The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware

Matthias Vallentin[3], Robin Sommer[2,1], Jason Lee[2], Craig Leres[2], Vern Paxson[1,2], and Brian Tierney[2]

[1] International Computer Science Institute
[2] Lawrence Berkeley National Laboratory
[3] TU München

Abstract. In this work we present a *NIDS cluster* as a scalable solution for realizing high-performance, stateful network intrusion detection on commodity hardware. The design addresses three challenges: *(i)* distributing traffic evenly across an extensible set of analysis nodes in a fashion that minimizes the communication required for coordination, *(ii)* adapting the NIDS's operation to support coordinating its *low-level* analysis rather than just aggregating alerts; and *(iii)* validating that the cluster produces sound results. Prototypes of our NIDS cluster now operate at the Lawrence Berkeley National Laboratory and the University of California at Berkeley. In both environments the clusters greatly enhance the power of the network security monitoring.

1 Introduction

The performance required to implement effective network security monitoring poses major challenges for the underlying hardware. Many network intrusion detection systems (NIDSs), both open-source and commercial, are based on inexpensive commodity hardware. However, today the processing required to analyze even a single well-loaded Gbps traffic stream at any significant depth is beyond the reach of single workstations, and the technology trends threaten to widen this gap in the future, not narrow it [1].

Faced with this performance gap, we must abide either *(i)* curtailing our analysis, *(ii)* turning to expensive, custom hardware, or *(iii)* employing some form of *load-balancing* to split the analysis across multiple commodity systems. In this work we pursue the third of these, because of the appeal of retaining the great flexibility and cost benefits of using commercial PC hardware. With this approach a "frontend" divides the traffic stream among the analysis nodes, each of which gets a share of the total network traffic to analyze in depth.

Conceptually, such a setup is easy to extend with increasing traffic volumes by simply deploying more boxes. However, the key challenge with such a system is how to *correlate* the analysis performed by each node, as otherwise attacks that span more than what one system sees will go undetected.

Unfortunately, this is where things get tricky. While all major NIDS provide support for multi-system configurations, typically individual instances (often termed *sensors*) connect to a central manager that correlates their results. The information exchanged tends to be very high-level: often just alerts that already present the *conclusion* of a sensor's analysis. Many inter-connection attacks, however, require much finer-grained correlation. As a simple example, to reliably detect a scan we need to track connection attempts across the full traffic stream. Hence, instead of correlating results, what we really need is to correlate the underlying *analysis*.

In this work, we build such a system. We term it the *NIDS cluster*: a set of commodity PCs that collaboratively analyze a traffic stream without sacrificing accuracy of the detection. Individual cluster nodes run instances of a NIDS and transparently exchange low-level analysis state to compose a global picture of the network activity. As a whole, the cluster transparently performs the same analysis a single instance of the NIDS would if it could cope by itself with the full network load.

When designing our system we faced three challenges: *(i)* distributing traffic across the nodes in a fashion that minimizes the communication required for correlation, yet avoids overloading any particular node; *(ii)* adapting the NIDS's operation to support coordinating its lower-level; and *(iii)* validating that the cluster produces sound results. We will discuss each of these in depth.

The original motivation for our work arose from the operational network monitoring setup at the Lawrence Berkeley National Laboratory (LBNL), which connects thousands of users/hosts to the Internet via a 10 Gbps access link. The lab's primary monitoring is done using Bro [2], an open-source NIDS running on commodity hardware. Since no single instance of the system can analyze LBNL's traffic in sufficient depth, over time the setup evolved into a configuration that uses a number of separate, uncoordinated Bro instances running on an inhomogeneous set of PCs (and even this setup still cannot analyze all traffic). Each instance performs a dedicated task (e.g., one analyzes only HTTP traffic) in isolation, and each system individually reports its results to the Lab's analysts. Thus, we desired to remedy the lack of coordinated analysis without sacrificing the very major benefits of using commodity, general-purpose hardware.

Thus, when designing the NIDS cluster we naturally targeted Bro as the underlying analysis engine for the backend nodes. In addition to fitting with the operational environment, Bro had the significant benefit that it already provides mechanisms for coordinating lower-level analysis (rather than only high-level results such as alerts) by means of its *independent state* framework [3]. Due to our choice of Bro, in the subsequent discussion we sometimes have to delve into particulars of the system. We note, however, that generally our approach applies well to other systems that support general low-level messaging functionality.

We now operate a prototype of our NIDS cluster at LBNL in parallel with the sites' operational monitoring, which it will eventually replace. Another prototype installation monitors the access link of the University of California at Berkeley (UCB), and an earlier prototype ran at *IEEE Supercomputing 2006*, the premier international conference on high performance computing, networking and storage. There, two separate clusters monitored the conference's primary 1 Gbps backbone network as well as portions of the 100 Gbps *High Speed Bandwidth Challenge* network.

We structure the remainder of this paper as follows. In § 2 we present the primary design objectives of the NIDS cluster. In § 3 we discuss the design and implementation of schemes for evenly distributing load across the cluster nodes, and in § 4 the design and implementation of the backend nodes. In § 5 we perform a trace-based evaluation to gauge the performance and accuracy of the cluster, followed in § 6 by a discussion of live performance as observed in our current cluster installations at LBNL and UCB. § 7 discusses related work, with conclusions in § 8.

2 Design Objectives and Resulting Architecture

With the NIDS cluster we aim to realize in-depth, yet inexpensive, network monitoring in high-performance environments. To do so in a manner suitable for *operational* security monitoring, the design must satisfy a number of objectives:

Transparency. The system should convey to the operator the impression of interacting only with a single NIDS, producing results as a single NIDS would *if* it could cope with the total load.

Scalability. Since network traffic volumes grow with time, we want to be able to easily add more nodes to the cluster to accommodate an increased load. Ideally, the cluster's performance scales *linearly*, i.e., the amount of additional resources necessary is a linear function of the increase in network load.

Commodity hardware. In general, we want to leverage the enormous flexibility and economies-of-scale that operation on commodity hardware can bring over use of custom hardware (e.g., ASICs or FPGAs). However, for monitoring very high-speed links, we may need to resort to specialized hardware for the *frontends*, as these need to process packets at full line-rate.

Ease of Use. The operator should interact with the system using a single host as the primary interface, both for accessing aggregated analysis results and for tuning the system's configuration.

Ease of Maintenance. Replacing failed components should be a straight-forward operation that does not impair the analysis of other components unaffected by the defect. If the hardware setup allows, hot-spares should be able to automatically take over.

Driven by these design objectives, we architect the NIDS cluster in terms of four main components (see Figure 1): *frontend nodes* distribute the traffic to a set of *backend nodes*. The backends run the NIDS instances that perform the traffic analysis and exchange state via *communication proxies*. Finally, a central *manager node* provides the cluster's centralized user interface.

There is typically one frontend per monitored link. Each frontend forwards each packet it receives to exactly one backend node in charge of the packet. Frontend nodes operate at line-speed and are therefore the most performance-critical components of the cluster. In the next section we discuss their operation and different options for implementing them.

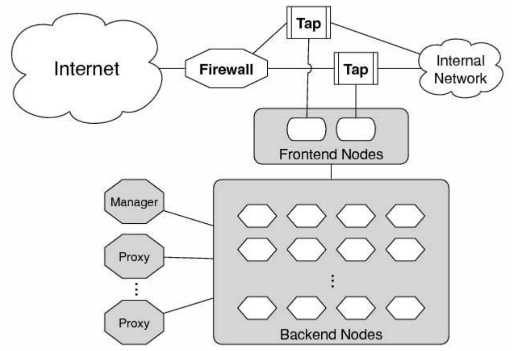

Fig. 1. Cluster architecture.

Backend nodes are commodity PCs with two NICs: one for receiving traffic from the frontend, and one for communication with the manager and the communication proxies. Each backend node analyzes its share of the traffic using an instance of the NIDS, forwards results to the rest of the cluster as necessary, and receives information from other backends required for global analysis of activity. All backend nodes are using the same NIDS configuration and thus perform the same analysis on their traffic slices.

The communication proxies are a technical tool to avoid direct communication among all backend nodes, which does not scale well due to requiring fully meshed communication channels. Often a single proxy node suffices. It connects to each backend, forwarding information it receives from any of them to the others.

The manager node provides the operator-interface to the cluster. It aggregates and potentially filters the results of the backend nodes to provide the user with a coherent view of the overall analysis. The manager also facilitates easy management of the cluster, such as performing configuration updates and starting/stopping of nodes.

While conceptually proxies and managers are separate entities, we can combine them on a single host or co-resident with a backend system if their individual workloads permit (as is the case for some of our current installations).

3 Distributing Load

We begin with the question of how to divide the network traffic across the cluster's backend nodes. We first discuss and evaluate different distribution schemes, and then present options to implement them in the frontend nodes, as well as implications for coping with failure.

3.1 Distribution Schemes

The most straight-forward approach by which the frontend can achieve an even distribution of the incoming traffic would be a packet-based, round-robin scheme:

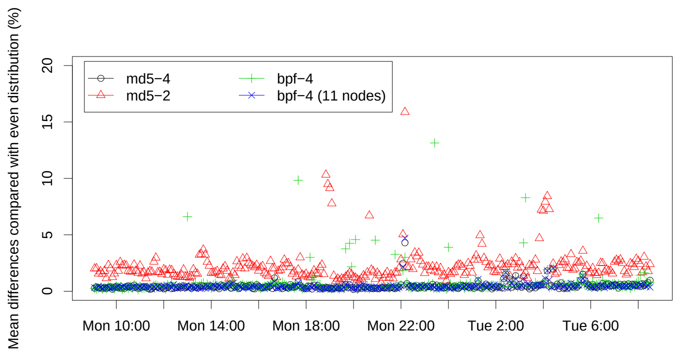

Fig. 2. Hash simulation.

each node will see the same number of packets, arriving at the same rate. However, all major NIDSs keep significant *per-flow state*, e.g., to facilitate reassembling TCP byte streams, and thus a packet-based scheme would entail major communication overhead.

Flow-based schemes (all packets belonging to the same flow go to the same backend node) hold more promise as they result in load-balancing at the same granularity as used by the lower layers of backend analysis. Such schemes avoid inter-node communication for any analysis which is limited to the scope of a single flow—by far the largest share in most NIDSs. For example, traditional Snort-style signature matching requires no communication if we employ flow-based distribution. In addition, recent research suggests that the resource usage of a NIDS scales primarily with the number of flows [4], and thus a flow-based distribution scheme should impose a similar processing load on all backend nodes. We assess this claim in § 5.

To keep the frontend nodes as simple as possible, we focus on stateless distribution schemes. A simple approach is hashing a *flow identifier* derived from each packet's header into the set $\{0, \ldots, n-1\}$, with n being the number of backend nodes. For TCP and UDP traffic, for example, the identifier might be the 4-tuple $(addr1, port1, addr2, port2)$. We could then use MD5 to generate a hash from such tuples: $h_{md5_4}(addr1, port1, addr2, port2) := MD5(addr1 + port1 + addr2 + port2) \mod n$. By using addition, this hash is commutative with respect to a flow's source and destination, so we map all packets of a flow in both directions to the same backend.

Figure 2 shows the result of applying several such hashes (for $n = 10$) to a day's worth of TCP traffic at UCB (231 million connections). Let N_i be the number of flows that began during each 5-minute interval i. An ideal distribution scheme across n backends assigns $M_i := N_i/n$ flows to each backend. For each interval, the plot shows the mean differences between a hash-generated distribution and M_i, as a percentage of M_i. As we would expect, we see that the h_{md5_4} hash (black circles) performs very well: the standard deviation σ of its variation from the ideal M_i is just 0.35%.

However, calculating MD5 for every packet is expensive. We therefore explore a simpler additive hash, $h_{bpf_4}(addr1, port1, addr2, port2) = (addr1 + port1 +$

addr2+*port2*) *mod* *n*.[4], shown in Figure 2 using green crosses. This scheme has $\sigma = 1.31$, not as good as for h_{md5_4}, resulting in a few more outliers. However, it turns out that the outliers arise due to our choice of $n = 10$: for such an additive hash, a non-prime modulus can lead to aliasing. If we instead use $n = 11$ (blue crosses in the figure), then the outliers disappear, and σ falls to 0.36, yielding essentially the same level of performance as does h_{md5_4}.

Yet, while hashing on the 4-tuple yields a nice distribution of flows across backends, it also has drawbacks. First, it relies on port numbers, which are not always well-defined (e.g., ICMP packets). Second, extracting the port numbers can be somewhat complicated for fast hardware (per § 3.2) since their location within a packet is not necessarily fixed (due to IP options or nested IPv6 headers). Third, and most importantly, 4-tuple hashing cannot cope with IP fragments, as not all fragments contain the ports. Thus, a frontend would need to reassemble fragments, which requires maintaining state.

Accordingly, we also examine a third type of hash based *only* on addresses: $h_{md5_2}(\text{addr1}, \text{addr2}) = md5(\text{addr1} + \text{addr2}) \mod n$. While this hash is easy to implement, we expect it to lack the evenness of the 4-tuple hashes, since all traffic between the same two hosts will map to the same hash. The question is whether in a large network the diversity of source and destination addresses already suffices to still yield similar loads on the backend nodes. We show the results in the figure using red triangles, finding that while h_{md5_2} has $\sigma = 1.55$, the unevenness is fairly mild. (We omit similar results for an analogous h_{bpf_2} hash to keep the plot legible.)

In addition, there are in fact some significant benefits to using 2-tuple hashing rather than 4-tuple: hashing just by addresses decreases communication overhead for per-host-pair forms of analysis. For example, a single cluster node can detect port scans without any inter-node correlation. In our cluster installations we therefore choose to rely on a 2-tuple hash despite its slightly lower performance. Accordingly, we also use it for our evaluation in § 5.

Finally, we note that in practice the load of a NIDS is quite difficult to predict even if we know exactly what traffic it processes [5]. Thus, even with a completely even distribution scheme the actual *processing* load on the backends differs. We return to this point in § 5.

3.2 Frontend Implementation

We can consider implementing the frontend using either specialized hardware, or purely in software. Regarding specialized hardware, we experimented with the P10 appliance from Force10 Networks, which features two 10 Gbps ports and the ability to draw upon an FPGA to inspect traffic across the ports at line rate. We programmed the FPGA to calculate hashes and in real-time rewrite the destination MAC address of each packet to be that of the corresponding backend

[4] We call this hash h_{bpf_4} because it can be implemented as a BPF filter [4].

node. We report on our experiences in § 6 below.[5] Alternatively, we find in our testing that software frontends, while slower, can provide sufficient performance for up to 2 Gbps. The most successful approach we have experimented with is based on Click [7] which, running entirely within a Linux kernel, can similarly rewrite the destination addresses.

3.3 Recovery From Failure

One of the design objectives for the NIDS cluster is *Ease of Maintenance*: if a component fails, it should be simple to replace it, ideally in an automated fashion and without interrupting unaffected components. For the front end, this is difficult without employing fully redundant hardware. However, for the other components, we can do so as follows.

If a backend node fails, the immediate effect is that its slice of the network traffic becomes unmonitored. All other components, however, continue to work unaffected. To prepare for backend failures, we can run additional nodes as hot spares. During normal operation, hot spares are configured the same as the other backends, but do not receive any traffic. Once the cluster detects the failure of a backend (e.g., via a heartbeat mechanism), a hot spare can assume the MAC address of the failed node and continue the node's monitoring. While the hot spare will not have the internal state of the failing node, it automatically receives a copy of all globally-shared state from its proxy when it connects to the cluster.

If a communication proxy fails, the backends connected to it will no longer be able to correlate their analysis with other nodes. However they continue analyzing their traffic slices locally, including further accumulation of local state. A hot-spare proxy then take over by connecting to all affected backends, which then automatically resume propagating their state updates via the new connections. However, the system will be in a state of partial inconsistency due to the state updates lost during the fail-over period.

If the manager fails, the cluster loses its reporting and logging facilities, but the backends and proxies continue their monitoring unaffected. If they also log locally, there is no loss of analysis results, even though they are no longer aggregated centrally during the manager outage. Once detected, a new manager can quickly takeover: like a proxy, it only needs to connect to the backends and they will automatically begin forwarding their results to the new manager.

4 Distributing Analysis

We now turn to devising the cluster's backend analysis. As noted above, we base our implementation on the Bro NIDS. Bro's flexibility makes it well-suited to the task. In contrast to most other NIDSs, Bro is fundamentally neither anomaly-based nor signature-based. Instead, it is partitioned into a *protocol*

[5] We have also developed an FPGA-based NIDS frontend that we term a "Shunt" that we can adapt to this purpose [6], but we do not yet have it at a stage to evaluate in this context.

analysis component ("event engine") and a *policy script component*. The former feeds the latter by generating a stream of *events* that reflect different types of activity detected using protocol analysis. For example, when the analyzer sees the establishment of a TCP connection, it generates a `connection_established` event; when it sees an HTTP request, it generates `http_request`, and for the corresponding reply, `http_reply`.

Bro's event engine is *policy-neutral*: it does not consider any particular events as reflecting trouble. It simply makes the events available to the policy script interpreter. The interpreter then executes scripts written in Bro's custom scripting language in order to define the response to the stream of events. Because the language includes rich data types, persistent state, and access to timers and external programs, the response can incorporate a great deal of context in addition to the event itself. The script's reaction to a particular event can range from updating arbitrary state (for example, tracking types of activity by address or address pair, or grouping related connections into higher-level "sessions") to generating alerts.

Almost all of Bro's event engine processing executes on a per-flow fashion, and thus does not require correlation of state across flows.[6] Therefore, we can restrict exchange of analysis state between backend nodes to script-level logic.

At the script-level, Bro provides extensive support for remote communication by means of its *independent state* framework [3]. The framework provides two communication primitives: *remote event subscription* and *synchronized variables*. With the former, one Bro instance can subscribe to any events from the event stream of remote peers; it will then transparently receive these events at its script-layer just as does the event source itself. With the latter, Bro instances can share *any* script-level data structure with a set of peers. Any change performed by one peer is transparently propagated to the others, creating a globally shared data structure. For our cluster, we use event subscription for the communication between the manager node and the backends, and synchronized variables for correlating analysis between backends.

To install Bro on the NIDS cluster, we need to set up three different types of Bro instances: *(i)* backends to analyze each slice of traffic; *(ii)* proxies to propagate state information across the backends; and *(iii)* a central manager to collect and aggregate the results of the backends.

Managers and proxies are straight-forward to construct. The manager is a Bro instance that connects to each backend and subscribes to the events corresponding to their analysis output: *alarm* events, generated upon detecting malicious activity, and *log* events, generated whenever the backend logs information to a local file.[7] When the manager receives a remote *alarm* event, it processes it according to its local alarm configuration (e.g., determining which

[6] There is one exception: Bro's stepping stone detector [8] correlates flows *inside* the event engine. However, this analysis could just as well be done at the script layer, and so for our evaluation we have decided to ignore supporting its functionality.

[7] Bro differs from many other NIDS in that it keeps extensive logs of all network traffic, independent of whether activity is deemed malicious or not. These logs include one-

merit paging the operator), just as if the alarm had been locally generated. This approach allows the operator to easily centralize (and dynamically change) the alarm policy for the entire cluster. When the manager receives a *log* event, it writes the content into a corresponding local file; thus, the manager's log files reflect the aggregation of all of the backends' log files in real-time.

Different from the manager, the proxies operate in a fully policy-neutral fashion. Each proxy connects to a subset of the backend nodes, as well as to all other proxies. Proxies subscribe to the stream of state operations from all of their peers. Once they receive an operation from a peer, they forward it to all of the others. The receivers then apply the operation locally (or propagate the operation further in the case of multiple proxies).

Setting up the backend Bro instances consists of two step: *(i)* choosing an analysis configuration suitable for the environment, and *(ii)* adapting the processing to correlate state across the cluster. The first step does not differ from setting up a traditional, single Bro instance, and we therefore do not discuss it in more detail. With regard to correlating state, per our observation above, Bro performs its inter-connection analysis (almost) exclusively at the script-level, and thus we focus on identifying script-level variables that require synchronization across the backends. To do so, we examined each of Bro's standard scripts to determine which variables require synchronization between peers. By providing this synchronization, each peer obtains the *full* decision context while processing only a subset of the entire traffic.

Our analysis of the scripts revealed that many of them in fact perform only intra-connection analysis, and hence do not require any modification. In particular, most of the scripts analyzing the content of application-layer protocols do not correlate information across connection boundaries. (For example, while Bro's primary SMTP script maintains a table of all active SMTP sessions, the analysis of each individual one does not require access to the state of any other SMTP session.)

Some scripts, however, do require information from multiple connections. A prominent example is the scan detector, which counts connection attempts per source address. If these reach a certain threshold, the system raises an alarm. In the cluster setup, the scan detector now must count across backends; we therefore synchronize the corresponding tables of counters (which simply entails annotating the corresponding script variables with the attribute `&synchronized`). Other examples of scripts needing synchronization are the worm detector (which maintains a global list of infected hosts) and the SMTP relay detector (which identifies open SMTP relays by associating incoming with outgoing mails). Overall, we needed to synchronize 29 script-level variables spanning 19 different types of analysis.

When adapting the scripts in this fashion, we sometimes leveraged the specific traffic distribution schemes implemented by our frontends (see § 3.1). Since the 2-tuple scheme we used directs all traffic between the same two hosts to a

line summaries of each flow, and transcripts of application-layer protocol dialogs for a wide range of protocols—invaluable for forensic analyses.

single backend, we do not need to synchronize state that only associates connections between the same two endpoints (for example, detection of port scans). However, since this optimization depends on the frontend distribution scheme, we structured our script modifications so that the user can selectively apply them based on *a priori* knowledge of how traffic will be distributed.

In general, there are trade-offs between the overhead that synchronization requires and the benefits one gains from it. For example, FTP data connections are *usually* instantiated between the same endpoints (addresses) as the corresponding FTP control connections. When that holds, a purely address-based distribution scheme obviates the need for inter-node communication. However, in principle FTP can involve a third address in such transfers, and thus not synchronizing knowledge between nodes can potentially lead to misclassifications. In our current configuration we still choose to *not* propagate such information in favor of avoiding the communication overhead. (Note that some, but not all, of the attack uses of such third-party FTP already manifest in the control session, and thus can be detected without synchronization.)

To adapt the backend analysis to the cooperative cluster setting, ideally it would suffice to simply go through all of Bro's analysis scripts and synchronize all variables used to correlate state across flows. However, in practice we encountered subtleties when doing so which merit discussion.

So far, we have assumed that synchronizing a variable works in a fully transparent and reliable fashion: at all times all peers agree on the variable's value; each operation is immediately reflected in all instances. However, in practice real-time requirements impede this model, as it would require global locking across mutually-exclusive data structures. The Bro system therefore relies on *loose synchronization* [3], which propagates state changes in a best-effort fashion without any kind of locking. Doing so can lead to race conditions, and therefore changes the semantics of the script processing.

While we cannot avoid such race conditions, we can mitigate their impact. We devised several strategies to do so, addressing the common situations we encountered. One example arises from the specific way in which some scripts use nested tables. The scan detector, for example, uses a table of source addresses mapping into sets of destination addresses to count how many unique victims a scanner has so far contacted. When a new source begins to scan, there is initially no entry for it in the outer table. However, the first connection attempts the source makes are often noticed by multiple backends at almost the same time, and thus each of them then assigns a freshly instantiated set of destination addresses to the corresponding source address. Because of loose synchronization, it is unpredictable what sort of picture these fresh creations plus additions will eventually result in at each peer. We addressed this problem by introducing *mergeable* sets into the scripting language: if new content is assigned to a mergeable set, the distributed result is the *union* of old and new content, rather than the new content replacing the old.

A number of other such examples arose; see [9] for a detailed discussion. (We note that all of our changes, including the mergeable sets, will be part of the next Bro release.)

Finally, we equipped the manager node with a set of tools to ease maintenance. Rather basically, a set of shell scripts provide means to control and monitor the cluster operation on all nodes (e.g., start and stop the cluster), based on a central configuration file that specifies which systems are backends, proxies, and manager. More interesting is on-the-fly reconfiguration, such as installing a modified alert policy, which such scripts also support via the *independent state* framework. Similarly, the operator can inspect the state of all nodes during runtime, e.g., to examine the contents of script-level variables on all of the backends. The scripts for doing so run a temporary Bro instance on the manager node that connects to all the cluster nodes, either sending out a configuration update or querying for internal state.

5 Evaluation

We evaluate the performance of our cluster setup as follows. We first first introduce our evaluation methodology. We then use it to assess the analysis accuracy of the cluster in comparison with running a single system. To understand how the cluster design scales, we next measure backend CPU load as we vary the number of nodes. Finally, we look at the overhead due to inter-node communication.

5.1 Methodology

For the bulk of our evaluation, we operate the system offline on previously captured traces. In contrast to running on live traffic (including traffic replayed into a testbed), this approach ensures reproducible results with multiple runs. When running multiple times on the same trace with different configurations, we can attribute any changes in performance directly to the configuration change.

However, the cluster's distributed processing introduces a few complications. First, as each backend node sees a different slice of traffic, we need one trace per backend node. Thus, we first capture a full trace and then split it up into slices using the h_{bpf_2} hash scheme discussed in § 3.1. We then copy one slice to every backend node and run each NIDS instance offline on its subset of the trace. The second difficulty arises due to inter-node communication: the NIDS, running offline, can process a trace more quickly than in real-time—since the nodes consume packets as fast as possible, even if in actuality the packets would not yet have arrived—leading to desynchronization between the backend nodes. To address this problem, the Bro system provides a *pseudo-realtime* mode [3]: if enabled, the trace analysis deliberately inserts delays into its processing that echo the inter-packet gaps observed when capturing the trace. That is, the analysis of the trace proceeds with the same speed as it would when running live, thereby synchronizing it with real-time, and hence with inter-node communication.

We conducted our evaluation in the LBNL environment, using the same cluster hardware that our live prototype runs (§ 6). The cluster consists of 10 backend nodes and one node for manager and proxy. All systems are 3.6 GHz dual-CPU Intel Pentium D with 2 GB RAM. We use a single Bro instance as both manager and proxy, and configured the backends to reflect the full operational Bro setup used at LBNL (the complete range of analysis that to date has been spread across a number of uncoordinated nodes). In addition, we enabled Bro's general capability to detect and analyze applications running on non-standard ports [10], which is infeasible for LBNL's current operational setup because it requires in-depth analysis of *all* packets.

We captured a 2+ hour, full-packet trace around noon on March 1st, 2007, comprising 97 GB, 134 M packets, and 1.5 M source/destination pairs. The average throughput corresponds to 113 Mbps, with a per-second peak of 261 Mb. 88% of the packets were TCP, 9.7% UDP, and 2.3% ICMP. The most prevalent TCP protocols were HTTPS (18.6% of the packets), followed by HTTP (12.0%), and SSH (10.3%). 40.8% of the TCP traffic was not classifiable by well-known ports, with a large share quite likely due to Grid protocols.

5.2 Accuracy

Ideally, a NIDS cluster produces the same output as a single NIDS would. Therefore, we first compared our cluster's output (as aggregated by the manager node) with the results of a single Bro instance running offline on the full trace. We examined both the *alarms* and the activity *logs* generated (per § 4).

Of the 2661 alarms reported by the single Bro, all were also raised by the cluster, i.e., the new setup does not miss any intrusion attempts. Upon closer inspection, we however found two differences. First, of the 252 address scans reported by the single Bro, two scanners were flagged significantly later by the cluster. The first scanner performed a quick but extremely short scan, contacting 39 different destinations within 1 sec but then no further contact for an interval exceeding the scan detector's time-out. While a single Bro can notice such a scan easily, the latency of the communication between the backends delays the cluster in doing so. The second initially missed scanner performed 5 small bursts of connection attempts: roughly 10 attempts each time, set 10 sec apart. This was only detected when the backends later generated summaries of their shared state (rather than upon propagation of the shared state, due to the activity-triggered nature of the current scan detection algorithm). In both cases, the final alarm generated by the backend agrees with the summary produced by the single Bro.

The second difference arose because the details of context accompanying an alarm can differ. Timestamps vary slightly (on the order of 0.1 sec) and, for example, the scan detector can report a different number of connections attempts in its initial alarm, with both effects due to communication delays and semantics slightly differing due to the distributed setup. Apart from these minor differences, we find that the cluster produces alarms closely matching those of a single Bro processing the same input.

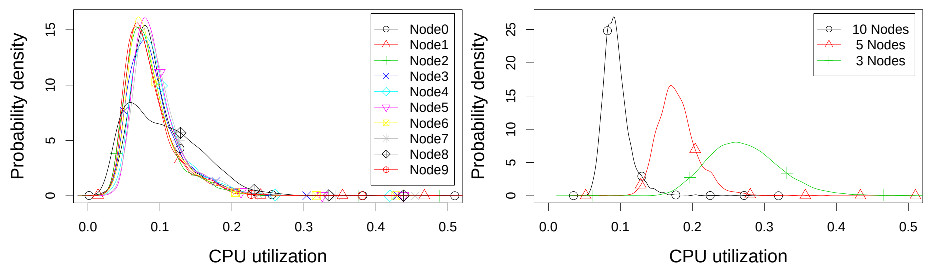

Fig. 3. Probability densities of backends' CPU load (left), and probability densities for varying numbers of backends (right).

Next, we compared the activity logs. The main discrepancies we encountered were differences in timing, e.g., the begin and duration logged for connections differed slightly. This is expected due to the pseudo-realtime mode, which can only approximately reproduce the exact timing of the trace. (Timing would similarly differ in a live setting.) Except for such timing issues, we found only one other major type of difference between the logs, which was also an artifact of our test-bed setup: when the Bros on the backends terminate, they generate a final spike of log activity. However, as Bro tears down the communication to the manager immediately at that point, the corresponding log events are not reliably forwarded anymore. Thus, the manager is missing some of the activity at the end of the processed trace. In a live setting, this problem does not occur because the nodes run continuously.

Overall, we conclude that the cluster yields very similar results as a single NIDS—well within the variation we see operationally for a single NIDS due to differences in timing and minor configuration variations—and therefore achieves an acceptable degree of transparency.

5.3 Performance

We now assess the performance of the NIDS cluster in terms of CPU load and communication overhead. We first examine how well our frontend balances the processing load across the backends. We then perform a series of measurements with different numbers of backends to assess the scalability of the approach. Finally, we take a look at the overhead introduced by the communication.

Load-balancing. In § 3.1 we found that overall the h_{bpf_2} hashing scheme yields a good distribution in terms of the number of flows assigned to each node. However, even assignment does not automatically imply even *processing* loads on all backends, as different types of connections require different degrees of analysis (see [5]). To examine the backend CPU load, we again run the cluster on the captured trace, using the same configuration as described above. For each backend, we logged the amount of user CPU time consumed per second

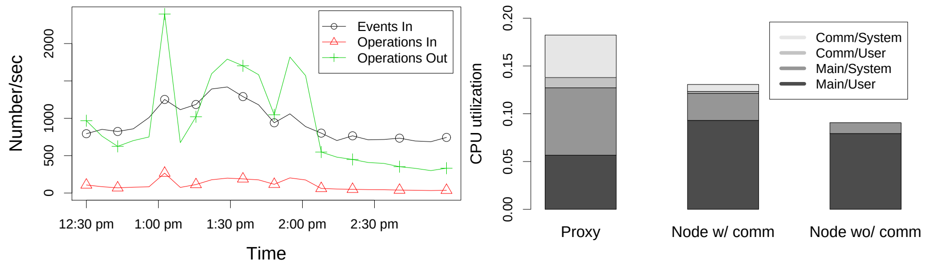

Fig. 4. State exchanged by manager/proxy (left), and CPU load of manager/proxy & one node (right).

by the NIDS's analysis. Figure 3(left) shows the distribution of these per-second load samples for each backend. We see that nine of the ten backends (all except node 8) show very similar distributions, indicating quite similar CPU loads. Across these nine backends, the *largest* mean CPU utilization was 10.0%, and the largest standard deviation $\sigma = 4.8\%$, reflecting that both the loads and the load fluctuations leave ample headroom for increases in traffic.

However, backend node 8 shows a notably different density shape (mean 10.7%, $\sigma = 5.7\%$). We examined the slice of the trace processed by node 8 and found that the slice contains a *single* TCP connection which makes up 86% of the trace's total bytes (33 GB of 38 GB!). Just by being assigned this one connection, node 8 receives a significantly larger share of the overall traffic (other nodes on average received 6.5 GB). Note, though, that pretty much any flow-based traffic distribution scheme will wind up introducing this disparity, since it manifests at even the finest flow-based granularity. However, even so node 8's CPU load 8 stayed well within a manageable range (below 30% for 99.5% of the time).

We conclude that overall our traffic distribution imposes quite similar processing loads across the nodes, and that the 10-node setup has sufficient headroom to easily cope with the occasional traffic spikes induced, even when performing the full range of operational analysis *plus* dynamic protocol detection.

Scaling. We next examine how the backend load scales with the number of analysis nodes. Figure 3(right) plots the CPU utilization for setups with 3, 5, and 10 backends. For each run, we first averaged the one-second CPU samples (see above) across all nodes, and then plotted the probability density of these mean CPU loads. In the plot we see that the load indeed scales nearly linearly with the number of nodes: the mean load for 3 nodes is 27.4%, for 5 nodes it is 18.0%, and for 10 nodes it is 9.4%, with the corresponding values of σ being 5.5%, 3.0%, and 2.0%.

Overhead. Compared to running a single Bro instance, the cluster setup introduces overhead in terms of communication. We now examine the volume of state exchanged between the cluster nodes and the additional amount of processing it requires. All measurements reported in this section use 10 backend nodes.

We first look at the amount of state exchanged between the cluster nodes. For the combined manager/proxy node, Figure 4(left) shows the number of incoming and outgoing state operations as well as the number incoming events (this node does not generate its own events). As *alarm* events are relatively rare, almost all of the incoming events are *log* events reflecting summaries of transport-level and application-level activity.

On average, one log event consumes 200 bytes when transmitted wire in its binary form. Incoming state operations correspond to updates to synchronized variables; each of these triggers 9 outgoing operations due to the proxy broadcasting the update to the other backends.[8] On average, one state operation consumes about 140 bytes.

Examining the state operations in more detail reveals that by far the largest fraction (97%) are triggered by the scan detector, unsurprising because scan detection is naturally quite expensive in terms of communication (to first order, *each* connection might be part of a scan and thus needs to be propagated).

To understand the processing burden that propagating events and operations imposes on the cluster nodes, Figure 4(right) shows the average CPU load over the course of our trace for *(i)* the manager/proxy, and *(ii)* an arbitrary backend node with and without any communication. For the proxy, we see that a significant amount of the processing time (11.5%) is system time. Apart from logging to disk, this time primarily reflects communication input/output: over the course of the trace, the proxy sends in total 101/918 MB in/out.

The mean CPU time consumed by the proxy is rather low in our evaluation (6.3%). However, as the proxy cannot do a real broadcast but has to individually send each operation to every receiver, its CPU usage increases with the number of backends. Depending on the traffic, this could in principle cause the proxy to become a bottleneck, especially during traffic spikes that suddenly generate a large number of events/operations. Yet, due to the flexibility of our cluster architecture, we can easily divide the load between multiple proxies. In our current installation (see § 6), we in fact run two proxies, and also separate the manager so that logging and operations broadcasting can be performed on different hosts.

Looking at the exemplary node in Figure 4(right), we see that enabling communication increases its mean total CPU usage by 42.9% (though still to below 15% in absolute terms). In fact, 23% of the increase occurs in a child process that Bro uses to manage inter-Bro communication; thus, on a dual CPU machine this portion does not decrease the processing capacity of the main process. Overall, the overhead for a node's communication is non-negligible but also is not dominant. Furthermore, due to the proxy architecture the amount of communication that a node performs is independent of the number of backend nodes, providing good scaling properties.

[8] Due to Bro's communication framework using TCP, this is not a network-layer broadcast.

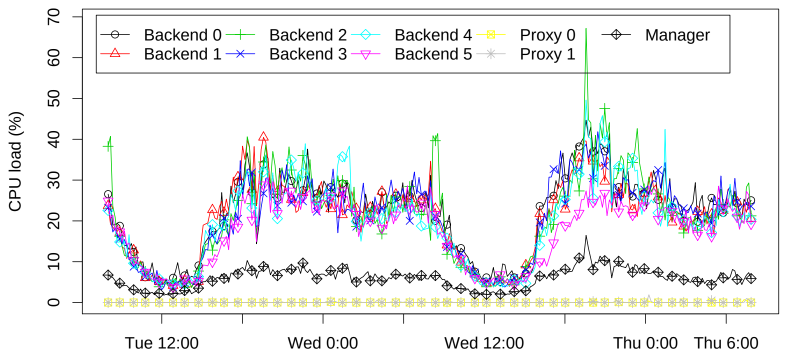

Fig. 5. CPU load on UC Berkeley cluster.

6 Installations

We have installed operational NIDS clusters at LBNL and UC Berkeley, which here we discuss in turn.

The Bro cluster at LBNL consists of one frontend (classifier) node, one node each for the manager and communication proxy, and ten backend nodes. Each is a 3.6 GHz dual-CPU Intel Pentium D with two GB of memory and two GigE network interface ports, one for packet capture and one for communication. Optical splitters provide copies of each direction of wide area traffic. Since LBNL's current aggregate utilization is less than 10 Gbps, we merge these into a single 10 Gbps stream, fed into a Force10 P10 appliance. The P10 classifies the packets according to a variant of the h_{bpf_2} hash (which uses xor rather than addition) and injects them into a 10 Gbps uplink port on a Force10 S50 switch. The switch distributes the packets to GigE-connected analysis nodes according to their rewritten MAC addresses.

We run the manager and communication proxy each on a dedicated node. The manager collects log files from all backends, archiving them for forensic analysis, and responding to real-time alarms. In typical operation, backend nodes consume less then 2% CPU for packet analysis and less than 1% CPU for communication, the manager consumes around 5% CPU, and the proxy node consumes around 2% CPU. We have seen bursts of traffic consume up to 40% CPU on the backends, 25% CPU on the proxy, and 15% CPU on the manager for short periods of time. The backends report very little packet loss (less than .0001%). On average we monitor 32K pkts/sec and 28 MB/sec of traffic on this cluster.

The Bro cluster at UC Berkeley monitors the campus's two 1 GigE upstream links, which are mirrored via SPAN ports from two separate routers. There are two frontend nodes running Click to distribute the traffic (Dell PowerEdge 850; Intel Pentium D 920 dual-core; Linux 2.6), one for each SPAN port; and currently six backend nodes (Sun Fire X2100; AMD Opteron 180 dual-core; FreeBSD 6.1). An HP ProCurve 3500 switch connects frontends and backends.

The traffic volume seen at UCB is huge, 3–5 TB per day. As our six backends do not suffice to analyze the total traffic in full, until we can add more nodes we limit the analysis to half of the traffic volume by enabling only one of the frontends. We use two proxy instances to balance the communication load. The proxies, as well as the manager, run in addition with the traffic analysis on one of the backends each. For the manager, it appears that its disk I/O decreases the analysis capacity of the backend process running on the same host (we see occasional packet drops at similar loads that the other backends have no trouble with). Figure 6 shows the processing load of the different processes over a time period of two days. The specifics of the UC Berkeley network posed some challenges. First, Bro's scan detector imposed significant load on the cluster due to the large number of connections in this environment (about 2500/sec on average). On the one hand, these generate large numbers of propagated state operations. On the other hand, counting connection attempts for all sources requires a great deal of memory. The latter highlights a drawback of our approach to clustering: while we split the CPU load across the multiple nodes, each backend keeps a complete copy of all (synchronized) state. To counter both effects, we added two new options to Bro's scan detector: the first limits the synchronization of scanner state to sources for which one of the backends has at least seen 5 (default) different destinations locally. The second option stops synchronizing scanner state for sources once they have scanned at least 500 (default) destinations. With this tuning, the scan detector performs well on the cluster.

We encountered a similar problem with Bro's IRC analyzer, which tracks a significant amount of state for each IRC user encountered. Being a campus network, the share of IRC traffic is relatively large, and therefore these data structures grow quickly. Since they are synchronized, each backend keeps its own copy of the full set. For now we have disabled parts of the IRC analysis in favor of having the memory available for other types of analysis.

More generally, these problems highlight how existing ways of structuring analyses are not always well-suited for a distributed setup. With the cluster platform now in place, we plan to investigate analysis algorithms specifically designed for multi-node processing. For example, a distributed, probabilistic scan detector has the potential to significantly reduce communication and memory requirements.

7 Related Work

To our knowledge, the approach we have framed in this work—employing a cluster of commodity systems to perform load-balanced intrusion detection that coordinates lower-level analysis across nodes—is a novel development. That said, the more general notion of applying clusters to construct scalable network services has seen significant exploration in prior work. Fox et. al. mention several advantages clusters provide, including *incremental scalability*, *high availability*, and the *cost-effectiveness* of commodity PCs [11]. The performance of network intrusion detection has been extensively studied in the past [12–15]. All studies conclude that it is imperative to cope with the induced load that growing network traffic imposes. Schaelicke and Freeland argue that system-level optimizations such as *interrupt coalescing* and *rule-set pruning* as well as architectural techniques can significantly improve performance and reduce packet loss [15]. While previous work primarily focuses on the design of a NIDS cluster processing frontend [14, 16], we look in addition at the challenges that intra-NIDS communication raises.

Numerous different NIDSs are available today. The focus and range of application vary for each system. To our knowledge, only a few systems feature a tunable and flexible communication sub-system that we can leverage to build a NIDS cluster. Snort [17] is the most widespread open-source NIDS. Snort runs on commodity hardware, utilizing *libpcap* to enable platform independent packet capturing. The detection engine is misuse-based. Around a core of numerous signatures, various plugins enhance its functionality. Despite the lack of a communication sub-system, Kruegel et. al. built a flow-based load-balancer on top of Snort [16]. Their approach maintains connection tables to forward packets belonging to the same flow to the corresponding sensor, but does not extend to inter-sensor communication. The *State Transition Analysis Technique (STAT)* tool suite [18] is a set of distributed intrusion detection tools based on misuse-detection. STAT models intrusions as sequences of attack scenarios reflected by state transition diagrams, and supports inclusion of network-based, host-based, and application-based sensors. The *MetaSTAT Infrastructure* [19] provides the communication sub-system and control infrastructure to enable distributed coordination of STAT-based applications. STAT-based tools fan out into $\{U,N,Net,Win,Web,Alert\}STAT$, each designed for a different application domain. In particular, *NetSTAT* [20] is the network-based component responsible for network communication. If it is impossible for the system to detect an attack completely, the NetSTAT propagates the partially configured scenario containing state information to other probes. EMERALD [21] is a distributed hybrid intrusion detection framework designed for large-scale enterprise network operation; it is not openly available. The architecture of EMERALD uses a layered approach to support hierarchical organization of monitors. Each monitor can subscribe to events and propagate correlated results. Prelude [22] is a distributed NIDS that relies on the IDMEF [23] standard to exchange events. In its framework, sensors are connected to managers, which process and correlate alerts. In a distributed setup, multiple managers can also act as relay managers that report to a central manager. However, none of the existing approaches provided a sufficiently flexible means to share arbitrary policy-neutral state, unlike the approach we pursue with our NIDS cluster.

8 Conclusion

In this work we set out to build a *NIDS cluster* as a scalable solution to realizing high-performance, stateful network intrusion detection on commodity hardware. The cluster consists of a frontend that distributes traffic evenly across an extensible set of backend nodes. Each backend examines its slice of the traffic in-depth and correlates its analysis with the rest of the cluster. Different from traditional multi-system NIDS setups, our cluster exchanges *low-level* state information across all the backends and thereby transparently creates the impression of interacting with a single NIDS.

In the process of developing the NIDS cluster, we examined different traffic distribution schemes for the frontend and experimented with both hardware and software implementations. We adapted the open-source Bro NIDS to run on the backends, and conducted a trace-based evaluation of the cluster to ensure that the cluster achieves *transparency* (output matches that of a stand-alone system) and good performance with respect to scalability and communication overhead.

A prototype of the cluster runs at the Lawrence Berkeley National Laboratory in parallel with the site's operational security monitoring, which it will eventually replace. Another prototype monitors the access links of UC Berkeley. With the cluster infrastructure now in place, we plan to further investigate the development of analysis algorithms specifically tailored for a distributed setting, allowing us to decrease communication overhead. Even without this, the NIDS cluster already increases the computational power of network security analysis far beyond what is currently feasible in these environments.

9 Acknowledgments

We would like to thank Mark Dedlow for testing and improving the cluster configuration, Christian Kreibich for valuable feedback, Livio Ricciulli and Force10 for implementing our hashing scheme on their P10 appliance, and Nicholas Weaver for his help with setting up the UCB cluster. Parts of this work were produced under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College, and supported under Award number 2003-TK-TX-0003 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security, the Science and Technology Directorate, the I3P, or Dartmouth College. This work was also supported by NSF Awards STI-0334088, ITR/ANI-0205519, and CNS-0627320. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

References

1. Paxson, V., Asanovic, K., Dharmapurikar, S., Lockwood, J., Pang, R., Sommer, R., Weaver, N.: Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In: Proc. USENIX Hot Security. (2006)
2. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks **31**(23–24) (1999) 2435–2463
3. Sommer, R., Paxson, V.: Exploiting Independent State For Network Intrusion Detection. In: Proc. Computer Security Applications Conference. (2005)

4. Dreger, H.: Operational Network Intrusion Detection: Resource-Analysis Tradeoffs. PhD thesis, TU München (2007)
5. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational Experiences with High-Volume Network Intrusion Detection. In: Proc. ACM Conference on Computer and Communications Security. (2004)
6. Weaver, N., Paxson, V., Gonzalez, J.M.: The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. In: Proc. ACM Symposium on Field Programmable Gate Arrays. (2007)
7. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, F.: The Click Modular Router. ACM Transactions on Computer Systems **18**(3) (2000)
8. Zhang, Y., Paxson, V.: Detecting Stepping Stones. In: Proc. USENIX Security Symposium. (2000)
9. Vallentin, M.: Transparent Load-Balancing for Network Intrusion Detection Systems. Bachelor's Thesis, TU München (2006)
10. Dreger, H., Feldmann, A., Mai, M., Paxson, V., Sommer, R.: Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In: Proc. USENIX Security Symposium. (2006)
11. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P.: Cluster-Based Scalable Network Services. In: Proc. Symposium on Operating Systems Principles. (1997)
12. Puketza, N.J., Zhang, K., Chung, M., Mukherjee, B., Olsson, R.A.: A Methodology for Testing Intrusion Detection Systems. IEEE Transactions on Software Engineering **22**(10) (1996) 719–729
13. Schaelicke, L., Slabach, T., Moore, B., Freeland, C.: Characterizing the Performance of Network Intrusion Detection Sensors. In: Proc. Symposium on Recent Advances in Intrusion Detection. (2003)
14. Schaelicke, L., Wheeler, K., Freeland, C.: SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In: Proc. Computing Frontiers Conference. (2005)
15. Schaelicke, L., Freeland, C.: Characterizing Sources and Remedies for Packet Loss in Network Intrusion Detection. In: Proc. IEEE Symposium on Workload Characterization. (2005)
16. Kruegel, C., Valeur, F., Vigna, G., Kemmerer, R.A.: Stateful Intrusion Detection for High-Speed Networks. In: Proc. IEEE Symposium on Research on Security and Privacy. (2002)
17. Roesch, M.: Snort: Lightweight Intrusion Detection for Networks. In: Proc. Systems Administration Conference. (1999)
18. Vigna, G., Eckmann, S.T., Kemmerer, R.A.: The STAT Tool Suite. In: Proc. DARPA Information Survivability Conference and Exposition. (2000)
19. Vigna, G., Kemmerer, R.A., Blix, P.: Designing a Web of Highly-Configurable Intrusion Detection Sensors. In: Proc. Symposium on Recent Advances in Intrusion Detection. (2001)
20. Vigna, G., Kemmerer, R.A.: NetSTAT: A Network-based Intrusion Detection System. Journal of Computer Security **7**(1) (1999) 37–71
21. Porras, P.A., Neumann, P.G.: EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In: Proc. National Information Systems Security Conference. (1997)
22. Blanc, M., Oudot, L., Glaume, V.: Global Intrusion Detection: Prelude Hybrid IDS. Technical report (2003)
23. Intrusion Detection Message Exchange Format. `http://www.ietf.org/html.charters/idwg-charter.html`