A Parallel Architecture for Stateful, High-Speed Intrusion Detection

Luca Foschini, Ashish V. Thapliyal, Lorenzo Cavallaro
Christopher Kruegel, Giovanni Vigna

Department of Computer Science
University of California, Santa Barbara
{foschini,ashish,sullivan,chris,vigna}@cs.ucsb.edu

Abstract. The increase in bandwidth over processing power has made stateful intrusion detection for high-speed networks more difficult, and, in certain cases, impossible. The problem of real-time stateful intrusion detection in high-speed networks cannot easily be solved by optimizing the packet matching algorithm utilized by a centralized process or by using custom-developed hardware. Instead, there is a need for a parallel approach that is able to decompose the problem into subproblems of manageable size. We present a novel parallel matching algorithm for the signature-based detection of network attacks. The algorithm is able to perform stateful signature matching and has been implemented only using off-the-shelf components. Our initial experiments confirm that, by making the rule matching process parallel, it is possible to achieve a scalable implementation of a stateful, network-based intrusion detection system.

1 Introduction

Intrusion detection is the process of analyzing a stream of events to identify the evidence of attacks. Intrusion detection can be applied to different domains and can be based on different techniques, which are usually characterized by the model used as the basis for detection. Systems used to specify what the "normal behavior" of an application is are usually called *anomaly-based*, because an attack will be detected as an event that does not fit the system's idea of what is "normal." On the other hand, systems used to specify what the manifestation of an attack is are known as *misuse-based*, and the models are often referred to as *signatures*.

Both anomaly-based and misuse-based systems have advantages and disadvantages. Anomaly-based systems are able to detect previously unknown attacks, but they traditionally produce more false positives (erroneous detections). Misuse-based systems, on the other hand, are usually more precise, but cannot detect attacks for which they do not have a description, and, therefore, they need continuous updating.

Because of their low false positive rate and their higher performance, misuse-based detection approaches are the basis for the majority of the existing network-based intrusion detection systems. Unfortunately, as the speed of the network links increases, keeping up with the pace of events becomes a real challenge.

This problem has been addressed by current intrusion detection systems by minimizing the amount of state that is associated with the matching process. In particular, many systems operate on single packets (e.g., [12]) and do not provide support for complex, *stateful* signatures, where the evidence necessary to detect an attack is spread across multiple packets at different points in time.

A number of approaches have been proposed to address the problem of matching stateful signatures in high-speed networks. Kruegel et al. proposed a parallel architecture to partition the traffic into slices of manageable size, which are then examined by a set of intrusion detection sensors [10]. This slicing technique takes into account the signatures used by the intrusion detection sensors, so that all the evidence needed to detect an attack is guaranteed to appear in the slice associated with the sensor responsible for the detection of that particular attack. Therefore, no communication among the intrusion detection sensors is necessary.

Unfortunately, even though the system was able to improve the overall performance of the detection process, it was not able to effectively partition the traffic in the case of complex stateful signatures. This is because, in general, the correct partitioning of the traffic is known only at runtime, and, therefore, any approach that uses a *static* partitioning algorithm needs to over-approximate the event space associated with a signature, possibly resulting in a degenerate partitioning.

We propose a novel technique to perform parallel matching of intrusion detection signatures. Our technique is similar to the one proposed in [10], since it uses a partitioning mechanism to reduce the traffic on a high-speed link to slices of manageable size, and, in addition, it uses a number of parallel sensors. However, in our approach, the sensors are able to communicate through a high-speed, low-latency, dedicated control plane. By doing this, they can provide feedback to each other in order to synchronize their scanning processes. Therefore, they can detect attacks that would be missed if the traffic partitions were analyzed separately as it happens in [10]. In addition, our system does not rely on any predetermined partitioning of the traffic and can optimally distribute the load over the available sensors.

In this paper, we make the following contributions:

– We introduce a novel architecture for the parallel matching of stateful network-based signatures.
– We present a novel algorithm that allows for the detection of complex, stateful attacks in a parallel fashion.
– We provide a precise characterization of the bottlenecks that are inherent to the parallel matching of stateful signatures in the most general case.
– We describe a prototype implementation of our system and we show that the proposed architecture allows for the parallel matching of network signatures.

The rest of this paper is structured as follows. In § 2, we present our rule matching model. Then, in § 3, we present the architecture of our parallel intrusion detection system. In § 4, we describe an algorithm for parallel rule matching that we implemented on the described architecture and in § 5 we show experimental results. Then, § 6 discusses related work in the field of high-speed intrusion detection and, finally, § 7 concludes.

2 Model

In general, an intrusion detection system scans a stream of events and detects attacks using signatures. Rules are matched against the stream of events by a rule matching system to identify signatures of attacks.

Usually, a rule r is described as composed of a predicate P, possibly a state S (in this case, the rule is said to be "stateful"), and an action A [5,17]. The predicate P describes the constraints on the values of the event and attributes of S. If the predicate evaluates to true for an event, then the action is triggered. The action A consists in modifying the rule state or raising an alert (that is, generating a new event for further processing). The rule maintains the state S in order to keep track of the steps of an attack. For example, a counter might be the state used by a rule to keep track of the number of TCP connection attempts to a host. This is information can be used to detect port scans. A number of architectures that implement a rule matching system as described above are publicly available [5,14,15]. These architectures are, in general, centralized, i.e., a single processing node deals with all the events.

A parallel architecture for intrusion detection should help lower the processing power demand and the memory footprint of packet analysis by distributing the network traffic to be analyzed among different sensors. We define a parallel model for intrusion detection by expanding the aforementioned event model, with some assumptions:

1. The network traffic is split and sent to one or more *sensors*.
2. Each sensor tries to match one or more signatures against the traffic it receives.
3. The system does not rely on any predefined mapping between packets and sensors, i.e., each packet could be sent to any sensor. This is a more general approach than the one described by Kruegel et al. in [10], where the traffic slicer enforced a mapping between packet features and sensors based on the analysis of the signatures' event spaces. Here, we take into consideration the most general case, that is, for any pair of packets, we have no *a priori* knowledge that they both belong to the same event space.

It is easy to see that, given these assumptions, stateful rules are difficult to implement in a parallel rule matcher. The reason is that, without any pre-existent mapping between packet features and sensors, there is no guarantee that two packets that are required to match a signature are processed by the same sensor.

A trivial solution to overcome this problem would be to replicate the traffic so that each packet is sent to every sensor. Unfortunately, this solution would render the parallel approach totally ineffective. A better approach is to minimize the amount of traffic to be scanned by processing each packet only once, which, in turn, entails that every sensor has to maintain the same rules with the same state loaded. Therefore, our parallel machine, by design, requires no replication of the incoming traffic, but a complete replication in the matcher state, which has to be the same for all the sensors at any time. To refine this observation, we express the predicate evaluation of a stateful rule in the following form:

Definition 1. *Predicate evaluation of a stateful rule:*

$$S_{R,t+1} = F(I_t, S_{R,t}),$$

where I_t is the input at time t, R is a rule, and $S_{R,t}$ is the state of the rule at time t.

If one sees the rule matching process as a finite state machine (FSM), F is nothing more than its *transition function*. The computationally expensive parts of the whole process are hidden in the formalism above and consist, most notably, of: (a) the evaluation of the input I_t, i.e., the packet scanning, which will possibly trigger a state transition, and (b) the corresponding state update that a transition might imply.

Evaluating the predicate is part of the packet scanning process, and it can be performed in parallel with no additional effort[1], since each sensor evaluates the same

[1] These kind of problems are also known as "embarrassingly parallel."

predicate on different packets. On the other hand, the state update is an operation that we aim to avoid in our system, because it needs to be repeated on all the sensors, in order to have the state of the parallel machine to evolve simultaneously.

These considerations are not very encouraging when building a parallel system. Amdahl's law [1] here fits perfectly: the time needed by the state update has to be spent by all the sensors for each packet that modifies the state, even though the packet matching occurs on only one sensor, and, therefore, cannot be parallelized. Simply put, during the state update triggered by a packet matching a rule, all the sensors do the same operation, behaving as if they were a single machine and without exploiting any parallelization. Therefore, it is already clear that the bottleneck of an architecture that has to address a general, parallel stateful signature matching system lies in updating the state of the rule(s) when a match occurs.

Note that the above considerations hold only for the scanned packets that match a rule and, thus, update the rule's state. However, not all the packets will match (hopefully, very few will). Therefore, we can significantly benefit from parallelizing the packet scanning process.

Intuitively, we need to make the update of the state of a rule as fast as possible. In order to do that, we can partition the rule's state. More precisely, we split the rule's state in two parts: a *scanning state*, or predicate state, and a *working state*. The *working state* does not modify the scanning state and thus does not affect the matching capability of a sensor. Only the predicate state needed for the evaluation has to be updated in the aforementioned non-scalable fashion. The actual working state can, on the other hand, live on a separate centralized machine, which we call "control node," and it can be updated asynchronously (in § 3, we will describe the system architecture in more detail). Ideally, we would like the scanning state to be as small as possible. More formally, we can functionally decompose a rule as below:

Definition 2. *State transition and output of a stateful sensor:*

$$S_{P,t+1} = F_{SP}(I_t, S_{P,t}); \quad O_{P,t+1} = F_{OP}(I_t, S_{P,t})$$

Definition 3. *State transition and output of a control node:*

$$S_{W,t+1} = F_{SW}(O_{P,t}, S_{W,t}); \quad O_{W,t+1} = F_{OW}(S_{W,t})$$

In the definition above, S_P is the *scanning (or predicate) state* and S_W is the *working state*, while O_P and O_W are, respectively, the output of the sensors (going to the control node) and the output of the control node, which could be an alert. $F_{S\cdot}$ are the state transition functions, which map states and inputs to new states, and $F_{O\cdot}$ are the output functions, which map states, or states and input, to output.

Def. 2 describes the behavior of a generic sensor in terms of a *Mealy Machine*, while Def. 3 describes the control node in terms of a *Moore Machine*. The fundamental point is that the control node can process the output from the sensors in an environment that is completely decoupled from the sensors.

Following the considerations above, we want to split the state management so that the state update process on the sensors is minimized and does not require any feedback from the control node to the sensors. For example, a rule that detects a port scan attack can be implemented as a TCP stream-based predicate on the sensors. The stream-based predicate and the predicate state take care of filtering out packets that are part of a flow (thus, maintaining some predicate state that mimics the TCP state machine for each stream and filters out packets when they are found to belong to a legitimate connection). The detection part (working state) on the control node takes care of keeping in memory and updating all the data structures needed for the detection of the scan, such as the list of targets and scanners, counters and timers, or, to detect coordinated port scans, a whole implicit graph representation of inter-host connectivity as described in [9], on which to run periodically complex set-covering algorithms.

3 Architecture

The proposed system uses a parallel architecture as shown in Figure 1. In our setup,

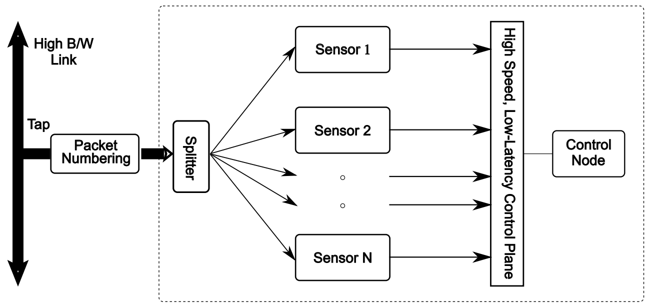

Fig. 1: The architecture of the parallel rule matcher.

packets are obtained from a tap (T) into the high-speed network link(s) that copies the packets and sends them to a packet numbering unit (PNU). The PNU, in turn, tags the packets with a logical timestamp (e.g., generated using a counter starting from zero) and forwards them to the splitter. The splitter sends the traffic in a round-robin fashion to N sensor nodes in charge of performing the detection.

The sensor nodes are managed by a control node that maintains the working state associated with the attack instances, as described in the previous section. The sensor nodes forward to the control node any packet[2] that is tagged as suspicious, i.e., that it is possibly part of a stateful attack.

The control node updates its state (the working state) according to the messages received from the sensors. In addition, the sensors are able to talk to each other through a low-latency broadcast channel. As a result, when one of the sensors matches a rule, the others can update their predicate state (once again, the same for all sensors) consistently.

In order for the parallel matcher to emulate a serial one, each sensor is provided with a buffer that we call *sensor match buffer* (SMB), which is utilized to store already-scanned packets. This buffer is required in case an earlier packet was matched by another node, which triggered a sensor state change. In § 4, an algorithm will be described that leverages the SMB on each sensor in order to make the parallel rule matcher behave like a serial one.

In this architecture, the traffic is split in a round-robin fashion, and, therefore, each node receives $1/Nth$ of the total traffic, without incurring any potential load-balancing problems that instead would be caused by alternative traffic partitioning

[2] More precisely, only the relevant attributes of the packet such as, for example, source or destination IP, are forwarded to the other sensors and the control node.

techniques employed in other frameworks, such as [2,10]. Moreover, since there are no constraints on the mapping between sensors and traffic, if a sensor is slower than the others and is overloaded, the excess traffic can be seamlessly diverted to other sensors in order to keep the load constant, without using any complicated load-balancing algorithm.

4 Algorithm and Protocol

We present a general algorithm, called "simple-sequencer," that we use to perform the parallel update of the state transition function of the predicate state as described in Def. 2. The algorithm is implemented on the sensor nodes and is used to keep synchronized their predicate state. The simple-sequencer parallel algorithm behaves like a serial algorithm performing the same task, as far as correctness is concerned, if the predicate state transition function of Def. 2 belongs to a class of *strict-sequence* state machine that we define below. The control node implements a serial rule-matching algorithm corresponding to the working state update function in Def. 3 and needs only to perform reordering of out-of-order packets sent by the sensors.

Definition 4. *Strict-sequence state machine:*
A strict-sequence state machine recognizes events that happen in a sequential fashion, i.e., any event in the sequence can be followed by one and only one different event in the event stream.

According to Kumar et al. [11], the majority of known intrusion patterns can be formulated as a set of events that happen in strict sequence. Nevertheless, the ability of the algorithm to match events in strict sequence can also be leveraged to recognize patterns not directly implying an attack. For instance, the simple-sequencer algorithm allows to identify the beginning of a TCP connection, described by the sequence (SYN, SYN-ACK, ACK) in this particular order. Even though the starting of a TCP connection is not a manifestation of an attack itself, it could help, for instance, filter out packets belonging to a legitimate TCP connection in a signature, which would reduce the rate of the traffic processed by the parallel matcher.

Note that the strict-sequence constraint above applies only to the portion of rule implemented on the sensors. The control node can evaluate more complex functions (order-variant, with partial order constraints, etc.) on the output of the sensor nodes, since it does not operate in parallel and leverages a centralized state.

4.1 Simple-Sequencer Matching Algorithm

Overview of the algorithm. From now on, we refer to packets as being "earlier" or "later," using the packet ID as the ordering parameter; with smaller corresponding to earlier and larger corresponding to later. In addition, when we say "packet x" we mean "packet with packet ID x". We also say that a packet "matches" the rule when it causes a state update on the associated FSM defined in Def. 2.

The intuition behind the inner works of the simple-sequencer algorithm is the following: if a packet causes a state change on a sensor, then any other packet following it in the input has to be scanned against the new state. This means that no packets can be discarded from the sensors' SMBs as long as they are needed.

The crux of the algorithm lies in the logic used to prune the sensors' SMBs, which, otherwise, would grow unbounded. The algorithm guarantees that no packet that could be needed at some sensor is ever discarded. This condition is enforced in two ways: (i) no packet later than the earliest packet currently scanned (i.e., the packet scanned by the slowest sensor) is discarded; (ii) if a packet caused a match and a state update on all the sensors, it is retained until all the matches triggered by that state update are completed.

The need for (ii) is clarified by an example. Suppose that the slowest sensor is scanning packet x and that packet x causes a match. At this point, we say that packet x has a pending match. The match is communicated to the other sensors, which will update their state accordingly. The sensors will rescan any packet later than x. If, during this operation the packet $x+1$ causes a new match on any sensor, then all the packets later than $x+1$ must be retained on the sensors. But in the meantime, the slowest sensor could have moved far beyond packet $x+2$, thus discarding all the earlier packets (and, $x+2$ itself) by (i). Therefore, packet $x+2$ needs to be retained in the slowest sensor's SMB by additional logic, as described by (ii).

A match remains pending until it has been acknowledged by all the sensors. A sensor acknowledges a match when all the matches triggered by this match have been acknowledged, or, no new matches have been triggered. Packet $x+2$ will become available to be discarded only when no packets earlier than it have pending matches.

We assume that all messages between a pair of sensors are transported using a first-in-first-out messaging system. Each sensor i keeps the following variables:

- $lastInspected_i$: The packet ID of the last packet inspected by sensor i.
- $pendingMatches_i$: A priority queue arranged in ascending order of the value of packet ID k_i. This queue contains the matches that are pending, i.e., not yet acknowledged by the other sensors.
- $earliestNeeded_i$: The head of the $pendingMatches_i$, i.e., the packet ID of the earliest packet that needs to be retained at sensor i.
- SMB_i: The buffer in which packets are stored, ordered by packet ID. Packets stored in this buffer include those yet to be scanned, as well as those already scanned but not yet discarded, i.e., all the packets later than $earliestNeeded_i$.

In addition to the previous per-sensor variables, we define $discardPtr$ to be the minimum of $earliestNeeded_j$ over all the sensors j.

Now, we are ready to specify the algorithm, and we do so for one sensor i. We rely on the following primitives:

BroadcastMatch($< Match(P_k, r_l, i) >$**):** If the packet matches a rule r_l then broadcast $< Match(P_k, r_l, i) >$ to all nodes. Add $< k, triggerMsg, timestamp >$ to the $pendingMatches_i$ priority queue. The parameter $triggerMsg$ may be *null* if no other match triggered this match. P_k is the packet that matched[3].

ProcessMatch($< Match(P_k, r_l, j) >$**):** A rule match $< Match(P_k, r_l, j) >$ is received by sensor i. Update rule r_l according to the action specified by the rule and the data present in packet P_k, then scan the packets up to and including $lastInspected_i$ in buffer SMB_i that are after packet k for a match against the updated rule. For each match that occurs at packet h, for rule m, call BroadcastMatch($< Match(P_h, r_m, i) >$).

ProcessMatchAck($< MatchAck(P_k, r_l, j) >$**):** Mark in $pendingMatches_i$ that sensor j has acknowledged $< k, triggerMsg =< Match(P_h, r_m) >, timestamp >$. If all the nodes have acknowledged this match, remove $< k, triggerMsg =< Match(P_h, r_m, j) >, timestamp >$ from the priority queue $pendingMatches_i$ and send a match acknowledgment $< MatchAck(P_h, r_m, i) >$ to node j.

[3] Of course, in a real-world deployment, only the relevant information needed by other sensors will be broadcast.

```
// Task 1: new packets are enqueued in the sensor's SMB
foreach new packet $P_k$ available from network do
    Insert $P_k$ at the end of buffer $SMB_i$;
end
// Task 2: packet inspection and communication with other sensors
foreach packet $P_k$ not yet scanned in $SMB_i$ do
    if a Match message is available wrt packet $P_h$ and rule $r_l$ for sensor $j$ then
        ProcessMatch($< Match(P_h, r_l, j) >$);
    else
        if a MatchAck message is available wrt packet $P_h$ and rule $r_l$ for
        sensor $j$ then
            ProcessMatchAck($< Match(P_h, r_l, j) >$);
        end
    end
    Scan $P_k$ against each rule;
    Set $lastInspected_i$ to $k$;
    if a match with rule $r_l$ occurred then
        BroadcastMatch($< Match(P_k, r_l, i) >$)
    end
end
// Task 3: synchronize sensors' buffers
Every $T$ packets scanned, run the bully algorithm to compute $discardPtr$;
```

Fig. 2: The simple-sequencer rule matching algorithm.

The algorithm, shown in Figure 2 for the i-th sensor, is composed of three tasks, which can be performed concurrently.

The bully algorithm [8] is utilized to compute the *discardPtr*. All the sensors can discard the packets from their SMBs periodically (an appropriate timer can be utilized to trigger discards). More precisely, the sensors can discard packets earlier of the last *discardPtr* seen. This deferred deletion ensures that the transitories associated with the contention have vanished, so that there is not any packet earlier than *discardPtr*, which is needed.

Properties of the Simple-Sequencer Algorithm. The simple-sequencer algorithm has an interesting property, i.e., it behaves as a serial algorithm in matching strict-sequence FSMs as described in Def.4. This property is important because it guarantees the matching of strict-sequence FSMs avoiding any race condition among the sensors' state.

Due to space constraints, a more thorough description of the simple sequencer algorithm with exemplifications can be found in [7]. Also, for the same reasons, an analysis of the scaling properties of the architecture when running the simple-sequencer algorithm is omitted and reported in [6], Section 4.4.

4.2 Control Node

The simple-sequencer algorithm works on the sensors by implementing the predicate state transition function of Def. 2. This function is constrained to belong to the class of strict-sequence FSM as defined in Def. 4. Nevertheless, the Moore machine of Def. 3 implemented on the control node and used to update the working state does not have any limitation. Therefore, any kind of rule, as decomposed in Def. 2 and Def. 3, can be implemented on the parallel architecture made of sensors and control node, as long as the rule is decomposed in such a way that guarantees that its predicate part is an FSM that abides to Def. 4.

Going back to the port scan detection example of § 2, now we can see how this signature can be detected by our parallel architecture. The sensor nodes are responsible of tracking established TCP connections and filter out packets belonging to a legitimate connection. Tracking the beginning of a TCP connection means identifying the three-way handshake between the involved hosts. The (SYN, SYN-ACK, ACK) pattern can be matched by a strict-sequence rule implemented on the sensors. Once the sensors identify a connection they start discarding packets belonging to it. Other packets are instead forwarded to the control node, which performs a thorough port scan detection on them. In this way, the joint action of sensors and control node can detect a signature that cannot be described by a strict-sequence FSM itself. We will review this setup more in depth in § 5.

We stress that the control node, being a serial component, simply processes packets sent to it by the sensors. However, sensors can still send to the control node packets out of order. The reordering is caused by the different relative instantaneous speeds at which sensors operate. To prevent the control node to analyze packets sent by the sensors in the wrong order, we use a simple reordering buffer.

The buffer must be large enough to accommodate as many packets as can fit the gap between the fastest and the slowest sensor. In the same fashion, the delay before processing packets must be large enough to allow late packets to come in and be put in the right order.

Different from the SMBs, where the correct size of the buffers were enforced as a byproduct of the simple-sequencer algorithm, the size of the control node front buffer cannot be easily predicted. We will therefore set the size of the buffer and the delay before processing to conservative values, which will be validated in § 5.

5 Experimental Validation

To test our model, following the model and algorithm presented in § 2 and § 4, we implemented a parallel architecture to detect port scans.

Our implementation of the sensors and the control node relies on Snort version 2.6 [15]. The Snort running on the sensors has two preprocessors loaded, namely `stream4_reassemble` and `sfportscan`, which have been modified to update their state synchronously, as described in the algorithm reported in Fig. 2. More precisely, the `stream4_reassemble` preprocessors have been enabled to communicate with each other to synchronize the state of the observed TCP connections.

The control node runs Snort 2.6 with the `sfportscan` preprocessor loaded and has a packet reordering buffer. The packet reordering buffer is used to reorder possibly out-of-order packets from the sensors. The sensors use the `stream4_reassemble` to check if a packet belongs to an established TCP connection. When this is not the case, the packet is forwarded to the control node that uses the `sfportscan` preprocessor to match it against the port scan signature.

We expect our parallel architecture to be able to detect port scans as a single-instance vanilla Snort with `sfportscan` preprocessor loaded would do. In addition, we expect the parallel machine to be able to keep up with a higher throughput than what a single instance detector can cope with.

At first, we have performed a functionality test in an emulated environment based on tap interfaces and the Virtual Distributed Ethernet [3,4]. A complete description of the emulated environment cannot be reported here due to lack of space and can be found in [6], § 6.1.

We have also set up a real-world network testbed to evaluate our algorithm, protocol, and architecture. From the network point of view, the system was built following the architecture described in § 3. More precisely, the system is composed of two networks:

The distribution network, through which packets are sent from the splitter to the sensors. This has been implemented as an Ethernet network with no IP-level processing. The splitter has a high throughput network interface connected to a switch, configured with static routes from the interface to which the splitter is connected towards the outgoing interfaces to the sensors.

The control network, through which the sensors exchange control packets among each other and with the control node. The sensors and the control node are connected through a hub, since there are no high throughput requirements for control traffic, and only low latency is required.

We analyzed the behavior of the sensors by following the journey of a packet captured from the live interface by a sensor.

1. When a new packet is captured on the live interface, it is passed to the Snort decoding engine.
2. Before sending a packet to the preprocessors, each sensor listens (in non-blocking mode) for incoming packets on the control interface and processes them, if any.
3. Packets received from the live interface are sent to the `stream4_reassemble` preprocessor. If the packet changes the state of the reassembly machine (i.e., initiates, tears down, or modifies the state of some connections), which in this case represents the scanning state described in our theoretical framework, then it is broadcast to the other sensors as a $Match$ beacon.
4. Packets received from the control interface (coming from other sensors) are stored into the SMB and will be used to re-create the same state changes on the `stream4_reassemble` preprocessor.
5. Packets coming from the live interface and sent through the stream reassembly preprocessor are then handed out to the port scan preprocessor. At the beginning of the process, some tests are performed on the packet in order to exclude benign packets. One of these tests consists in checking if the packet is part of a legitimate connection. If this is the case, the packet surely cannot be part of a scan, and, thus, no further checks are performed on it.
   Here is where the scanning state of the `stream4_reassemble` preprocessor is exploited. Packets that are not tagged as benign at this stage do not go through a complete analysis against port scan detection heuristics, but are simply forwarded to the control node, which takes care of them. Therefore, sensors do not need to maintain any state associated with the port scan preprocessor, such as scanners and scanned hosts lists. Moreover, only a small fraction of the input traffic will reach the control node.
6. The control node puts the packets received from the sensors in the reordering buffer and, after a delay of 0.2 seconds, scans them and identifies scan attacks. In our experiments we have seen that a delay of 0.2 second was sufficient to allow the control node for processing all the packets in order.

It is easy to recognize that the aforementioned setup is a particular case of the general architectural model described in § 2. The preprocessor `stream4_reassembly` on the sensors maintains the predicate state and uses the simple-sequencer algorithm to keep the state of TCP connections synchronized among sensors. The `sfportscan` preprocessor loaded on the sensors performs the rule predicate evaluation to filter out benign traffic. The working state is only kept in the `sfportscan` preprocessor on the control node in the form of all the relevant structures maintained by the preprocessor itself, such as the scanner hosts and the scanned hosts lists.

5.1 Evaluation Dataset

To evaluate our system, we have crafted an artificial dataset by merging attack-free traffic collected on a real-world network at UCSB with a dump of a port scans performed against a host inside the home network. The scans were performed using the `nmap` tool. The attack-free traffic contains 3.46 million packets, and includes a mix of web traffic, mail traffic, IRC traffic, and other common protocols.

We rewrote, by means of the `tcprewrite` tool [20], the MAC address of every frame in order to induct a round-robin scattering pattern on the splitter.

5.2 Hardware and Software Configuration

We set up four Linux sensors running Snort 2.6. Each sensors is equipped with four Intel Xeon CPUs X3220 at 2.40GHz and 4GB of RAM.

The control node is deployed on another Linux host with the same hardware characteristics of the sensors. The control node runs a version of Snort with the reordering packet buffer mentioned before and a vanilla version of the `sfportscan` preprocessor. Another Linux box is used only to inject the traffic into the testbed using a Gigabit Ethernet card connected to a PCI-Express bus.

Each sensor box mounts two Gigabit Ethernet network cards, one to receive the traffic from the splitter, and another one to communicate with the other sensors and the control node. Sensors are connected to the splitter box through a Cisco Catalyst 3500 XL switch configured with static routes to the sensors capture interfaces. The switch has a 1000-BaseT GBIC module that receives the traffic from the splitter output interface and is connected to sensors through FastEthernet ports. The sensors' control interfaces are connected to each other through an ordinary FastEthernet hub.

5.3 Experiments

We performed several experiments to validate our model. We compared the parallel rule matcher setup with a single instance of Snort with the `stream4_reassemble` and `sfportscan` preprocessor loaded. The Snort community ruleset [19] (version "CURRENT" as of July 2008) was loaded on both the parallel sensors and the single instance. We note that none of the rules loaded on the sensors were extended to work correctly on the parallel machine. The rules were used only to represent a realistic per-packet workload in a real-world intrusion detection system.

We expected that, at a certain traffic rate, the single instance Snort would not be able to keep up with the traffic, and, therefore, would miss the port scan attack, while the parallel version of the sensor would be able to catch it. Each experiment has been repeated 10 times and the results reported are the average of the outcomes. More precisely, the setups compared are as follows.

Single Snort: In this setup, the traffic has been replayed via a cross cable to a single box mounting a Gigabit Ethernet interface.

Parallel architecture with four sensors: In this setup, we implemented the complete parallel architecture with four sensors and a control node.

Parallel architecture with two and three sensors: In this setup, we changed the number of sensor utilized to study the scalability of our approach.

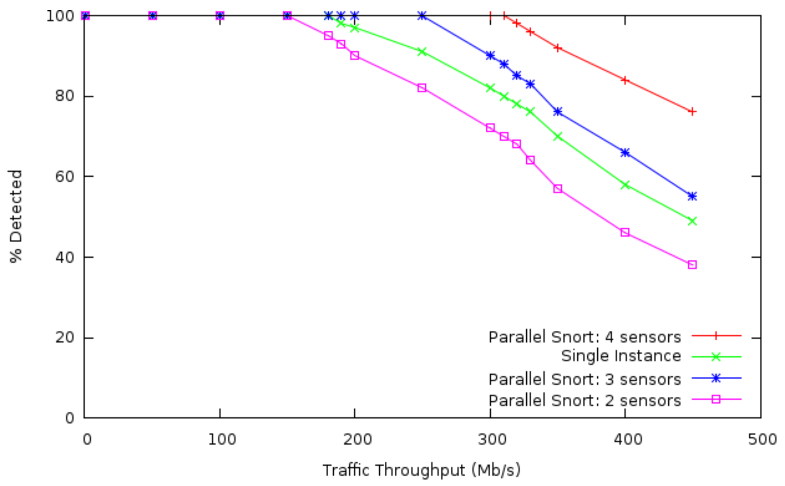

Fig. 3: % of correct detection when varying the number of sensors and the traffic speed.

Results are reported in Figure 3. For each setup, the percentage of attacks detected versus the input traffic throughput is plotted. The percentage of detected attacks is computed as the average number of port scan packets detected with respect to the total present in the trace. It can be seen how the four-sensors Snort outperforms the single instance starting from a traffic throughput of 190Mb/s. At this speed, the single instance of Snort starts discarding packets. The percentage of discarded packets grows almost linearly as the throughput of the traffic increases. On the other hand, the parallel Snort starts discarding packets at around 320Mb/s. This happens because we saturated the aggregated capacity of the four outbound ports of the Catalyst switch going to the sensors. In fact, we found that none of the four Snort sensors employed dropped any packet during their analysis; instead, packets were dropped by the switch. We expect our parallel architecture to be able to achieve higher throughputs, but we were not able to prove our expectation due to the physical limitations of the hardware employed. The same problem of capacity saturation was also encountered with the parallel architecture using two and three sensors, as can be seen in Figure 3. For two sensors, in particular, the performance in detection is worse than using a single Snort. This is easily explained by the fact that the single Snort was tested using a crossover connection between the splitter and the instance, therefore bypassing, the FastEthernet switch.

CPU Usage. We measured the time spent by the parallel-enabled Snort in the portion of code implementing the communication and parallel synchronization logic. The RDTSC instruction was used to achieve a precise time measure. RDTSC returns time in arbitrary units (count of ticks from processor reset). Therefore, it can be used only to get relative measurements. From our experiment, we have measured that the average time spent by the sensors in the additional logic (at the highest possible speed) is about 5.3% of the total time spent in the Snort engine. The CPU usage measurements were performed without the community ruleset used for the throughput experiments mentioned above, which is the worst case. In fact, if the ruleset had been loaded, the fraction of time spent in the parallel logic would have dropped even more, as it is independent of the loaded rules.

Latency and Cost of Communication. Another important factor to take into consideration is the latency introduced by the SMBs. In our experimental setup, we chose to limit the length of the SMBs to 1,000 packets. Therefore, this number determines the upper bound of the latency that a traffic packet can experience in its journey to the control node. With four sensors scanning the traffic at 100Mb/s, the maximum latency experienced would be ~ 0.05 seconds, which is negligible. Moreover, the latency decreases with the increasing of throughput, becoming less significant at higher throughputs.

As far as the communication cost introduced by the communication among sensors and control node is concerned, we report that only 58K packets out of 3.46 million were forwarded to the control node when the setup with four sensors was employed at 100Mb/s (no packet loss). 17K packets were exchanged among the sensors to update the predicate state. Most of the rest of the communication cost of the parallel machine is spent for the buffer synchronization protocol. More precisely, 127K packets were exchanged among sensors as part of the bully algorithm. We note that, since we chose to limit the SMB length to 1,000 packets, we could have disabled the buffer synchronization mechanism. However, we decided to leave it active to determine experimentally the cost of the number of packets exchanged for buffer synchronization purposes. We also note that the total cost spent in communication among the sensors and the control node accounts for 203K packets, out of the 3.46 million present in the traffic. This means that only a small fraction of the input traffic ($\sim 6\%$) reached the control node and the majority of packets were filtered out by the sensors.

6 Related Work

As we mentioned in the introduction, Kruegel et al. proposed a distributed architecture to partition the traffic into slices of manageable size, which were then examined by dedicated intrusion detection sensors [10]. Our technique is different because the sensors can communicate with each other in order to keep a synchronized version of the matching state, which, in turn, allows the packets to be sent independently to any sensors.

Sekar et al. [16] describe an approach to perform high-performance analysis of network data. They propose a domain-specific language for capturing patterns of both normal and abnormal packet sequences (therefore, their system encompasses both misuse and anomaly detection). Moreover, they developed an efficient implementation of data aggregation and pattern matching operations. A key feature of their implementation is that the pattern-matching time is insensitive to the number of rules, thus making the approach scalable to large rule sets. The shortcomings in their work are that the processing of the packets is done on a central sensor, and, therefore needs to be able to keep up with the fast pace of packets on high-speed networks.

Building up from the foundations laid by Sekar et al., but using a more general approach, Meier et al. [13] propose a method for system optimization in order to detect complex attacks. Their method reduces the analysis run time that focuses on complex, stateful, signatures.

Sommer et al. [18] propose a way to exchange state among sensors by serializing it into a file and exchanging it between sensors. Their approach is focused on providing a framework to transfer fine-grained state among different sensors in order to enhance the IDS with a number of features. Although the authors provide a way to exchange fine-grained state among sensors, no emphasis is put on performance.

However, the mechanisms introduced in [18] were leveraged by Vallentin et al. to build a cluster-based NIDS [21]. This design has a number of similarities with our architecture, as it includes a component that splits the traffic across sensors, which, in turn, exchange information to detect attacks whose evidence span multiple slices. However, the focus of the work of Vallentin et al. is on the engineering challenges associated with the creation of a high-performance, cluster-based NIDS, while the focus of the research we described in this paper is on the modeling and analysis of the general problem of performing parallel detection of stateful signatures. For example, a substantial part of the complexity of our algorithm is due to the analysis of previously processed packets when a new rule triggers a change in the replicated state. This problem is simply avoided by Vallentin et al. by partitioning the traffic according to source/destination pairs, using loose synchronization, and by limiting the possibility of race conditions by means of signature-specific techniques. Even though the two approaches have different foci, many of the lessons learned by implementing each approach can be used as a basis to improve the respective designs.

Other approaches, such as the ones from Colajanni et al. [2] and Xinidis et al. [22] propose optimization based on load-balancing and early filtering to reduce the load of each sensor. However, their work does not focus on the design of a truly parallel matching algorithm.

7 Conclusions

The speed of networking technologies has increased faster than the speed of processors, and, therefore, centralized solutions to the network intrusion detection problems are not scalable. The problem of detecting attacks in high-speed environment is made more difficult by the stateful nature of complex attacks.

In this paper, we have presented a novel approach to the parallel matching of stateful signatures in network-based intrusion detection systems. Our approach is based on a multi-sensor architecture and a parallel algorithm that allow for the efficient matching of multi-step signatures. We have analyzed the feasibility and described a proof-of-concept implementation of a parallel, stateful intrusion detection system for high-speed networks.

The architecture has been deployed on a real network using four Linux boxes as sensors and it has been tested for port scan detection in various configurations. The result of the evaluation have confirmed the validity of the theoretical model, since the parallel rule matcher, composed of four sensors, has successfully outperformed a single sensor instance, which we used as a baseline for comparison, performing the same kind of detection on the same dataset.

References

1. G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, 1967.
2. M. Colajanni and M. Marchetti. A parallel architecture for stateful intrusion detection in high traffic networks. september 2006.
3. R. Davoli. Vde: Virtual distributed ethernet. Technical report, 2004.
4. R. Davoli. Vde: Virtual distributed ethernet. In *TRIDENTCOM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220, Washington, DC, USA, 2005. IEEE Computer Society.
5. S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *Proceedings of the ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.
6. L. Foschini. A formalization and analysis of high-speed stateful signature matching for intrusion detection. 2007.
7. L. Foschini, A. V. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A Parallel Architecture for Stateful, High-Speed Intrusion Detection. Technical report, 2008.
8. H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, 1982.
9. C. Gates. *Co-ordinated Port Scans: A Model, A Detector and An Evaluation Methodology.* PhD thesis, Dalhousie University, Halifax, Nova Scotia, February 2006.
10. C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, Oakland, CA, May 2002. IEEE Press.
11. S. Kumar and E. H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.
12. H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu. A Memory-Efficient Parallel String Matching Architecture for High-Speed Intrusion Detection. *IEEE Journal on Selected Areas in Communication*, 24(10), October 2006.
13. M. Meier, S. Schmerl, and H. Koenig. Improving the Efficiency of Misuse Detection. In *Proceedings of RAID*, 2005.
14. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th Usenix Security Symposium*, 1998.
15. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the Large Installation System Administration Conference (LISA)*, Seattle, WA, November 1999.
16. R. Sekar, V. Guang, S. Verma, and T. Shanbhag. A High-performance Network Intrusion Detection System. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, November 1999.
17. Snort - The Open Source Network Intrusion Detection System. http://www.snort.org, 2004.
18. R. Sommer and V. Paxson. Exploiting Independent State For Network Intrusion Detection. In *Proceedings of ACSAC*, 2005.
19. The open source community. *Snort Community ruleset.*
20. A. Turner. tcprewrite trac page. http://tcpreplay.synfin.net/trac/wiki/tcprewrite.
21. M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proceedings of RAID*, 2007.
22. K. Xinidis, I. Charitakis, S. Antonatos, K. Anagnostakis, and E. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE TDSC*, 3(1):31, 2006.