Conservative vs. Optimistic Parallelization of Stateful Network Intrusion Detection *

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

Abstract

This paper presents and experimentally analyzes the performance of three parallelization strategies for the popular open-source Snort network intrusion detection system (NIDS). The parallelizations include 2 conservative variants and 1 optimistic scheme. The conservative strategy parallelizes inspection at the level of TCP/IP flows, as any potential inter-packet dependences are confined to a single flow. The flows are partitioned among threads, and each flow is processed in-order at one thread. A second variation reassigns flows between threads to improve load balance but still requires that only one thread process a given flow at a time. The flow-concurrent scheme provides good performance for 3 of the 5 network packet traces studied, reaching as high as 4.1 speedup and 3.1 Gbps inspection rate on a commodity 8-core server. Dynamic reassignment does not improve performance scalability because it introduces locking overheads that offset any potential benefits of load balancing.

Neither conservative version can achieve good performance, however, without enough concurrent network flows. For this case, this paper presents an optimistic parallelization that exploits the observation that not all packets from a flow are actually connected by dependences. This system allows a single flow to be simultaneously processed by multiple threads, stalling if an actual dependence is found. The optimistic version has additional overheads that reduce speedup by 25% for traces with flow concurrency, but its benefits allow one additional trace to see substantial speedup (2.4 on five cores).

1 Introduction

Network intrusion detection systems (NIDSes) run on a server at the edge of a LAN to identify and log Internet-based attacks (called exploits) against a local network. Unlike firewalls, which shut off external access to certain ports, NIDSes can monitor attacks on externally-exposed ports used for network services. The most popular NIDS is the open-source Snort, which identifies intrusion attempts by comparing every inbound and outbound packet against a *ruleset* [14]. Rules in the set represent characteristics of known attacks, such as the protocol type, port number, packet size, packet content (both strings and regular expressions), and the position of the suspicious content. Such rules target various forms of exploits, including buffer overruns, cross-site scripting, and denial-of-service (DoS). Each new exploit leads to new rules, with rulesets growing rapidly. The most recently-released Snort rulesets have over 4000 rules.

NIDSes such as Snort are computationally intensive since they must decode packet data, inspect it according to a ruleset, and log intrusions. An NIDS receives input as packets, but must aggregate distinct (and possibly noncontiguous) network packets into TCP streams to uncover attacks that span several packets. Additionally, an NIDS often saves asides state information that affects its processing of later packets. For example, if a given sequence of characters represents a possible attack in the body of an HTML document but may appear normally in an image, the NIDS should not trigger an alert on that attack if an earlier packet indicated that this data transfer was an image. Such constraints are incorporated into Snort as *stream reassembly* and *flowbits*, respectively. All TCP data is reassembled into streams, and about 36% of rules require flow tracking (90% of which are related to NetBIOS). These overheads limit Snort to an average packet processing rate of about 699 Mbps on a modern host machine (1.8 GHz Xeon processor) — well below link-level bandwidths that are already commoditized at 1 Gigabit and are now approaching 10 Gigabits. For higher performance, companies and researchers have proposed scalable solutions based on clustering, but these require an expensive load-balancing switch [8, 13, 15].

This paper presents and evaluates three methods to parallelize Snort for multicore PC-based servers. Although the stream reassembly and flowbits phases described above require in-order packet processing, any information sharing between packets only applies to packets in the same IP flow (which include not only TCP streams but also source/destination communication pairs in other protocols). The first parallelization strategy is called the *flow-concurrent* parallelization. This scheme exploits concurrency by parallelizing ruleset processing on a flow-by-flow basis, since this represents the minimum granularity at which dependences can be maintained conserva-

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

tively. All packets are initially received by a "producer" thread. That thread inspects the IP headers to determine the flow to which the packet belongs and then steers that packet to the appropriate "consumer" thread based on whether or not that flow has already been assigned to a thread. Since each given flow is only processed by one thread at any given time, the dependences required for proper stream reassembly and flow tracking are maintained easily at the queue between the producer and consumer. A variation on this conservative scheme aims to improve load balance by allowing a flow to be dynamically reassigned from one thread to another, but only if there are currently no packets from that flow still waiting to be processed. This reassignment still preserves the invariant that a specific flow is only processed by one thread at any given time.

The above conservative schemes exploit parallelism well if there are enough independent flows, but provide no benefits if all packets are from the same flow. The latter case is not likely in a high-bandwidth edge NIDS, but does represent a limitation of these schemes. The alternative parallelization is an optimistic variant on flow concurrency. This scheme starts with the basic flow-concurrent parallelization but can dynamically reassign a flow to a different thread even while earlier packets of the flow are still being processed, potentially exploiting parallelism even with just one flow. This optimistic version relies on two key observations. First, TCP stream reassembly will still take place even if a stream is broken at some arbitrary point; reassembly is triggered by various flush conditions, one of which is a timeout. It is also easy to force additional flushes if needed for correctness. Consequently, any unprocessed earlier packets will still go through stream reassembly at their thread even though later packets are being reassembled and processed in another thread. (This property also allows the conservative reassignment described above). Second, most packets do not match rules that use flowbits tracking, so enforcing ordering across all packets in a flow just to deal with a few problematic rules is too restrictive. To precisely deal with the rules that do use flowbits, the optimistic system stalls processing in any packet that sets or checks flowbits unless it is the oldest packet in its flow. This condition is checked by adding per-flow reorder buffers. This system is optimistic in the sense that it reassigns threads under the assumption that the actual use of flowbits is uncommon, but is still conservative in maintaining correct ruleset processing without speculation and rollbacks.

All the parallelizations use most of the same packet processing code as the current Snort (version 2.6), with minor modifications to make certain code segments re-entrant and well-synchronized using Pthreads. The resulting NIDS tools are evaluated and analyzed on two different systems: a 2U rack-mounted x86-64 Linux system with two quad-core Xeon processors (eight cores in total) and a similar system with two dual-core Opteron processors. The flow–concurrent parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 4.1 speedup on 8 processor cores and processing at speeds up to 3.1 Gbps. The extra overheads introduced by allowing reassignment actually degrade performance by about 4.5% for the traces that exhibit good flow concurrency, and additional overheads in the optimistic parallelization degrade performance by an additional 16%, limiting speedup to 3.0 on six cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.4 on five cores), with a peak traffic rate over 2.6 Gbps. All schemes achieve an average traffic rate of over 1.7 Gbps for the 5 traces, providing several options for increasing NIDS performance by exploiting multicore processors.

2 Background

Snort is the most popular intrusion-detection system available. The system and its intrusion-detection ruleset are freely available, and both are regularly updated to account for the latest threats [14]. Snort rules detect attacks based on traffic characteristics such as the protocol type (TCP, UDP, ICMP, or general IP), the port number, the size of the packets, the packet contents, and the position of the suspicious content. Packet contents can be examined for exact string matches and regular-expression matches. Snort can perform thousands of exact string matches in parallel using one of several multi-string pattern matching algorithms, including the well-known Aho-Corasick algorithm [1] and a modified version of the Wu-Manber algorithm [21], which can be selected by the user. Additionally, Snort includes preprocessors that perform certain operations on the data stream. Some important preprocessors include **flow**, **stream4**, and **HTTP Inspect**. The flow preprocessor associates each scanned packet to a specific network traffic flow between a source and destination pair and allows rules to set, clear and check flags (called *flowbits*) associated with the flow based on packet contents. For example, one rule checks for a GIF image header and sets a specific flowbit, and another checks for a heap overflow exploit that may occur in a later packet in flows which have that bit set. The stream4 preprocessor tracks TCP connection states for use by rules (for example, only alert for packets in an established TCP connection). Stream4 also performs stream reassembly, concatenating multiple scanned packets from a given direction into a single conceptual packet so that rules may match content that spans packet boundaries. Without stream reassembly, an attacker may hide attacks by splitting them across packets. Packets are reassembled into stream buffers and sent into the inspection process after a "flush point" is triggered by conditions such as processing a certain randomly-selected amount of data or a timeout. The HTTP Inspect preprocessor converts URLs to a normalized canonical form so that rules can specifically match URLs rather than merely strings or regular expressions.

The Snort ruleset has been growing at a nearly constant rate over the past 6 years, with over 4000 attack signatures in the most recent rulesets. Although the performance of multi-string content matching does not depend directly on the number of rules, a greater number of rules does require a greater amount

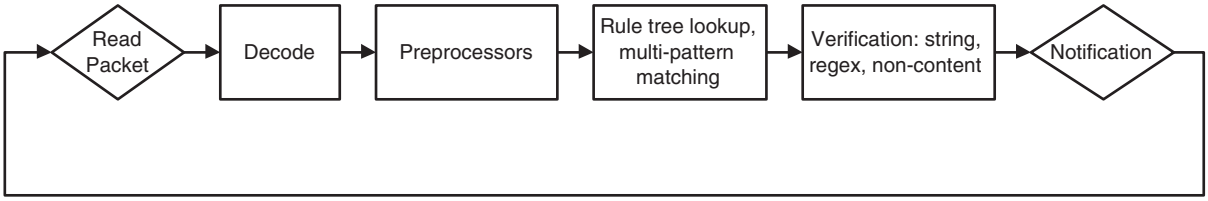


Figure 1. The Snort packet processing loop

of memory and may require more time to be spent in other stages of intrusion detection. Over 86% of the rules are for TCP-based exploits (including over 30% for HTTP rules), with UDP at about 9%, ICMP at 4%, and other IP rules at just over 1%. All rules specify several conditions, alerting only if all such tests are met. Nearly all of the higher-level protocol rules (HTTP, TCP, and UDP) check for string matching content, but a large fraction of the ICMP and general IP rules do not. The multi-string pattern match generally serves as a first-order filter; other tests are not performed unless the corresponding pattern matches. This is particularly important for rules that specify time-consuming regular expressions. About 10% of HTTP rules, 40% of all TCP rules, and 30% of UDP rules test for regular expression matches.

The following rules demonstrate some of the kinds of traffic characteristics used by Snort to detect attacks:

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

Figure 1 depicts Snort's packet processing loop. Snort first reads a packet from the operating system using the `pcap` library (also used by `tcpdump` and other analysis tools). The decode stage interprets the packet's tightly-encoded protocol headers and stores the results in Snort's loosely-encoded packet data structure. Snort then invokes the preprocessors, which use and manipulate packets in various ways. The rule-tree lookup and pattern matching stage determines which rules are relevant for the packet at hand (based on port number) and checks the packet content for the attack signatures defined in the string rules using the multi-string matching algorithms. Packets may match one or more strings in the multi-string match stage, each of which is associated with a different rule. For each of those rules, all the remaining conditions are checked, including other strings, non-content conditions, and regular expressions. Because each match from the multi-pattern algorithm may or may not result in a match for its rule as a whole, this stage may be thought of as a "verification" stage. This stage also calls detection functions for any rules without exact string matches. The last stage alerts the system owner when rules match.

Figure 2 categorizes the execution time of a system running Snort into various components for five different network test patterns. Most of these components correspond to the stages shown in Figure 1. The component labeled **Other** includes utility code and library functions shared among several components, the overall packet processing loop and other code between the stages, operating system activity, and other processes running on the system (such as the profiler itself). Most of the time spent in this category consists of shared library calls (`malloc`, `memset`, etc.) and code that calls the other stages and transitions between them. The remaining part is very small, and its effect is not considered further. The Snort code tested here is modified to read all of its packets sequentially from an in-memory buffer to allow the playback of a large network trace representing communication from various hosts to a local network. The network traces used in these tests and their significance are described in more detail in Section 5.

The profiles shown here were gathered using the `oprofile` full-system profiling utility running on a Dell Poweredge 2950 with two quad-core Xeons (8 processors total), but Snort only runs on 1 processor. The system has 16 GB of DRAM and uses Linux version 2.6.18. The profile was gathered using the `oprofile` full-system profiling utility, and the Snort configuration included the most important preprocessors: flow, stream4, and HTTP Inspect as described above. The overall performance of this system averages 699 Mbps for these traces with a peak of 1094 Mbps.

As Figure 2 shows, string content matching ranges from 29–77% of the execution time of the system with an average of 48%. For all traces except DEF1, the combination of ruleset processing components (string match, verification, and regular expression) make up over 54% of execution time. Thus, any performance optimization strategy must effectively target those components. At the same time, the other components cannot be ignored since they make up an average of 41% of execution time.

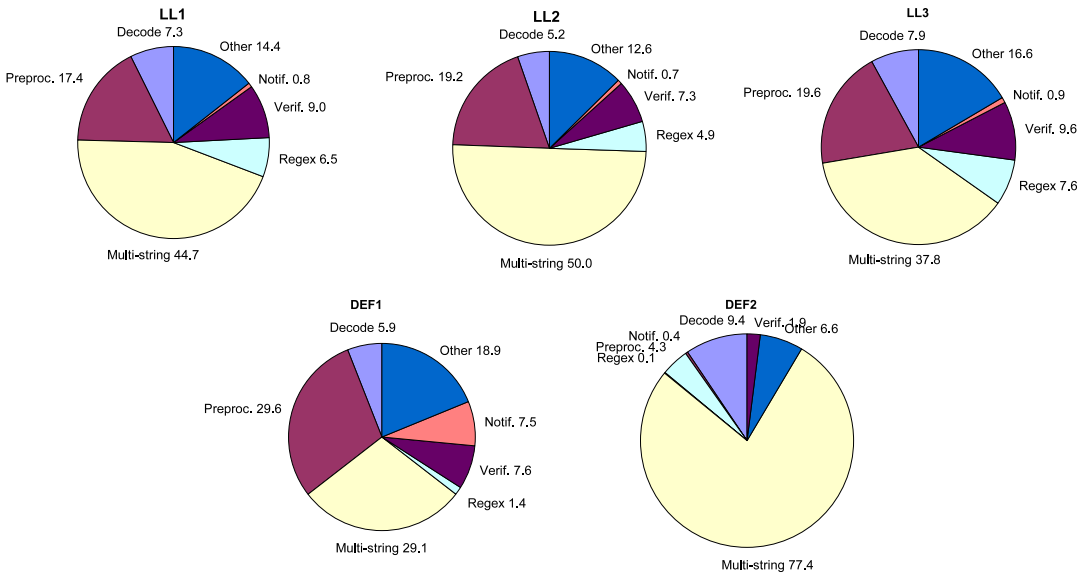


Figure 2. Execution time of Snort categorized into principal components when run using various traces on an Intel Xeon-based system

3 Concurrency Opportunities

As discussed in Section 2, the main loop of the Snort NIDS works on one packet at a time. Figure 3 illustrates the interactions between the processing of two separate packets in Snort. The center column of Figure 3 depicts resources shared by the processing of multiple packets, with dashed lines indicating the accesses to these resources by specific stages in the Snort processing loop. Although this paper focuses on Snort, the same basic steps and resources are present in any intrusion detection system that reassembles packets from the same stream and tracks flow-specific state.

Even though Snort receives its input as packets, the depicted resource sharing prevents packet-level parallelization. In particular, the flow tracking table must be consulted while processing all packets and must be updated any time a packet arrives from a flow that is not currently being tracked. The rules associated with flowbits are tested and set in the verification stage. Similarly, each TCP packet's stream must have its state checked and set (loosely following TCP's state transition table [7], but with provisions for handling missing packets and picking up streams in mid-session) in the stream reassembly preprocessor, the actual reassembly information must be updated for each packet in an established connection, and the TCP stream state must again be checked in verification since some rules depend on this. Both flowbits processing and stream reassembly thus require in-order processing of packets to maintain correct state information, limiting the available concurrency.

4 Parallelization Strategies

4.1 Flow-level Parallelization

Although packet-level parallelization is impractical because of ordering requirements on shared data structures, any actual information sharing only applies to packets in the same flow. Since packets from one flow will never affect the flowbits or TCP connection state of another flow, different flows can be processed by independent processing threads with no constraints on ordering. Moreover, the data sharing requirement of the stream preprocessor can be eliminated by simply maintaining separate stream tables for each thread. Then each thread can be responsible for different flows, as long as packets from the same flow are always steered to the same processing thread in-order. This steering process should consist of a minimal amount of code to determine the flow associated with any given packet and then enqueue the packet for processing by the appropriate thread. In-order processing by each thread guarantees that the packets from each flow will be processed in the same relative sequence as in the serial code. Consequently, this flow-based concurrency model maintains all required ordering constraints between packets in a flow.

Effective flow-level parallelization depends on the existence of concurrent flows in the network stream. However, this should be a normal situation for high-bandwidth NIDS sensors because they usually protect many machines or an entire network.

Figure 4 shows the stages of the flow-based concurrency model. This model consists of two components: the producer routine and the consumer routine. The producer reads the packet from the interface and assigns the packet to its thread

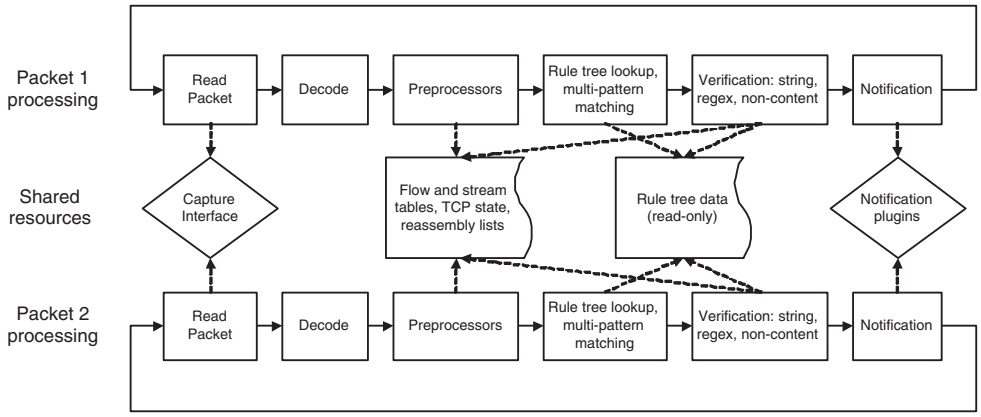


Figure 3. Multiple instances of the Snort packet processing loop, with access to shared resources explicitly identified

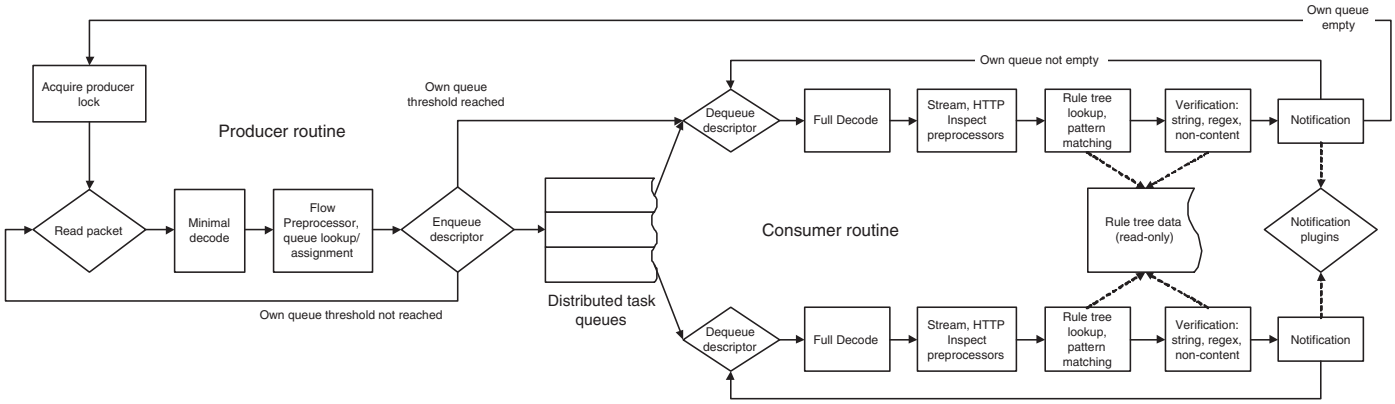


Figure 4. Flow-level parallelization strategy for Snort

based on its flow. The consumer routine processes the remaining stages of the NIDS just as in the single-threaded Snort. Each thread has its own work queue and consumes packets from it as long as there are packets waiting. If its own queue becomes empty, it then attempts to become the producer, and begins reading packets and assigning them to their proper work queues. These work queues can be made quite large since each entry only requires 3 pointers; consequently, it is unlikely that a work queue will fill up and cause head-of-line blocking by stalling the producer. Any thread can be the producer, but the producer code is protected by a mutex lock so that only one thread may do so at a time. A thread will continue to act as the producer until its own queue size reaches a threshold, at which point it gives up the producer lock and returns to processing from its work queue. The threshold prevents the producer lock and shared data structures from passing back and forth between processors (which causes expensive cache-to-cache transfers) too often.

To assign packet flows to threads properly, the producer must never allow packets from the same flow to be queued or processed at multiple consumer threads at the same time. The stream reassembly preprocessor (stream4) separates TCP sessions based on their IP addresses and TCP ports, and the flow preprocessor considers IP addresses, layer 4 protocol, and ports if applicable, meaning that any assignment scheme that satisfies flow's requirements will also satisfy stream4's.

Static hashes of flows to threads based on IP addresses and TCP/UDP ports are fast and simple, but are prone to uneven assignments and load imbalance if the flow distribution on the network is unfavorable. Instead, the producer assigns new flows to the currently least-busy consumer. Since the information needed for flow lookup and assignment is identical to that used by the flow preprocessor, this preprocessor is made part of the producer routine and its data structures are augmented to include information on flow-to-thread mapping. The flow preprocessor requires decoded packets, but only uses a few fields (IP addresses, ports, and protocol). Thus, the system must only run a small subset of the decode stage before the flow preprocessor.

Parallelization requires some other minor changes. For example, the notification interface is shared across threads, but the specific ordering of alert reports is not important. Conse-

quently, simple mutual exclusion suffices to protect this code (and leads to little serialization since notification accounts for less than 2% of time). The Snort code for stages such as multi-pattern matching assumes only one packet at a time and thus keeps only one common structure for all processing; these structures must now be associated with a specific thread or packet to make the code reentrant.

Deadlock avoidance. This flow-based parallelization cannot encounter deadlock because there is no cycle of dependences that cause stalling. The consumer routines do not affect each other, so they never stall for each other. Thus, any cycle would have to include the producer. A consumer can stall waiting to receive work from the producer (empty queue) or because its thread is currently serving as producer. The producer can only stall if it is trying to insert a work item into a consumer queue that is already full. If this consumer is not the same as the producer thread, it cannot be stalled since such a consumer will only stall for an empty queue. Thus, a cycle can only be formed if the producer is trying to insert into *its own* full consumer queue. The system avoids this case by causing the producer routine to return control to the consumer once its own work queue is sufficiently full.

4.2 Conservative Flow Reassignment

The assignment of flows to threads is critical, but the parallelization described above only considers the length of each thread queue when balancing load. In reality, some flows will require more inspection time than others, and some will end while others continue. The second parallelization scheme, conservative flow reassignment, targets situations where one thread has too many active flows while another thread has few or none. The scheme balances load by reassigning flows when safe.

Reassignment is safe when all the packets from a given flow drain out of the system and finish processing. Changing threads is not harmful in this case because there are no packets from the same flow being *simultaneously* processed by different threads. Changing the assignment here has the same effect as reaching a stream flush point in single-threaded Snort; in addition, neither the flush point selection (which is randomized) nor the flow reassignment (which is essentially random because it depends on the processing speed of Snort and the state of all flows in the network) can be predicted (and therefore exploited) by an attacker. Reassignment also has no impact on deadlock avoidance since it introduces no new stall conditions.

To implement reassignment, the producer increments a per-flow counter when a packet passes through the flow preprocessor, and the consumer decrements it when the packet finishes processing. This is essentially a reference count for each flow, and a flow with zero packet counts no longer needs a thread assignment. If the flow reappears, the producer will then assign it to the least-loaded consumer at the time of reappearance.

While improving load balance, reassignment introduces two complications. First, the stream4 session data remains in the stream table of the original thread, not the new one. This data will be flushed from the stream table after a timeout elapses, ensuring that all packets are inspected. Second, flow reassignment introduces locking overheads, since counters for each flow must be incremented and decremented concurrently.

4.3 Optimistic Flow Reassignment

The prime limitation of the flow-based parallelization is that it offers no opportunity for speedup on data streams with only a single network flow. As discussed in Section 3, data sharing between packets in the same flow stems from two key components: stream reassembly and flowbits. However, these subsystems have certain favorable properties that may enable a relaxation of the requirement. First, packets from the same flow that are separated by a stream reassembly flush point actually have no reassembly-related dependences between them since they will be reassembled into separate stream buffers. Second, only 36% of the rules actually test or set the flowbits used in flow-tracking (and more than 90% of these only apply to NetBIOS packets); rules that do not consider the flowbits have no dependences caused by flow-tracking. If few packets have content matches for these rules, there will be no dependence most of the time.

To allow intra-flow parallelization, the producer must be able to steer packets from the same flow to different consumers while also maintaining flowbits dependences when needed. Spreading a single flow across threads is only valuable when the base flow-concurrent version has a load imbalance. Consequently, the approach studied here starts with the flow-concurrent version and opts to reassign a flow to a different thread if the number of packets in the current thread's queue belonging to the flow are over a certain threshold, called the *reassignment threshold* (providing flow affinity to avoid problems in stream reassembly). The flow will then be reassigned to the least-loaded thread. The first packet after reassignment is then marked with a special flush point indicating that all previous packets from this flow should be reassembled and sent to inspection before attempting to process this packet in stream reassembly. This flush insures proper stream reassembly even when a flow is reassigned to a thread to which it has previously been assigned, making sure that the older packets are not reassembled with the newer ones.

Reassignment must not alter the behavior of flow tracking. However, it only needs to enforce flowbits dependences for packets that actually match rules that use flowbits. Detailed statistics show that only about 3% of the packets match flowbits rules for the traces shown in Figure 2 except DEF1. The new parallelization stalls the actual testing or setting of flowbits until the packet which has actually matched the rule is the oldest packet from that flow in the system. The system determines the oldest packet by maintaining per-flow reorder buffers, which are simply circular arrays of bits representing the completion state of packets in that flow. The flow preprocessor adds an incomplete entry bit to the tail of a flow's reorder buffer whenever it processes a packet. A packet's bit is marked complete when the verification stage completes. If the newly completed packet

is at the head of the circular array, the head pointer advances through as many complete entries as possible. Only the packet corresponding to the head of the circular array is allowed to test or set flowbits, but any packet that does not require flowbits may simply mark itself complete and then exit the system. (Unlike register renaming in superscalar processors, intrusion detection cannot use the reorder buffers to rename the flowbits because any given operation that sets flowbits only changes some of the bits. Consequently, such an operation must be considered both a read and a write, making renaming useless.)

This parallelization is optimistic because it assumes that intra-flow dependences will not be common. It then uses that assumption to assign flows to multiple threads. If the optimistic assumption is correct, packets from the same flow need not have any ordering imposed on them and will thus achieve intra-flow parallelism. If the optimistic assumption is incorrect, the system will stall until the dependences are met, providing correct detection of attacks.

Deadlock avoidance. Proving the optimistic parallelization deadlock-free is somewhat more complicated than the conservative version. In addition to the stall cases possible in the conservative case, one consumer may now wait for another as a result of a flowbits condition. Further, the producer may stall because it is trying to process a packet from a flow that has a full reorder buffer. Consumer-to-consumer stall cycles are avoided because each queue is processed in-order. Consequently, the oldest packet in the system is also the oldest in its flow and thread and thus will never have to wait for flowbits condition testing or setting. The processing of this packet can only stall if its consumer thread is currently acting as the producer. As in the conservative case, the producer will revert control to the consumer if its queue is sufficiently full. The only remaining case is if the producer thread is stalled waiting to insert an entry into a full reorder buffer. If the producer ever encounters a full reorder buffer, it does not immediately know if it is responsible for processing the oldest packet in the flow since the reorder buffer is nothing more than an array of bits. The producer avoids deadlock by checking each entry in its own queue to see if it is responsible for resolving the dependence. If so, the producer puts back the current packet (so that some later producer can handle it) and reverts control to the consumer. Although traversing its own queue is potentially an $O(N)$ operation, it is an extremely unlikely event; additionally, it takes place when the producer would already be stalled waiting for reorder-buffer space, so the cost is not a concern.

5 Experimental Methodology

The system studied here is based on Snort version 2.6RC1, downloadable from `www.snort.org`. A few modifications were made to snort that are independent of the parallelization strategy. The most important of these is the use of a large in-memory buffer from which to read the packets while processing, to minimize the system-dependent effects of reading directly from a file or network interface. In practice, this may be an important component of IDS performance, but separate solutions exist to address this problem, such as a version of `libpcap` that uses the `mmap()` system call to map a kernel ring buffer into Snort's address space, thus avoiding the overhead of copying packets to userspace. The measured runtimes for the tests do not include copying the packets from the trace file into the memory buffer, nor printing out statistics data after processing, but only cover reading the packets from the memory buffer, processing them, and generating alerts.

Snort is designed on a plugin architecture for almost all aspects of packet processing. Preprocessors, detection mechanisms, and notification methods are all based on modular plugins that may be mixed and matched according to the specifications of the user and rule writer. The system supports the stream4, flow, and HTTP Inspect preprocessor plugins, all the standard detection plugins (those that are not required to be explicitly enabled in the configuration file), and the "fast" alert method, which consists of writing a line to a text file for each alert generated. Packet logging was disabled. In practice, it is common for large installations to use an external database for collecting alert and log data, which may be on another machine. These and other methods can be supported by making their plugins reentrant. The multi-pattern matching algorithms are also abstracted from the rest of Snort's architecture; the modified Wu-Manber algorithm (the default up until version 2.6) is used in the parallel Snort.

As of Snort version 2.4, the rules and signatures are no longer distributed and released along with Snort itself. Instead, they are updated more often and may be downloaded separately. This paper uses the ruleset released on March 29, 2006. All rules are enabled except those marked as deleted or deprecated. In addition, many rules refer to a variable, such as "home net" or "HTTP servers" (for example, to check for patterns on streams that are only incoming or only bound for a user's web servers), which may be configured to refer to the user's own systems or network. In this study, however, these terms were set to "any" to catch all possible attacks. Other configuration variables exist to define which ports run particular services, and these were left at their defaults. The preprocessors' configurations were also left at their defaults.

Tests were run and analyzed two different types of machines: The first is a 2U Dell Poweredge 2950 server with two 1.8 GHz quad-core Intel Xeon E5320 processors based on the Core 2 microarchitecture (8 processors total). This system has 16 GB of system RAM and 4 MB L2 caches shared between pairs of processors. This system runs Linux kernel 2.6.18 and GNU C library 2.3.6 with the Native POSIX Threads Library in the Debian AMD64 distribution. The second is a 1U rack-mounted Sun Fire X4100 server with two 2.2 GHz dual-core AMD Opteron processors (4 processors total). This system has 4 GB of system RAM and 1 MB private L2 cache per processor core. The system is run using Linux kernel version 2.6.15 and the GNU C library 2.3.6 with the Native POSIX Threads Library in the same Debian AMD64 distribution. The Linux kernel supports affinity scheduling to maintain threads

Table 1. Packet traces used to evaluate the system and the throughput achieved by uniprocessor Snort (measured in Mbps)

Trace Name	Source	Date	Size (MB)	Packets	Alerts	Intel Rate	AMD Rate
LL1	Lincoln Lab	4/9/99	991	3,393,919	2,074	694	525
LL2	Lincoln Lab	4/08/99	740	3,201,382	3,567	469	369
LL3	Lincoln Lab	3/24/99	694	2,453,967	142	801	566
DEF1	DEFCON	7/14/01	687	3,960,264	127,672	434	345
DEF2	DEFCON	7/14/01	842	1,050,364	395	1094	933

on the same processor whenever possible. Instead of the standard pthread mutex locks, both parallelization methods use the pthread spin locks provided by the GNU C library. Unlike the standard mutex locks, these lock primitives do not suspend the calling thread when they encounter a lock that is already held by another thread; instead, they simply spin-wait until the lock is free. Because the system uses only as many threads as there are processors, the threads do not need to yield the processor, and critical sections are short enough that the overhead of invoking the operating system to block the thread is much higher than simply spinning until the lock is free. On x86 platforms, the spin locks are implemented using an atomic compare-and-swap instruction.

The packet traces used to test the system come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab and from the Defcon 9 Capture the Flag contest [10, 16]. The Lincoln Lab traces are simulations of large military networks generated during an online evaluation of IDSes and are available for download. Because they were generated specifically for IDS testing, (including anomaly-based detection systems, which require realistic traffic models to be useful) the traces have a good collection of ordinary-looking traffic content and also contain attacks that were known at the time. The traces used here are the largest available in the set, and come from the 1999 test. The Defcon traces are logs from a contest in which hackers attempt to attack and defend vulnerable systems. Consequently, these traces contain a huge amount of attacks and anomalous traffic, representing a sort of pathological case for intrusion detection systems. For example, DEF1 generates a very large number of alerts (even compared to the LL traces, which are seeded with real attacks). Table 1 shows a summary of the traces used, their source, their capture date, the number and total size of the packets they contain, the resulting number of alerts, and the throughput rates in Mbps achieved by uniprocessor Snort using the Intel-based server (averaging 699 Mbps) and the AMD-based server (averaging 548 Mbps).

6 Results and Discussion

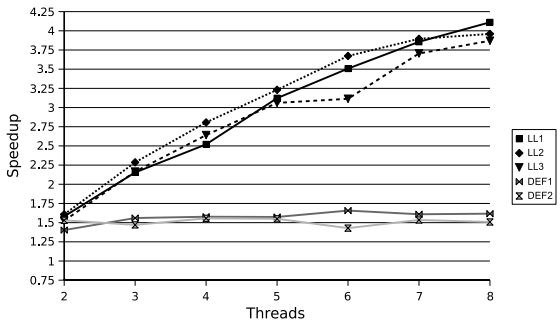
This section gives experimental results for the parallelization strategies, using the hardware platform and traces described in Section 5 and comparing to the performance results given in Table 1.

6.1 Flow-concurrent Parallelization

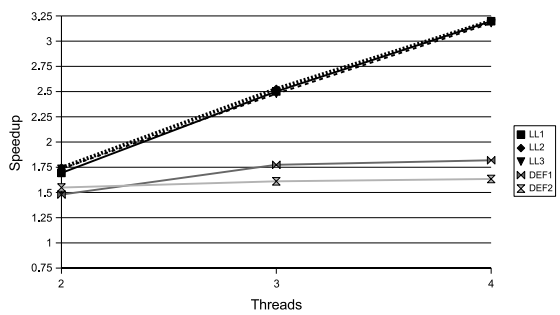
Figure 5 shows the parallel speedup achieved by the most conservative scheme on the Xeon-based Dell system and the Opteron-based Sun Fire system. Each line represents a packet trace, with parallel speedup plotted against number of threads. The plots show that the flow concurrent scheme achieves good speedup on the three LL traces but not on the DEFCON traces. The conservative parallelization sees an average throughput of 2.03 Gbps across the five traces for the Intel machine, and 1.36 Gbps for the AMD.

The three LL traces have similar speedup characteristics, achieving 63–79% of the theoretical ideal linear speedup for 2–4 threads, but decreasing to as low as 48% for 8 threads, with a peak speedup of 4.1 on the Intel machine. The AMD machine was slightly more efficient, achieving 79–87% of linear speedup for 2–4 threads with approximately 3.2 speedup at 4 threads. All 3 traces see processing rates in excess of 1 Gbps with 4 threads even on the slower AMD platform; LL1 and LL3 achieve this rate with 3. The peak processing rate is 3.1 Gbps for the Intel machine and 1.8 Gbps for the AMD machine. The two factors that limit performance in these cases are a small amount of imbalance (sometimes more than one thread ran out of work at the same time) and synchronization and data transfer overheads (primarily in the form of cache-to-cache transfers between processors). In particular, the work queues for each thread require transfers of the packet inspection descriptors from the producer to the consumers, as well as the descriptors for the queues themselves. Consequently, the amount of coherence traffic on the memory bus may become significant with larger numbers of processors. Although its absolute inspection rate is lower, the parallel speedup seen on the AMD machine is significantly better than that of the Intel machine — on average 20% better across the LL traces for 4 threads. This may be because the Opteron's HyperTransport interconnect allows lower-latency cache transfers than the Xeon's front-side bus.

In contrast, the DEFCON traces, and in particular DEF2, achieve little speedup in any case. As discussed previously,



(a) Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)



(b) Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

Figure 5. Parallel speedup for pure flow-concurrent parallelism on two systems

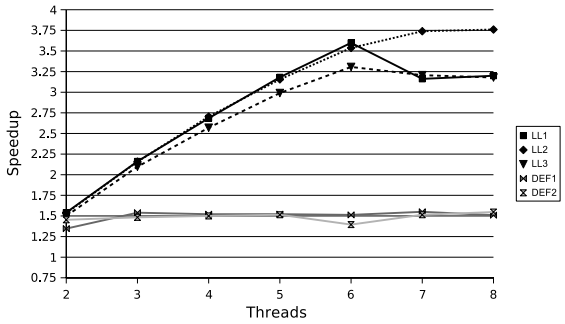
these workloads behave very differently from the others. DEF2 has extremely poor flow concurrency; most of the trace sees only one active flow, so no purely flow-based parallelization scheme can provide substantial speedup. DEF1 has several factors which contribute to poor performance. First, it triggers an extreme number of alerts; because alerts require synchronization, significantly more time is spent waiting for locks with DEF1. Second, DEF1 apparently contains attack attempts which create and abandon huge numbers of flows. This is the cause of the large preprocessing time seen in figure 2; in fact, for the single-threaded case, over 17% of the total time is spent in the flow preprocessor searching and updating the hash table containing the flows. This limits the speed of the producer routine, and thus the whole system. Lastly, despite the load on the flow preprocessor, DEF1 also has relatively poor flow concurrency, because the created flows are quickly abandoned and most of the actual traffic is concentrated in a relatively small number of flows. These last two problems exacerbate each other, because threads acting as consumers are more likely to run out of data when the producer is slower, further causing these threads to compete for the lock which protects the producer routing. Of course, the source and nature of the DEFCON workload mean that diminished performance is to be expected; it reflects a very small network and an extremely adversarial environment where nearly all traffic is malicious. A system that detects such a high rate of alerts may respond quickly by more aggressive firewalling to shut off traffic on vulnerable ports or from IP addresses observed to participate in malicious behavior. The other traces are actually more likely to be dangerous since they have a small number of attacks hidden in a larger amount of "normal" traffic. Consequently, such workloads are more important and realistic for IDS testing, and the conservative parallelization performs quite well on 3 of those 4 workloads.

6.2 Conservative flow reassignment

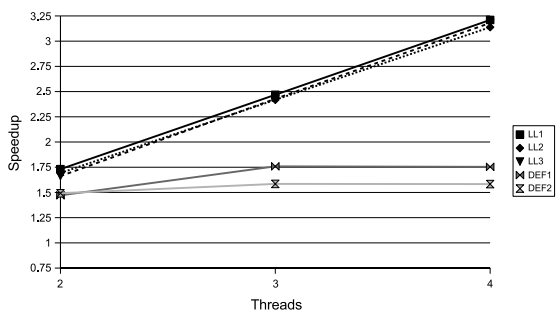
Figure 6 shows the parallel speedup achieved by the Dell and Sun servers with conservative flow reassignment, which should improve load balance without affecting the amount of packet inspection work. Figure 6(b) is nearly identical to Figure 5(b), indicating no improvement on the AMD platform. Moreover, Figure 6(a) indicates that in most cases adding conservative reassignment actually reduces performance on the Intel platform. With 7 or 8 threads, flow reassignment degrades complexity by as much as 22% and sometimes underperforms 6 threads. These effects arise from the bookkeeping required for reassignment. For each packet, the producer must acquire the lock on the flow data structure to which the packet belongs, make a new assignment if necessary, and increment the flow's reference count. The consumer must likewise acquire the lock, decrement the reference count, and remove the flow's thread assignment if its reference count has reached zero. This results in more synchronization overhead, remote cache transfers, and coherence traffic for every packet; combined with the overheads discussed in Section 6.1, the cost of this strategy is enough to offset any benefits gained by more flexible flow assignment.

6.3 Optimistic flow reassignment

DEF2, as mentioned, has poor flow concurrency, and is thus a good candidate for improvement using optimistic flow reassignment. Figure 7 shows the parallel speedup with optimistic reassignment for the Intel and AMD platforms, with a reassignment threshold of 100 (about 10% of the queue length). DEF2 indeed shows benefits over the conservative method, improving performance on the Intel machine by as much as 68% for 6 threads, with a 2.4 speedup and a 2.63 Gbps rate. Likewise the performance on the AMD machine improved by about 45% for 4 threads to achieve a factor of 2.38 parallel speedup and a peak traffic rate over 2.2 Gbps. The value chosen for the reassignment threshold should be small enough so that when a packet matches a flowbit rule and must wait for previous packets, it does not have to wait too long (since the number of packets ahead of it can be no more than the threshold multiplied by the number of other queues). However it must be large enough to avoid excessive switching (which causes too much flushing

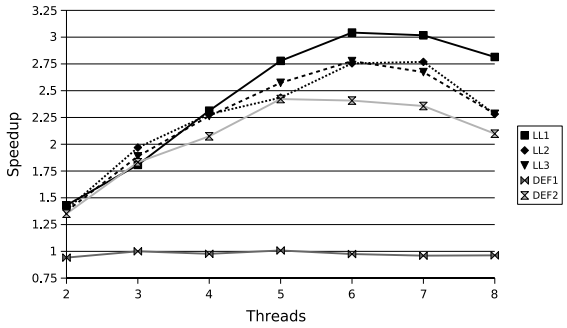


(a) Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)

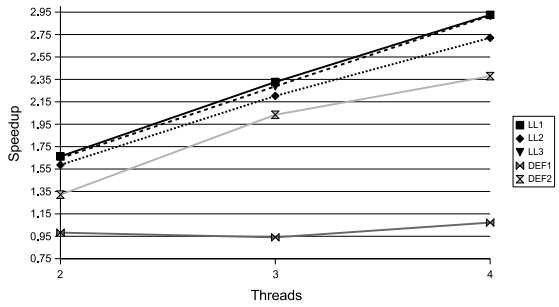


(b) Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

Figure 6. Parallel speedup for conservative flow reassignment on two systems



(a) Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)



(b) Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

Figure 7. Parallel speedup for optimistic flow reassignment on two systems

and other overhead in stream reassembly). In practice, 100 is a good balance.

Since the LL traces already have good flow concurrency, optimistic flow reassignment provides no benefit; in fact, their performance is degraded by 26.5% on average because of the overhead of maintaining the reorder buffers. In addition to the overhead of the conservative flow reassignment, updating and checking the bits in the buffer and the head and tail pointers must be done even when packets in a flow are serialized, and this must be done while holding the mutex lock associated with the flow, leading to additional synchronization overhead. DEF1 also does not benefit from reassignment because most of its packets actually set or check flowbits. Thus, any advantage gained by reassigning flows is erased because so many flows must serialize themselves. Further, since the serialized flows are spread across all the threads, they even block flows behind them that might otherwise have been able to pass them. Consequently, the overall rate in DEF1 is reduced to approximately that of the single-threaded case.

As with the non-reassignment scheme, the AMD machine gets better parallel speedups in both reassignment schemes than the Intel machine for the same number of threads. In fact the advantage increases to almost 26% on average for 4 threads with optimistic reassignment. As synchronization overhead and read-write sharing of data increase with the addition of conservative and then optimistic reassignment, so does the parallel speedup advantage of the AMD system over the Intel system, providing more evidence that the AMD interconnect handles this sharing better.

Because overheads increase with the number of processors, the throughput achieved peaks at 5 or 6 processors and then decreases as more are added. Despite the degradations in some of the traces, the average of the best traffic rates achieved for each trace using the optimistic reassignment is roughly the same as that of conservative reassignment at 1.74 Gbps, but does not reach the 2.03 Gbps average seen by the non-reassignment scheme. The optimistic parallelization sees good parallel speedup for 4 out of 5 traces, though these are somewhat lower than the conservative version for 3 traces.

6.4 Discussion

The results in this section indicate substantial benefits from parallelization in the Snort NIDS. For most realistic scenarios with many simultaneous packet flows, conservative flow-based parallelism without reassignment is sufficient. Conservative reassignment provided no scalability benefits because of the over-

head of maintaining per-flow counters. Networks with poor flow concurrency can see benefits from optimistic reassignment of flows, provided that the number of packets that must check flowbits is limited. All of these parallelization strategies use multicore hardware that is increasingly becoming commoditized, allowing for fast single-node edge-based NIDS. As architectures continue to evolve, all expectations are for more multicore and multiprocessor solutions and less potential benefit from ramping up clock frequency. Thus it is essential for an application as important as NIDS to achieve its performance by exploiting fine-grained flow-level and intra-flow parallelism.

The effectiveness of optimistic flow reassignment may vary substantially with the conditions used for reassignment. Additional modifications were tested that would prevent reassignment if the queue length at the reassignment target were at least a certain percentage of the queue length of the origin (indicating some load balance factor), but this conditionhad had little impact on performance as this threshold was varied from 10–50%. Further modifications seemed promising, but had little or no impact (for example, disabling reassignment on NetBIOS flows as these account for most flowbits rules, or moving the flowbits tests to the end of rule processing so that they would only be checked if all other conditions matched).

The optimistic system is also still conservative in how it manages rules that actually use flowbits since it stalls when a flowbit is to be set or checked out-of-order. An alternative would be to speculatively perform the flowbit operation and then roll back if there was a violation, but such rollbacks would require the reprocessing of many packets and a great deal of stored state.

7 Related Work

Clustered intrusion-detection systems achieve performance using multiple low-cost, identically-configured and administered PCs along with a load-balancing switch. Schaelicke et al. proposed SPANIDS, a system that combines a specially-designed FPGA-based load-balancing switch that considers flow information and system load when redirecting packets to commodity PCs that run intrusion detection software [15]. Commercial offerings by F5 and Radware use the companies' L4–7 load-balancing switches to redirect traffic to a pool of intrusion-detection nodes, allowing high overall throughput scalability [8, 13]. In contrast, parallelized Snort exploits trends toward multicore processing to achieve good performance for small to mid-scale deployments without the expense of a load-balancing switch. For larger deployments, parallelized Snort could be used in a clustered IDS with greater per-system performance and higher space-efficiency.

The research community has also proposed distributed NIDS, in which nodes at various points in the network track anomalies and collaboratively collect data that may indicate a system-level intrusion even if no specific host triggers an alert [17, 9]. Efforts in distributed NIDS have invariably targeted gathering additional information to *identify* intrusions, rather than processing packets at a faster rate. Thus, distributed NIDS is largely orthogonal of the parallel processing approach.

Because matching multiple simultaneous strings is such an important component of intrusion detection and can potentially exploit extensive hardware concurrency, several works have proposed hardware support for this stage (and, in some cases, regular expression processing) using FPGAs, ASICs, and TCAMs [2, 3, 6, 11, 18, 19, 22]. Such works are valuable but do not solve the entire NIDS problem. Figure 2 shows an average of 48% of processing time for multi-string and 4% for regular expression matching. By Amdahl's Law, even infinite hardware acceleration for these stages would lead to a limit of 2.1 speedup for the entire NIDS application. The software approach to parallelizing the various stages of intrusion detection should work synergistically with hardware that speeds up string matching and regular expression matching, increasing the effectiveness of both.

There has also been some investigation on running intrusion detection software on network processors: Bos and Huang implement a rudimentary IDS using the Intel IXP network processor architecture and its parallel microengine processor cores to perform Aho-Corasick string matching, stream reconstruction, and I/O operations [4]. Vermeiren et al. propose several strategies for a multithreaded Snort with the aim of running it on high-end network microprocessors [20] such as the Broadcom BCM1250 [5]. That work does not discuss any solutions for maintaining dependences across the stages of Snort processing or for insuring that packets from the same flow are processed in an appropriate order.

Other intrusion detection software systems also exist, such as the Bro IDS from Lawrence Berkeley National Labs [12]. Bro rules can detect all standard Snort traffic signatures as well as anomalies such as an excessive number of connections. This paper chooses a Snort-based system primarily because of its popularity and greater update frequency. Despite the differences among systems, the problem of and need for fine-grained parallelism applies to all NIDS software, and the fundamental challenges and solutions discussed here apply to any system that employs stream reassembly and flow tracking to provide stateful ruleset processing.

8 Conclusions

This paper presents and evaluates two conservative and one optimistic parallelization strategies for network intrusion detection. Although this paper specifically targets Snort, the challenges and solutions described here apply to any NIDS that performs stream reassembly and flow-tracking. Each parallelization schemes has its limitations, but each performs well for most of the workloads that it targets. The most conservative flow-concurrent parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 4.1 speedup on 8 processor cores and processing at speeds up to 3.1 Gbps. The extra overheads in the optimistic parallelization degrade performance by about 26% for the traces

that exhibit flow concurrency, limiting speedup to 3.0 on six cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.4 on five cores), with a peak traffic rate over 2.6 Gbps. The most conservative scheme achieves an average traffic rate of 2.03 Gbps for the 5 traces, and both reassignment-based schemes average over 1.7 Gbps, providing major improvements in intrusion detection performance using hardware that is cost-effective, space-efficient, and increasingly being commoditized.

References

[1] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.

[2] M. Aldwairi, T. Conte, and P. Franzon. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Comput. Archit. News*, 33(1):99–107, 2005.

[3] Z. K. Baker and V. K. Prasanna. A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, Apr. 2004.

[4] H. Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. *Recent Advances in Intrusion Detection. 8th International Symposium, RAID 2005. Revised Papers (Lecture Notes in Computer Science Vol. 3858)*, pages 102 – 23, 2005.

[5] Broadcom. *BCM1250 Product Brief*, 2006.

[6] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.

[7] DARPA Internet Program Protocol Specification. Transmission Control Protocol. IETF RFC 793, Sept. 1981.

[8] F5 Networks. Securing the Enterprise Perimeter – Using F5's BIG-IP System to Provide Comprehensive Application and Network Security. White paper, Oct. 2004.

[9] R. Gopalakrishna and E. H. Spafford. A Framework for Distributed Intrusion Detection using Interest Driven Cooperating Agents. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, Oct. 2001.

[10] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. Technical Report 1062, MIT Lincoln Laboratory, 2001.

[11] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 31–38, Apr. 2003.

[12] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, Dec. 1999.

[13] Radware Inc. FireProof Security Activation. White paper, Sept. 2004.

[14] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238, 1999.

[15] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 315–322, 2005.

[16] Shmoo Group. Defcon 9 Capture the Flag Data, Sept. 2001.

[17] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype. In *Proceedings of the 14th National Computer Security Conference*, pages 167–176, Washington, DC, Oct. 1991.

[18] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pages 880–889, Sept. 2003.

[19] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 112–122, June 2005.

[20] T. Vermeiren, E. Borghs, and B. Haagdorens. Evaluation of software techniques for parallel packet processing on multi-core processors. *IEEE Consumer Communications and Networking Conference, CCNC*, pages 645 – 647, 2004.

[21] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[22] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proc. of the 12th IEEE International Conference on Network Protocols*, pages 174–183, Oct. 2004.