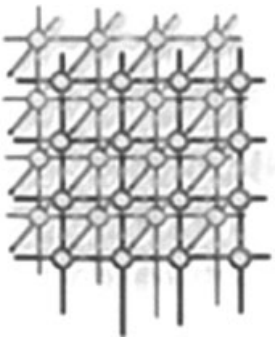
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:1255–1279
Published online 14 May 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1422



An architecture for exploiting multi-core processors to parallelize network intrusion prevention

Robin Sommer[1,2,*,†], Vern Paxson[1,3] and Nicholas Weaver[1]

[1]*International Computer Science Institute, Berkeley, CA, U.S.A.*
[2]*Lawrence Berkeley National Laboratory, Berkeley, CA, U.S.A.*
[3]*UC Berkeley, Berkeley, CA, U.S.A.*

SUMMARY

It is becoming increasingly difficult to implement effective systems for preventing network attacks, due to the combination of the rising sophistication of attacks requiring more complex analyses to detect; the relentless growth in the volume of network traffic that we must analyze; and, critically, the failure in recent years for uniprocessor performance to sustain the exponential gains that for so many years CPUs have enjoyed. For commodity hardware, tomorrow's performance gains will instead come from *multi-core* 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', a custom device based on an inexpensive FPGA platform. 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 asynchrony into the analysis while still maintaining good cache locality. A preliminary evaluation demonstrates the potential of this architecture. Copyright © 2009 John Wiley & Sons, Ltd.

Received 20 November 2008; Revised 30 January 2009; Accepted 12 February 2009

KEY WORDS: network intrusion detection; event-based system; concurrent processing; evaluation

*Correspondence to: Robin Sommer, International Computer Science Institute, Berkeley, CA, U.S.A.
†E-mail: robin@icsi.berkeley.edu

Contract/grant sponsor: NSF; contract/grant numbers: CNS-0716636, NSF-0433702
Contract/grant sponsor: Intel Corporation

1. INTRODUCTION

The performance pressures on implementing effective network security monitoring are growing fiercely in multiple dimensions. First, the adversarial nature of network security gives an evolutionary impetus to the entire problem: 'attacks never get worse, only better'. 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. Moving beyond signature-matching requires sophisticated analysis of protocols (i) at higher semantic levels and (ii) incorporating *context* correlated across multiple connections, hosts, sensors, and over time. For such an analysis, the monitor must both perform much more computation and, crucially, undertake sophisticated management of large quantities of complex *state*.

Second, the need to alter traffic ('normalization' [1]) to eliminate broad classes of *evasion* threats, and, even more critically, to progress beyond simply detecting attacks to instead realizing intrusion *prevention* systems, forces the analysis to move beyond the domain of passive processing of network streams, and into the forwarding path itself. With such *in-line* processing, computationally intensive analysis systems run the risk of imposing direct limits on the performance of production network traffic.

Third, traffic volumes and rates continue to race forward, incessantly shrinking the processing budget available for computing a given type of network analysis. Thus, even if we could stick with the computational simplicity of signature-matching, we would find our processing capabilities stretched increasingly thin.

Finally, we have 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 15 prior years, the uniprocessor performance increased by 50–60% per year. But by 2006, the performance was a *factor of three* slower than had the pre-2002 curve continued.

When single processors can no longer track the necessary growth curve, one naturally turns to multiple, concurrent processing. Until recently, this has meant embracing either expensive custom designs (ASICs) or diminished (network processors) or alternate (FPGAs) execution models. Such hardware offers the raw parallelism necessary to address *half* of the problem, namely the incessant growth of network traffic volumes and rates. But the highly deliberate, customized programming they require is directly at odds with the other half of the problem: the inexorably growing need to perform more and more sophisticated forms of analysis.

To perform such an analysis, it would be hugely advantageous if we could somehow draw upon the flexibility and inexpensive system costs of using general-purpose CPUs. Recently, hardware vendors have begun delivering commodity CPUs that again reflect Moore's law-style scaling, with the parallelization gains coming from multi-core/multi-thread architectures.

Today one can buy dual-core [2], dual-core dual-thread [3], quad-core [4], six-core [5], and 8-core with 8 threads/core [6] CPUs. These designs promise to continue scaling into the future; for example, there are already specialized 64-core processors for network processing [7] and upcoming $\times 86$-based many-core architectures that may contain 64 discrete $\times 86$ cores with vector extensions [8].

The *aggregated* throughput of such processors does in fact still follow Moore's law. However, to exploit the full power of a modern multi-core hardware platform, we must explicitly structure

DOI: 10.1002/cpe

our applications in a highly parallel fashion: dividing the processing into concurrent tasks while minimizing inter-task communication.

In our previous work with colleagues [9], we have argued that we can extract a potentially enormous degree of parallelism from the task of network security monitoring. However, doing so requires rethinking on how we pursue the parallelism. Historically, parallelization of intrusion detection/prevention analysis has been confined to coarse-grained load-balancing (with little or no fine-grained communication between the analysis units) and fast string-matching. These approaches buy some initial speed-ups, but Amdahl's law prevents significant gains for more sophisticated analyses that require fine-grained coordination.

Taking advantage of the full power of multi-core processors requires a more in-depth approach. Obviously, we need to structure the processing into separate, low-level threads that are suitable for concurrent execution. To do so, however, we need to address a number of issues:

- 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.
- To perform global analysis (e.g. scan detection [10,11], worm contact graphs [12], stepping-stone detection [13], content sifting [14], botnet command-and-control [15]) 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 of the memory and threading parameters of a given CPU, so we can automatically retarget the implementations of analysis algorithms to different configurations.
- 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. Ultimately, we aim to prove the power of such designs in terms of enabling network intrusion prevention to reap both the benefits of executing on general-purpose commodity hardware, and the exponential scaling that Moore's law for aggregate parallel processing continues to promise.

We start with a high-level overview of our architecture in Section 2. Next we argue for the large potential of parallel processing for network security analysis in Section 3. In Section 4 we discuss our architecture in more concrete terms and outline how we plan to implement and evaluate a full NIPS built according to the approach. In Section 5, we evaluate the potential of a crucial part of our architecture with real-world network traffic. Section 6 covers the rich related work in this area. We conclude in Section 7.

2. OVERVIEW

We begin our discussion with an overview of the architecture we envision; Figure 1 illustrates its overall structure. At the bottom of the diagram is the 'active network interface' (ANI). This

DOI: 10.1002/cpe

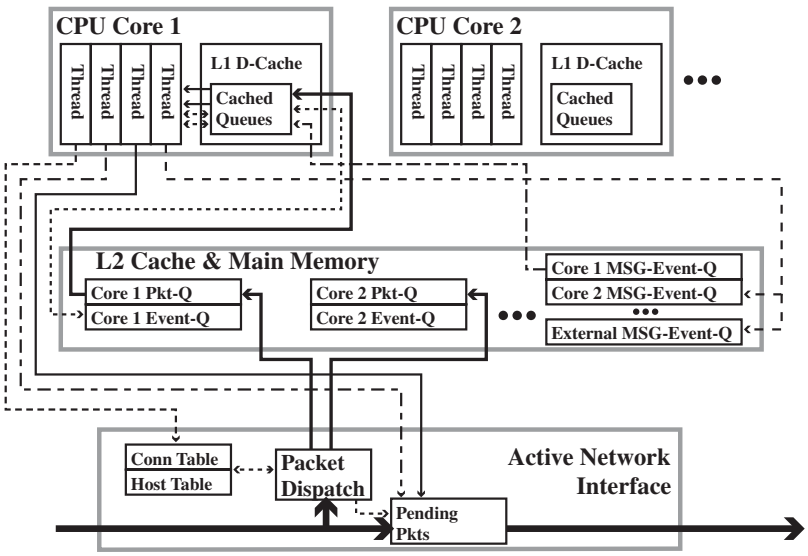


Figure 1. Structure of architecture for parallel execution of network attack analysis.

component provides an 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 in different 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 of load-balancing or to improve the locality of reference when analyzing a set of activity.

The ANI dispatches packets for analysis by writing them into queues in memory associated with the thread assigned to analyze the corresponding flow. It also sends corresponding descriptors 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[‡].

Conceptually, the packet queues reside in the processor's shared memory. In general, these writes can directly target the processor's shared L2 cache. On modern multi-core systems, such a write will

[‡]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 [1]. The ANI cannot normalize packets itself because it lacks sufficient state to perform the necessary analysis.

DOI: 10.1002/cpe

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 can be *custom* hardware, specialized for the task. Our previous work has shown that we can construct such hardware efficiently and affordably using a simple FPGA design [16]. Because the functionality has quite a limited complexity, it can tolerate single-sided errors, and does not require general CPU-like flexibility, by employing such hardware we can gain major performance gains without incurring much of the programmability burden that using custom hardware for the entire task would cost.

We design the functionality to be conceptually straightforward and amenable to execution in parallel if the processor fabric supports sufficiently fast packet delivery and automatic load-balancing, as there are no inter-packet dependencies. Thus, if a multi-core fabric includes embedded network interfaces [6,7], we envision that the ANI functionality could also be realized in a small program running on one or more cores without needing to access data beyond the processor caches.

We structure the analysis components as an *event-based* system. We have extensive experience with the power of applying an event-based approach to network security analysis, as it forms the heart of our 'Bro' intrusion detection system [17]. As we will develop, the focus on an event-oriented architecture 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 analyses for a given packet have completed, and thus it is safe to forward the pending packet, assuming none of the analysis elements has previously signaled that the packet should instead be discarded.

Parallelizing event execution requires care, however. First, temporal relationships exist between events, which means that 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* that collect together semantically related events for in-order 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. We discuss event scheduling in more detail in Section 5.1.

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 (since normalization already requires header analysis) and payload byte streams (for TCP). The next stage performs application-layer protocol parsing. As we will develop below, this stage can significantly benefit from parallelizable execution. The outputs from this stage are events reflecting application-level control information (requests and responses) with associated protocol data units. Finally, these events are consumed by multiple high-level analyzers that detect attacks both within application dialogs and across multiple connections and hosts.

In the figure, we do not show these stages, but only the abstract structure of how the cores coordinate their processing to achieve them. 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. (The queues are shared across all of the threads within the core, since using per-thread

DOI: 10.1002/cpe

queues rather than per-core risks thrashing 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'.

As discussed above, a thread of execution can signal the ANI to forward a pending packet, or alternatively to discard it regardless of the outcome of further analysis. A thread can also update the connection and host tables in the ANI to alter the dispatching associated with a given flow or address, as shown by the dashed line from Core 1 to the left-hand part of the ANI in the diagram.

3. UNCOVERING PARALLELISM IN NETWORK SECURITY ANALYSIS

In this section we discuss how the task of performing high-level network security analysis exhibits a great deal of potential parallelism. To effectively extract it, however, we must take care how we structure the workflow of the analysis.

Figure 2, taken from our previous work [9], 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 to gain the maximal performance gain. In the figure, vertical boxes reflect different types of analyses, increasing in semantic level and breadth from left to right. The progression of arrows indicates 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 less opportunities for parallel execution, but *also* will be presented with 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 easily be leveraged by multi-core and multi-threaded processors. Even at the highest level of global analysis, there are potentially tens to even hundreds of independent tasks.

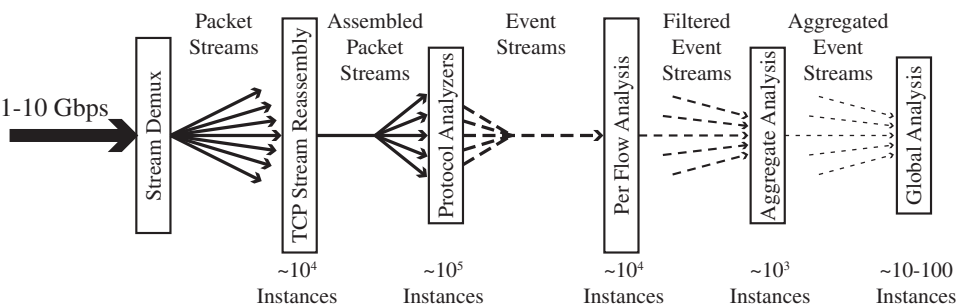


Figure 2. The spectrum of parallelism present in a high-level network security analysis pipeline.

DOI: 10.1002/cpe

Fan-out of arrows indicates multiple analyses that for a given flow 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 to convey a sense of opportunity, rather than as concrete values.)

We work through the figure as follows. The first stage ('Stream Demux') demultiplexes incoming packets to per-flow processing. This is the only fully sequential task, which we assign to a custom front-end, the ANI, as discussed in Section 2. On a link of 1–10 Gbps, after processing this stage we have now decomposed the problem into, say, 10^4 concurrent flows, and thus we can then parallelize and/or pipeline the process of TCP stream reassembly and normalization among these 10^4 independent streams. In a multi-core context, now each core only works on a subset of the streams.

After performing TCP stream reassembly, we then forward the resulting flows for protocol analysis. Perhaps surprisingly, even this state exhibits a large degree of parallelism, as we developed in [15]. To reliably determine the application protocol in use for a given flow, it is insufficient to just consider the transport-layer port numbers (as in fact most systems still do). Instead, for a variety of reasons—some benign, some malicious—many flows explicitly avoid the use of well-known ports [18]. However, as per our previous work, a powerful means to analyze application protocols without relying on port numbers is to *run all the possible application parsers in parallel* to determine which parser finds the flow syntactically and semantically correct. Thus, as the figure shows, here we have fan-out as execution tries a plethora (10 in the figure) of different application parsers, and then fan-in as only one of those parsers that 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, signature matches, and so on. We then analyze these events on a per-flow basis, maintaining the earlier parallelism we gained during the demux stage.

Next, a subset of these events, gathered across multiple flows involving a given host or a given flow type, feed into analyses that execute at an aggregate level. For example, for scan detection we assess as 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.

Finally, at a higher level of aggregation we execute analyses that use events drawn across not only multiple flows but also multiple hosts. An example at this level would be 'content sifting' [14], which needs to analyze elements of the contents of disparate traffic flows in order to detect the propagation of a network worm.

An important observation about the parallelization potential is that many related tasks share the same basic working set. Thus, although we may have 10 different application parsers decoding the same TCP stream, these 10 threads share a great deal of state. On the other hand, the events spawned by *different* flows will have largely disjoint working sets.

This observation fits well with the multi-core model, as each core has its own memory cache. Thus, we can achieve good memory performance by scheduling threads that share the same working set onto the same core, while executing unrelated threads on another core.

DOI: 10.1002/cpe

4. BUILDING SCALABLE PARALLEL INTRUSION PREVENTION SYSTEMS

Given the context presented in the previous sections, we now revisit the architecture sketched in Section 2 to develop it in greater detail, including specifics of the concrete instance of such a system. Recall that the architecture consists of two key components: a front-end, the ANI; and a backend, the analysis engine that executes on a multi-core/multi-threaded hardware platform. We discuss each in turn.

4.1. Active network interfaces

In contrast to conventional network interface cards, the ANI is a stateful device whose functionality can be dynamically refined by the backend analysis engine. The ANI is responsible for (i) routing copies of packets to the appropriate analysis threads; (ii) retaining packets until signaled by the analysis engine to either forward or drop them; and (iii) supporting alteration of packet content. We discuss these tasks below.

Our overall goal is to facilitate the development of high-performance, highly flexible, *inexpensive* network intrusion prevention systems. To this end, we keep the one non-commodity component of our architecture—the ANI—structurally simple, to enable implementing it in relatively low-cost specialized hardware. In [19], we present an implementation of a more restricted version of the ANI, and we envision the use of the same NetFPGA platform [20] here. A single unit should cost roughly $2000. We find this an acceptably low price, since the functionality it provides enables us to build the rest of our system using off-the-shelf commodity hardware.

In addition, the algorithms used in the hardware implementation can also run in pure software. Depending on the development of the commodity-hardware market, this may enable us at some point to forgo having any custom hardware in the system, and instead rely solely on general-purpose CPUs. For example, the Sun Niagara 2 [6] includes, in addition to its 8 CPU cores, two directly attached 10 Gbps Ethernet controllers, and the Tilera Tile64 processor [7] has similar features. Since our ANI algorithms should exhibit predictable and bounded memory access patterns, such a processor might prove capable of delivering enough general-purpose performance to execute the ANI using one or more of its cores directly attached to its high-performance network interfaces.

4.1.1. Thread-aware routing

As shown in Figure 2, the first task of the parallel analysis pipeline is flow demultiplexing: routing packets to analysis threads. We assign this task to the ANI. For each packet, it first decides which thread(s) is in charge of the corresponding flow. The ANI then appends a copy of the packet to the packet queue of the core running that thread. Technically, the NIC does so by directly copying the packet into the thread's memory (i.e. the corresponding core's L2 cache). This is highly efficient as it avoids the need to have the operating system move the packets from a single queue over to the proper thread. The resulting savings in memory bandwidth are substantial, eliminating one of the two main packet transfers.

The ANI needs to determine to which thread to route a packet. A simple approach is to use a *static* scheme, e.g. hashing the flow into a thread identifier. We have used such a scheme very successfully for building a *NIDS cluster* [21], a set of commodity PCs jointly analyzing a

DOI: 10.1002/cpe

high-volume network stream that is load balanced across them by a high-performance front-end system. However, the drawback of such a static scheme is that the backend engine cannot influence the decision, for example, to route a flow to a thread analyzing related communication on demand. *Dynamic* approaches, based on the ANI maintaining a table of per-flow routing decisions, offer far more flexibility. If a packet arrives that corresponds to a flow already having an entry in the table, the ANI will directly route it to the appropriate destination. If the ANI does not find a corresponding table entry, it forwards the packet to a dispatcher thread that computes which thread should assume responsibility for packet-level analysis of the corresponding flow. The dispatcher thread then updates the ANI's table for direct routing of further packets belonging to the flow.

In a previous effort [16] we developed an architecture that demonstrates the effectiveness of such an approach. An important performance observation from that work is that the tables the ANI uses need not be 'perfect'. That is, we can view the ANI's tables as caches rather than full data structures; we can then tolerate occasional inconsistent entries in the tables, if the result of those entries is that packets are forwarded to the dispatcher thread, as it can then correct the inconsistent entry without detriment to the security analysis of the packet. This form of 'one-sided' error means we can use more aggressive, cache replacement-style policies to manage the ANI's tables, rather than requiring that they always perfectly match the routing requested by the backend; this approach in turn allows us to significantly simplify the hardware implementation (for example, the ANI does not need to worry about implementing lengthy hash chains to deal with hash collisions).

4.1.2. Selective packet forwarding

The ANI is an in-line element that for a given packet either forwards it or drops it. The ANI itself does not decide which (unless its initial table lookup for the packet explicitly indicates such an action). Instead, the forward-or-drop decision is made by the backend, with the ANI holding each packet until the backend signals how to treat it. To avoid requiring the backend to transmit entire packets back to the ANI for forwarding, when the ANI routes a packet to a thread, it includes a packet descriptor that the backend can subsequently use to refer to the particular packet.

In addition, some extensions to this basic scheme can short-cut the process. For example, if the backend wants to block a flow completely, it instructs the ANI to mark the flow appropriately in its connection/flow table as an immediate drop for all subsequent packets matching the flow. Similarly, the backend might instead conclude that a flow is benign and does not require further inspection (for example, it corresponds to a URL that the backend has already analyzed, or to a TLS connection that has now negotiated encryption for which the NIPS lacks the session key), in which case it instructs the ANI to forward all of its future packets directly without dispatching them for analysis. Our previous work showed that in some environments such cut-through for partially analyzed flows can allow a security monitor to skip over a great deal of its total traffic volume [19].

Similar to caching per-flow decisions, the ANI can also remember decisions at other granularities. For example, it can use a table of IP addresses to immediately drop all traffic from certain sources (or, alternatively, 'white-list' them). As for flows, we can implement such tables as imperfect caches, provided the default action (no entry in the table, i.e. a cache miss) results in again forwarding the packet to the dispatcher to ensure its proper disposition.

DOI: 10.1002/cpe

4.1.3. Normalization

The ANI needs to also support packet *normalization*, i.e. removing ambiguities from network traffic that can undermine effective security analysis [1]. Normalization can require altering the header or payload contents of packets. Similar to determining forwarding/dropping decisions (see the previous section), the ANI does not perform such packet manipulation itself, but instead provides a mechanism by which the backend can instruct it to alter packets cached in its pending queue and potentially re-dispatch them for repeated analyses now that they have been rendered unambiguous.

4.2. Parallelized network analysis

As outlined in Section 3, there is an enormous potential of parallel processing inherent in analyzing network traffic. However, if we want to adapt traditionally serialized monitoring to effectively use multi-core CPUs to exploit this potential, we must address several challenges. We need to:

- structure the data flow in a fashion that can fully take advantage of the multi-core CPU's potential, by identifying the optimal thread granularity, and considering the effects of the hardware architecture;
- devise scalable communication schemes between threads for correlation of global activity;
- factor in intrusion *prevention* functionality: with concurrent packet processing, it is significantly more difficult to resolve go/no-go decisions in a timely and reliable fashion; and
- support effective evaluation, profiling and debugging of such systems, to identify and remove performance bottlenecks.

We now discuss these areas in turn, assuming that the number of threads is not tied to the number of CPU cores; we may have fewer, equal, or more threads than we have cores running at any specific time; and that initially there is exactly one thread responsible for the packets of a particular flow. This initial thread is the one to which the ANI dispatches the flow's packets; however, the thread may delegate work to other threads, either for follow-on analysis (after it has completed its own analysis) or to replace its own analysis. In this latter case, the initial thread may redirect dispatch for the flow by updating the corresponding table in the ANI if it does not want to process any subsequent packets itself first.

We note that in practice, to achieve optimal performance we might want to limit the number of threads to the number of available CPU hardware threads, especially if we optimize memory access patterns as outlined below. In this model we would statically associate one thread with each core (for single-threaded cores) or for each hardware thread (for multi-threaded cores), which would multiplex its processing across all tasks it gets assigned to.

4.2.1. Architecturally aware threading

Assuming that a single thread initially processes each flow's packets, there are two orthogonal ways to optimize the processing performance. First, the flow's analysis may involve tasks that can execute concurrently, and thus would benefit from delegation to additional threads. Second, we can reap significant performance gains by optimizing the state management across these threads to best match the underlying memory system.

DOI: 10.1002/cpe

Protocol analysis. As per the discussion in Section 3, 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. It is fairly straightforward to structure these blocks into individual threads by following the data flow of the processing, which proceeds along the edges of an *analyzer tree* [15]. For example, the packets of a TCP connection are first passed to the IP analyzer; then to the TCP analyzer which tracks connection state transitions and performs stream reassembly; and finally to one or more analyzers which decode application-layer protocols[§].

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). In general, at this point we do not require any inter-thread communication between threads working on different flows. However, threads come at a cost, and thus one thread per analyzer might not be the best choice. Generally, we need to find a processing granularity that gives us the best trade-off between the benefits of pipelining/parallelizing and the overhead imposed by additional threads—and also taking into account changes in memory access behavior (see text below).

Event processing. After the initial, fairly fixed stages of analysis comes the execution of handlers for the events produced by the protocol parsers. These next stages are considerably harder to effectively parallelize. 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. For example, event handlers called upon session establishment must run to completion before handlers for that session's tear-down event can execute. It is crucial to preserve this order, as otherwise we would undermine the soundness of any stateful analysis.

To control the parallel execution of events, our architecture defines 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 an FIFO order.

In our design, we assign one such event queue to each CPU core. For each flow, the low-level protocol analysis will put its generated events into a particular core's event queue, and each such core will have an event-processing thread that dequeues events, serializing the execution of their handlers. This approach guarantees that cores process their events in the order they are raised, while the event processing of independent flows can proceed concurrently if the events associated with the flows wind up allocated to separate cores.

However, event handlers can generate new events that semantically might no longer be tied to a particular flow anymore (for example, a synchronicity match between two SSH sessions for 'stepping-stone' detection). For these, our architecture also includes global event queues into which analyzers can insert such events. Again, we dedicate a thread to each global queue to oversee the

[§] As noted in Section 3, one needs to run *multiple* application-layer simultaneously analyzers to identify protocols independent of transport-layer ports.

DOI: 10.1002/cpe

sequential execution of its corresponding event handlers. This approach allows us to structure event processing in a very flexible way.

While concurrent event processing already promises a large gain in performance by itself, there is a 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 *non-uniform access* model. As previous work shows [22], memory caching has a *major* impact on performance for highly stateful processing. That effective network security monitoring requires a great deal of dynamic state [23] makes it particularly susceptible to such effects.

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.

We can envision further cache optimization of state management. One possibility regards *event reordering*: if we can identify event handlers that access the same state working set, we might see considerable performance gains by executing them in immediate succession, rather than inter-mixing their execution with that of unrelated handlers. However, as discussed above we cannot arbitrarily reorder events because we must ensure to avoid violating temporal ordering constraints. Still, by identifying such constraints (perhaps with the help of user-provided annotations), we anticipate that such reordering can achieve significant gains.

4.2.2. Scalable communication

Global correlation requires significant communication between individual threads. In an earlier work, we developed and implemented a clusterized version of the Bro network monitoring system [17] that spreads its processing of high-volume network streams over a set of commodity PCs, each analyzing a share of the overall network traffic and synchronizing state via an interconnection network [21].

In many ways, the cluster exploits the same parallelism inherent in network analysis that we discussed in Section 3. However, we found that the global synchronization of the individual cluster nodes quickly threatens to become a bottleneck in large networks. The cluster uses a message-passing approach to state exchange: the synchronization layer propagates each operation on a cluster-global state element to all of the cluster nodes. For some forms of analysis, this rapidly leads to messages traffic that scales as $O(n^2)$ for n state updates. Mitigating this effect required switching to a star topology, introducing a relay node that takes charge of broadcasting updates. However, now the proxy can become a bottleneck as we attempt to scale up the size of the cluster.

Within a single multi-core system, however, we can take advantage of its shared-memory semantics, and thus do not need to rely on explicit message passing for thread communication. However, our evaluation of the Bro cluster shows we must still very carefully consider the potential costs of state coordination. For our multi-core effort, this in particular reflects on the need to align the execution locality of elements in the network analysis chain with the non-uniformities present due to the underlying system's cache hierarchy.

To this end, we need to analyze the communication requirements for threaded operation in particular detail. Our goal is to confine inter-thread communication to a bare minimum so that

DOI: 10.1002/cpe

threads can run with the greatest possible degree of independence. Clearly, for any communication that we cannot avoid, we need to ensure that synchronization points—which can potentially block operation of one or more threads—are well-defined and short-term.

There are several potential strategies to this end. One approach is restructuring the detection algorithms in terms of how they modify or interpret shared state. Our work on the Bro cluster uncovered a number of simple ways to re-code network security analysis algorithms to make them more conducive to concurrent execution. For example, checks for counters reaching specific thresholds can suffer from race conditions when external entities can also increment the counter; a problem that is easy to address—once recognized—by recasting the code to check whether the counter reaches *or surpasses* the threshold.

When restructuring the code does not help, one can also change the semantics of the communication primitives. One approach, which we have explored already within the Bro system, is the concept of *loose* synchronization [23]: due to the large number of messages exchanged between the nodes of the Bro cluster and potential delays imposed by network latency, it is infeasible to fully lock each data structure before every access to ensure global consistency. Any exclusive lock potentially suspends the operation of one or more cluster nodes and can easily lead to packet drops in a high-speed network. Therefore, we introduced into Bro deliberately weakened synchronization semantics, to *expose* the possibility of such race conditions rather than try to ensure they cannot occur.

In a shared-memory system, we likely can employ *some* data structure locks, but certainly will still want to minimize them. Thus, we need to analyze detection algorithms for opportunities to trade-off the requirement of tight synchronization of their data structures versus the overhead that this involves. One approach is to deploy *two-stage* strategies: first prefilter traffic for potentially interesting activity, and only then perform global synchronization for the (presumably much smaller) output set. A simple example is a scan detector that first only looks for potential scanners within a small slice of traffic, but with a high probability for false positives. We can combine the output of multiple such detectors to report scanners with high reliability.

Another approach we can additionally pursue is the use of randomized algorithms (e.g. [24]), which by design can cope with occasional irregularities. With these, intermittent race conditions that sometimes introduce such irregularities do not perturb the reliability of the algorithm's analysis.

Overall, from our experience we find that many detection algorithms exhibit significant potential to be optimized in a communication-efficient fashion. We return to this point in Section 5, where we experimentally verify the parallelization potential of the Bro NIDS.

4.2.3. Prevention functionality

For the event-based analysis model presented in Section 3, we face the significant challenge of realizing intrusion *prevention* functionality, i.e. blocking malicious packets from reaching their destination. The primary problem is that the events—on which the analysis is based—are decoupled from packets that ultimately trigger their generation. A particular packet may trigger from zero to many events, and several packets may all contribute to a single event. For example, since we must first reassemble TCP packets into byte-streams before performing application-layer analysis, if a packet is missing then an entire byte-stream derived from a large number of packets might only become available for analysis upon retransmission of the missing packet.

DOI: 10.1002/cpe

Ideally, the front-end ANI would retain each packet until we have fully processed all events to which the packet contributes in any way. However, this is not practical: high-level analysis algorithms might generate events reflecting aggregated activity significantly after the arrival of the individual packets that comprise the activity. Such events can occur arbitrarily later than the arrival of particular packets that contribute to the generation of the event.

On the other hand, all events *directly* triggered by lower-level analysis will be generated very shortly after the ANI receives the corresponding packet. These events reflect activity visible on a per-flow basis, which is typically manifested in small, localized protocol data units. For these, it *is* feasible to have the ANI hold each packet until all of the events it engenders execute to completion. Because the chain of event processing follows per-flow locality and is directly triggered by the arrival of the packet, the end-to-end analysis latency should remain quite low (e.g. 1–2 ms or less). Such additional latency is essentially invisible to all but the most persnickety applications.

The model of multiple event queues (see Section 4.2.1) makes this approach straightforward to achieve. The system places all events directly triggered for a flow into the same event queue. Once all of them are processed, i.e. the queue has fully drained, then if none of the event handlers has signaled that the ANI must drop or modify the packet, the ANI can safely forward the packet.

This approach does not apply for more global forms of analysis, however. For example, a scan detector can only report a scan after observing some number of connections; it is infeasible for it to block *all* of the probes that a scanner sends, since part of its analysis might well require seeing the degree to which the source's initial attempts succeed or fail [10]. However, due to the global nature of such an 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. In this example, it is not a significant loss if the particular packet that triggered the analysis decision is forwarded; what really matters is blocking the originating host. Accordingly, the scan detector can propagate the offending address to the ANI, which will then discard any future packets originating from that address[¶].

In general, event handlers that raise events themselves need to decide whether these events require processing before a packet can be safely forwarded. The handler can do so by choosing the corresponding event queue: the core's thread queue presumes blocking semantics (i.e. requires all events to be processed before a packet gets a go/no-go decision), while global event queues do not. Since the most apt trade-off between reliable blocking decisions and introduced delays is not obvious up front, our proposed effort will include analyzing the properties of existing detection schemes in this regard. We note that the best choice might change, depending on capabilities of the hardware at hand: for a many-thread-per-core processor, such as the Niagara line, it is critical to make as many events non-blocking as possible, in order to best utilize the additional hardware threads available. However, this is less critical for few-threads-per-core systems such as the Intel Core family.

[¶] Such active blocking of scanners has been in operational use at the Lawrence Berkeley National Laboratory for many years now, implemented by means of a utility that the institute's Bro system executes to contact the site's border router and install a corresponding ACL.

DOI: 10.1002/cpe

4.2.4. *Evaluation, profiling, and debugging*

The concurrent nature of the analysis outlined in the previous sections poses new challenges with regard to the evaluation, profiling, and debugging of network security analysis algorithms. Relating to these issues, in the past we have invested significant efforts into developing tools to instrument our Bro system in order to assess its performance in a sound manner.

We are particularly interested in two areas: (i) identifying race conditions and (ii) understanding memory access patterns. The former reflects a frequent problem often present in concurrent processing: with race conditions, results depend on the order of execution, which is not tolerable. The latter problem area is important in terms of optimizing memory locality to provide optimal performance (see Section 4.2.1).

The key to systematically analyzing a program's behavior is *repeatability*. To this end, we have extensively relied on trace-based evaluation in the past: we first capture a packet trace on a live network link, which we then feed into the system offline as often as required. Ideally, the output should be identical for each run, and also match the results we would have attained during live analysis.

To conduct such assessments with the multi-threaded architecture, we likewise need to rely to a significant extent on trace-based evaluation. However, we cannot readily assess the system directly from traces, since our ANI is a hardware device that operates on *live* traffic. As such, it cannot directly execute on traces. We thus need to develop a second, software-only implementation that fully matches the device's operation, but can also operate from trace files as input.

Another problem for achieving repeatability is *timing*: even when reading a trace, the communication between threads still proceeds in real time. Therefore, we also need to adapt the speed of the packet processing to real time. This can become tricky since slight variations in communication times can also lead to discrepancies. In past work, we faced similar problems when working with multiple communicating instances of the Bro NIDS [25]: results were not reproducible even when we fed in all system inputs via a trace. To solve this problem, we introduced a *pseudo-real-time* mode to the Bro system. When activated, packets from a trace are artificially delayed to match real-time semantics. In addition, the mode 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.

Once we have repeatable settings, we can start analyzing the system's overall behavior. Particularly interesting are memory access patterns, as our most significant concern regarding realizing the potential parallel performance is with respect to working sets and cross-core communication. To this end, we will add instrumentation to track the number and time of accesses to global state, as well as causality-tracking back to trace within which we find accesses triggered by the given events. The results will enable us to fine tune the system's architecturally aware threading for optimal performance.

5. EVALUATION

To understand the parallelization potential that our architecture is in principle able to exploit, we performed a series of simulations based on the Bro NIDS' processing. We picked Bro for our

DOI: 10.1002/cpe

experiments because it already provides many of the same abstractions that our architecture relies on, in particular the internal separation into the two main components: *protocol analysis* and *event processing* (cf. Section 4.2.1). In our simulations we focus on the latter, as the protocol analysis is rather straightforward to parallelize by distributing connections individually across threads. As discussed in 4.2.2, the NIDS Cluster [21] takes a conceptually similar approach and has already demonstrated its promising scaling properties.

It is however much less clear whether event processing can scale similarly well. In the following, we first frame a model for concurrent event execution that is able to address the intricate constraints we face with regards to order-of-execution and inter-event state correlation, and then simulate this scheme using an abstraction of Bro's processing. Based on these simulations, we predict that our concurrent event model, and thus our architecture, is able to scale to large numbers of independent CPUs.

5.1. Concurrent event model

To understand the parallelization potential of Bro's event processing, we use an execution model that, while simplified, captures the main conceptual bottlenecks in processing events concurrently.

We assume that a set of n threads is available for processing events, running independently on different CPUs. Each thread has an incoming *event queue* from which it pops events for sequential execution. The processing of a single event involves the execution of zero, one, or more event *handlers* written in Bro's scripting language. The execution of each handler occupies the thread's CPU for a certain (non-constant) amount of time during which no other handlers can be processed.

During execution, a handler can access *global* variables[‖]. These globals might also be accessed concurrently by handlers running in *other* threads and therefore require synchronization. For more straightforward simulation, in our model we assume a *single* global lock for inter-thread synchronization: whenever a handler that might need access to a global variable starts to execute, it must first acquire the global lock, potentially blocking until that becomes available.

While easy to implement, this locking approach is too coarse to scale well. Nearly all handlers in Bro's standard scripts potentially access some global state, and thus threads would spend most of their time blocking. Therefore, we introduce an optimization. Examining Bro's standard event handlers, we observe that while many of them access globals, most do so only to remember state about their *current unit of processing*. For example, events generated by the protocol analysis tend to store information about the triggering connection (e.g. Bro's HTTP script remembers the URLs that have been requested within a particular connection). However, typically these handlers do not access the corresponding information for *other* connections. Similarly, scripts like the scan detector keep state about individual IP addresses (such as the number of distinct destinations contacted), but do not correlate it *across* sources.

We leverage this observation by slicing such state into individual pieces. Rather than storing all of the information globally, each thread keeps only locally any state that only it needs to access. We introduce this notion into the event model by optionally annotating global variables with *scopes* that define visibility of any changes with regard to processing units. For example, for a global of

[‖] In our simplified model we do not differentiate between read and write accesses.

DOI: 10.1002/cpe

scope `connection`, an update is only guaranteed to be visible to handlers subsequently triggered by the same connection as the one doing the update. Likewise, a global of scope `originator` will reflect modifications only to events triggered by the same originating IP address. In addition to these two, we further introduce scopes `responder` and `host pair` to cover the most common patterns of access to global state present in Bro's scripts.

Going one step further, we extend the notion of scopes from variables to *event handlers*: for each handler, we derive a scope based on the globals it accesses. First, we restrict each handler to access only globals of *one* type of scope (in addition to any not-scoped globals)**. This scope then becomes the scope of the handler. If a handler does not access any global state, we define its scope as `any`.

Now we can incorporate these scopes into our execution model. We schedule event handlers to threads *based on their scopes*: all handlers of the same scope will be processed by the same thread if triggered by the same processing unit (e.g. for a particular connection all handlers of scope `connection` are guaranteed to be run by the same thread). This scheduling strategy localizes all accesses to the scoped global to a single thread, and therefore allows the state to reside in thread-local storage. As a result, all handlers that access *only scoped globals* do not need to acquire the global lock.

To summarize the concurrent event model, whenever an event is raised, we first determine all relevant handlers. We then schedule each of them to a thread determined based on the handler's scope and the current processing unit††, and schedule the handler for execution by inserting it into the corresponding event queue. As the threads process their queues, they only acquire the global lock for handlers accessing non-scoped globals, processing all others directly.

We note that this concurrent event model makes a few simplifications. For example, not all uses of global state in Bro directly map to one of the scopes we have defined so far. Bro's scan detector, for instance, sometimes flips the direction of a connection internally when it believes that the first packet of a connection might have been missed. In these cases, the roles of originator and responder are reversed, which we cannot directly capture in the model laid out so far. We also neglect any negative effects introduced by memory/cache latencies. However, we believe that the model captures the essence of event processing by order of execution requirements and global state synchronization constraints.

5.2. Simulation

To predict the performance of the described approach, we performed a series of experiments with a Python-based implementation of the event model. Based on an actual event log from a Bro run, the simulator schedules events across a specified number of simulated threads according to the constraints identified above. In the following, we first describe the necessary instrumentation of Bro, and then present the results.

**Limiting an event handler in such a way does not impose any significant restriction. Typically event handlers can easily be split into multiple handlers of different scopes; see Section 5.2. Furthermore, the restriction can be enforced via static checking and violations are therefore easy to spot.

††In our simulation, we generally track which connection triggered each event, and then use the handler's scope to extract the relevant components from the connection's 5-tuple. We finally hash these into the set of available threads.

DOI: 10.1002/cpe

5.2.1. *Instrumenting the Bro NIDS*

To perform the simulation, we first defined execution scopes for the global variables most commonly accessed by Bro's default script handlers. When selecting scopes, we started with a set of heuristics to infer the granularity of accesses automatically. For example, most of Bro's script-level tables store information about entities that are derived from the current connection 5-tuple, like the involved IP addresses. By observing which components of a tuple are used during run-time to build the table index for a table operation, we can often identify the right scope. We then further adjusted some scopes manually where the heuristics failed to identify the correct granularity (e.g. a trace might lack the traffic triggering the use of a particular global). In total, we assigned 74 scopes to global script variables, each of one of the types such as `connection` mentioned above.

Next, we modified some of Bro's default scripts to comply with the restriction that only globals of *one* scope can be accessed by each handler (see Section 5.1). Generally, this proved to be easy to achieve, usually by splitting non-conforming handlers up into two or more separate ones, with the first handler raising new events to trigger the subsequent ones[‡‡]. To ensure that we did not affect Bro's analysis semantics with our changes, we used its standard test suite to confirm that Bro's output still matched the original one.

Finally, we instrumented Bro to record each event handler execution with the connection that triggered it, the time when the event was raised, and the time it took to process the event. As the latter figure requires a high-precision clock, we leveraged the open-source PAPI library [26] for reading the CPU's cycle counter.

5.2.2. *Results*

With these changes in place, we ran Bro on a trace captured in the early afternoon of a weekday at the 10GE access link of the Lawrence Berkeley National Laboratory. Owing to the large number of events Bro generates, we restricted the analysis to a duration of 15 min, which resulted in a trace of 24 GB. We configured Bro to perform an extensive analysis using most of the standard analysis scripts that come with the distribution, yielding an event log of about 50 million entries, on which we then ran the Python simulator, specifying increasing numbers of threads to simulate.

The two plots in Figure 3 show the simulation results. In both plots, we show the times a thread spent either processing, blocked waiting for the global lock, or idle due to a lack of event handlers to process. The left plot averages these times over all threads of a particular run, whereas the right plot shows them for one selected thread across all setups to illustrate individual variability. We see that while the single-thread configuration fully utilizes the CPU, idle times increase with the numbers of threads, as we would expect. We also see that the time spent blocking is negligible in all configurations, demonstrating that our concurrent, scope-based event model works quite well.

[‡‡] In rare cases, such a restructuring would have been more difficult to achieve and we then modified the code slightly to work around the problem, sometimes by disabling code seldom exercised. We note that we did not encounter any conceptual problems but only wanted to reduce the effort required to eventually perform the simulations.

DOI: 10.1002/cpe

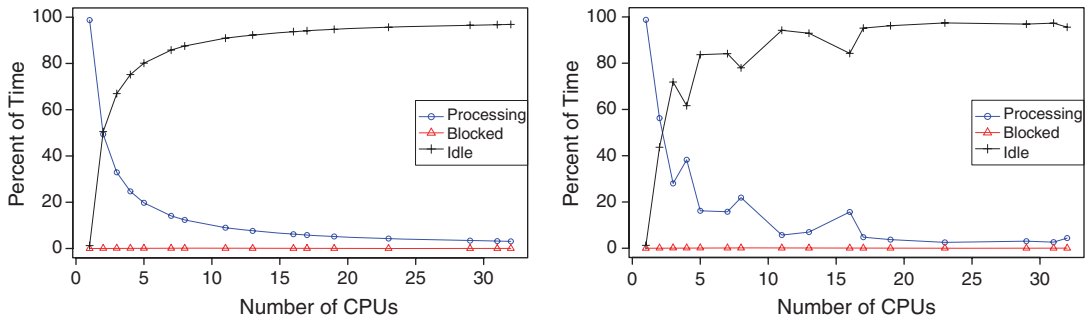


Figure 3. Processing/idle/blocking times with increasing numbers of simulated threads. Left plot shows times averaged over all threads and right plot shows times for a selected individual thread across all configurations.

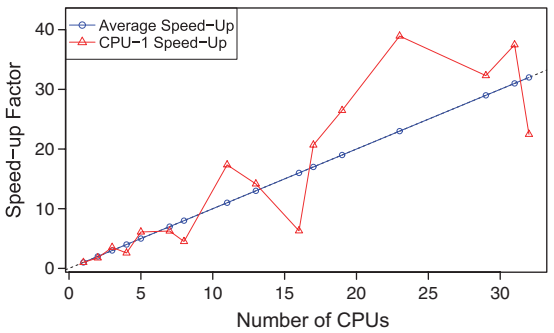


Figure 4. Speed-up with increasing numbers of simulated threads on average and for the first thread in all configurations.

To better visualize the speed we achieve, Figure 4 shows the same simulation results converted into a speed-up factor by relating each thread's processing time to the processing time of the single-thread baseline. Again we show both the speed-up averaged over all threads (circles) and specifically for the first thread in each configuration (triangles). We see that on average, the performance scales almost exactly with the increase in the number of available threads. This is not surprising given the earlier observation that the threads spend hardly any time in a blocking state. Looking at just the first thread, however, we also see that by averaging speed-up factors across threads, we smooth out some of the imbalances introduced by the specifics of how we distribute event handlers across threads. The larger variability indicates that the load-distribution scheme might have some head room for further fine tuning.

Overall, we can conclude from these simulations that the concurrent event model we sketched in Section 5.1 promises to scale very well with increasing numbers of available threads. While in practice further effects, such as memory performance, will affect the achievable performance gain, conceptually our approach appears well suited to exploit the parallel potential we find in large-scale network traffic.

DOI: 10.1002/cpe

6. RELATED WORK

Parallel 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' [27] and its large set of byte-level signatures.

FPGA-based work in this regard has investigated the use of non-deterministic finite automata to match regular expressions [28,29], compiling regular expressions into deterministic finite automata [30], and then quickly generating new, compiled FPGA binaries [31]. Other custom hardware efforts, not specific to FPGAs, have investigated building optimized Aho-Corasick trees for sets of strings [32] and specialized architectures based on collections of highly optimized tiny state machines, each of which looks for a portion of a string [33].

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. As shown in our previous work, this latter operation actually turns out to be more difficult than the string matching itself, particularly when considering the problem of adversaries who target the memory available to the reassembler [34].

Parallelizing richer, stateful hardware elements, such as TCP stream reassembly, have not been explored in as much depth. Schuehler *et al*. developed a TCP processor that maintains a small, fixed amount of state per connection for several thousand concurrent connections at OC-48 speeds [35]. This was subsequently integrated into a signature-matching system [36], as well as being combined with a Bloom filter-based system [37] to construct a simplified version of Snort in hardware.

Unfortunately, TCP processors constructed in this way suffer from significant limitations. First, they are subject to *evasion* attacks [1,38]: if the processor operates in a passive monitoring role, an attacker can easily evade detection by fragmenting and reordering packets[§§].

Along with research efforts, 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, whereas others rely on FPGAs or network processors. However, the analysis provided by vendor systems appears heavily focused on high-speed signature detection (e.g. [39,40]), rather than higher-level semantic analysis, with the custom hardware serving simply to parallelize low-level matching operations. For example, Kruegel and colleagues were able to reverse engineer the signatures used by ISS RealSecure in order to construct variants that evade detection by it [41], and in our own operational experiences with McAfee's Intrushield product, we found we could readily trigger false alarms regarding purported file-sharing traffic by issuing particular HTTP requests [42].

By relying on customized hardware rather than general-purpose CPUs, these technologies have difficulty in tracking Moore's law-style scaling. Network processors have not enjoyed the smooth,

[§§] Some systems counter these attacks by dropping out-of-order packets. This, however, can impose a huge performance penalty due to TCP's congestion response. Even more seriously, such dropping can *amplify* packet loss during times of congestion, increasing the work the network must perform at the very moment when it lacks resources for its existing load.

DOI: 10.1002/cpe

continual evolution of microprocessors, with new versions of a given processor often requiring rewriting substantial portions of the code. (Wun *et al*. present an approach toward unifying the programming interface in [43]). They also remain difficult to program for high-level analysis due to their lack of the powerful cached memory semantics of a commodity microprocessor. Owing to the low level at which FPGA designs express parallelism, scaling such designs up requires major recoding efforts. ASIC designs likewise embed parallelism deep within the execution model, so scaling them up can require complete reengineering, at great expense for mask sets.

In terms of *higher-level* network security analysis, several existing approaches focus on executing a small number of components concurrently on independent cores, achieving some speed-up yet not a generally scalable, concurrent execution model. Like with hardware-supported solutions, Snort has seen particular attention in this regard. For example, recently Verdu *et al*. presented *Snort-MT* [44], which identifies a set of processing layers sharing related sets of states to then localize their memory accesses by executing them in separate threads. In [45], Vasiliadis *et al*. outsource parts of Snort's processing to a graphical processing unit. Endace provides a commercial solution that load balances a packet stream across multiple Snort instances running on a multi-core platform [46].

For our work, we take our main inspiration from the work of Kruegel *et al*., who explored the design of front-end NIDS load balancers [47]. 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. For example, an element performing scan detection (a form of global analysis that requires observing all connection requests and responses) will receive copies of all connection requests; however, these also need to be sent to elements performing parsing of the corresponding application protocols.

The issue of such front-end dispatch becomes subtle because there are many such forms of global analysis. For example, *content sifting* [14] requires looking at a large pool of potentially suspicious strings that may be taken from any connection; *contact graph* analysis [12] can efficiently detect new worms, but requires a global connection history within a time window; *stepping-stone* detection [13] needs to correlate packet timing across connections that may have no hosts in common; and, as mentioned above, scan detection [10,11] needs to track all connection initiation requests. At a simpler level, we note that many attacks seen today involve complex application-level sessions that span multiple connections and sometimes multiple hosts. For example, of the 66 different types of worms detected by our GQ honeyfarm over a four-month period, *all* used exploits that required more than one connection—sometimes as many as *seventy two* (for BAT.Boohoo.Worm) [48].

In [21], we have used a connection-based load-balancing approach to build a *NIDS cluster* out of commodity PCs. The *stateless* load-balancing algorithm is able to operate at very high line rates yet it does not allow dynamically rerouting packets for analysis with a different target than the one initially chosen. In [49], Guo *et al.* also deploy a connection-based scheduling scheme to parallelize the operation of L7-filter [50], finding that the resulting maximum throughput is close to linear speedup compared with the sequential version.

Because intrusion prevention requires *in-line* operation, to realize parallelizable intrusion prevention we need to go significantly further than the slicing approach developed by Kruegel *et al*. Their architecture allows intelligent front-end load-balancing. What we need in addition are (i) ways of

DOI: 10.1002/cpe

structuring the analysis itself such that it is amenable to multi-core parallelization (addressed by our event-based structure) and (ii) support for *prevention* functionality (addressed by incorporating control feedback from the analysis elements back to our front-end, the ANI).

Modern general-purpose CPUs. As developed in the Introduction, today the gains provided by Moore's law manifest not in uniprocessor execution speed but in performance aggregated across a number of concurrent execution units. Modern designs include symmetric *multi-threaded* CPU cores [51,52], which allow a single CPU to switch between multiple independent threads of execution, and *multi-core* systems, where a single die holds multiple CPUs [52,53]. Recent systems support both: multiple CPUs each executing multiple threads, as discussed below. (For simplicity, we refer to CPUs within this spectrum of multiple threads and cores as simply 'multi-core.')

In concrete terms, AMD is shipping dual-core systems [2]; Intel has shipped dual-thread SMT systems (hyperthreading) Pentium 4 designs [51], dual-core designs for server, desktop, and portable use [53], dual-core designs with each core supporting two threads [3], quad-core [4], and six-core [5] CPUs. Sun's UltraSPARC T1 (Niagara) contains 8 CPU cores, with each core supporting four threads [52], for a total of 32 simultaneous threads of execution. Its successor, the T2 (Niagara2) [6], already supports eight threads per CPU core, for a total of 64 threads. On the slightly more specialized front, Tilera [7] has 64 core processors with dedicated network interfaces, and Intel has announced a graphics-card family, Larrabee [8], which uses up to 64×86 cores with vector extensions. Both Tilera and Larrabee also include dedicated processor-to-processor interconnect networks.

It is critical to recognize that to exploit the power of such processors, programs must be specifically designed to have a parallelizable structure; otherwise, one will not see any substantial performance gains. However, when developing 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, since main-memory latency can be 70 ns for random access (over 100 clock cycles on a 1.5 GHz processor), compared with L1/L2-cache clock cycles of 3 and 14, respectively (all numbers are for the Intel Core Duo microprocessor).

In a multi-threaded core, the threads must share a common working set. If they do not, cache thrashing will significantly degrade performance [54]. This is one reason why the Pentium 4 HT in fact performs better on some benchmarks when multi-threading is *disabled* [22]. In general, significant care is required when scheduling threads on a multi-threaded system.

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 [2,7], it becomes vital to create and feed the threads in a memory-aware manner. As an example of the approach we envision to use for the ANI, Kumar *et al*. discuss how packets can be directly placed into a CPU's cache to be then further processed by a network stack.

All these factors create a very difficult programming problem. Not only are there complications imposed by concurrent execution (deadlock, livelock, difficulties in identifying and restructuring bottlenecks), but the parallel execution also needs to be parameterized for the target system's memory architecture.

DOI: 10.1002/cpe

7. CONCLUSION

Network traffic continues to grow at rates that outpace flattening CPU performance curves, even more so as the speed of uniprocessor execution—so long blessed with Moore's law doubling—has finally 'hit the wall'. Recently, however, hardware vendors have begun delivering commodity CPUs whose *aggregated* throughput again reflects Moore's law-style scaling, with the parallelization gains coming from multi-core/multi-thread architectures. Yet 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 such parallel execution of network attack analysis. At its lowest layer, the architecture relies on an 'active network interface' that provides an in-line interface to the network, reading in packets and forwarding them only after they are fully inspected and deemed safe. The device dispatches packets to a set of threads that structure the processing as an event-based analysis model well suited to exploit many of the opportunities for concurrent execution. In this model, we capture event dependencies by scheduling related events to the same thread, serializing their execution without needing to resort to expensive inter-thread synchronization. We demonstrated via trace-driven simulation that this approach holds promises to scale quite well to large numbers of independent threads.

The presented concurrent architecture shows how we can transform the traditionally serial network security analysis pipeline into a highly parallelizable form that is able to take full advantage of the multicore concurrency offered by modern and future commodity CPUs. Given how parallel hardware appears likely to evolve in the future, we expect such a paradigm to ultimately prove highly beneficial, and we are now in the process of implementing these concepts within the framework provided by the Bro network intrusion detection system.

ACKNOWLEDGEMENTS

This work was supported in part by NSF Awards CNS-0716636 and NSF-0433702, as well as by a grant from Intel Corporation. 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.

REFERENCES

1. Handley M, Paxson V, Kreibich C. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, 2001.
2. Advanced Micro Devices. AMD Athlon 64 X2 Dual Core Processor. Available at: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html [20 November 2008].
3. Intel Corporation. Dual Core Intel Xeon Processor 7100 Sequence. Available at: http://www.intel.com/products/processor/xeon/7100_prodbrief.htm [20 November 2008].
4. Intel Corporation. Quad Core Intel Xeon Processor 5300 Sequence. Available at: ftp://download.intel.com/products/processor/xeon/dc53kprodbrief.pdf [20 November 2008].
5. Intel Corporation. Six Core Intel Xeon Processor. Available at: http://ark.intel.com/cpu.aspx?groupID=36937 [20 November 2008].
6. Sun Microsystems. UltraSPARC T2 Overview. Available at: http://www.sun.com/processors/UltraSPARC-T2/features.xml [20 November 2008].
7. Tilera Corporation. Tilera Tile64 Processor. Available at: http://www.tilera.com/products/TILE64.php [20 November 2008].

DOI: 10.1002/cpe

8. Seiler L, Carmean D, Sprangle E, Forsyth T, Abrash M, Dubey P, Junkins S, Lake A, Sugerman J, Cavin R, Espansa R, Grochowski E, Juan T, Hanrahan P. Larrabee: A many-core ×86 architecture for visual computing. *SIGGRAPH*, Los Angeles, CA, 2008.
9. Paxson V, Asanovic K, Dharmapurikar S, Lockwood J, Pang R, Sommer R, Weaver N. Rethinking hardware support for network analysis and intrusion prevention. *Proceedings of the USENIX Hot Security Workshop*, Vancouver, BC, Canada, August 2006.
10. Jung J, Paxson V, Berger AW, Balakrishnan H. Fast portscan detection using sequential hypothesis testing. *IEEE Symposium on Security and Privacy*, Oakland, CA, 2004.
11. Weaver N, Staniford S, Paxson V. Very fast containment of scanning worms. *USENIX Security Symposium*, San Diego, CA, August 2004.
12. Ellis D, Aiken J, Attwood K, Tenaglia S. A behavioral approach to worm detection. *Workshop on Rapid Malcode*, Fairfax, VA, 2003.
13. Zhang Y, Paxson V. Detecting stepping stones. *Proceedings of the USENIX Security Symposium*, 2000.
14. Singh S, Estan C, Varghese G, Savage S. Automated worm fingerprinting. *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.
15. Dreger H, Feldmann A, Mai M, Paxson V, Sommer R. Dynamic application-layer protocol analysis for network intrusion detection. *USENIX Security Symposium*, Vancouver, BC, Canada, 2006.
16. Weaver N, Paxson V, Gonzalez JM. The shunt: An FPGA-based accelerator for network intrusion prevention. *ACM Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2007.
17. Paxson V. Bro: A system for detecting network intruders in real-time. *Computer Networks* 1999; **31**(23–24):2435–2463.
18. Zhang Y, Paxson V. Detecting backdoors. *Proceedings of USENIX Security Symposium*, 2000. Available at: http://www.aciri.org/vern/papers/stepping-sec00.ps.gz.
19. Gonzalez J, Paxson V, Weaver N. Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention. *ACM Communications and Computer Security* (*CCS*) *Conference*, Washington, DC, 2007.
20. Watson G, McKeown N, Casado M. NetFPGA: A tool for network research and education. *The 2nd Workshop on Architectural Research using FPGA Platforms* (*WARFP*), Austin, TX, 2006.
21. Vallentin M, Sommer R, Lee J, Leres C, Paxson V, Tierney B. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, Gold Coast, Australia, 2007.
22. Kim J, Settle A, Janiszewski A, Connors D. Understanding the impact of inter-thread cache interefernce on ILP in modern SMT processors. *Journal of Instruction-Level Parallelism* 2005; **7**:1–28.
23. Sommer R, Paxson V. Enhancing byte-level network intrusion detection signatures with context. *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, 2003. Available at: http://www.net.in.tum.de/~robin/papers/ccs03.ps.
24. Venkataraman S, Song D, Gibbons PB, Blum A. New streaming algorithms for fast detection of superspreaders. *Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, 2005.
25. Sommer R, Paxson V. Exploiting independent state for network intrusion detection. *ACSAC*, Tucson, AZ, December 2005.
26. Performance Application Programming Interface (PAPI). Available at: http://icl.cs.utk.edu/papi [20 November 2008].
27. Roesch M. Snort: Lightweight intrusion detection for networks. *Proceedings of the Systems Administration Conference*, 1999.
28. Sidhu R, Prasanna VK. Fast regular expression matching using FPGAs. *IEEE Symposium on Field-programmable Custom Computing Machines* (*FCCM*), Rohnert Park, CA, U.S.A., April 2001.
29. Franklin R, Carver D, Hutchings B. Assisting network intrusion detection with reconfigurable hardware. *Proceedings from Field Programmable Custom Computing Machines*, Napa, CA, 2002.
30. Moscola J, Lockwood J, Loui R, Pachos M. Implementation of a content-scanning module for an Internet firewall. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), Napa, CA, U.S.A., April 2003; 31–38.
31. Lockwood JW, Moscola J, Kulig M, Reddick D, Brooks T. Internet worm and virus protection in dynamically reconfigurable hardware. *Military and Aerospace Programmable Logic Device* (*MAPLD*), Washington, DC, September 2003; E10.
32. Tuck N, Sherwood T, Calder B, Varghese G. Deterministic memory-efficient string matching algorithms for intrusion detection. *IEEE Infocom*, Hong Kong, China, March 2004.
33. Tan L, Sherwood T. A high throughput string matching architecture for intrusion detection and prevention. *The 32nd Annual International Symposium on Computer Architecture* (*ISCA*), Madison, WI, 2005.
34. Dharmapurikar S, Paxson V. Robust TCP stream reassembly in the presence of adversaries. *USENIX Security Symposium*, Baltimore, MD, August 2005.
35. Schuehler DV, Lockwood JW. Tcp-splitter: A TCP/IP flow monitor in reconfigurable hardware. *Hot Interconnects*, Stanford, CA, August 2003; 89–94.

DOI: 10.1002/cpe

36. Schuehler DV, Moscola J, Lockwood JW. Architecture for a hardware-based, TCP/IP content-processing system. *IEEE Micro* 2004; **24**(1):62–69.
37. Attig ME, Lockwood J. SIFT: Snort intrusion filter for TCP. *Symposium on High Performance Interconnects* (*HotI*), Stanford, CA, August 2005; 121–127.
38. Ptacek TH, Newsham TN. Insertion, evasion, and denial of service: Eluding network intrusion detection. *Technical Report*, Secure Networks, Inc., January 1998.
39. Cisco Systems. Block ASA 5500 Series Adaptive Security Appliances. Available at: http://www.cisco.com/en/US/products/ps6120/prod_brochure0900aecd80402ef4.html [20 November 2008].
40. Juniper Networks. Juniper Networks ISG Series With IDP. Available at: http://www.juniper.net/products_and_services/intrusion_prevention_solutions/isg_series_with_idp/ [20 November 2008].
41. Kruegel C, Mutz D, Robertson W, Vigna G, Kemmerer R. Reverse engineering of network signatures. *Proceedings of the AusCERT Asia Pacific Information Technology Security Conference*, Gold Coast, Australia, May 2005.
42. Sommer R. Viable network intrusion detection in high-performance environments. *Ph.D. Thesis*, TU Muenchen, 2005.
43. Wun B, Crowley P, Raghunath A. Design of a scalable network programming framework. *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, CA, 2008.
44. Verdu J, Nemirovsky M, Valero M. MultiLayer processing—An execution model for parallel stateful packet processing. *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, CA, 2008.
45. Vasiliadis G, Antonatos S, Polychronakis M, Markatos EP, Ioannidis S. Gnort: High performance network intrusion detection using graphics processors. *Proceedings of the Recent Advances in Intrusion Detection*, Boston, MA, 2008.
46. NinjaBox-Z, Applied Watch. Available at: http://www.endace.com/assets/files/ninjaBoxZ_applied_watch.pdf [20 November 2008].
47. Kruegel C, Valeur F, Vigna G, Kemmerer RA. Stateful intrusion detection for high-speed networks. *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.
48. Cui W, Paxson V, Weaver N. GQ: Realizing a system to catch worms in a quarter million places. *Technical Report TR-06-004*, International Computer Science Institute, 2006.
49. Guo D, Liao G, Bhuyan L, Liu B, Ding J. A scalable multithreaded L7-filter design for virtualized multi-core servers. *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, CA, 2008.
50. Application Layer Packet Classifier for Linux. Available at: http://l7-filter.sourceforge.net [20 November 2008].
51. Intel Corporation. Intel Pentium 4 Processor. Available at: http://www.intel.com/products/processor/pentium4/index.htm [20 November 2008].
52. Sun Microsystems. UltraSPARC T1 Overview. Available at: http://www.sun.com/processors/UltraSPARC-T1/ [20 November 2008].
53. Intel Corporation. The Intel Core Duo Processor. Available at: http://www.intel.com/products/processor/coreduo/ [20 November 2008].
54. Hily S, Seznec A. Standard memory hierarchy does not fit simultaneous multithreading. *Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation* (*with HPCA-4*), Las Vegas, NV, January 1998.

DOI: 10.1002/cpe