An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention

Vern Paxson
International Computer Science Institute
Lawrence Berkeley National Laboratory
vern@icir.org

Robin Sommer
International Computer Science Institute
Lawrence Berkeley National Laboratory
robin@icir.org

Nicholas Weaver
International Computer Science Institute
nweaver@icsi.berkeley.edu

Abstract

It is becoming increasingly difficult to implement effective systems for preventing network attacks, due to the combination of (1) the rising sophistication of attacks requiring more complex analysis to detect, (2) the relentless growth in the volume of network traffic that we must analyze, and, critically, (3) the failure in recent years for uniprocessor performance to sustain the exponential gains that for so many years CPUs enjoyed ("Moore's Law"). For commodity hardware, tomorrow's performance gains will instead come from multicore *architectures in which a whole set of CPUs executes concurrently.*

Taking advantage of the full power of multi-core processors for network intrusion prevention requires an in-depth approach. In this work we frame an architecture customized for parallel execution of network attack analysis. At the lowest layer of the architecture is an "Active Network Interface" (ANI), a custom device based on an inexpensive FPGA platform. The ANI provides the in-line interface to the network, reading in packets and forwarding them after they are approved. It also serves as the front-end for dispatching copies of the packets to a set of analysis threads. The analysis itself is structured as an event-based system, which allows us to find many opportunities for concurrent execution, since events introduce a natural, decoupled asynchrony into the flow of analysis while still maintaining good cache locality. Finally, by associating events with the packets that ultimately stimulated them, we can determine when all analysis for a given packet has completed, and thus that it is safe to forward the pending packet—providing none of the analysis elements previously signaled that the packet should instead be discarded.

1 Introduction

In previous work [13], we have argued that the performance pressures on implementing effective network security monitoring are growing fiercely in multiple dimensions: (1) attacks continue to improve due to the adversarial nature of network security; (2) the power of simple "signature matching"—looking for specific strings or regular expressions within packets or reassembled byte streams—has drastically dwindled due to the major problems of false positives, polymorphism, and "zero day" attacks; (3) moving beyond signature-matching requires sophisticated analysis of protocols at higher semantic levels, and incorporating *context* correlated across multiple connections, hosts, sensors, and over time; (4) increasingly, we need to not only analyze traffic but *transform* it ("normalization" [8]) to eliminate broad classes of *evasion* threats, and, even more critically, to realize intrusion *prevention* systems); and (5) all of this in the presence of ever-increasing traffic volumes and rates.

In addition, in the face of all these performance pressures we have also lost our traditional ace-in-the-hole, Moore's Law for uniprocessors. Starting around 2002, the performance scaling curve for single CPUs has slowed precipitously. Over the fifteen prior years, uniprocessor performance increased 50–60% per year. But by 2006, performance was a *factor of three* slower than had the pre-2002 curve continued.

To perform sophisticated network analysis, it is hugely advantageous if we can draw upon the flexibility and inexpensive system costs of using general-purpose CPUs rather than custom hardware such as FPGAs and ASICs. Recently, hardware vendors have begun delivering commodity CPUs that again reflect Moore's Law-style scaling—but with the parallelization gains coming from *multi-core/multi-thread* architectures. However, while the *aggregated* throughput of such processors does in fact still follow Moore's law, to exploit the full power

of these architecture we must explicitly structure our applications in a highly parallel fashion, dividing the processing into concurrent tasks while minimizing inter-task communication.

Taking advantage of the full power of multi-core processors requires an in-depth approach in order to realize speedups for sophisticated analyses that require fine-grained coordination between concurrent threads. First, to provide intrusion *prevention* functionality (i.e., active blocking of malicious traffic), we must ensure that packets are only forwarded if *all* relevant processing gives approval. Second, to perform global analysis (e.g., worm contact graphs [6] or content sifting [18]) we must support exchange of state across threads, but we must minimize such inter-thread communication to maximize performance. Similarly, we must understand how the memory *locality* of different forms of analysis interacts with the ways in which caches are shared across threads within a CPU core and across cores. We need to be able to express the analysis in a form that is independent to the memory and threading parameters of a given CPU, so we can automatically retarget the implementations of analysis algorithms to different configurations. Finally, we must ensure that our approach is amenable to analysis by *performance debugging tools* that can illuminate the presence of execution bottlenecks such as those due to memory or messaging patterns.

In this work we frame an architecture customized for parallel execution of network attack analysis. The goal is to support the construction of highly parallel, inline network intrusion prevention systems that can fully exploit the power of modern and future commodity hardware. We first discuss related work, including the large potential of parallel processing for network security analysis (§ 2). We then sketch a high-level overview of our architecture (§ 3), and explore what pursuing it would mean in more concrete terms (§ 4), before briefly summarizing (§ 5).

2 Related Work

Parallelizing analysis. To date, efforts on exploiting parallelism for network security monitoring have focused heavily on *signature scanning*, i.e., detecting whether a packet (or sometimes a reassembled byte stream) contains a string of interest or matches a regular expression, and executing an action (such as drop or alert) associated with the signature. Much of this work has drawn inspiration from the popularity of "Snort" [16] and its large set of byte-level signatures. This work includes use of nondeterministic finite automata to match regular expressions [17], compiling regular expressions into deterministic finite automata [12], building optimized Aho-Corasick trees for sets of strings [21], and specialized architectures based on collections of highly optimized tiny state-machines [20].

A vital point regarding much of the previous parallel hardware design research is that it presumes a nearly *stateless* approach to attack detection. The systems either operate on single packets or assume that a separate process reassembles the TCP byte stream. Parallelizing richer, stateful hardware elements, such as TCP stream reassembly, have not been explored in as much depth; see our previous work in [4] and the discussion of prior efforts therein, including the vulnerability to such TCP processors to *evasion* attacks [15, 8].

In terms of parallelizing *higher-level* network security analysis, we take our main inspiration from the work of Kruegel et al, who explored the design of front-end NIDS load balancers [10]. They introduced the notion of *slicing*: splitting up traffic not simply at a per-connection granularity, but in a NIDS-analysis-aware fashion to ensure that packets germane to possible attack scenarios are all available to the processing element that assesses their associated scenarios.

However, the issue of such front-end dispatch becomes subtle because of the many forms of *global analysis*. For example, *content sifting* [18] requires looking at a large pool of potentially suspicious strings that may be taken from any connection, and *contact graph* analysis [6] can efficiently detect new worms, but requires a global connection history within a time window. Many attacks seen today involve complex application-level sessions that span multiple connections and sometimes multiple hosts.

Thus, for *in-line* intrusion prevention operation, we need to go significantly beyond Kruegel's slicing approach to also incorporate *(i)* ways of structuring the analysis itself such that it is amenable to multi-core parallelization, and *(ii)* support for *prevention* (blocking) functionality.

Modern general-purpose CPUs. By relying on customized hardware rather than general-purpose CPUs, commercial systems have difficulty in tracking Moore's Law-style scaling, due to the low level at which parallelism must be expressed (FPGA and ASIC designs) or weak memory caching semantics (network processors).

Modern CPU designs include symmetric *multi-threaded* CPU cores [1, 11], which allow a single CPU to switch between multiple independent threads of execution, and *multi-core* systems, where a single die holds multiple CPUs [2, 11]. Recent systems support both: multiple CPUs each executing multiple threads.

It is critical to recognize that to exploit the power of such processors, programs must be specifically designed to have a parallelizable structure. However, when devel-

oping software for these systems, not only is it crucial to parallelize the program's execution structure, but also its *memory access patterns*. Although multi-thread/multi-core CPUs preserve the semantics of shared memory with cache-coherence, memory locality and behavior can completely dominate a program's ultimate performance.

In a multi-threaded core, the threads must share a common working set, lest thrashing significantly degrade performance [9]. In contrast, on a multi-core system having disjoint working sets on different cores can be a *benefit*, as the L1 and often also the L2 caches are independent. When coupled with independent memory controllers [3], it becomes vital to create and feed the threads in a memory-aware manner.

3 Overview of the Architecture

Figure 1 illustrates the overall structure of our architecture. At the bottom of the diagram is the "Active Network Interface" (ANI). This component provides the in-line interface to the network, reading in packets and later (after they have been approved) forwarding them. It also serves as the front-end for dispatching copies of the packets to the analysis components executing on different cores/threads.

The ANI drives its dispatch decisions based on a large connection table indexed by packet header five-tuple. The table yields a *routing decision* for each packet: either *(i)* which thread will analyze the packet, *(ii)* that the ANI should drop the packet directly without further processing, or *(iii)* that the ANI should forward the packet directly (to enable some forms of off-loading, as discussed below). There is an analogous table indexed by IP addresses to provide per-host blocking, and also default routing for packets not found in either table.

The analysis components populate the ANI's table entries to control its dispatch procedure. For example, a component can install a *drop* action to cut off a misbehaving connection, or alter the thread associated with a connection for purposes to improve locality of reference.

The ANI dispatches packets for analysis by writing them into queues in memory associated with the thread (and core) assigned to analyze the corresponding flow. It also sends a corresponding descriptor used to subsequently refer to the packets. The ANI holds copies of the packets locally pending approval to forward them, which an analysis component can signal by sending a control message that includes the descriptor back to the ANI.[1]

[1] As shown by the solid line from CPU Core 1 to the ANI in the figure, the analysis components can also rewrite pending packets. This functionality is necessary to support *normalization*, which may require altering the contents of packets [8].

Conceptually, the packet queues reside in the processor's shared memory. In general, these writes will directly target the processor's shared L2 cache. On modern multi-core systems, such a write will invalidate the L1 cache entries local to the individual cores, enabling the threads executing in that core to detect that they have a new packet waiting for them and load it from L2 cache to L1 cache.

An important point is that unlike for the rest of the architecture, we make the presumption that the ANI is *custom* hardware, specialized for the task. Our recent work has shown that we can construct such hardware efficiently and affordably using a simple FPGA design [24]. There are already at least two suitable Gbps Ethernet FPGA platforms available, the four-port GigE copper NetFPGA [23] and the 6 SPF (fiber) port HyperTransport-based HTX board [7].

We structure the analysis components as an event-based system, which we have developed in previous work as offering great power for network security analysis [14]. Doing so allows us to find many opportunities for concurrent execution, since events introduce a natural, decoupled asynchrony into the flow of analysis. By associating events with the packets that ultimately stimulated them, we can determine when all analysis for a given packet has completed, and thus whether it is safe to forward the pending packet.

Parallelizing event execution requires care, however. First, temporal relationships exist between events, which means that their their subsequent handlers cannot execute in arbitrary order. Second, event handlers tend to share a large amount of state, and thus need to access the same memory, potentially blocking execution of other threads. Our architecture envisions addressing these issues by introducing multiple *event queues* which collect together semantically related events for FIFO execution. Because the events are related, keeping them within a single queue localizes memory access to shared state. This in turn allows for efficient threaded execution of events since the threads can efficiently communicate (and lock data structures, when necessary) by exploiting the per-core memory caches.

The analysis proceeds in stages. The initial stages concern low-level tasks such as TCP stream reassembly and normalization, suitable to a single thread of execution. This stage requires very little inter-thread communication. It outputs events parameterized with parsed packet headers and payload byte streams. The next stage performs application-layer protocol parsing. The outputs from this stage are events reflecting application-level control information (requests and responses) with associated ADUs. Finally, these events are consumed by multiple high-level analyzers that detect attacks both within

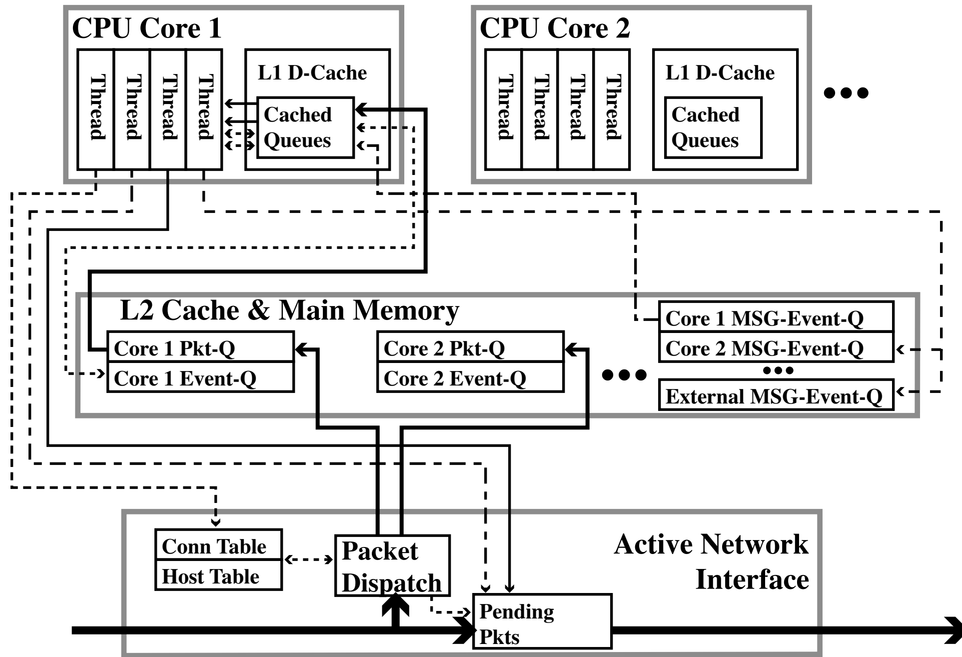

Figure 1: Structure of proposed architecture for parallel execution of network attack analysis.

application dialogs and across multiple connections and hosts.

In the figure, each core has two queues associated with it, one for receiving packets from the ANI and one for managing the events that its analysis generates and consumes. (Sharing queues across all of a core's threads minimizes potential thrashing of the limited L1 cache.) Communication between threads occurs either via the shared memory or by passing events. Events exchanged between threads executing in the same core generally use the core's event queue, while communication across cores can use separate per-core queues (e.g., "Core 1 MSG-Event-Q" in the figure). The figure shows Core 1 inserting elements into the queue for Core 2, and reading from its own MSG-Event-Q. Similarly, the system can receive externally generated events (e.g., from a host-based IDS) and send events to external agents (e.g., a global management console such as HP OpenView) via "External MSG-Event-Q".

4 Building Scalably Parallel Intrusion Prevention Systems

Given the architecture presented in the previous section, we now discuss what would go into a concrete instance of such a system.

First, in contrast to conventional network interface cards, the ANI is a stateful device whose functionality can be dynamically refined by the backend analysis engine. This is the one non-commodity component of our architecture, which we keep structurally simple to enable implementing it in low-cost (≈ $2,000) specialized hardware, as already somewhat explored in [24].

As argued in [13], the first task of a parallel analysis pipeline is flow demultiplexing: routing packets to analysis threads. For each packet, the ANI consults its flow table to decide which thread(s) is in charge of the corresponding flow and appends the packet to the packet queue of that thread's core, directly copying the packet into the thread's memory (L2 cache). This avoids the operating system having to move the packets from a single queue over to the proper thread. If the ANI does not find a flow-table entry, it forwards the packet to a dispatcher thread that computes which thread should assume responsibility for the flow's packet-level analysis and update's the ANI's flow table accordingly. An important performance observation is that the tables the ANI uses needn't be "perfect" [24]—we can tolerate occasional inconsistent entries in the tables, since the result of those entries is that packets are forwarded to the dispatcher thread—which simplifies the hardware.

For in-line operation and potential intrusion prevention blocking, analysis threads inform the ANI about their go/no-go using packet descriptors that the ANI includes with copies of the packets that it dispatches. The ANI also supports packet rewriting necessary for *normalization* [8].

Regarding the higher-level analysis components, to effectively use multi-core CPUs to exploit the potential parallelism in network monitoring, we must: (1) identify the optimal thread granularity for a given hardware architecture so we can structure the data-flow accordingly, (2) devise scalable inter-thread communication schemes, (3) resolve intrusion-prevention go/no-go decisions in a timely and reliable fashion, and (4) support effective evaluation, profiling and debugging of such systems.

In our envisioned approach, we assume there is exactly one thread responsible for the packets of a particular flow, to which the ANI dispatches the flow's packets; however, this thread may instantiate new threads on demand, to either supplement or replace its analysis.

The first stages of analyzing a flow consist of relatively fixed blocks of functionality, such as reassembling a TCP stream or decoding a particular application-layer protocol. We can structure these blocks into individual threads by following the data-flow of the processing, which proceeds along the edges of an *analyzer tree* [5]. Assuming a supply of inexpensive threads, the natural approach promises the greatest gain: one thread per analyzer will exploit the benefits of both data pipelining (for serial components of the dataflow, e.g., TCP decoding after IP decoding) and parallel processing (for computations that we can perform concurrently, e.g., running multiple application-layer analyzers). At this point we do not require any inter-thread communication.

After the initial, fairly fixed stages of analysis comes the execution of handlers for the events produced by the protocol parsers. Each packet can stimulate execution of multiple event handlers, and these handlers can generate further events, or cause side effects such as changing global state. We cannot blithely execute in parallel the event handlers triggered by an arriving packet because events have a *temporal* order among them. To control the parallel execution of events, we define multiple, independent *event queues*. Within the architecture, the semantics of these queues allows processing of events from separate queues to execute concurrently; but all events inside a single queue are processed sequentially.

In our design, we assign one such event queue to each CPU core. However, event handlers can generate new events which semantically might no longer be tied to a particular flow anymore. For these, we include global event queues into which analyzers can insert such events. Again, we dedicate a thread to each global queue to oversee the sequential execution of its corresponding event handlers.

While concurrent event processing already promises a large gain in performance by itself, there is further, major performance consideration: patterns of memory accesses. While a general-purpose processor presents a single shared memory to all of its cores and their threads, the system's cache hierarchy imposes a *nonuniform access* model. Memory caching has a *major* impact on performance for highly stateful processing. Our architecture's use of event queues promises to prove valuable here, too. By processing all events that relate to the same flow on the same core, we localize memory accesses, and thus can benefit from that core's memory cache. Similarly, by placing related events into the same global event queue, we can localize access patterns when executing inter-flow analysis.

Global correlation requires significant communication between individual threads. We have explored tightly coupled multi-CPU intrusion analysis in our work on "Bro Cluster," where a set of commodity PCs each analyze a share of the overall network traffic and synchronize state via an interconnection network [22]. The synchronization traffic between the cluster nodes can exhibit significant overhead; however, within a single multi-core system we can take advantage of its shared memory semantics rather than explicit message-passing for thread communication. However, we still need to carefully align the execution-locality of elements in the network analysis chain with the nonuniformities present due to the underlying system's cache hierarchy. We can pursue this via restructuring detection algorithms in terms of how they modify or interpret shared state; or by changing the semantics of the communication primitives, such as introducing explicit *loose* synchronization [19] and emphasizing randomized analysis algorithms that by design can cope with occasional irregularities.

For our system to realize intrusion prevention functionality, a key problem is that the analysis events are decoupled from the packets that ultimately trigger their generation. A particular packet may trigger any from zero to many events, and several packets may all contribute to a single event. However, events *directly* triggered by lower-level analysis will be generated very shortly after the ANI receives the corresponding packet. For these events it is feasible for the ANI to hold each packet until all of the events it engenders execute to completion. This approach does not apply for more global forms of analysis; however, due to the global nature of such analysis, the blocking associated with detection will in general refer to *more coarse-grained entities than flows*. For example, upon detecting a scan it is very likely tolerable that the packets of the scan (so far) have already reached their destination—as long as one can *ensure* that the system will block any further activity by the originating host.

Finally, for profiling and debugging of such systems, we are particularly interested in: *(i)* identifying race conditions, and *(ii)* understanding memory access patterns.

The key to systematically analyzing the behavior of such programs is *reproducibility*. We have explored trace-based reproducibility by augmenting the Bro analysis system [14] with a *pseudo-real-time* mode: when activated, packets from a trace are artificially delayed to match real-time semantics. The mode also introduces synchronization points at regular time intervals to ensure that the reproducibility of individual instances do not drift too far from the trace they process.

5 Summary

The goal of our effort is to develop a framework to support the construction of highly parallel, inline network intrusion prevention systems that can fully exploit the power of modern and future commodity hardware. Ultimately, we aim to enable network intrusion prevention to reap both the benefits of executing on general-purpose CPUs, and the exponential scaling that Moore's Law for aggregate parallel processing continues to promise.

The key elements of achieving this vision are *(i)* identifying the optimal thread granularity for a given hardware architecture so we can structure the data-flow accordingly, *(ii)* devising scalable inter-thread communication schemes, *(iii)* resolving intrusion-prevention go/no-go decisions in a timely and reliable fashion, and *(iv)* supporting effective evaluation, profiling and debugging of such systems. We have sketched how we believe we can achieve all of these by employing a custom front-end component (the Active Network Interface) and structuring our network analysis in an event-oriented fashion.

If successful, our approach will offer a design point in terms of price and ease of scalability that significantly differs from that offered by current approaches based on FPGA, ASIC, or network processor hardware. Given how parallel hardware appears likely to evolve in the future, this paradigm could ultimately prove highly influential to industry.

6 Acknowledgments

This work was supported by NSF Awards STI-0334088, ITR/ANI-0205519, and CNS-0627320, as well as by the Office of Science and Technology at the Department of Homeland Security. We are grateful for the support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation, or the official position of the U.S. Department of Homeland Security or the Office of Science and Technology.

References

[1] Intel Corporation. Intel Pentium 4 Processor, http://www.intel.com/products/processor/pentium4/index.htm.

[2] Intel Corporation. The Intel Core Duo Processor.

[3] Advanced Micro Devices. AMD Athlon 64 X2 Dual Core Processor, http://www.amd.com/us-en/processors/productinformation/0,,30_118_9485_13041,00.html.

[4] Sarang Dharmapurikar and Vern Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *USENIX Security Symposium*, August 2005.

[5] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *USENIX Security Symposium*, 2006.

[6] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. A behavioral approach to worm detection. In *Workshop on Rapid Malcode*, 2003.

[7] H. Froning, M. Nussle, D. Slogsnat, H. Litz, and U Bruning. The HTX-Board: A Rapid Prototyping Station. In *3rd annual FPGAWorld Conference*, 2006.

[8] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[9] S. Hily and A. Seznec. Standard memory hierarchy does not fit simultaneous multithreading. In *Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4)*, 1998.

[10] C. Kruegel, F. Valeur, G. Vigna, and R.A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[11] Sun Microsystems. UltraSPARC T1 Overview, http://www.sun.com/processors/ultrasparc-t1/.

[12] James Moscola, John Lockwood, Ronald Loui, and Michael Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38, Napa, CA, USA, April 2003.

[13] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *Proceedings of the USENIX Hot Security Workshop*, August 2006.

[14] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[15] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[16] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[17] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, USA, April 2001.

[18] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the sixth Symposium on Operating Systems Design and Implementation*, December 2004.

[19] R. Sommer and V. Paxson. Enhancing byte-level network intruson detection signatures with context. In *ACM Communications And Computer Security (CCS) Conference*, 2003.

[20] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *The 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[21] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE Infocom*, Hong Kong, China, March 2004.

[22] Matthias Vallentin. Transparent Load-Balancing for Network Intrusion Detection Systems. Bachelor's Thesis, TU Muenchen, 2006.

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

[24] Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. The Shunt: An FPGA-based accelerator for network intrusion prevention. In *ACM Symposium on Field Programmable Gate Arrays*, February 2007.