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 evaluates two parallelization strategies for the popular open-source Snort network intrusion detection system (NIDS). Snort identifies intrusion attempts by processing a ruleset, a file which specifies various protocol-based, string-based, and regular-expression-based signatures associated with known attacks. As attacks proliferate, NIDS becomes increasingly important. However, the computational requirements of intrusion detection are great enough to limit average achievable throughput to 557 Mbps on a commodity server-class PC — just over half the link-level bandwidth. The strategies studied in this paper accelerate the performance of Snort by parallelizing rule processing while still maintaining the shared state information required for correct operation.

The conservative version proposed here parallelizes ruleset processing at the level of TCP/IP flows, as any *potential* inter-packet dependences are confined to a single flow. Any single flow is processed in-order at one thread, but the flows are partitioned among threads. This solution provides good performance for 3 of the 5 network packet traces studied, reaching as high as 3.0 speedup and 1.7 Gbps inspection rate when implemented on x86-64 Linux for a server with two dual-core Opteron processors (four cores total). Conservative parallelization allows an average inspection rate of 1.07 Gbps across all 5 traces – nearly twice the serial performance. However, it is too restrictive to achieve good performance if there are not enough concurrent flows in the traffic stream. To handle this case, an optimistic version of Snort is also designed that exploits the observation that not all packets from a flow are actually connected by dependence orders (although these dependences cannot be discovered until deep in packet inspection). The optimistic version thus allows a single flow to be simultaneously processed by multiple threads, stalling processing only if an *actual* dependence is found. The optimistic version has additional overheads that reduce speedup by 7–13% for traces that have flow concurrency. However, the benefits of the optimistic appproach allow one additional trace to see substantial speedup (2.2 on four cores). The average inspection rate stays nearly unchanged at 1.09 Gbps, but the peak increases to over 2 Gbps. Consequently, this may be a good option for protecting systems and networks with few flows.

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

1. INTRODUCTION

Internet-based security attacks have proliferated in recent years, with buffer overruns, cross-site scripting, and denial-of-service among the most prominent and damaging forms of attack (commonly called *exploits*). A successful attacker can then initiate follow-on exploits, such as compromising the local system to gain administrator-level access or initiating denial of service attacks against other machines. The most popular operating systems regularly publish security updates, but the combination of poorly-administered machines, uninformed users, a vast number of targets, and ever-present software bugs has allowed exploits to remain ahead of patches.

Network intrusion detection systems (NIDSes) run on a server at the edge of a LAN to identify and log Internet-based attacks against a local network. Unlike firewalls, which work by shutting off external access to certain ports, NIDSes can monitor attacks on externally-exposed ports used for running 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. Each new type of attack leads to new rules, with rulesets growing rapidly. The most recently-released freely-available Snort rulesets have over 4000 rules.

The processing required by a network intrusion detection system such as Snort is quite high, since the system must decode the data, inspect the data according to the ruleset, and log intrusions. These requirements limit Snort to an average packet processing rate of about 557 Mbps on a modern host machine (2.2 GHz Opteron processor) — just over half of the Gigabit link-level bandwidth. Consequently, it is not possible to deploy Snort directly at a high-end network access point that requires a data rate of 1 Gbps or more. To address this problem, various companies and researchers have proposed solutions based on clustering [8, 13, 15]. Clustered NIDS potentially allows high scalability, but requires the use of an expensive load-balancing switch.

This paper presents and evaluates methods to parallelize Snort for PC-based systems with modern multiprocessor architectures. Although an NIDS like Snort receives its input on a packet-by-packet basis, an NIDS must seek to aggregate distinct packets into TCP streams to prevent an attacker from disguising malicious communications by breaking the data up across several packets. Additionally, an NIDS must process later packets in a given commu-

.

nication based on decisions made when analyzing earlier 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). Although these phases require packets to be processed in-order, a key observation is that any ordering or data sharing between the processing of separate 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). Although this paper specifically targets the Snort NIDS, the parallelization challenges and strategies discussed here apply to any intrusion detection system that uses TCP stream reassembly to merge packets together for inspection or preserves other state across different packets from the same flow.

The parallelization strategies studied in this paper take different approaches to splitting the Snort NIDS into threads: one conservative and one optimistic. The conservative scheme, called the *flow-concurrent* parallelization, exploits concurrency by parallelizing ruleset processing on a flow-by-flow basis. All packets are initially received by one thread. That thread inspects the IP headers to determine the flow to which the packet belongs and then steers that packet to the appropriate processing 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. This scheme works well if there are enough independent flows, but provides no benefits if all packets are from the same flow. The latter case is not a likely situation in a high-bandwidth edge NIDS, but does represent a limitation of this scheme.

The alternative parallelization is an optimistic variant on flow concurrency. This scheme starts with the basic flow-concurrent parallelization but then has the ability to 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. 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 requiring rollbacks and redundant processing.

Both 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 on a 1U rack-mounted x86-64 Linux system with two dual-core Opteron processors (four processor cores in total). The conservative parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 3.0 speedup on 4 processor cores and processing at speeds up to 1.7 Gbps. The extra overheads in the optimistic parallelization degrade performance by about 10% for the traces that exhibit flow concurrency, limiting speedup to 2.8 on four cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.2 on four cores), with a peak traffic rate over 2 Gbps. Both schemes see an average traffic rate of just over 1 Gbps for the 5 traces, nearly doubling the performance of the serial version with only a slight increase in hardware cost and no increase in space. Either parallelization allows the benefits of high-performance intrusion-detection without relying either on higher clock frequencies (which are reaching a stage of diminishing returns) or costly and space-consuming load balancers.

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 and allows rules to take them into consideration for rule matches (for example, only match the rule if the packet is part of an established TCP connection). Stream4 also performs stream reassembly, taking multiple scanned packets from a given direction of a stream and builds a single conceptual packet by concatenating their payloads, allowing rules to match packet content that spans packet boundaries. This is one of the most important preprocessors because without it, an attacker can trivially hide attacks by simply splitting them across more than one packet. Packets are reassembled into stream buffers and sent into the inspection process after a “flush point” is reached. Flush points are 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 regularly updated over the past 5 years, quadrupling in size from approximately 1000 rules in 2001 to over 4000 in March of 2006, and indicating a roughly constant rate of increase of new attack signatures over that period of time. Although the performance of the multi-string content matching does not depend directly on the number of rules, a greater number of rules does require a greater amount of memory and may require more time to be spent in the stages of intrusion detection other than content matching. Over 86% of the rules are for TCP-based ex-

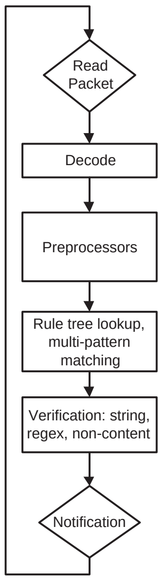
Read Packet
Decode
Preprocessors
Rule tree lookup, multi-pattern matching
Verification: string, regex, non-content
Notification

Figure 1: The Snort packet processing loop

ploits (including over 30% for HTTP rules), with UDP at about 9%, ICMP at 4%, and other IP rules at just over 1%. 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. Most rules specify several conditions on packet content, and all of them must be met for a match to occur. The exact string match (implemented by the multi-pattern matching algorithm) is used as a first-order filter; tests for a specific rule are not performed unless its corresponding pattern is matched first. 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:[ˆ\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 the packet processing loop used by Snort. 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 associated information, storing the results in Snort's loosely-encoded packet data structure. Snort then invokes the preprocessors that use and manipulate packet data in various ways. The rule-tree lookup and pattern matching stage determines which rules are relevant for the packet at hand (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 notifies the system owner through alerts related to the specific rule matches.

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 it 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 4.

The profiles shown here were gathered using the `oprofile` full-system profiling utility running on a Sun Fire X4100 with two dual-core Opterons (4 processors total), but Snort only runs on 1 processor. The system has 4 GB of DRAM and uses Linux version 2.6.16. 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 557 Mbps for these traces with a peak of 951 Mbps.

As Figure 2 shows, string content matching is a very important component of intrusion detection, ranging from 20–70% of the execution time of the system with an average of 38%. For all traces except DEF1, the combination of ruleset processing components (string match, verification, and regular expression) make up over 56% 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 43% of execution time.

3. PARALLELIZATION ALTERNATIVES

Parallelizing any application requires first identifying the available concurrency. As discussed in Section 2, the main loop of the Snort NIDS works on one packet at a time. Consequently, packet-level concurrency seems a natural granularity for parallelizing this application. However, several problems need to be addressed for parallelism to be feasible.

3.1 Packet-level Parallelization

Figure 3 illustrates the interactions between the processing of two separate packets in the Snort NIDS, following the basic Snort processing loop depicted in Figure 1. 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.

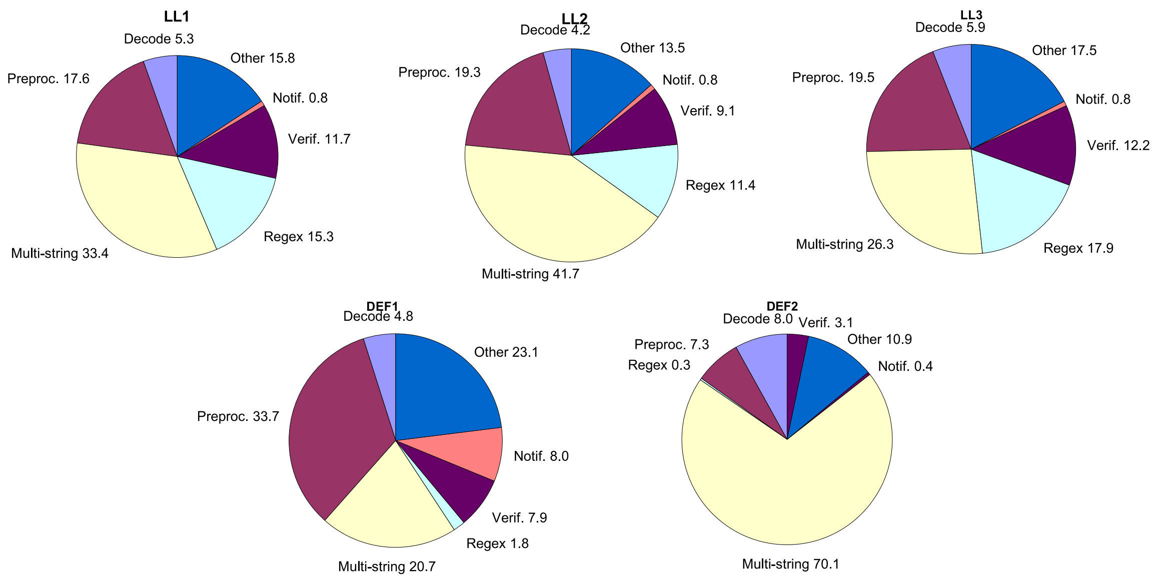

Figure 2: Execution time of Snort categorized into principal components when run using various traces on an Opteron-based system

For packet-level parallelization to be practical, no resource should actually share common information across separate packets. However, the data structures used by the flow and stream reassembly preprocessors must be shared by different packets. 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 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 to decide if it even needs assembly — streams that have not yet completed connection establishment should only be processed packet-by-packet. The reassembly list must be searched and updated on each packet, and the TCP stream state must again be checked in verification to decide if a rule applies. Processing packets completely independently could allow packets to arrive out-of-order at stream reassembly, disrupting the tracking of TCP connection states and possibly causing the system to miss an attack. However, maintaining proper ordering on shared data structures would require multiple expensive synchronization operations on every packet.

The notification stage must also share a common notification interface whenever an alert actually arises. However, the specific ordering of alert reports is not important, so simple mutual exclusion will suffice to enable the parallelization.

Other minor changes are also required for parallelization. For example, stages such as multi-pattern matching currently assume only one packet at a time and thus keep only one common structure for all processing; these structures must now be associated with a specific thread or packet to make the code reentrant.

3.2 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.

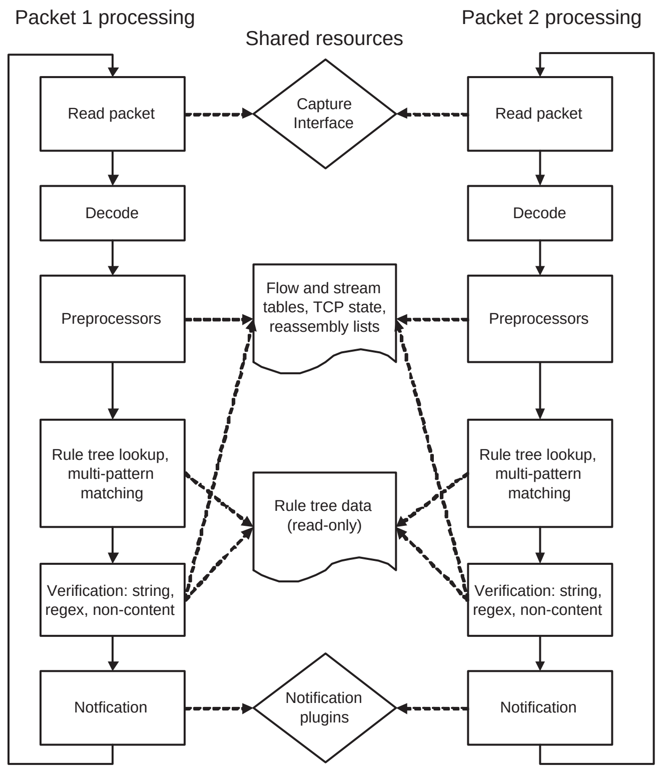
Packet 1 processing
Shared resources
Packet 2 processing
Read packet
Capture Interface
Read packet
Decode
Decode
Preprocessors
Flow and stream tables, TCP state, reassembly lists
Preprocessors
Rule tree lookup, multi-pattern matching
Rule tree data (read-only)
Rule tree lookup, multi-pattern matching
Verification: string, regex, non-content
Verification: string, regex, non-content
Notfication
Notification plugins
Notification

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

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 dependence constraints between packets in a flow. Note that this is actually conservative because a dependence might not actually exist between successive packets in a flow. For example, if the stream reassembly process encounters a flush point between those packets, they will actually be reassembled into separate stream buffers. Similarly, if a packet only matches rules that do not use flowbits, it is irrelevant whether earlier packets have set any flowbits. However, these conditions cannot be detected until the system is already deep into packet inspection. Consequently, this parallelization chooses to err on the side of caution by enforcing an ordering between packets in the same flow.

Note that the effectiveness of flow-level parallelization depends on the existence of multiple concurrent flows in the network stream. If there is only one flow on the network, then it can only be inspected by one thread. However, because NIDS sensors usually protect many machines or an entire network, it seems reasonable to assume that many flows will be present at a time, particularly if a large amount of bandwidth is being consumed.

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 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 becomes the producer, and begins reading packets and assigning them to their proper work queues. These work queues can be quite large since each entry only includes 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. If a second thread's work queue empties and it tries to become the producer simultaneously, it must waste time waiting with nothing to do. To reduce the likelihood of such wasted time, the producer routine uses a lower threshold to start consuming its own data if another thread also has no work queue items. This minor modification reduces the amount of time that the other thread must spend waiting to either receive tasks or become the producer.

Completing the process of flow-level parallelization requires a policy for assigning packet flows to threads. At a minimum, the system 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 require-

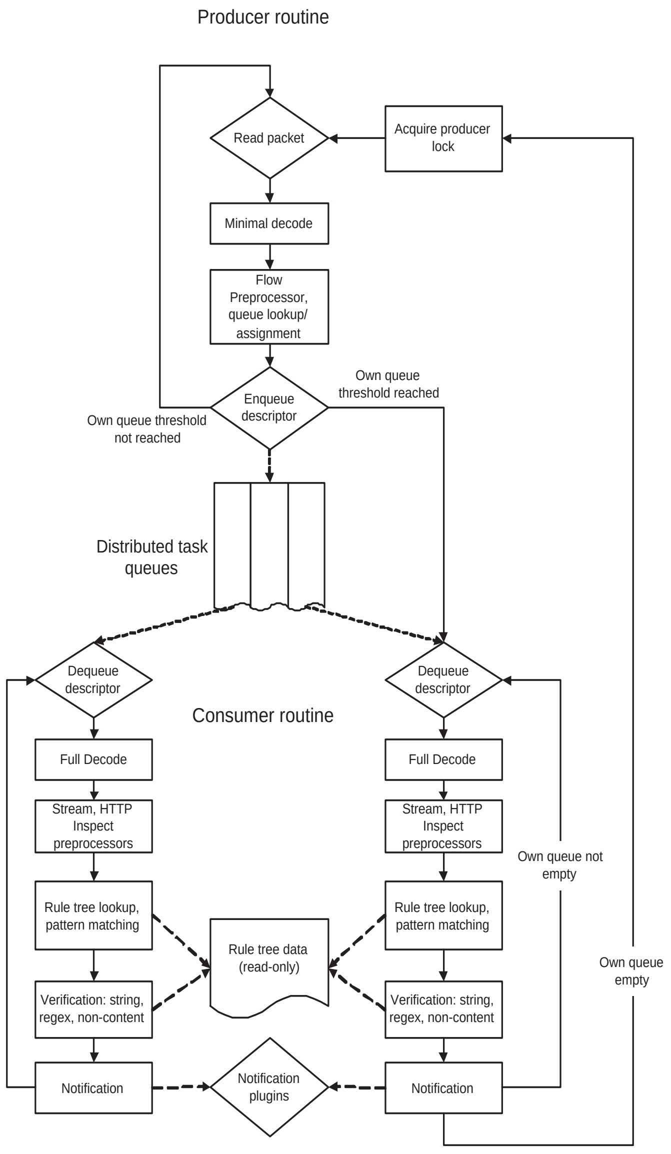
Producer routine
Read packet
Acquire producer lock
Minimal decode
Flow Preprocessor, queue lookup/ assignment
Enqueue descriptor
Own queue threshold reached
Own queue threshold not reached
Distributed task queues
Dequeue descriptor
Dequeue descriptor
Consumer routine
Full Decode
Full Decode
Stream, HTTP Inspect preprocessors
Stream, HTTP Inspect preprocessors
Own queue not empty
Rule tree lookup, pattern matching
Rule tree data (read-only)
Rule tree lookup, pattern matching
Own queue empty
Verification: string, regex, non-content
Verification: string, regex, non-content
Notification
Notification plugins
Notification

Figure 4: Flow-level parallelization strategy for Snort

ments will also satisfy stream4's. The thread assignment scheme must aim to avoid load imbalance. Static hashes based on IP addresses are fast and simple, but are prone to uneven assignments if the flow distribution on the network is unfavorable. A better approach is to dynamically assign flows to threads, based on information available at runtime.

When a new flow arrives at the system, it is assigned to the thread with the shortest work queue and this assignment is entered into a table. The producer routine must steer later packets from the same flow to the appropriate thread. The producer must determine the packet's flow information, consult a hash table to determine if the flow has an existing assignment, or create an assignment otherwise. Conveniently, these are exactly the actions taken by the flow preprocessor to allow its tracking of flowbits. So, the flow preprocessor is simply run before thread assignment, assigning new flows to the thread that is currently least busy. The flow tracking structure is augmented to include the thread assignment for the flow. Flows are identified using the IP addresses of the hosts and the TCP/UDP port numbers, allowing different flows between the same two hosts to be separated. Using this data requires the decoding of the IP and TCP/UDP headers. However, this is the only part of the overall decode stage that is required. Consequently, only a minimal subset of decode is peformed before the flow preprocessor, and a full decode is performed by the consumers in parallel. The flow preprocessor itself need not be run again since a pointer to the flow information is passed in the queue along with the pointer to the packet data.

If all the packets from a given flow drain out of the system, this flow may be reassigned to a different thread. Changing threads is not harmful in this case because there are no packets from the same flow being *simultaneously* processed by different threads. This reassignment is implemented by counting the packets in the system for each flow, incrementing the counter when the packet passes through flow preprocessor and decrementing it when the packet finishes processing. When a flow's packet count reaches zero, its assignment is removed, and it is free to be reassigned to the least-busy thread if the flow reappears. This additional flexibility improves performance slightly by improving the flow workload balance. However, it introduces one complication: the stream4 session data remains in the stream table of the original thread, not the new one. However, this data will be flushed from the stream table after a timeout elapses.

3.3 Optimistic Parallelization

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 3.1, 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 (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; the flowbits state is irrelevant for other packets. Detailed statistics show that only about 3% of the packets match flowbits rules for the all of 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 to the tail of a flow's reorder buffer whenever it processes a packet. A packet's entry 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 reassign flows to different 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.

Claim: Despite sometimes stalling for a packet to reach the head of its flow, the above system avoids deadlock.

Proof: Each consumer queue is processed in-order. Consequently, either the oldest element still under inspection for the entire system is being processed by its consumer thread or that thread is currently acting as the producer. The producer for the optimistic version is carefully designed never to wait on a consumer thread; if the consumer to which its current packet is destined already has a full queue, the producer will automatically reassign that flow to another consumer. Even if all other consumers have full queues, the current producer thread must still have space in its consumer queue since it would have already switched back to being a consumer if its queue was full. Once the thread responsible for the oldest packet is a consumer, it must process the oldest element since allowing it to wait in a queue further would imply a still older element that requires processing (since consumer queues are processed in-order). Even if the oldest packet tests or sets flowbits, the algorithm specified above would never stall its processing (since it is obviously the oldest element in its flow). Consequently, this packet will never have to stall for any other, guaranteeing that it will achieve forward progress and that there will be no deadlocks.

Although the optimistic parallelization would allow for concurrency within a single flow, its performance rests upon the idea that most rule matches do not use flowbits. If this were not actually true, the amount of stalling in this system could be prohibitive.

4. 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

Table 1: Packet traces used to evaluate the system

Trace Name	Source	Date	Size (MB)	Packets	Alerts
LL1	Lincoln Lab	4/9/99	991	3,393,919	2,382
LL2	Lincoln Lab	4/08/99	740	3,201,382	3,693
LL3	Lincoln Lab	3/24/99	694	2,453,967	186
DEF1	DEFCON	7/14/01	687	3,960,264	128,897
DEF2	DEFCON	7/14/01	842	1,050,364	396

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 on a 1U rack-mounted Sun Fire X4100 server with two 2.2 GHz dual-core Opteron processors (4 processors total). The system has 4 GB of system RAM and 1 MB 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 Debian AMD64 distribution. The Linux kernel supports affinity scheduling to maintain threads 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.

Table 2: Uniprocessor performance levels achieved by Snort for traces and hardware platform described in Section 4.

Trace Name	Throughput ($\frac{packets}{sec}$)	Throughput (Mbps)
LL1	220,202	539
LL2	196,184	380
LL3	239,670	568
DEF1	239,267	348
DEF2	141,375	951

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, and the resulting number of alerts.

5. RESULTS AND DISCUSSION

This section gives experimental results for the parallelization strategies, using the hardware platform and traces described in Section 4. Table 2 gives the throughput achieved by the standard uniprocessor Snort for each of the traces; the results for parallel speedup are relative to these performance levels. The average throughput of the uniprocessor Snort is 557 Mbps.

5.1 Flow-concurrent Parallelization

Figure 5 shows the parallel speedup achieved by the conservative scheme on the Opteron-based Sun Fire system. Each group of bars represents one of the packet traces, while the bars themselves show the performance running with 2, 3, and 4 threads. The bars 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 traffic rate of 1.07 Gbps across the five traces.

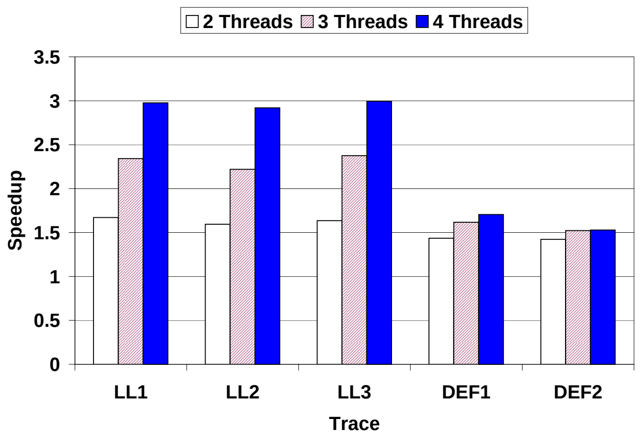

Figure 5: Parallel speedup for conservative parallelism on Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

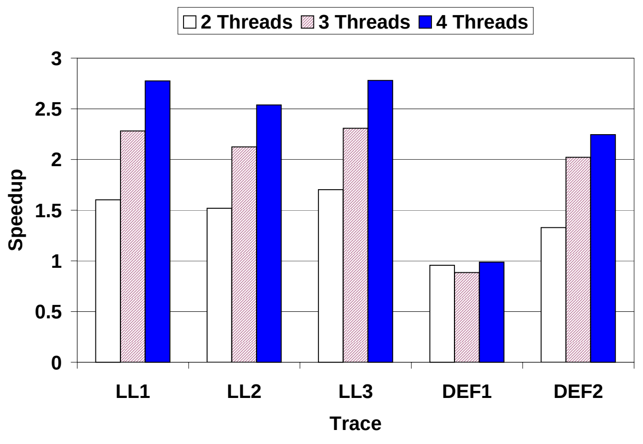

Figure 6: Parallel speedup for optimistic parallelism on Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

The three LL traces have similar speedup characteristics, achieving 73–83% of the theoretical ideal linear speedup for 2–4 threads and achieving 2.9–3.0 speedup at 4 threads. All 3 traces see processing rates in excess of 1 Gbps with 4 threads; LL1 and LL3 achieve this rate with 3. The peak processing rate is 1.7 Gbps. The two factors that limit performance in these cases are a small amount of imbalance (occasionally 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 contrast, the DEFCON traces, and in particular DEF2, achieves little speedup in any case. As discussed previously, the DEF1 and DEF2 workloads behave very differently from the others. As it turns out, DEF2 has extremely poor flow concurrency; in fact, for much of the trace there is only one active flow, so no flow-based parallelization scheme can hope for any significant improvement. DEF1 has several factors which contribute to poor performance. First, it triggers an extreme number of alerts, and 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. These effects also combine to greatly increase the lock and cache-transfer overhead because multiple threads with empty queues may compete for the lock which protects the producer routine, and the data used by the producer routine is transferred frequently. 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 require extreme vigilance to prevent compromise, and the conservative parallelization performs quite well on 3 of those 4 workloads.

5.2 Optimistic flow reassignment

DEF2, as mentioned, has poor flow concurrency, and is thus a good candidate for improvement using the optimistic flow reassignment method. Figure 6 shows the performance using the optimistic reassignment method, with a reassignment threshold of 100 (about 10% of the queue length). DEF2 indeed shows benefits over the conservative method, improving performance by about 50% for 4 threads to achieve a factor of 2.2 parallel speedup and a peak traffic rate over 2 Gbps. The value chosen for the 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 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 7–13% compared to the pure flow-concurrent method because of the overhead of maintaining the reorder buffers. In particular, 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 when compared to the conservative model. DEF1 also does not benefit from reassignment because it matches so many rules, and most of the packets actually trigger setting or checking of 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 slightly less than that of the single-threaded case.

Despite the degradations in processing some of the traces, the average traffic rate of the optimistic parallelization is roughly the same as the conservative at 1.09 Gbps. The peak rate is actually much higher, at over 2 Gbps. 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.

5.3 Discussion

The results in this section indicate substantial benefits from parallelization in the Snort network intrusion detection system. For most realistic scenarios with many simultaneous packet flows, conservative flow-based parallelism is sufficient. 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. This parallelization is achieved while using hardware that is increasingly becoming commoditized, allowing for fast single-node edge-based network intrusion detection. 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 network intrusion detection to achieve its performance by exploiting fine-grained flow-level and intra-flow parallelism.

The flow reassignment scheme considered here is quite simple, and variations on it may improve its responsiveness or limit its overhead. For example, 90% of the flowbits rules are related to NetBIOS, so it may be reasonable to prevent a flow on a NetBIOS port from being reassigned since such a reassigned flow will likely have to stall in setting or checking flowbits. Indeed, preliminary experiments with DEF1 indicate that preventing reassignment on NetBIOS ports mitigates some of the negative effects of reassignment and allows the optimistic parallelization to have some speedup relative to the single-threaded case. More detailed experiments would be needed to study the impact of such changes on the other traces and determine if such strategies could be effectively automated by analyzing the ruleset. As another example, the effectiveness of reassignment may vary substantially with the conditions used for reassignment. Experiments with reducing or increasing the flow reassignment threshold suggest that thresholds below 100 suffer from the overhead of additional flushes in stream reassembly while larger thresholds cause excessive stall times when flowbits rules actually match. 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). However, this condition had little impact on performance even as this threshold was varied from as high as 50% all the way down to 10%.

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. The impact of flowbit stalls can be mitigated by exploiting the fact that verification of rule options occurs in the order they are specified; if any of the tests fail or do not match, the remaining tests are skipped. Flowbits are often specified early among the options because on a single-threaded machine they are very fast. However for the optimistic scheme they can be moved to the end of the rule and not checked unless all other conditions match. Such ruleset modifications seem more practical than using a purely optimistic solution that speculatively performs the flowbit operation and then rolls back if there was a violation, since such rollbacks would require the reprocessing of many packets and a great deal of stored state.

6. RELATED WORK

Clustered intrusion-detection systems are among the most popularly deployed approaches to high-performance intrusion detection because they exploit multiple low-cost, identically-configured and administered PCs along with a load-balancing switch. Schaelicke et al. have proposed SPANIDS, a system that combines a specially-designed FPGA-based load-balancing switch that considers flow information and system load when redirecting packets to commodity PCs that run intrusion detection software [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]. By exploiting current architectural trends toward low-cost multicore and multiprocessor PCs, the parallelized Snort achieves good performance for small to mid-scale deployments without the expense of a load-balancing switch. For larger deployments, the parallel Snort versions presented here 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]. For example, Snapp et al. point out that an individual host experiencing a few failed logins may be normal, but a pattern of failed logins across a domain may indicate an intrusion attempt [17]. Collaborative information sharing may thus improve the overall rate of detecting intrusions. However, 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 because it can potentially exploit extensive hardware concurrency, several works have proposed hardware support for this stage. Several works use reconfigurable FPGAs since the hardware can simply be resynthesized on ruleset updates. Moscola et al. match not only exact strings but also regular expressions, exploiting packet-level parallelism across independent scanning engines [11]. Sourdis and Pnevmatikatos have used independent comparison pipelines in an FPGA to perform exact string matching at a rate of multiple characters per clock cycle [18]. Baker and Prasanna have given an FPGA-based synthesis algorithm that optimizes the set of characters to those that actually exist in matching patterns and then uses several independent matching pipelines for individual bits of those characters [3]. Instead of depending on the reconfigurability of FPGAs for ruleset updates, several works use SRAM for storage of string tables. Aldwairi et al. presented a network processor architecture with string-matching accelerators based on simple FSMs [2]. Tan and Sherwood use a compact SRAM-based representation of the string table along with a special-purpose hardware engine that processes the data by exploiting parallelism on a bit-level granularity [19]. Brodie et al. developed a pipelined FSM representation that allows high-speed matching of regular expressions [6], and can be implemented in an FPGA or an ASIC.

Though there has been much work that can effectively accelerate the multi-string and regular expression matching tasks, it is not sufficient. Figure 2 shows an average of 35% of processing time for multi-string and 15% for regular expression matching, for a combined total of almost 51% for the LL traces. According to Ahmdal's Law, even if it were made infinitely fast, only a 54% overall speedup could be obtained with accelerated string-matching, or 96% if regular expressions were accelerated as well. Clearly, though these components are the most important, they cannot be treated in isolation from the rest of the system, and an IDS is not complete if it has only these components. 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 its effectiveness.

There has also been some investigation on running intrusion detection software on network processors: Bos and Huang implement a rudimentary IDS which uses 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.

7. CONCLUSIONS

This paper presents and evaluates conservative and 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. Both parallelization schemes have their limitations, but both perform quite well for most of the workloads that they target. The conservative parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 3.0 speedup on 4 processor cores and processing at speeds up to 1.7 Gbps. The extra overheads in the optimistic parallelization degrade performance by about 10% for the traces that exhibit flow concurrency, limiting speedup to 2.8 on four cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.2 on four cores), with a peak traffic rate over 2 Gbps. Both schemes see an average traffic rate of just over 1 Gbps for the 5 traces, nearly doubling the performance of the serial version with only a slight increase in hardware cost and no increase in space. These results can be achieved while using hardware that is cost-effective, space-efficient, and increasingly being commoditized.

8. 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.