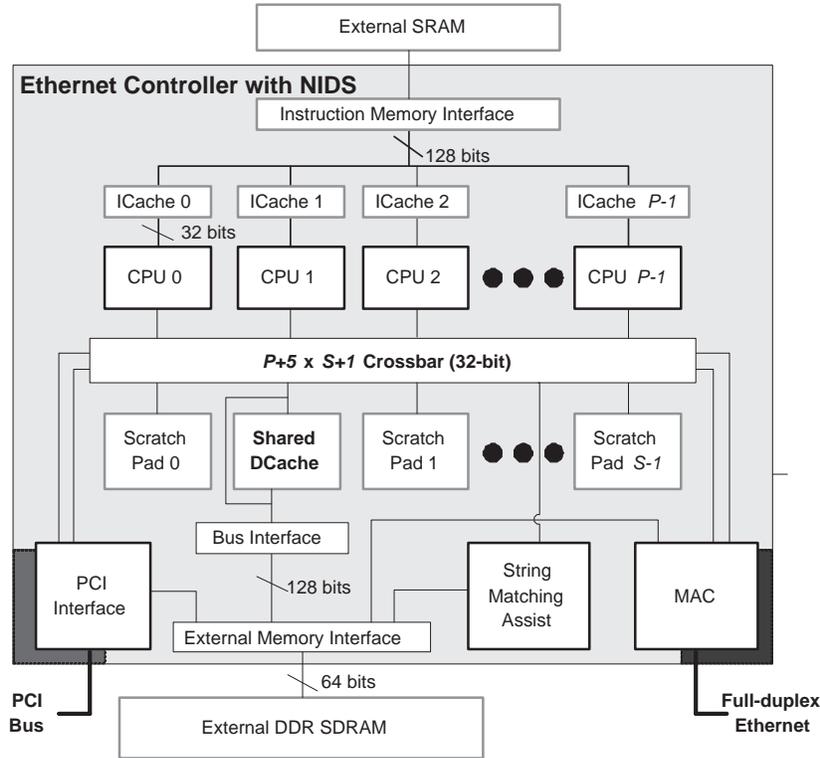


Figure 2. Block diagram of proposed Ethernet controller architecture

3 A Self-Securing Network Interface Architecture

As discussed in Section 2, programmable network interfaces have been proposed as resources for offloading various forms of network-based computation from the host CPU of high-end PC-based network servers. Offloading intrusion detection from the host could be quite valuable because this service potentially requires processing every byte sent across the network. This paper proposes a self-securing Ethernet controller that uses multiple RISC processors along with event-driven protocol processing firmware and special-purpose hardware for string matching. Figure 2 shows the architecture of the proposed Ethernet controller, which is based upon the 10 Gigabit Ethernet design by Willmann et al. [34]. The individual components target the following uses:

- Programmable processors: control-intensive computation, needs low latency
- Memory transfer assists (DMA and MAC): data transfers to external interfaces, need high bandwidth
- String matcher: data processing, needs high bandwidth
- Graphics DDR (GDDR) SDRAM: for high capacity and high bandwidth access to network data and dynamically allocated data structures
- Banked SRAM scratchpads: for low latency on accesses to fixed control data
- Instruction caches: for low latency instruction access

- Data cache: for low latency on repeated data accesses

Of these components, the data cache did not appear in the previous works discussed. The remainder of this section discusses the architectural components in greater detail.

Programmable Processing Elements. Ethernet protocol processing and the portions of Snort other than string matching are executed on RISC processors that use a single-issue five-stage pipeline and an ISA based on a subset of the MIPS architecture. Parallelism is a natural approach to achieving performance in this environment because packets from independent flows have no semantic dependences between them. Additionally, alternatives to parallelism such as higher frequency or superscalar issue are not viable options because network interfaces have a limited power budget and limited cooling space.

Memory Transfer Assists. As in previous programmable network interfaces, this system includes hardware engines to transfer data between the NIC memory and the host memory or network efficiently [4, 19, 34]. These assists, called the DMA and MAC assists, operate on command queues that specify the addresses and lengths of the memory regions to be transferred to the host by DMA or to the network following the Ethernet medium-access control policy. The DMA assist is also used to offload TCP/IP checksumming from the host operating system [20].

String Matching Hardware. High-performance network intrusion detection requires the ability to match string content patterns efficiently. Thus, the controller integrates

string matching assist hardware. This assist could be based on almost any of the previously-proposed ASIC, TCAM, or FPGA-based designs mentioned in Section 2. The string-matcher operates by reading from a dedicated task queue describing regions of memory to process and then reading those regions of memory (just like the DMA/MAC assists). Since previously-reported string and regular expression matchers have been shown to match content at line-rate (e.g., [3, 5, 24, 29, 30, 36]), the only potential slowdown would be reading the input data from memory. This penalty can be managed by allocating data appropriately in the memory hierarchy.

One major difference from conventional Snort string matching, however, is the use of a single table for all strings as described in Section 2; in contrast, standard Snort uses separate tables for each set of network ports. Such a solution may not be feasible for embedded hardware string matchers that have limited storage capacity. Consequently, fast multipattern string matching alone will likely generate false positives for rules that only apply to other ports.

Memory System. The memory system consists of small instruction caches for low-latency access to the instruction stream, banked SRAM scratchpads for access to protocol processing and Snort control data, and an external GDDR SDRAM memory system for high-bandwidth access to high-capacity frame data. Two Micron GDDR SDRAM chips can provide a 64 MB memory with a 64-bit interface operating at speeds up to 600 MHz, double-clocked [23]. Additionally, the memory system includes a shared data cache to allow the programmable processors to use DRAM-based frame data when needed without incurring row activation overhead on every access. The cache is not used for scratchpad accesses or by the assists. For simplicity, the cache is write-through; consequently, it is important to avoid allocating heavily-updated performance-critical data in the DRAM. The cache does not maintain hardware coherence with respect to uncached accesses from the assists (which are analogous to uncached DMAs from I/O devices in a conventional system). Instead, the code explicitly flushes the relevant data from cache any time it requires communication with the assists. The system adopts a sharing model based on lazy release consistency (LRC), with explicit flushes before loading data that may have been written by the hardware assists [17]; unlike true LRC, this system need not worry about merging updates from multiple writers because the cache is write-through. An alternate design is also evaluated which uses private caches for each processor. Unlike the shared cache, the private caches can provide single-cycle access. However, flushes are also required to maintain coherence between processors. Each time a processor acquires a lock that protects cacheable data, it checks if it was the last processor to own the lock. If not, it must flush its cache before reading because another processor may have written to the data.

Table 1 summarizes the default architectural configuration parameters for this controller.

| Parameter | Value |
|------------------------|---------------|
| Processors | 2–8 @ 500 MHz |
| Scratchpad banks | 4 |
| Private I-cache size | 8 KB |
| Shared D-cache size | 64 KB |
| Cache block size | 32 bytes |
| Cache associativity | 2-way |
| GDDR SDRAM chips | 2 × 32 MB |
| DRAM frequency | 500 MHz |
| Row size | 2 KB |
| Row activation latency | 52 ns |

Table 1. Default architectural parameters

4 Integrating IDS into Network Interface Processing

Section 2 described the steps in the Snort network intrusion detection system and in programmable Ethernet protocol processing firmware. A self-securing programmable Ethernet NIC integrates these two tasks. Achieving high performance requires analyzing and extracting the concurrency available in these tasks and mapping the computation and data to the specific resources provided by the architecture. Previous work has shown strategies to extract frame-level and task-level concurrency in programmable Ethernet controller firmware [19, 34]. Consequently, the following will focus on executing Snort efficiently for the architecture of Section 3. The resulting self-securing Ethernet network interface is called LineSnort.

The architecture of LineSnort requires parallelism to extract performance. Since both Snort and Ethernet protocol firmware process one packet at a time, packet-level concurrency seems a natural granularity for parallelization. Recall the Snort processing loop depicted in Figure 1. Unlike conventional processors, LineSnort does not need a separate stage to read packets since this is already done for protocol processing itself. The remaining stages, however, present some obstacles that must be overcome. In particular, the data structures used by stream reassembly must be shared by different packets, making packet-level parallelization impractical.

Stream reassembly is implemented using a tree of TCP session structures that is searched and updated on each packet so that the packet’s payload is added to the correct stream of data. A TCP session refers to both directions of data flow in a TCP connection, considered together. The information related to the TCP session must be updated based on each TCP packet’s header and can track information such as the current TCP state of the associated connection. Because the session structure tracks the TCP state and data content of its associated connection, packets within each TCP stream must be processed in-order. However, there are no data dependences between different sessions, so a parallelization that ensured that each session would be handled by only one processor could access the different sessions without any need for synchronization. Such a parallelization scheme implies that the Snort processing loop should

only use session-level concurrency (rather than packet-level concurrency), at least until stream reassembly completes.

It is worth noting that the IDS itself can be the target of denial-of-service attacks, and the stream reassembly stage is a vulnerable target component, which is doubly true when detection is hardware-assisted. Fortunately, Snort’s stream preprocessor already contains code which attempts to detect and mitigate the effects of such attacks, and this code can also be used by LineSnort.

Figure 3 shows the stages of Snort operating using this parallelization strategy. The parallel software uses distributed task queues, with one per processor. When a packet arrives in NIC memory and is passed to the intrusion detection code, it must be placed into the queue corresponding to its flow. The source and destination IP addresses are looked up in a global hash table. If the stream has an entry, the queue listed in the table is used. Otherwise the stream is assigned to whichever queue is currently shortest and the entry is added to the table, ensuring subsequent packets from the stream will go into the same queue. Stream reassembly is then performed for each session using the steps described in Section 2. The processor assigned to the queue places a pointer to the incoming packet data in its stream reassembly tree. Eventually, enough of the stream is gathered that Snort decides to flush it. This processor then finds a free stream reassembly buffer and copies the packet data from DRAM into the free buffer. These stream reassembly buffers are allocated in the scratchpad for fast access; the buffers are then passed to the hardware pattern matching assist by enqueueing a command descriptor. The reassembler also enqueuees a descriptor for HTTP inspection if the stream is to or from an HTTP port.

Although the above assignment strategy parallelizes stream reassembly and other portions of Snort using session-level concurrency, it provides no benefit for situations where there are fewer flows than processors (such as a high-bandwidth single-session attack). Such a situation can be supported by exploiting a property of stream reassembly: packets from the same flow that are separated by a stream reassembly flush point actually have no reassembly-related dependences between them since they will be reassembled into separate stream buffers. Consequently, if the processor enqueueing a new packet finds that its destination queue is too full (and that another queue is sufficiently empty), it can change the assignment of the flow to the emptier queue. This works because the stream reassembler is robust enough to handle flows encountered mid-stream (in the new queue), and, after a timeout occurs, will inspect and purge any streams that have not seen any recent packets (in the old queue). It is also possible that a flow may be assigned away from a queue and then later assigned back to it. To prevent improper reassembly of these disjoint stream segments, a flush point is inserted before the new section is added to the reassembly tree. Reassignment achieves better load balance and supports intra-flow concurrency for higher throughput; the cost is additional overhead in stream reassembly. In addition, if reassignment were to be too frequent, (i.e., on almost every packet), the benefits of stream reassembly would be negated; the requirement that the shortest queue be suf-

ficiently empty prevents this.

The string content matching assist dequeues the descriptor passed in from stream reassembly, scans the reassembled stream in hardware, and notes any observed rule matches. For each rule matched by the stream, it writes a descriptor into the global match queue, containing a pointer to the rule data. The next processor to dequeue from the match queue must verify each match reported by the content matcher. Each Snort rule specifies several conditions (which may include multiple strings), but the matcher only checks for the longest exact content string related to each rule. The verification stage checks each condition for the rule and handles alerting if necessary. At the same time, another processor may also dequeue the HTTP inspect descriptor if one was generated. The HTTP inspect portion of the firmware performs URL normalization and matches “URI-Content” rules to check for HTTP-specific attacks. If any “URI-Content” matches are found, verification is performed in the same manner as for normal content rules. The HTTP Inspect and verification stages are essentially unmodified from the original Snort. The rule data structures are allocated in DRAM since they are not updated while processing individual streams; the HTTP inspection data structures are allocated in the scratchpad, with one per processor. The notification stage shares common alerting mechanisms; no attempt was made to privatize this stage since it represents only 1.5% of the total computation even with complete serialization.

The Snort stages described above integrate into Ethernet protocol processing after new data arrives at the NIC (transmit or receive). Currently, LineSnort only acts to *detect* intrusions, just like the standard Snort; thus, it passes the frame onto the remainder of Ethernet processing regardless of whether or not there was a match. LineSnort could be used to *prevent* intrusions by dropping a suspicious frame or sending a reset on a suspicious connection, but that choice is orthogonal to the basic design of the system.

The firmware uses the scratchpads for all statically-allocated data (e.g., Ethernet processing control data, inter-stage queues, and target buffers for stream reassembly and HTTP Inspect). DRAM is used for frame contents and all dynamically-allocated data (e.g., the stream reassembly trees, the session assignment table, and the per-port rule-tree data). This data allocation does not suffer as a result of the simple write-through cache in the system because very little data in DRAM is actually updated by the processors during operation.

5 Evaluation Methodology

The architecture and firmware described in the previous sections are evaluated using the Spinach simulation infrastructure [33]. Spinach is a toolkit for building network interface simulators and is based on the Liberty Simulation Environment [32]. Spinach includes modules common to general-purpose processors (such as registers and memory) as well as modules specific to network interfaces (such as DMA and MAC assist hardware). Modules are connected through communication queues akin to the structural and

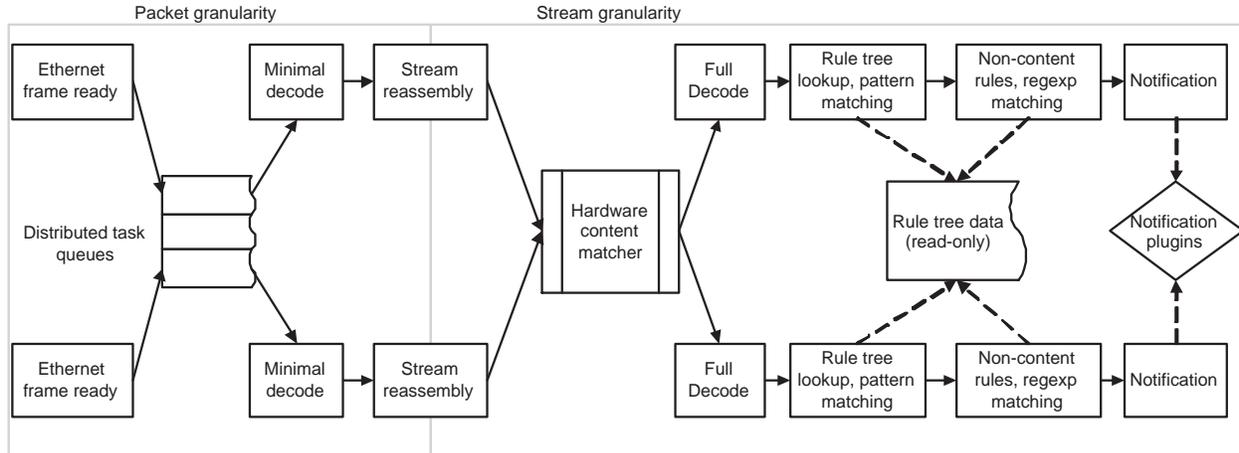


Figure 3. Firmware parallelization strategy used for LineSnort

hierarchical composition of an actual piece of hardware. Spinach has previously been validated against the Tigon-II programmable Gigabit Ethernet controller, has been used for studying architectural alternatives in achieving 10 Gigabit Ethernet performance, and has been extended to model graphics processing units [16, 33, 34].

Evaluating IDS performance is a difficult problem for several reasons. IDS performance is sensitive to the ruleset used and to packet contents, but there are very few network traces available with packet contents because of size and privacy concerns. In addition, programmable NIC performance is sensitive to packet size. Because of its method of distributing flows to processors, LineSnort is also sensitive to *flow concurrency*, the number of active flows running at one time. Hence, the traces used for evaluation should be realistic in terms of the packet and flow contents and the average packet sizes. The rulesets used should also be reasonable for the target machine.

The rulesets used are taken from those distributed by Sourcefire, the authors of Snort. Sourcefire's ruleset contains over 4000 rules describing vulnerabilities in all kinds of programs and services. Although an edge-based NIDS would need to protect against all possible vulnerabilities, a per-machine NIDS such as LineSnort only needs to protect against the vulnerabilities relevant to that particular server. For example, a Windows machine has no need to protect against vulnerabilities that exist only in Unix, and vice versa. Two rulesets are used in these tests: one with rules for email servers and one with rules for web servers. Both rulesets also include rules for common services such as SSH. The mail ruleset contains approximately 330 rules, and the web ruleset approximately 1050. Tailoring the ruleset to the target machine allows LineSnort to consume less memory and see fewer false positives.

LineSnort is evaluated using a test harness that models the behavior of a host system interacting with its network interface. The test harness plays packets from a trace at a specified rate, and Section 6 reports the average rate sustained by LineSnort over the trace. The rules used for testing do not distinguish between sent and received packets, so

all traces are tested using only the send side. Although all steps related to performing DMAs and network transmission are included in the LineSnort firmware, the actual PCI and Ethernet link bandwidths are not modeled, as these are constantly evolving and can be set according to the achieved level of performance.

The packet traces used to test LineSnort come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab, which simulates a large military network [13]. Because they were generated specifically for IDS testing, the traces have a good collection of traffic and contain attacks that were known at the time. However, because they were designed for testing IDS efficacy rather than performance, they are not realistic with respect to packet sizes or flow concurrency. To address this problem, a variety of flows were taken from several traces and reassembled to more closely match average packet sizes (≈ 778 bytes) seen in publicly available header traces from NLANR's Passive Measurement and Analysis website (pma.nlanr.net). Two traces, called LL1 and LL2, were created in this manner, using different ordering and interleaving among the flows.

6 Experimental Results

Figure 4 shows the base results for the Lincoln Lab traces using the web and mail rulesets for varying numbers of CPUs. There are many factors, both in hardware and software, that affect performance.

Rulesets. Rulesets have an important effect on any IDS, and the choice of rules will always involve performance tradeoffs. In LineSnort, the ruleset affects how much work the firmware must do in verifying content matches. Since the content matcher has only one string table, many matches will be for rules that do not ultimately lead to alerts; these "false positives" will be filtered out by the verification stage. A false positive can arise because the rule does not apply to the TCP/UDP ports in use or because it specifies additional conditions that are not met by the packet. If the ports do not match, the verification will complete quickly. However, if additional string or regular expression matching is required,

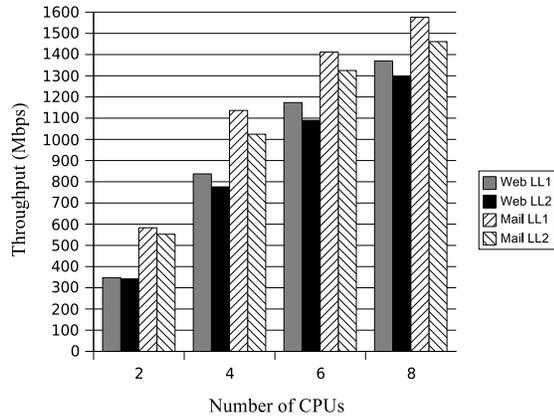


Figure 4. LineSnort throughput results achieved with 2–8 CPUs for mail and web rulesets with LL1 and LL2 input traces

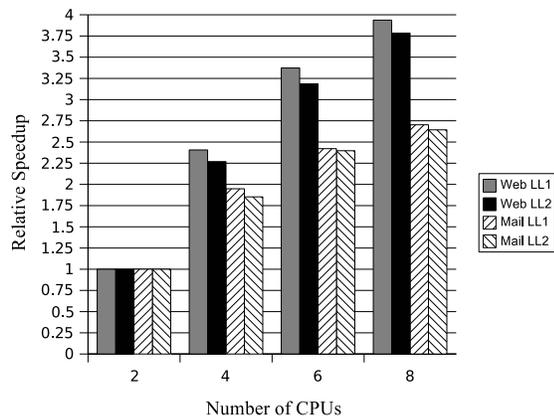


Figure 5. Relative speedup for increasing numbers of processors

verification will be much slower as the processor will need to read the packet and process its data using the matching algorithms. The mail ruleset actually generates more false positives at the string matching hardware because this ruleset contains many short strings that appear in many packets; however, the traces used contain far more web flows than email flows, so verification for the mail ruleset typically completes quickly after a simple port mismatch. In contrast, many web rules continue on to the more expensive checks, causing the web ruleset to spend nearly twice as much of its time on verification as the mail ruleset. Within the verification stage, the largest fraction of CPU time is consumed by regular expression matching.

Processor and Frequency Scaling. Figure 5 shows the speedup obtained by increasing the number of processors, normalized to the 2-processor throughput, for each trace and ruleset. Although the mail ruleset has higher raw throughput in all cases, the web ruleset scales much better with increasing numbers of processors. For the mail ruleset, two main factors contributed to the limitation in scalability. Because the mail ruleset requires a comparatively

small amount of verification work per packet, its throughput is more limited by the rate at which the processors can perform the TCP stream reassembly and copy the reassembled data from the DRAM into the scratchpad for inspection. Thus, there is more contention for the shared cache with the mail ruleset even with 4 processors than there is with the web ruleset with 8. The mail ruleset also triggers a larger increase in lock contention as the number of processors increases. Conversely, the web ruleset requires much more verification work per packet, and thus has lower overall throughput. Verification reads the reassembled packet data from scratchpad rather than DRAM, so the overall workload is more balanced between the scratchpad and the DRAM for the web ruleset, making increased DRAM contention less important. Moreover, because the scratchpad is banked, the increase in processors causes less contention there than for the cache.

Figure 6 shows the relationships between processor frequency and throughput for the LL1 trace with the web and mail rulesets. Scaling the processor frequency gives less than linear speedups because the DRAM latency and bandwidth is unchanged, so processors at higher frequencies spend more cycles waiting for DRAM accesses. The web ruleset scales better with frequency for the same reason it scales better with additional processors; it is slower overall and DRAM access makes up a smaller fraction of its time.

Flow Assignment. One of the most important influences on overall performance is the assignment of packet flows to processor queues. The assignment must be as balanced as possible to keep the processor load balanced. Otherwise, if the load imbalance is enough to fill up one of the stream reassembly queues, head-of-line blocking can occur, and all incoming packets must wait for the full queue. We evaluate two primary methods for queue assignment. In the *static assignment* method, the source and destination IP addresses are hashed in a symmetric manner, with the queue determined directly from the hash, so that packets from the same TCP session are always in the same queue. In *dynamic assignment*, the source and destination IP addresses are looked up in a hash table. If the stream has an entry, the queue stored in the table is used. Otherwise the stream is assigned to whichever queue is currently shortest, and the entry is stored, ensuring subsequent packets from the stream will go into the same queue. Dynamic assignment is then further enhanced with reassignment as discussed in Section 4.

The static assignment method is simpler and faster (which is good because flow assignment is serialized); however, dynamic assignment does a better job balancing the flows, and the reduction in head-of-line blocking compensates for the cost of using the hash table, particularly as more processors are added. The addition of dynamic reassignment provides further improvements for all combinations with 4 or more processors. Reassignment enforces balance and prevents head-of-line blocking, which compensates for the extra stream reassembly overhead; improvements varied from 2-16%, with an overall average of 7%.

Hardware String Matching. One of the main features of the design is the use of a hardware assist for multi-string

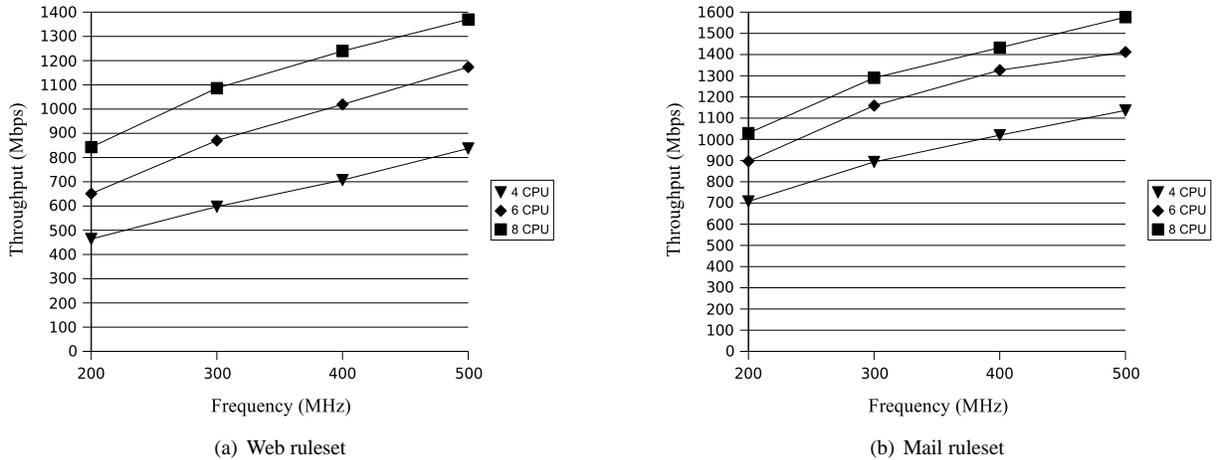


Figure 6. Impact of CPU frequency for LL1 trace

| | Web Ruleset | Mail Ruleset |
|-----------|-------------|--------------|
| LL1 Trace | 1.8 | 2.8 |
| LL2 Trace | 1.4 | 2.4 |

Table 2. Speedup for hardware-assisted string matching

matching. According to Amdahl’s Law, the benefit of accelerating string matching is determined primarily by the fraction of time spent in software string matching. Table 2 shows the actual speedups for each ruleset and trace combination when comparing hardware-assisted string matching with Snort’s baseline software string matcher. As discussed in Section 3, the hardware string matcher uses only a single rule table, while the software matcher uses separate rule sets per target port; consequently, the hardware string matcher sees more false positives for content checks on ports other than the actual target. Nevertheless, hardware string matching provides benefits of 1.4–2.8x for these tests. The mail ruleset benefits much more from the hardware matcher because, as previously discussed, the mail ruleset requires less time in the verification stage for our traces than the web ruleset. Thus, a larger fraction of time is spent in the multi-string matching phase, allowing a larger benefit by accelerating it. A common practice to improve IDS performance is to eliminate unneeded rules, or those that often trigger expensive verification such as regular expressions. Doing so would serve to increase the benefit of the hardware string matcher even more.

Caches. Processor cache sizes were evaluated using the static queue assignment method. Willmann et al. showed that the simple NIC firmware does not benefit much from data caches [35]; however, the same is not true of intrusion detection. Figure 7 shows the impact of cache size on throughput for LL1 trace. The working set of the firmware is small; even a 512 byte cache is about twice as fast as a system with no cache, and the largest cache is only 20-30%

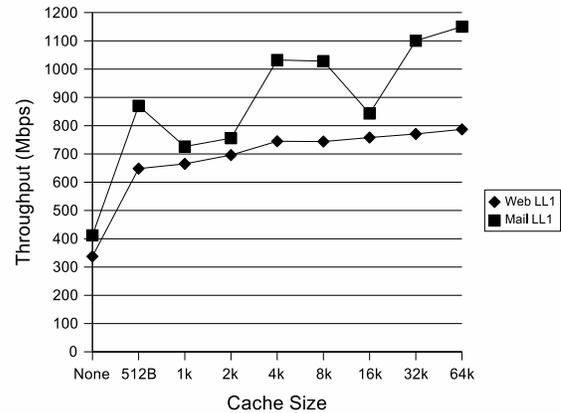


Figure 7. Effect of shared cache size on 4-processor throughput

faster than the smallest. Important data in the DRAM includes the the stream reassembly tree (which is shared for static assignment) and flow assignment table, which are accessed frequently, as well as the packet data itself; the payload is only accessed once but the IP and TCP headers are reused during flow assignment and stream reassembly. The 32-byte cache blocks also help to exploit spatial locality, reducing DRAM row activations by almost 40% with 1 kB of cache. Since the mail ruleset spends less time in verification than the web ruleset, it spends a relatively greater fraction in stream reassembly and thus depends more on DRAM access for its performance. Consequently, it benefits more from the presence of and increased size of the cache.

Private per-processor caches were also tested. For most tests however, the shared cache performed 10-30% faster for the same total amount of cache, despite the fact that the private cache had single-cycle access. The important shared data structures are frequently accessed and updated, and the private cache must often be flushed before accessing them to maintain coherence. By contrast, sharing the data cache

gives the benefits of inter-processor prefetching for these structures, and processors using the shared cache need only flush specific lines when reading frame data written by the assists.

The addition of flow reassignment requires that the stream reassembly tree be made private per processor. This means that although the overall number of packets and packet descriptor structures in the system is the same, the number of stream descriptors will be larger when reassignment occurs because one stream can be in multiple reassembly trees at once. This has two potential effects. First, it may increase the demand for cache because of the duplication of stream descriptors. Second, it may reduce the fraction of shared data accessed by the processors, potentially benefiting configurations with private caches, reducing or even reversing the performance gap between shared and private cache configurations.

Summary and Discussion. Performance of any intrusion detection system is heavily dependent on the ruleset and workload. For our traces the mail ruleset requires less time in the verification phase, and is thus more dependent on bandwidth and contention for the cache and DRAM. The web ruleset requires more verification time and so runs slower overall, but benefits more from increased number and frequency of CPUs. Load balancing and assignment of flows to processors for stream reassembly is also of primary importance for our parallelized approach. Dynamic assignment, though less predictable and more variable than static assignment, performs better, particularly as the number of processors increases. Hardware-assisted string matching was found to be essential to approach Gigabit speeds, as was at least a small amount of cache. However the DRAM footprint and amount of reuse are small enough that increasing cache size provides only limited benefit. In addition, a shared cache configuration performed better than private caches since coherence-related flushes were not needed for inter-processor sharing. Outside of these considerations, several paths can be taken to achieve near-Gigabit speeds; for simpler rulesets, such as the mail ruleset, 4 processors can be used at 400MHz, 6 at 300MHz, or 8 at 200MHz, depending on costs and power budgets. Likewise, a workload like the more demanding web ruleset would require 6 processors at 400MHz or 8 at 300MHz. Scaling the architecture beyond 8 processors would probably require improvements to the cache; a banked architecture for the shared cache, or private caches with hardware coherence could potentially address any contention problems. Alternatively, adding hardware assistance for stream reassembly (such as “gather” support at the string matcher) could substantially reduce the overall workload by eliminating copy overheads.

7 Related Work

Section 2 gives the background information related to the architectures and software used in this work. This section discusses other related issues in high-performance intrusion detection. Other researchers have also considered incorporating intrusion detection software onto network interfaces, but they have focused on the use of network pro-

cessors (NPUs). Clark et al. used the multithreaded microengines of an Intel IXP1200-based platform to reassemble incoming TCP streams that were then fed to an FPGA-based string content matcher [10]. Bos and Huang also propose a solution based on an IXP1200 that performs both stream reassembly and string content matching on the microengines [7]. Both of these systems target traffic rates of no more than 100 Mbps, although the former performs the string matching portion at Gigabit speeds. Although neither of these NPU-based works include the HTTP preprocessor, they could most probably add it with some performance degradation. Although NPUs can move network data at high rates, their multithreaded latency tolerance features are targeted toward the memory latencies seen in routers rather than the much higher DMA latencies seen when NICs must transfer data to and from the host. Consequently, previous work on using NPUs in NICs has seen performance imbalances related to the cost of DMA transfers [21].

In their work proposing self-securing network interfaces, Ganger et al. suggest integrating the secure network interfaces into switch ports rather than keeping them in individual machines [11]. This strategy aims to protect against physical intruders swapping out the NICs with standard ones. Although LineSnort’s design includes host-specific elements such as DMA, it would not need to change substantially to support switch integration.

The primary approach to high-performance intrusion detection today is through clustering multiple PCs that execute IDS using a load-balancing switch. Schaelicke et al. have proposed SPANIDS, a system that combines a specially-designed FPGA-based load-balancing switch that considers flow information and system load when redirecting packets to commodity PCs that run intrusion detection software [26]. Commercial offerings by companies such as Top Layer use L4–7 load-balancing switches to redirect traffic to a pool of intrusion-detection nodes, allowing high overall throughput scalability [31]. The expense and size of these clusters tends to make them practical only at edge-based deployments. LineSnort supports a different model, with the self-securing NIC as a key defense against both external and LAN-based attacks.

The research community has also proposed distributed NIDS, in which nodes at various points in the network track anomalies and collaboratively collect data that may indicate a system-level intrusion even if no specific host triggers an alert [12, 28]. Efforts in distributed NIDS have targeted collaboratively gathering additional information to *identify* intrusions, rather than processing packets at a faster rate. Thus, distributed NIDS is largely orthogonal to LineSnort.

8 Conclusions

This paper presents the architecture and software design of LineSnort, a programmable network interface card (NIC) that offloads the Snort network intrusion detection system (NIDS) from the host CPU of a high-end PC-based network server. The paper investigates and analyzes design alternatives and workloads that impact the performance of Line-

Snort, including the Snort rulesets used and the number and frequency of processor cores.

Leveraging previous work in programmable NICs and hardware content matching, LineSnort protects a single host from both LAN-based and Internet-based attacks, unlike edge-based NIDS which only guards against the latter. LineSnort exploits TCP session-level parallelism using several lightweight processor cores and a dynamic assignment of TCP flows to cores. LineSnort also exploits intrasession concurrency through flow reassignment, providing better load balance and higher throughput as the number of cores increases, or for workloads with poor flow concurrency. Simulation results using the Spinach toolkit and Liberty Simulation Environment show that LineSnort can achieve Gigabit Ethernet network throughputs while supporting all standard Snort rule features, reassembling TCP streams, and transforming HTTP URLs. To achieve these throughput levels, LineSnort requires a small shared cache, a string-matching assist in the hardware, and one of several options for the number and frequency of processors. Lightweight rulesets can achieve Gigabit throughput with 4 CPU cores at 400MHz, 6 at 300MHz, or 8 at 200MHz, while a more demanding ruleset requires 6 CPU cores at 400MHz or 8 at 300MHz.

Substantial prior work has considered the use of network interface cards as a resource for optimizing the flow of communication in a system, with targets ranging from simple checksumming to full protocol offload and customized services [1, 15, 18, 20, 22, 27]. LineSnort helps to deliver on the promise of programmable network interfaces by demonstrating that efficiently offloading the challenging problem of intrusion detection enables the new service of per-machine NIDS for network servers, providing a higher level of protection against both Internet-based and LAN-based attacks.

References

- [1] Adaptec. ANA-7711 Network Accelerator Card Specification, Mar. 2002.
- [2] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.
- [3] M. Aldwairi, T. Conte, and P. Franzone. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Comput. Archit. News*, 33(1):99–107, 2005.
- [4] Alteon Networks. *Tigon/PCI Ethernet Controller*, Aug. 1997. Revision 1.04.
- [5] Z. K. Baker and V. K. Prasanna. A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proc. of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004*, Apr. 2004.
- [6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF RFC 2396, Aug. 1998.
- [7] H. Bos and K. Huang. Toward software-based signature detection for intrusion prevention on the network card. In *Proc. of the Eighth International Symposium on Recent Advances in Intrusion Detection*, September 2005.
- [8] R. S. Boyer and J. S. Moore. A Fast String Search Algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.
- [9] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.
- [10] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Koné, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proc. of the Third Workshop on Network Processors and Applications*, February 2004.
- [11] G. R. Ganger, G. Economou, and S. M. Bielski. Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces. Technical Report CMU-CS-03-109, Carnegie Mellon School of Computer Science, Jan. 2003.
- [12] R. Gopalakrishna and E. H. Spafford. A Framework for Distributed Intrusion Detection using Interest Driven Cooperating Agents. In *Proc. of the 4th International Symposium on Recent Advances in Intrusion Detection*, Oct. 2001.
- [13] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. Technical Report 1062, MIT Lincoln Laboratory, 2001.
- [14] R. N. Horspool. Practical Fast Searching in Strings. *Software: Practice and Experience*, 10(6):501–506, 1980.
- [15] Y. Hoskote et al. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, Nov. 2003.
- [16] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics*, 24(4):1462–1482, 2005.
- [17] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
- [18] H. Kim, V. S. Pai, and S. Rixner. Improving Web Server Throughput with Network Interface Data Caching. In *Proc. of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250, October 2002.
- [19] H. Kim, V. S. Pai, and S. Rixner. Exploiting Task-Level Concurrency in a Programmable Network Interface. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.
- [20] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In *Proc. of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98, Aug. 1995.
- [21] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An Intel IXP1200-based Network Interface. In *Proc. of the 2003 Annual Workshop on Novel Uses of Systems Area Networks*, Feb. 2003.
- [22] K. Z. Meth and J. Satran. Design of the iSCSI Protocol. In *Proc. of the 20th IEEE Conference on Mass Storage Systems and Technologies*, Apr. 2003.
- [23] Micron. 256Mb: x32 GDDR3 SDRAM MT44H8M32 data sheet, June 2003. Available from www.digchip.com.
- [24] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages

- 31–38, Apr. 2003.
- [25] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proc. of the 13th USENIX Conference on System Administration*, pages 229–238, 1999.
 - [26] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proc. of the 2nd Conference on Computing Frontiers*, pages 315–322, 2005.
 - [27] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, Nov. 2001.
 - [28] S. R. Snapp et al. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype. In *Proc. of the 14th National Computer Security Conference*, pages 167–176, Washington, DC, Oct. 1991.
 - [29] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In *Proc. of the 13th International Conference on Field Programmable Logic and Applications*, pages 880–889, Sept. 2003.
 - [30] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 112–122, June 2005.
 - [31] Top Layer Networks. Network intrusion detection systems: Important IDS network security vulnerabilities. White Paper, September 2002.
 - [32] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, November 2002.
 - [33] P. Willmann, M. Brogioli, and V. S. Pai. Spinach: A Liberty-Based Simulator for Programmable Network Interface Architectures. In *Proc. of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 20–29, June 2004.
 - [34] P. Willmann, H. Kim, S. Rixner, and V. S. Pai. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 96–107, February 2005.
 - [35] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.
 - [36] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proc. of the 12th IEEE International Conference on Network Protocols*, pages 174–183, Oct. 2004.