Rethinking Hardware Support for Network Analysis and Intrusion Prevention

V. Paxson,[1] *K. Asanović,*[2] *S. Dharmapurikar,*[3] *J. Lockwood,*[4] *R. Pang,*[5] *R. Sommer,*[1] *N. Weaver*[1]

Abstract

The performance pressures on implementing effective network security monitoring are growing fiercely due to rising traffic rates, the need to perform much more sophisticated forms of analysis, the requirement for inline processing, and the collapse of Moore's law for sequential processing. Given these growing pressures, we argue that it is time to fundamentally rethink the nature of using hardware to support network security analysis. Clearly, to do so we must leverage massively parallel computing elements, as only these can provide the necessary performance. The key, however, is to devise an abstraction of parallel processing that will allow us to expose the parallelism latent in semantically rich, stateful analysis algorithms; and that we can then further compile to hardware platforms with different capabilities.

1 Introduction

Effective network security monitoring is growing increasingly difficult. Given the adversarial—and hence constantly evolving—nature of the problem space, plus the inherently limited power of signature matching in the face of false positives, polymorphism, and zero day attacks, we need tools capable of sophisticated analysis of protocols *(i)* at higher semantic levels, and *(ii)* incorporating context correlated across multiple connections, hosts, sensors, and over time. For such analysis, the monitor must both perform much more computation and, crucially, undertake sophisticated management of large quantities of complex state.

Such monitors also need to *alter* traffic to eliminate broad classes of evasion threats (i.e., "normalization" [HPK01]) and to progress beyond simply detecting attacks to instead realizing intrusion *prevention* systems. But forcing the monitoring into the forwarding path as an active, inline element now runs the risk of imposing direct limits on the performance of production network traffic.

Worse, traffic volumes and rates continue to race forward, outpacing raw CPU performance now that uniprocessor speeds no longer track Moore's Law, rendering the detection task beyond the performance that traditional hardware approaches can provide. While the main alternatives—ASICs and FPGAs—support vastly more parallelism, they require highly deliberate, customized programming, which is directly at odds with the pressing need to perform diverse, increasingly sophisticated forms of analysis.

Given these growing pressures—more sophisticated forms of analysis, conducted inline, at higher rates, on non-uniprocessor hardware—we argue that it is time to fundamentally rethink the nature of using hardware to support network security analysis. Clearly, to do so we must leverage massively parallel computing elements, as only these can provide the necessary performance. The key, however, is to devise an *abstraction* of parallel processing that will allow us to expose the parallelism latent in semantically rich, stateful analysis algorithms; and that we can then further compile to hardware platforms with different capabilities.

In the next section we sketch the fundamentally limited nature of current hardware approaches for supporting network security analysis. We then in Section 3 develop an argument that the task of performing such analysis includes a great deal of potential parallelism, even for quite sophisticated forms of analysis, if only it can be exposed. In Section 4 we describe in high-level terms how we envision such an alternative paradigm for hardware support could be achieved. We offer final thoughts in Section 5.

[1]International Computer Science Institute. vern@icir.org, robin@icir.org, nweaver@icsi.berkeley.edu.

[2]Massachusetts Institute of Technology. krste@cag.csail.mit.edu.

[3]Nuova Systems. sarang@nuovasystems.com.

[4]Washington University, St. Louis. lockwood@arl.wustl.edu.

[5]Princeton University. rpang@CS.Princeton.EDU.

2 The Limited Nature of Current Hardware Approaches

To date, the literature of hardware designs to support network security analysis has 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" [Roe99] and its large set of byte-level signatures. Along these lines, within the past five years approaches have been devised based on NFAs (e.g., [SP01, FCH02]) and DFAs (e.g., [MLLP03]), supplemented by optimizations such as character decoding [CS04], architectural features [SP04, CMS05], TCAMs [LNZ+03], Bloom filters [DKSL04], optimized Aho-Corasick trees [TSCV04], and "tiny" state machines [TS05].

A vital point regarding nearly all of the previous hardware design research is that it presumes a nearly *stateless* approach to attack detection. The implementations either operate on single packets, or require some other process to reassemble the TCP byte stream. This leaves them vulnerable to evasion attacks [PN98, HPK01], which, if addressed at all, are countered by dropping out-of-order packets, with potentially major detrimental effects on performance and network load under congestion. Richer, stateful hardware elements, such as TCP stream reassembly, have not been explored in nearly as much depth[1] (e.g., using a small, fixed amount of state per connection [SL03]). It is in fact striking that one of the simplest, most basic next steps—TCP stream reassembly coupled with consideration of an adversary—had been wholly unexplored in the literature until less than a year ago [DP05]. This appears to reflect an historical gulf between hardware expertise and network security expertise that could yield immense benefits once bridged.

What is missing? A great deal. First, signature-matching is *fundamentally* limited in its expressive power. Its nature is syntactic rather than semantic, a weakness that leads to intrinsic difficulties in interpreting the significance of a match (resulting in false positives, and also the easy ability of an adversary to overstimulate the system) and in detecting variants of attacks that differ syntacticly but have semantics equivalent to known attacks. We have previously argued that we can ameliorate some of these deficiencies by incorporating *context* into byte-level signature matching—but doing so requires either maintaining significant *state*, and/or performing significant semantic processing [SP03].

Second, network protocols contain a wide range of ambiguous cornercases that create evasion opportunities beyond those that arise just due to the syntactic nature of signatures [PN98]. To prevent attackers from exploiting these ambiguities to elude detection, we need to remove the ambiguities via *normalization*, in which an inline network element selectively *rewrites* traffic to disambiguate it [HPK01]. However, performing such normalization again requires maintaining significant per-connection state.

Third, once we have a security element operating inline rather than passively, we again have an opportunity to do significantly more powerful analysis: intrusion *prevention* (blocking traffic that bears an attack as determined by NIDS analysis) rather than mere *detection*. When doing so, however, our willingness to tolerate false positives diminishes precipitously, again calling out for detection approaches more powerful than the weak one of signature-matching.

Fourth, if we can bring more sophisticated, stateful processing to bear, then we enable a wealth of richer types of analysis. One large class comes from *protocol parsing*: recovering the higher-layer semantics conveyed in a byte stream in order to understand the specific requests, responses, status messages, error codes, and data items embedded in a connection dialog. Analyzing these at the application layer, as opposed to the syntactic byte-string layer, opens up much greater insight into the nature and context of the exchange between two hosts, allowing much more precise detection decisions—but at a cost of the hardware managing a great deal more state, and performing much more processing.

Finally, other higher-level forms of analysis evaluate activity *across* multiple connections or hosts. For example, "content sifting" identifies worms by detecting substrings that repeatedly appear in traffic between multiple hosts [SEVS04]; "stepping stone" detection can determine that two connections, potentially with no host in common, in fact represent elements of a chain along which a single set of commands and responses propagate [ZP00]; and "scan detection" attempts to rapidly determine that the activity of a host reflects malicious probing with high probability [JPBB04]. Supporting these again requires significant state and processing.

3 Opportunities for Much Richer Forms of Parallelism

We could provide hardware support for each of the many desirable forms of in-depth analysis by handcraft-

[1] Hardware-based intrusion detection is an area abuzz with commercial activity. Almost nothing is available in the peer-reviewed literature regarding the designs that underly these systems. From vendor literature, it appears clear that some of the systems use extensive, expensive ASIC components, but whether these serve simply to parallelize low-level matching operations similar to the work described above, or also exhibit true innovations for higher-level processing, remains uncertain.

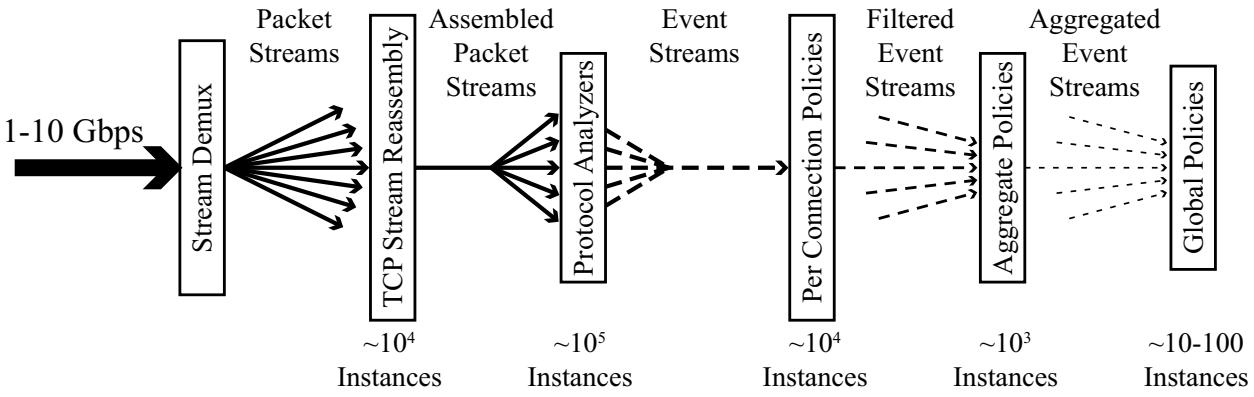

Figure 1: The spectrum of parallelism present in a high-level network security analysis pipeline.

ing intricate, individual hardware designs. But the labor involved—both for accommodating different platforms with different capabilities, and as the types of analysis continue to mushroom—renders this a losing game, requiring enormous effort and resources. Instead we need to find ways to *(i)* express such analysis in high-level terms, allowing us to readily refine our detection algorithms and develop new ones, *(ii)* with a means for translating the high-level descriptions into hardware realizations, *(iii)* for which we can effectively take advantage of the hardware's capabilities.

This final consideration is crucial: the only benefits we gain from implementing analysis in hardware come from the alternate execution model that hardware can offer. Otherwise, we might as well simply execute the analysis on a general purpose CPU—which is no longer viable given the failure for such CPUs to track the rates at which both network traffic and the desired richness of analysis grow.

Indisputably, the key to unlocking potential hardware performance gains is parallelism. Indeed, the massive parallelism of FPGAs is what has driven many of the efforts in boosting NIDS performance with hardware to date, as outlined above. However, we argue that we can find *further* exploitable parallelism in network analysis tasks, even very high-level ones, *if* we structure the tasks to expose it.

Figure 1 illustrates the parallelism potentially available across a pipeline of increasingly higher levels of network security analysis. A crucial point is that we need to extract parallelism at each stage of the pipeline or else Amdahl's Law will fundamentally limit the speedup that hardware can provide. In the figure, vertical boxes reflect different types of analysis, increasing in semantic level and breadth from left to right. The progression of arrows indicate how information flows from one level to the next, with the thickness of an arrow indicating the relative volume of data within the flow. Thinner arrows thus indicate fewer threads of analysis that need to execute at the next stage relative to the previous stage; hence if the later stage offers fewer opportunities for parallel execution, but *also* receives fewer flows to analyze, then we can still "keep the pipeline full" as we analyze flows at increasingly high levels. Of particular note is the large degree of *task-level* parallelism, which can be leveraged either by pipelining or by constructing multiple, independent functional units. Even at the highest level of global analysis, there are potentially tens to even hundreds of independent tasks.

Fanout of arrows indicates multiple analyses that can be executed in parallel with little conflict between the threads of processing. Fan-in indicates multiple sources of information flows being analyzed together at a higher level. Finally, the numbers shown such as "$\approx 10^4$ instances" convey an order-of-magnitude sense of the volume of parallelism available if we are monitoring a busy link with a capacity of 1–10 Gbps. (We use these numbers just to convey a sense of opportunity.)

We work through the figure as follows. The first stage ("Stream Demux") demultiplexes incoming packets to per-connection processing. Although a sequential task, experience has shown that one can efficiently implement stream demultiplexors in line-rate, pipelined hardware. On a link of 1–10 Gbps, we might have, say, 10^4 concurrent connections, and thus we can then parallelize and/or pipeline the process of TCP stream reassembly and normalization amongst these 10^4 independent streams.

After performing TCP stream reassembly, we then forward the resulting flows for protocol analysis. This stage exhibits still more parallelism: today, we can no longer reliably determine the application protocol used for a given connection based on the transport port numbers associated with the flow. However, a powerful means by which to analyze application protocols even in the ab-

sence of reliable port numbers is to *run different possible application parsers in parallel* to determine which parser finds the flow syntactically and semantically consistent [DFM+06]. Thus, here we have fanout as execution tries a plethora (10 in the Figure) of different application parsers, and then fan-in as only one of those parsers actually accepts the flow.

The output of the application analysis is a series of "events" reflecting a distillation of application-level activity such as the parameterization of requests, items and status codes associated with replies, error conditions, and so on. We can then analyze these events on a per-connection basis, maintaining the earlier parallelism we gained during the demux stage, and allowing us to evaluate per-connection policies.

Next, a subset of these events, gathered across multiple connections involving a given host or a given connection type, feed into analysis policies that execute at an aggregate level. For example, for scan detection we assess to how many different servers a given host has attempted to connect, and with what success. The parallelism available here is a function of how many such analyses we perform, and to what degree they can execute without conflict.

At the last stage, at a still higher level of aggregation we execute analyses that use events drawn across not only multiple connections but also multiple hosts, such as the "content sifting" and "stepping stone" analyses mentioned in the previous section.

Finally, an important point concerning this pipeline is that its processing does *not* have to execute in only a few microseconds in order to ensure that we can forward packets at line rate. Rather, due to the exploitation of parallelism we can pipeline the processing of many concurrent streams, allowing potentially hundreds of microseconds or even more for any one stream, while still sustaining line rate for the aggregate. For each given stream, incurring a millisecond or two of additional latency is quite acceptable, and allows us to perform much richer analysis than what we could realize in the absence of parallelism.

4 Extracting the Parallelism

The previous section argues that the task of network security analysis includes a great deal of *potential* parallelism, but the major challenges of exposing it and mapping the execution to hardware remain. We envision pursuing these using three fundamental elements: (1) a high-level language for expressing rich forms of network analysis; (2) a powerful abstraction of parallel execution to which we target compilation of the elements of the language; and (3) a final step of compilation from that abstraction to concrete hardware implementations. We briefly examine these now, top-down.

We can partition the first element into three components: a set of fixed function blocks, a set of protocol analyzers, and a domain-specific analysis language. The function blocks implement universal analysis prerequisites such as connection state management and stream reassembly (second stage of Figure 1). These might be implemented as handcrafted hardware modules. For protocol analyzers, if we specify them using a language such as *BinPAC* [Pan06, PPSP06] (which supports describing protocol syntax and semantic logic declaratively, at a high level), then we could retarget the compiler for the language to generate code for our parallelism abstraction rather than uni-threaded C++ (third stage). Finally, for higher levels of analysis (remaining stages), we picture expressing them in a language such as that used by our *Bro* intrusion detection system [Pax99], which includes asynchronous event-oriented execution, sophisticated state management primitives, and explicit support for coordinating concurrent execution among multiple processes. Such a set of language features offers natural opportunities for mapping high-level analysis scripts to our parallelism abstraction. Compilation of both protocol analyzers and higher-level analysis requires aggressively optimizing compilers that can identify the dependencies we must address for concurrent processing. We can consider the compilation in the other direction, too: what kind of features or restrictions might we incorporate into an existing language to facilitate the discovery of parallelism?

For the model of parallel execution to target, we envision an intermediary form such as the *Transactor* abstraction we have been developing, and which is used as the basis of the RAMP project design framework [GSA06]. This architecture represents a parallel computation as a communicating network of stateful "transactional actor" processing units. Each unit has four components: local architectural state; buffered input and output channels that provide decoupled connections between units; a set of atomic transactions that can read from the input channels, mutate private state, and write to the output channels; and a scheduler that selects the next transaction to perform.

The basic model is built around guarded atomic actions [CM88], which provide clean handling of nondeterministic input streams and shared mutable state, allowing us to model a concurrent computation as a network of transactor units connected by message queues. Execution of the transactor network has serializable semantics, in that any concurrent execution produces the same result as some sequential execution in which one transactor executes one transaction atomically at each step.

The transactor model simplifies the generation of high quality hardware in two main ways. First, by decoupling global communication between units from local computation within units, the model naturally captures locality in the computation and simplifies the generation of efficient global interconnect fabrics. Second, by describing computation within a unit as a set of guarded atomic actions rather than as a fixed execution schedule, the specification frees the implementation to map transactions onto a variety of sequential, pipelined, or parallel execution structures. For example, the same specification could be mapped to use greater or fewer resources on an FPGA depending on the area available, and mappings can change the area allocation between different pieces of the analysis depending on the relative throughput requirements.

Of course, a great deal of careful design work is still required, paying particular attention to the capabilities of different execution targets, private vs. shared memory access, and state and timer management. But if we can achieve such an approach, then we can *(i)* express our analyses in a high-level, domain-specific language, and *(ii)* decouple the labor and expertise of compiling the elements of our analysis language to the parallelism abstraction, from the labor and expertise of compiling that abstraction to platforms with different hardware capabilities.

5 The Bigger Vision

Finally, we note that little in our resulting architectural approach relates to the particular problem of network security analysis. Rather, the core of the new paradigm can enable sophisticated, stateful network processing *in general*. Thus, if successful, we could enable network processing rich in semantics and context as a *routine* capability provided by a network's routers. The same hardware will support reconfiguration for a wide range of processing using recompilation rather than redesign. With this shift will then come the opportunity to rethink the full edifice of the network processing architecture—which today begins with the fundamental restriction "keep the primary forwarding path as simple as possible," but tomorrow might transform to "keep the million parallel computational units busy if possible."

6 Acknowledgments

This work was supported by NSF Awards STI-0334088, ITR/ANI-0205519, and CCF-0541164, as well as a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD).

References

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[CMS05] Y. Cho and W. Mangione-Smith. Fast reconfiguring deep packet filter for 1+ gigabit network. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2005.

[CS04] Christopher R. Clark and David E. Schimmel. Scalable multi-pattern matching on high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2004.

[DFM+06] 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.

[DKSL04] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, January 2004.

[DP05] Sarang Dharmapurikar and Vern Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *USENIX Security Symposium*, August 2005.

[FCH02] R. Fanklin, D. Caraver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings from Field Programmable Custom Computing Machines*, 2002.

[GSA06] G. Gibeling, A. Schultz, and K. Asanović. RAMP architecture and description language. In *2nd Workshop on Architecture Research using FPGA Platforms*, February 2006.

[HPK01] 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.

[JPBB04] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy*, 2004.

[LNZ+03] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, September 2003.

[MLLP03] 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.

[Pan06] Ruoming Pang. *Towards Understanding Application Semantics of Network Traffic*. PhD thesis, Princeton University, in preparation, 2006.

[Pax99] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[PPSP06] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. BinPAC: A Yacc for Writing Application Protocol Parsers, 2006. In submission.

[Roe99] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[SEVS04] 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.

[SL03] David V. Schuehler and John W. Lockwood. Tcp-splitter: A tcp/ip flow monitor in reconfigurable hardware. In *Hot Interconnects*, pages 89–94, Stanford, CA, August 2003.

[SP01] 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.

[SP03] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[SP04] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[TS05] 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.

[TSCV04] 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.

[ZP00] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *Proc. USENIX Security Symposium*, 2000.